

Дэн Гасфилд

С троки, деревья и последовательности в алгоритмах

Информатика
и вычислительная биология



НЕВСКИЙ
ДИАЛЕКТ

Dan Gusfield

Algorithms on String, Trees, and Sequences

Computer science
and computational biology

University of California, Davis

Cambridge
University Press

Дэн Гасфилд

С
троки,
деревья
и последовательности
в алгоритмах

Информатика
и вычислительная биология

Перевод с английского И. В. Романовского

Невский Диалект
БХВ-Петербург

Санкт-Петербург
2003

УДК 519+575
ББК 22.19+28.070
Г 22

Гасфилд Дэн

Г 22 Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диалект; БХВ-Петербург, 2003. — 654 с.: ил.

Книга Д. Гасфилда написана на основе лекций, которые автор читает в Университете Дэвиса, Калифорния.

В ней, по-видимому, впервые подробно излагается круг математических вопросов, связанных с применением математики и информатики в задачах вычислительной молекулярной биологии. В этом направлении за последнее десятилетие накопилось целое море фактов, в том числе замечательных новых постановок задач, теоретических исследований и данных. Предлагаемая книга — их первое систематическое изложение.

Книга полезна научным работникам, студентам многих специальностей (не только по молекулярной биологии и по информатике) и даже старшеклассникам, которые захотят самостоятельно познакомиться с современными алгоритмами обработки практической информации. Она станет хорошим подарком талантливому подростку.

ISBN 0-521-58519-8 (англ.)
ISBN 5-7940-0103-8 (“Невский Диалект”)
ISBN 5-94157-321-9 (“БХВ-Петербург”)

© Dan Gusfield, 1997
© “Невский Диалект”, 2003

Оглавление

Предисловие переводчика	10
Предисловие	12
Часть I. Точное совпадение строк: основная задача	19
1. Точное совпадение	25
1.1. Наивный метод	25
1.2. Препроцессная обработка	27
1.3. Основной препроцессинг образца	27
1.4. Основной препроцессинг за линейное время	29
1.5. Простейший алгоритм с линейным временем	31
1.6. Упражнения	33
2. Точное совпадение: классические методы	38
2.1. Введение	38
2.2. Алгоритм Бойера–Мура	39
2.3. Алгоритм Кнута–Морриса–Пратта	46
2.4. Поиск строк в реальном времени	51
2.5. Упражнения	53
3. Более глубокий взгляд	59
3.1. Метод Бойера–Мура с линейной оценкой времени	59
3.2. Линейная оценка Коула	64
3.3. Исходный препроцессинг по методу КМП	74
3.4. Точный поиск набора образцов	78
3.5. Три приложения точного множественного поиска	88
3.6. Поиск образца, заданного регулярным выражением	93
3.7. Упражнения	95
4. Получисленное сравнение строк	99
4.1. Что использовать: сравнения или арифметику?	99
4.2. Метод <i>Shift-And</i>	99
4.3. Задача о счете совпадений и FFT	103
4.4. “Дактилоскопические” методы	107
4.5. Упражнения	115
Часть II. Суффиксные деревья и их использование	117
5. Введение в суффиксные деревья	119
5.1. Краткая история	120
5.2. Основные определения	121
5.3. Побуждающий пример	122
5.4. Наивный алгоритм построения суффиксного дерева	124

6. Построение суффиксных деревьев за линейное время	126
6.1. Алгоритм Укконена	126
6.2. Алгоритм Вайнера	141
6.3. Алгоритм Мак-Крейга	150
6.4. Обобщенное суффиксное дерево для набора строк	151
6.5. Вопросы практической реализации	152
6.6. Упражнения	155
7. Первые приложения суффиксных деревьев	158
7.1. APL1: Точное совпадение строк	159
7.2. APL2: Суффиксные деревья и множественное точное совпадение	159
7.3. APL3: Задача о подстроке для базы образцов	160
7.4. APL4: Наибольшая общая подстрока двух строк	162
7.5. APL5: Распознание загрязнения ДНК	162
7.6. APL6: Общие подстроки более чем двух строк	164
7.7. APL7: Построение меньшего ориентированного графа	166
7.8. APL8: Обратная роль суффиксных деревьев	170
7.9. APL9: Эффективный по памяти алгоритм нахождения наибольшей общей подстроки	174
7.10. APL10: Проверка совпадения суффикса с префиксом во всех парах	174
7.11. Введение в повторяющиеся структуры в молекулярных строках	177
7.12. APL11: Нахождение максимальных повторяющихся структур	183
7.13. APL12: Линеаризация циклической строки	189
7.14. APL13: Суффиксные массивы — большее сокращение памяти	190
7.15. APL14: Суффиксные деревья в геномных проектах	199
7.16. APL15: Подход Бойера–Мура к множественному совпадению	200
7.17. APL16: Сжатие данных по методу Зива–Лемпеля	208
7.18. APL17: Код минимальной длины для ДНК	212
7.19. Другие приложения	213
7.20. Упражнения	213
8. Общий наименьший предшественник	227
8.1. Введение	227
8.2. Предполагаемая модель машины	229
8.3. Полные двоичные деревья: очень простой случай	229
8.4. Как разрешать запросы об <i>lca</i> в \mathcal{B}	230
8.5. Первые шаги в отображении \mathcal{T} в \mathcal{B}	231
8.6. Отображение \mathcal{T} в \mathcal{B}	234
8.7. Препроцессинг \mathcal{T} за линейное время	235
8.8. Ответы на запросы об <i>lca</i> за константное время	237
8.9. Двоичное дерево не очень нужно	240
8.10. Для пуристов: как избежать битовых операций	240
8.11. Упражнения	241
9. Дополнительные приложения суффиксных деревьев	245
9.1. Наибольшее общее продолжение: мост к неточному совпадению	245
9.2. Нахождение всех максимальных палиндромов за линейное время	247
9.3. Точное совпадение с джокерами	249
9.4. Задача о k несовпадениях	250
9.5. Приблизительные палиндромы и повторы	251
9.6. Более быстрые методы для tandemных повторов	252

9.7.	Решение задачи о множественной общей подстроке за линейное время	256
9.8.	Упражнения	259
Часть III.	Неточное сопоставление, выстраивание последовательностей и динамическое программирование	261
10.	Значение сравнения (под)последовательностей	265
11.	Ядро методов редактирования строк и выстраивания	269
11.1.	Введение	269
11.2.	Редакционное расстояние между двумя строками	269
11.3.	Вычисление расстояния динамическим программированием	272
11.4.	Редакционные графы	278
11.5.	Взвешенное редакционное расстояние	279
11.6.	Сходство строк	281
11.7.	Нахождение подстрок высокого сходства	286
11.8.	Пропуски	292
11.9.	Упражнения	303
12.	Улучшение процедур выстраивания	313
12.1.	Вычисление выравниваний в линейной памяти	313
12.2.	Ускорение для ограниченного числа различий	320
12.3.	Методы исключения	332
12.4.	Еще о суффиксных деревьях и гибридном ДП	343
12.5.	Быстрый алгоритм для задачи <i>lcs</i>	351
12.6.	Вогнутый вес пропусков	359
12.7.	Метод “четырех русских”	369
12.8.	Упражнения	375
13.	Развитие основных задач	379
13.1.	Параметрическое выравнивание последовательностей	379
13.2.	Вычисление субоптимальных выравниваний	391
13.3.	Сцепление различных локальных выравниваний	395
13.4.	Упражнения	399
14.	Сравнение многих строк — Святой Грааль	403
14.1.	Зачем нужно множественное сравнение строк?	403
14.2.	Три “крупномасштабных” применения	408
14.3.	Представление семейств и суперсемейств	408
14.4.	Выводы о структурах	414
14.5.	Введение в вычисление множественных выравниваний строк	416
14.6.	Выравнивание с целевой функцией типа суммы пар	417
14.7.	Выравнивание с консенсусными целевыми функциями	426
14.8.	Множественное выравнивание по (филогенетическому) дереву	430
14.9.	Замечания о приближениях с ограниченной ошибкой	434
14.10.	Обычные методы множественного выравнивания	436
14.11.	Упражнения	444

15. Базы данных для последовательностей	449
15.1. Истории успешного поиска в базах данных	450
15.2. Промышленность баз данных	453
15.3. Алгоритмические вопросы поиска данных	455
15.4. Реальный поиск в базе данных для последовательностей	456
15.5. FASTA	457
15.6. BLAST	459
15.7. PAM: первые главные матрицы подстановки аминокислот	462
15.8. PROSITE	466
15.9. BLOCKS и BLOSUM	467
15.10. Матрицы подстановки BLOSUM	468
15.11. Дополнительные вопросы поиска в базе данных	469
15.12. Упражнения	474
Часть IV. Другие задачи: текущие, родственные и просто изящные	475
16. Карты, картирование, упорядочение и надстроки	478
16.1. Взгляд на задачи картирования и секвенирования ДНК	478
16.2. Картирование и геномный проект	479
16.3. Физические и генетические карты	479
16.4. Физическое картирование	481
16.5. Физическое картирование: STS и библиотеки клонов	482
16.6. Физическое картирование: радиационно-гибридное	485
16.7. Физическое картирование: дактилограммы	490
16.8. Вычисление самой плотной раскладки	492
16.9. Физическое картирование: последние замечания	497
16.10. Введение в выравнивание карт	499
16.11. Крупномасштабная расшифровка и сборка последовательности	501
16.12. Направленная расшифровка	502
16.13. Нисходяще-восходящая расшифровка: картина, использующая YAC	503
16.14. Дробовая расшифровка ДНК	508
16.15. Сборка последовательности	508
16.16. Заключительные комментарии о нисходяще-восходящей расшифровке	513
16.17. Задача о кратчайшей надстроке	514
16.18. Расшифровка гибридизацией	527
16.19. Упражнения	534
17. Строки и эволюционные деревья	539
17.1. Ультраметрические деревья и расстояния	542
17.2. Деревья с аддитивными расстояниями	551
17.3. Бережливость: символьно-ориентированное эволюционное воссоздание	554
17.4. Центральное место ультраметрической задачи	562
17.5. Максимальная бережливость, штейнеровы деревья	567
17.6. Снова филогенетическое выравнивание	569
17.7. Связи между множественным выравниванием и построением деревьев	572
17.8. Упражнения	573

18. Три короткие темы	579
18.1. Сравнение ДНК с белком при смещениях рамки	579
18.2. Предсказание гена	582
18.3. Молекулярные вычисления: при помощи (а не ради) строк ДНК	585
18.4. Упражнения	591
19. Модели мутаций на геномном уровне	594
19.1. Введение	594
19.2. Инверсионные перестройки генома	596
19.3. Знакопеременные инверсии	601
19.4. Упражнения	602
Эпилог	604
Библиография	608
Толковый словарь	632
Англо-русский словарь терминов	644
Предметный указатель	646

Предисловие переводчика

Эта книга вышла из печати в 1997 году. Через пять лет, в сентябре 2002 года, в интернетовском книжном магазине amazon.com она стоит на 8960-м месте по продажам, причем первые 10 тысяч мест — это высшая категория. По отзывам читателей, книга получила самую высокую оценку — ★ ★ ★ ★ ★.

Ну что ж. Можно сказать, что “для них”, на Западе, такая популярность этой объемистой научной книги достаточно естественна. Работы по вычислительной молекулярной биологии (в просторечии — по расшифровке ДНК) идут чрезвычайно интенсивно, и за последнее десятилетие накопилось целое море фактов, в том числе замечательных новых постановок задач, теоретических исследований и данных. Требуются обобщающие книги различной направленности. Обзор математической и вычислительной стороны вопроса нужен безусловно.

А “для нас”? Есть ли у нас такой фронт работ в молекулярной биологии? Я сначала думал: если нет, то должен быть. Казалось, что в любом случае нужно привлечь внимание молодых математиков и программистов к этой важной теме. Но пока готовился перевод книги, я узнал, что и у нас в стране есть центры, где исследования ведутся на самом современном уровне (некоторые из них цитируются автором), и при этих центрах готовятся молодые кадры, которым тоже нужна обобщающая литература. Им эта книга заведомо необходима.

Итак, предлагаемая вашему вниманию книга Дэна Гасфилда, профессора Университета Дэвиса (Калифорния), подводит итог математическим исследованиям в вычислительной молекулярной биологии. В книге описываются наиболее важные постановки задач и приводящие к ним модели биологических явлений, алгоритмы для решения этих задач, результаты математического анализа задач и алгоритмов, результаты практических вычислений.

Можно надеяться, что круг ее читателей не ограничится теми, кто работает в вычислительной молекулярной биологии. Книга может быть замечательным пособием по применению математического моделирования и вычислительных средств в трудных современных задачах. В ней на важном, актуальном и поразительно интересном примере вы увидите:

какой сложный путь проходят исследователи, разрабатывающие и использующие математическую модель;

как эта модель постепенно изменяется, начиная от простейших, совсем наивных постановок, и насколько несовершенной она еще остается, когда с ее помощью уже получаются осмысленные результаты;

как совершенствуются численные методы, как удается оценивать их трудоемкость и насколько различным — в зависимости от обстоятельств — может быть подход к этой оценке;

каково соотношение между методами с хорошей теоретической оценкой и методами, используемыми на практике;

откуда берутся исходные данные, с какими формами этих данных нужно иметь дело, как эти данные хранятся.

В таком своем качестве книга будет очень полезна научным работникам, студентам многих специальностей и даже старшеклассникам, которые захотят самостоятельно познакомиться с современными алгоритмами обработки практической информации. Думаю, что книга будет хорошим подарком талантливому подростку.

Для облегчения сопоставления английских и русских терминов мы поместили в конце книги небольшой англо-русский словарь.

Как обычно, некоторые термины было трудно переводить. Совершенно неожиданно возникли сложности со словом *строка*: автор занимается анализом строк символов (*strings*) и при этом создает таблицы, состоящие из отдельных строк (*rows*). Чтобы избежать путаницы, пришлось строки таблиц называть *строчками*.

Возможно, биологи будут порицать меня за некоторые отклонения от терминологии, стандартной для биологической литературы: зачастую термин получается транслитерацией. Например, *pattern* “переводится” как *паттерн*, хотя слово *образец*, употребляемое в математических книгах, ничем не хуже. Для слова *sequencing* здесь использовано и принятное у биологов, но пугающее *секвенирование* и более простое *расшифровка*, встречающееся в нашей литературе. Метод *shotgun* мы предпочли назвать *дробовым*, а не *шотган*. Мы старались упоминать все существующие варианты перевода.

Трудно было решиться (но я решился) на изменение использования терминов *выпуклая функция* и *вогнутая функция*. Автор называет функцию, у которой отрицательна вторая производная, выпуклой. Так как в русскоязычной математической литературе разброд терминологии в этом вопросе уже ликвидирован и начинать его заново явно не стоит, терминология была сменена — функция называется *вогнутой*.

При подготовке издания в библиографию были добавлены ссылки на работы, вышедшие после выхода оригинала книги, и на работы на русском языке. Они включены в основной список в алфавитном порядке с номерами, сопровождаемыми буквой, например [286а], чтобы избежать смены номеров ссылок и упростить поиск.

*И. В. Романовский,
профессор математико-механического факультета
С.-Петербургского университета*

*Посвящается памяти Гена Лаулера,
учителя, коллеги и друга*

Предисловие

История и мотивация

Я начал писать летом 1988 г., даже не предполагая тогда, что результатом станет эта книга. Я входил тогда в исследовательскую группу по информатике *) (сначала по биоинформатике) в Центре человеческого генома Лаборатории Lawrence Berkeley **). Наша группа придерживалась стандартного предположения, что можно получить биологически осмысленные результаты, рассматривая ДНК как одномерную строку символов и абстрагируясь от той реальности, что ДНК — это гибкая трехмерная молекула, взаимодействующая в динамической среде с белками и РНК и проходящая жизненный цикл, в котором даже классическая линейная хромосома существует только часть времени. Аналогичное, но более сильное предположение делалось и о белках; утверждалось, например, что вся информация, нужная для правильного трехмерного складывания, содержится в самой последовательности белка и в основном не зависит от биологической среды, в которой белок живет. Это предположение недавно модифицировали, но в значительной степени оно остается актуальным [297].

Для небиологов эти гипотезы были (и остаются) божьим даром, позволяющим быстро войти в восхитительную и важную область. Необходимость исследований на уровне последовательностей еще усиливалась такими высказываниями, как:

“Цифровая информация, которая лежит за биохимией, клеточной биологией и развием, может быть представлена обычной строкой из символов G, A, T и C. Для биологии организмов эта строка является исходной структурой данных” [352];

и

“В самом прямом смысле молекулярная биология касается только последовательностей. Прежде всего, она старается свести сложные биологические феномены к взаимодействиям между определенными последовательностями...” [449];

и

“Окончательная подоплека всех целенаправленных структур и поведения живых существ заложена в последовательности остатков нарождающихся полипептидных цепей... Это в прямом смысле тот уровень организации, на котором должен быть найден (если он существует) секрет жизни” [330].

Итак, не очень беспокоясь о более трудных химических и биологических аспектах ДНК и белков, наша программистская группа получила возможность рассматривать множество биологически важных задач, определенных прежде всего на *последовательностях*, или (если быть ближе к языку информатики) на *строках*: воссоздание длинных строк ДНК по перекрывающимся фрагментам строк; определение физических и генетических карт по опытным данным, взятым из протоколов различных экспериментов; хранение, выдача и сравнение строк ДНК; поиск сходства в двух или более строках; поиск родственных строк и подстрок в базах данных; формализация и использование различных понятий родственности строк; поиск новых или плохо

*) Так мы будем переводить computer science. — *Прим. перев.*

**) Другими постоянными членами этой группы были Уильям Чанг (William Chang), Ген Лаулер (Gene Lawler), Дейлит Наор (Dalit Naor) и Френк Олкен (Frank Olken).

определенных образцов (паттернов), часто встречающихся в ДНК; поиск структурных образцов в ДНК и белках; определение вторичной (двумерной) структуры РНК; нахождение сохраненных, но слабо выраженных паттернов во многих последовательностях ДНК и белков и др.

Мы организовали нашу работу в двух направлениях. Во-первых, нам нужно было изучить соответствующие разделы биологии, лабораторные протоколы и существующие алгоритмические методы, применяемые биологами. Во-вторых, мы старались собрать литературу по информатике об идеях и алгоритмах, которые еще не используются биологами, но, *вероятно, могли бы* пригодиться в уже рассматриваемых задачах или в задачах, которые, по нашему мнению, могут возникнуть, когда станет доступен большой объем расшифрованных ДНК и белков.

Наша задача

Никто из нас не был знатоком строковых алгоритмов. К этому времени я знал по учебникам алгоритм Кнута–Морриса–Пратта и глубоко заблуждался в отношении алгоритма Бойера–Мура (при каких условиях он является алгоритмом с линейным временем и как выполнить в линейное время *сильную* предварительную подготовку). Я догадывался о пользе динамического программирования для вычисления редакционного расстояния, но в остальном был мало знаком с конкретными строковыми алгоритмами в биологии. Я специализировался в области комбинаторной оптимизации, но всегда интересовался алгоритмами построения эволюционных деревьев и для удовлетворения этого интереса немножко изучал генетику и молекулярную биологию.

В чем мы тогда действительно нуждались и чего не имели — это всеобъемлющий связный текст по строковым алгоритмам, который руководил бы нашим образованием. В то время было несколько руководств по информатике, содержащих одну–две главы о строках, обычно со строгим изложением метода Кнута–Морриса–Пратта и поверхностным изложением метода Бойера–Мура, а также, возможно, элементарным обсуждением поиска совпадения с ошибками. Имелось также несколько хороших обзорных статей, которые охватывали предмет шире, но недостаточно глубоко. Было несколько учебников и сборников статей по применению компьютеров и алгоритмов для анализа последовательностей, написанных с точки зрения биолога. Некоторые из них прекрасно демонстрировали преимущества использования компьютеров в биологии и возможные ловушки, но опускали вопросы алгоритмической строгости и охватывали лишь узкий диапазон технических средств. Наконец, имелся плодотворный текст *Временные отклонения, редактирование строк и макромолекулы: теория и практика сравнения строк* [389] под редакцией Д. Санкова и Дж. Краскала, который служил мостиком между алгоритмами и биологией и содержал много примеров применения динамического программирования. Однако он также был значительно *уже наших интересов* и к тому же немного устарел.

Более того, большинство доступных источников с обеих сторон — и информатики, и биологии — концентрировало внимание на поиске совпадений строк, задаче поиска точной или “почти точной” копии образца в данном тексте. Задачи совпадения занимают центральное место, но, как мы увидим в этой книге, они составляют только часть совокупности важных вычислительных задач, определенных на строках. Таким образом, тем летом мы осознали потребность в строгом и фундаментальном изложении общего аспекта алгоритмов, оперирующих на строках, наряду с точным

изложением конкретных методов, потенциально и ныне значимых в вычислительной биологии. Эта книга является попыткой такого двойственного, и интегрированного, изложения.

Зачем мешать в одной книге информатику и вычислительную биологию?

Мой интерес к вычислительной биологии возник в 1980 г., когда я начал читать статьи о построении деревьев эволюции. Этот побочный интерес позволил мне случайно выйти за рамки лихорадочных, гиперсоревновательных “горячих” тем, на которых сфокусировалась теоретическая информатика. В то время вычислительная молекулярная биология была для информатики большой непознанной областью, хотя статистики и математики уже действовали в ней активно (особенно Майкл Уотермен и Дэвид Санков, которые в значительной степени создали эту область). В раннем периоде важные работы о вычислительных аспектах биологии (такие, как работа Бунемана [83]) появлялись не в основных средоточиях работ по информатике, а в самых неожиданных местах вроде конференций по *вычислительной археологии* [226]. Но семнадцать лет спустя вычислительная биология сама стала “горячим” местом, и многие информатики уже спешат в эту (теперь более горячечную и более соревновательную) область [280]. Что им нужно учить?

Проблема в том, что возникающая область вычислительной молекулярной биологии недостаточно хорошо определена, и ее определение делается все более трудным из-за быстрых изменений в самой молекулярной биологии. Но все же алгоритмы, которые оперируют с данными о молекулярных последовательностях (строках), находятся в самом сердце вычислительной молекулярной биологии. Глобальный вопрос вычислительной молекулярной биологии — это как “извлечь” по возможности больше “настоящей биологии” из данных о молекулярных последовательностях (ДНК, РНК и белков). Получать данные о последовательностях можно гораздо дешевле и быстрее (и в большем объеме) по сравнению с традиционными лабораторными методами исследования. Использование данных в форме последовательностей уже стало центральным в нескольких подобластях молекулярной биологии, и полное воздействие больших объемов данных в таком виде мы еще увидим. Следовательно, алгоритмы, которые оперируют со строками, сохранят свое место в теснейшем пересечении и взаимодействии между информатикой и молекулярной биологией. В этом случае информатикам нужно изучать наиболее успешно применяемые методы работы со строками. Но этого недостаточно.

Информатики должны учиться *фундаментальным* идеям и методам, которые будут жить долго после того, как забудутся сегодняшние основные мотивирующие приложения. Необходимо осваивать методы, чтобы уметь выделить будущие задачи и приложения и справиться с ними. Значительных достижений в вычислительной биологии можно было бы добиться, обобщая и адаптируя алгоритмы из информатики, даже когда исходный алгоритм не имеет очевидного приложения к биологии. Это иллюстрируется несколькими недавними сублинейными по времени методами приближенного сравнения, предназначенными для поиска в базах данных и использующими взаимодействие между точными методами сравнения из информатики и методами динамического программирования, уже применяемыми в молекулярной биологии.

Поэтому информатику, который хочет войти в общую область вычислительной молекулярной биологии и изучает строковые алгоритмы с этой конечной целью, следовало бы получить подготовку в строковых алгоритмах, которая много шире, чем обзорный тур по методам известных сейчас приложений. Молекулярная биология и информатика меняются слишком быстро для того, чтобы такой узкий подход был успешен. Более того, информатики-теоретики пытаются развивать эффективные алгоритмы несколько иначе, чем другие алгоритмисты. Мы обращаем больше внимания на доказательства корректности, анализ наихудших случаев, обоснование нижних границ, рандомизированный анализ алгоритмов и результаты с гарантированным приближением (среди других методов) для *управления развитием* практических эффективных алгоритмов. Наше “относительное преимущество” частично лежит в специальных знаниях и опыте их использования. Таким образом, даже если бы я писал книгу для информатиков, которые хотят работать только в вычислительной биологии, я бы все равно предпочел включить широкий диапазон алгоритмической техники из чистой информатики.

В этой книге я охватил широкий спектр строковой техники, выйдя за границы их установленной полезности; однако я отобрал из многих возможных иллюстраций те подходы, от которых ожидаются наибольшие *потенциальные применения* в будущей молекулярной биологии. Потенциальные применения, особенно идей, а не конкретных методов, и притом к *предполагаемым*, а не к существующим задачам, — дело мнений и спекуляций. Несомненно, часть материала этой книги никогда не найдет прямого применения в биологии, а другая часть найдет совершенно неожиданное употребление. Некоторые строковые алгоритмы, которые всего несколько лет назад считались не относящимися к биологии, стали приниматься биологами и в крупномасштабных проектах, и в более узких технических задачах. Методы, ранее отвергавшиеся из-за того, что они первоначально работали с (точными) строковыми задачами, где предполагались *идеальные данные*, стали частью более грубых методов, справляющихся с неидеальными данными.

Что в этой книге есть

В соответствии со сказанным эта книга представляет собой строгое и не специализированное изложение всей области детерминированных алгоритмов, которые работают на строках и последовательностях. Многие из этих алгоритмов используют деревья в качестве структур данных или встречаются в биологических задачах, относящихся к деревьям эволюции. Это вызвало включение слова “деревья” в название.

Предполагаемый читатель этой книги — профессионал-информатик, исследователь, аспирант или студент, хотя ее могут читать многие биологи (и, конечно, математики) с достаточной алгоритмической подготовкой. Эта книга должна быть и справочником, и основным учебником по курсу чистой информатики и по курсу вычислительной биологии, ориентированной на информатику.

Обсуждения биологических приложений встречаются на протяжении всей книги, но больше сосредоточены в последних параграфах части II и в большинстве — частей III и IV. Я обсуждаю подробно ряд биологических вопросов для того, чтобы дать читателю детальное понимание причин, по которым многие биологические задачи рассматриваются как задачи на строках, и чтобы показать то многообразие (часто

очень воображаемых) технических приемов использования в молекулярной биологии строковых алгоритмов.

Эта книга охватывает все классические вопросы и большинство важных методов в области строковых алгоритмов, за тремя исключениями. Она только слегка касается вероятностного анализа и не обсуждает параллельных алгоритмов, а также элегантных теоретических результатов об алгоритмах для бесконечных алфавитов и об алгоритмах, использующих только константную вспомогательную память^{*)}. В книге не обсуждаются также методы стохастического типа, которые идут от машинного обучения, хотя некоторые из рассматриваемых алгоритмов широко используются в таких методах в качестве вспомогательных средств. Кроме названных исключений книга включает в себя все важные способы рассмотрения строковых алгоритмов. Читатель, который освоит предлагаемый материал, достигнет широкого и глубокого понимания этой области и достаточной искусности, чтобы предпринимать свои собственные исследования.

Отражая мой собственный опыт, книга строго обсуждает каждый из вопросов, обычно с полными доказательствами поведения (корректность, время в наихудшем случае и память). Более важно, что она делает акцент на идеях и представляемых вариациях методов, а не на простом перечислении возможных алгоритмов. Для лучшего изложения идей и побуждения к открытиям я часто представляю сложный алгоритм, начиная с наивного неполноценного варианта, а затем последовательно ввожу дополнительные улучшения и детали реализации, которые приводят к желаемому результату.

Книга содержит новые подходы, которые я развел для объяснения излагаемого материала. В частности, я представил методы предварительной обработки, используемые в методах Кнута–Морриса–Пратта, Бойера–Мура и некоторых других алгоритмах поиска образца за линейное время, иначе, чем в классических методах, унифицировав и упростив препроцессные действия, необходимые для этих алгоритмов. Я полагаю, что мои (надеюсь, более простые и ясные) изложения построений за линейное время суффиксных деревьев и алгоритма нахождения наименьшего общего предка за константное время сделают эти важные методы более доступными и лучше понимаемыми. Я связал теоретические результаты из информатики относительно алгоритмов с сублинейным временем и широко используемые методы поиска в биологической базе данных. При обсуждении множественного выстраивания последовательностей я собрал вместе три основные целевые функции, которые предлагались для множественного выстраивания, и показал свойство непрерывности в приближенных алгоритмах для получающихся трех задач множественного выстраивания. Аналогично, глава о построении деревьев эволюции демонстрирует общность нескольких различных задач и методов их решения неизвестным ранее путем. На протяжении всей книги я рассматриваю многие вычислительные задачи, связанные с повторяющимися подстроками (феномен, широко распространенный в ДНК). Я исследую несколько различных способов определения повторяющихся подстрок и использую каждое конкретное определение при анализе вычислительных задач и алгоритмов на повторяющихся подстроках.

^{*)} Память, используемая алгоритмом, — очень важный практический аспект, и мы будем часто обсуждать его, но константная память представляется слишком серьезным ограничением в большинстве важных применений.

В этой книге я пытаюсь подробно объяснить, и с разумной скоростью, многие сложные методы, которые раньше были написаны исключительно для специалистов по строковым алгоритмам. Я избегаю детализированного программного кода, так как считаю, что он редко способствует объяснению интересных идей^{*)}, и предлагаю более 400 упражнений как для подкрепления материала книги, так и для более широкого охвата тематики.

Чего в этой книге нет

Позвольте мне ясно сформулировать, чего в этой книге нет. Ее нельзя рассматривать как *полный* текст по вычислительной молекулярной биологии, так как я считаю, что эта область включает в себя также вычисления над объектами, отличными от строк, деревьев и последовательностей. Но вычисления на строках и последовательностях образуют сердцевину вычислительной молекулярной биологии, и эта книга дает глубокое и широкое изложение вычислительной биологии, ориентированной на последовательности. Эта книга также и не справочник “рецептов” по анализу строк и последовательностей. Существует несколько книг, которые дают обзоры конкретных компьютерных пакетов, баз данных и вычислительных услуг и наряду с этим дают общее представление о том, как они работают. Наша книга с ее акцентом на идеи и алгоритмы не может с ними соревноваться. Наконец, с другой стороны, эта книга не пытается дать историю развития области строковых алгоритмов и ее создателей. Есть обширная литература, со многими повторами, независимыми переоткрытиями, противоречиями и конфликтами. Я делал некоторые исторические замечания и указывал читателю на источники, которые кажутся мне полезными, но я еще слишком рано пришел в эту область и недостаточно смел, чтобы претендовать на полную ее систематику. Поэтому я заранее приношу свои извинения тем многочисленным лицам, работа которых оказалась отражена не в должной мере.

В итоге

Эта книга представляет собой общий строгий текст по детерминистическим алгоритмам, оперирующем со строками, деревьями и последовательностями. Она охватывает весь спектр строковых алгоритмов от классической информатики до современной молекулярной биологии и, когда нужно, связывает эти две области. Это та книга, которую мне хотелось бы иметь, когда я начинал учиться строковым алгоритмам.

Благодарности

Мне бы хотелось поблагодарить Программу “Геном человека” Департамента энергии, Лабораторию Lawrence в Беркли, Национальный научный фонд, Программу по математической и молекулярной биологии и организаторов специального года по вычислительной биологии DIMACS (центра по дискретной математике и информатике) за поддержку моей работы и работы моих студентов и постдоков.

^{*)}Однако многие из алгоритмов этой книги запрограммированы в Си и доступны по адресу <http://wwwcsif.cs.ucdavis.edu/~gusfield/strpgms.html>

Персонально я выражаю свою глубокую признательность Уильяму Чангу (William Chang), Джону Кесесиоглу (John Kecesoglu), Джиму Найту (Jim Knight), Гену Лаулеру (Gene Lawler), Дэйлит Наор (Dalit Naor), Френку Олкену (Frank Olken), Р. Рави (R. Ravi), Полу Стэллингу (Paul Stelling) и Лушень Вангу (Lusheng Wang).

Я хотел бы также поблагодарить следующих лиц за помощь, которую они мне оказали на этом пути: Стивена Альтшуля (Stephen Altschul), Дэвида Аксельрода (David Axelrod), Дуга Брутлега (Doug Brutlag), Арчи Коббса (Archie Cobbs), Ричарда Коля (Richard Cole), Расса Дулилтта (Russ Doolittle), Мартина Фараха (Martin Farach), Джейн Гитчье (Jane Gitschier), Джорджа Хартцелла (George Hartzell), Пола Хортона (Paul Horton), Роберта Ирвинга (Robert Irving), Сорин Истраил (Sorin Istrail), Тао Джияня (Tao Jiang), Дика Карпа (Dick Karp), Дину Кравец (Dina Kravets), Гэда Ландау (Gad Landau), Юди Манбер (Udi Manber), Марси Мак-Клур (Marci McClure), Кевина Мэрфи (Kevin Murphy), Гена Майерса (Gene Myers), Джона Нгуена (John Nguyen), Майка Патерсона (Mike Paterson), Уильяма Пирсона (William Pearson), Павла Певзнера (Pavel Pevzner), Фреда Робертса (Fred Roberts), Гершеля Сейфера (Hershel Safer), Баруха Шибера (Baruch Schieber), Рона Шамира (Ron Shamir), Джая Снодди (Jay Snoddy), Элизабет Свидык (Elizabeth Sweedyk), Сильвию Спенделер (Sylvia Spengler), Мартина Томпа (Martin Tompa), Эско Укконена (Esko Ukkonen), Мартина Вингрона (Martin Vingron), Тэнди Уорнгу (Tandy Warnow) и Майка Уотермана (Mike Waterman).

Часть I

Точное совпадение строк:
основная задача

Точное совпадение: в чем задача?

Пусть задана строка P , именуемая *образцом* или *паттерном* (pattern), и более длинная строка T , именуемая *текстом* (text). Задача о **точном совпадении** заключается в отыскании всех вхождений образца P в текст T .

Например, если $P = aba$ и $T = bbabaxababay$, то P входит в T , начиная с позиций 3, 7 и 9. Заметим, что два вхождения P могут перекрываться, как это видно по вхождениям P в позициях 7 и 9.

Важность задачи о точном совпадении

Практическое значение задачи о точных совпадениях должно быть очевидно каждому, кто использует компьютер. Эта задача возникает в широком спектре приложений, настолько обширном, что всего даже не перечесть. Некоторые из наиболее общих приложений встречаются в текстовых редакторах; в таких утилитах, как *grep* в UNIX; в информационно-поисковых текстовых системах, таких как Medline, Lexis и Nexis; в поисковых системах библиотечных каталогов, которые в большинстве больших библиотек заменили обычные карточные каталоги; и интернетовских браузерах и краулерах, которые просеивают огромные количества текстов в поисках материалов, содержащих ключевые слова^{*)}; в программах, которые читают интернетовские новости и могут отыскивать статьи на требуемую тему; в гигантских цифровых библиотеках, которые уже запланированы на ближайшее будущее; в электронных журналах, которые уже “публикуются” интерактивно; в обслуживании телефонных справочников; в интерактивных энциклопедиях и других средствах обучения на компакт-дисках; в интерактивных словарях и тезаурусах, особенно таких, где организованы перекрестные ссылки (проект *Oxford English Dictionary* создал электронную интерактивную версию этого словаря, содержащую 50 миллионов слов), и в различных специализированных базах данных. В молекулярной биологии есть несколько сотен специализированных баз данных, содержащих исходные строки ДНК, РНК и аминокислот, а также обработанные образцы (называемые motifs — мотивами), полученные из исходных строк. Некоторые из этих баз данных мы рассмотрим в главе 15.

Несмотря на то что практическое значение задачи о точных совпадениях бесспорно, вас могут спросить, действительно ли эта задача сохраняет еще исследовательский или учебный интерес. Разве задача о точных совпадениях не решена так хорошо, что ее можно поместить в черный ящик и считать само собой разумеющейся? Вот сейчас я редактирую файл в девяносто страниц, используя “древний” текстовый редактор и персональный компьютер (486), и каждая команда точного совпадения, которую я подаю, выполняется так быстро, что я моргнуть не успеваю. И это погрывает тебя, если ты пишешь книгу с большим разделом, посвященным точным совпадениям. Действительно, осталось ли здесь что-нибудь еще не сделанное?

^{*)} Я недавно зашел в Интернете на страницу Alta Vista, которую ведет Digital Equipment Corporation. База данных Alta Vista содержит более 21 миллиарда слов, собранных в более чем 10 миллионах сайтов. Поиск всех сайтов, упоминающих “Mark Twain”, занял пару секунд и сообщил, что этому запросу удовлетворяют 200 тысяч сайтов.

В ответ нужно сказать, что, и правда, для типичных приложений, связанных с редактированием текстов, уже остается сделать не так много. Задача поиска точных совпадений для них уже решена (хотя текстовым процессорам более изощренные средства работы со строками пригодились бы). Но дело радикально меняется, если обратиться к другим приложениям. Пользователи Melvyl, интерактивного каталога библиотечной системы Университета Калифорнии, часто сталкиваются с долгими утомительными задержками при обработке даже совсем простых поисковых запросов. Даже *grepование*^{*)} большого каталога может продемонстрировать, что задача о точных совпадениях еще не тривиальна. Недавно мы использовали GCG (очень популярный интерфейс для поиска ДНК и белков в базах данных), разыскивая в Genbank (это основная в США база данных ДНК) строку из тридцати символов. Потребовалось больше четырех часов поиска (на локальной машине, использующей локальную копию банка данных), чтобы обнаружить, что искомой строки в банке нет.^{**)} А сегодняшний Genbank — это скромная доля того объема, который накопится, когда различные геномные программы заработают на полную мощность, выплескивая огромные количества расшифрованных ДНК. Конечно, существуют более быстрые программы поиска в базах данных (например, BLAST) и можно использовать более быстрые машины (например, доступен сервер электронной почты для точного и неточного поиска совпадений в базе данных, который работает на компьютере MasPar с 4 тысячами процессоров). Но дело в том, что задача о точных совпадениях решена еще не настолько эффективно и универсально, чтобы не нуждаться в дальнейшем внимании. Она будет оставаться интересной, поскольку размер баз данных растет, а также потому, что поиск точных совпадений будет оставаться *подзадачей* во многих комплексных задачах поиска, которые еще будут разрабатываться. Многие из них будут обсуждаться в этой книге.

Но, возможно, подробное изучение задачи *точных* совпадений нужно прежде всего для понимания различных идей, разработанных в связи с ней. Даже предполагая, что задача о точных совпадениях сама по себе имеет уже достаточно способов решения, вся в целом область строковых алгоритмов остается жизненной и открытой, и навыки, полученные от изучения задачи точных совпадений, могут оказаться решающими при работе с менее изученными задачами. Эти навыки приобретают три формы: специфические алгоритмы, общие алгоритмические подходы и техника анализа и доказательств. Все три формы представлены в этой книге, но наибольшее внимание уделяется стилю и технике доказательств.

Обзор части I

В главе 1 мы представим наивные решения задачи о точных совпадениях и предложим основные средства, необходимые для получения более эффективных методов. Хотя классические методы решения этой задачи не будут продемонстрированы до главы 2, мы покажем в конце главы 1, что использование одних только основных

^{*)} Автор образует здесь глагол *to grep*, используя название уже упоминавшейся раньше известной утилиты UNIX. Эта утилита применяется для поиска вхождений в текст образцов, заданных регулярным выражением. — *Прим. перев.*

^{**)} Позднее мы повторили эксперимент, используя алгоритм Бойера-Мура, на нашей собственной сырой копии Genbank. Поиск занял меньше десяти минут, из которых большая часть была израсходована на перекачку текста между диском и компьютером, а собственно поиск занял меньше минуты.

средств дает простой алгоритм для задачи о точных совпадениях с линейным временем. Глава 2 развивает некоторые классические методы поиска совпадений, используя основные средства, развитые в главе 1. Глава 3 более глубоко анализирует эти методы и их обобщения. Глава 4 продвигает нас в совершенно ином направлении, разрабатывая методы для поиска точных совпадений, основанные на операциях арифметического типа, а не на сравнениях символов.

Хотя главное внимание в части I уделяется точным совпадениям, в ней все же затрагиваются некоторые аспекты неточных совпадений и использования шаблонов (*wild cards*). Задача о точном совпадении будет обсуждаться также и в части II, где она и ее обобщения будут решаться с использованием суффиксных деревьев.

Основные строковые определения

Мы будем вводить новые определения главным образом там, где они впервые используются, но некоторые определения настолько важны, что их мы введем сейчас.

Определение. Стока S — это упорядоченный список символов, записанных подряд слева направо. Для любой строки S обозначим через $S[i..j]$ (сплошную) подстроку S , которая начинается в позиции i и заканчивается в позиции j строки S . В частности, $S[1..i]$ называется префиксом строки S , кончающимся в позиции i , а $S[i..|S|]$ — суффиксом строки S , начинающимся в позиции i . Всюду $|S|$ обозначает число символов в строке S .

Определение. Стока $S[i..j]$ пуста, если $i > j$.

Например, *california* — это строка, *lifo* — это подстрока, *cal* — это префикс, а *ornia* — это суффикс.

Определение. Собственные префикс, суффикс и подстрока S — это, соответственно, префикс, суффикс и подстрока S , не совпадающие с S и не пустые.

Определение. Для любой строки S через $S(i)$ обозначается ее i -й символ.

Обычно мы будем использовать символ S для обозначения произвольной фиксированной строки, которой не приписывается дополнительных свойств или роли. Однако если известно, что строка играет роль образца или текста, она, соответственно, будет обозначаться через P или T . Строчные греческие буквы ($\alpha, \beta, \gamma, \delta$) будут обозначать переменные строки, а строчные латинские буквы — отдельные переменные символы.

Определение. При сравнении двух символов мы скажем, что они совпадают (*match*), если они равны; в противном случае мы скажем, что они не совпадают (*mismatch*).

Терминологическая путаница

Слова “строка” и “слово” часто в информатике используются как синонимы, но для ясности в этой книге мы никогда не будем использовать “слово”, если подразумевается “строка”. (Однако мы используем “слово” в его обычном грамматическом смысле.)

Еще больше запутывает синонимичное использование слов “строка” и “последовательность”, особенно в биологической литературе. Это может вызвать путаницу, поскольку “подстроки” и “подпоследовательности” — абсолютно различные математические объекты, и алгоритмы для задач, связанных с подстроками, совершенно не такие, как для задач с подпоследовательностями. Символы в подстроке строки S должны идти *подряд*, тогда как символы подпоследовательности могут в строке перемежаться символами, не принадлежащими подпоследовательности. Хуже того, в биологической литературе часто можно встретить слово “последовательность”, используемое вместо “подпоследовательность”. Поэтому для ясности в этой книге мы будем всегда проводить различие между “подпоследовательностью” и “подстрокой” и никогда не будем употреблять “последовательность” вместо “подпоследовательности”. Мы будем использовать слово “строка” при обсуждении чисто математических вопросов, а в контексте биологических приложений — вперемешку “последовательность” или “строка”. Конечно, мы будем применять термин “последовательность” в его стандартном математическом смысле.*)

Первые две части этой книги в основном имеют дело со строками и подстроками. Задачи, связанные с подпоследовательностями, рассматриваются в частях III и IV.

*) В русском переводе появляется еще одна трудность: нам придется различать строки текста (*strings*) и строки таблицы (*rows*). Мы были вынуждены говорить всегда не о *строках*, но о *строчках таблицы*. — *Прим. перев.*

Глава 1

Точное совпадение

1.1. Наивный метод

Почти все обсуждения поиска точного совпадения начинаются с *наивного метода*, и мы следуем этой традиции. В наивном методе левый конец образца P ставится вровень с левым концом текста T и все символы P сравниваются с соответствующими символами T слева направо, пока либо не встретится несовпадение, либо не исчерпается P . В последнем случае констатируется, что найдено вхождение P в текст. При любом исходе P сдвигается на одну позицию вправо, и сравнения возобновляются, начиная с левого конца P . Процесс продолжается, пока правый конец P не зайдет за правый конец T .

Если обозначить через n длину P и через m длину T , в самом худшем случае число сравнений, выполняемых этим методом, равно $\Theta(nm)$. В частности, если и P и T состоят из одного и того же повторяющегося символа, то P входит в текст в каждой из $m - n + 1$ позиций и метод выполняет ровно $n(m - n + 1)$ сравнений. Например, если $P = aaa$ и $T = aaaaaaaaaaa$, то $n = 3$, $m = 10$ и делается 24 сравнения.

Наивный метод, конечно, просто и понимать, и программировать, однако его максимальная оценка $\Theta(nm)$ неудовлетворительна и может быть улучшена. Но даже фактическое время работы наивного метода может оказаться недопустимым для текстов и образцов большего размера. Начнем с того, что есть несколько взаимосвязанных идей улучшения наивного метода — и реально, и по максимальной оценке. Оценка наихудшего случая $\Theta(nm)$ может быть уменьшена до $O(n + m)$. Замена знака умножения на знак сложения предельно существенна (попробуйте взять $n = 1000$ и $m = 10\,000\,000$ — появление таких чисел в некоторых приложениях вполне возможно).

1.1.1. Первые идеи ускорения наивного метода

Первые идеи ускорения заключались в попытках сдвинуть P при несовпадении больше чем на один символ, но так, чтобы ни в коем случае не пропустить вхождения P в T . Увеличение сдвига экономит сравнения, так как P продвигается вдоль T быстрее. Кроме того, некоторые методы экономят сравнения, пропуская после сдвига какую-то часть образца. Мы детально разберем многие из этих идей.

На рис. 1.1 демонстрируется пример с $P = abxyabxz$ и $T = xabxyabxyabxz$. Отметьте, что вхождение P в T начинается с позиции 6. Наивный алгоритм прикладывает P с левого конца T , немедленно находит несовпадение и сдвигает P на одну позицию. Далее семь сравнений дают совпадения, а следующее (девятое по счету!) — несовпадение. Затем P сдвигается на единицу, результат — несовпадение, и этот цикл повторяется еще два раза, после чего левый конец P становится вровень с шестым символом строки T . Здесь выполняется восемь сравнений с совпадением и обнаруживается вхождение P в T , начиная с позиции 6. В этом примере наивный алгоритм делает двадцать сравнений.

0	1	0	1	0	1
1234567890123		1234567890123		1234567890123	
T: xabxyabxyabxz		T: xabxyabxyabxz		T: xabxyabxyabxz	
P: abxyabxz		P: abxyabxz		P: abxyabxz	
*		*		*	
abxyabxz		abxyabxz		abxyabxz	
^^^^^*		^^^^^*		^^^^^*	
abxyabxz		abxyabxz		abxyabxz	
*		^^^^^		^^^^^	
abxyabxz					
*					
abxyabxz					
*					
abxyabxz					
^^^^^					

Рис. 1.1. Первая колонка иллюстрирует наивный алгоритм в чистом виде, а две следующие — более разумные сдвиги. “Крышечка” под символом соответствует совпадению, а звездочка — несовпадению, установленному алгоритмом

Более разумный алгоритм мог бы после девятого сравнения установить, что три очередных сравнения, сделанные наивным алгоритмом, дадут несовпадения. Этот более разумный алгоритм пропускает и экономит три следующих сдвиг/сравнения, прямо сопоставляя левый конец P с позицией 6 текста T . Как он это делает? После девятого сравнения отмечается, что первые семь символов P совпадают с символами от 2 до 8 строки T . Если известно также, что первый символ P (именно a) больше в P не встречается до позиции 5, то уже достаточно информации для вывода, что символ a не появится в T до позиции 6. Отсюда следует, что P заведомо не будет совпадать с T , пока левый конец P не выровняется с позицией 6 текста T . Рассуждения такого типа и позволяют сдвигать образец больше чем на один символ. Мало того, что увеличивается сдвиг, мы увидим, что для некоторых пар символов не требуется проверять соответствие.

Еще более разумный алгоритм обнаруживает, что следующее вхождение в P первых трех символов P (именно abx) начинается в позиции 5. Так как известно, что первые семь символов P совпадают с символами от 2 до 8 строки T , у алгоритма достаточно информации для вывода, что при выравнивании левого конца P с позицией 6 строки T следующие три сравнения пройдут. Этот более разумный алгоритм не будет делать таких сравнений. Вместо них после выравнивания левого конца P с позицией 6 строки T он сопоставит четвертый символ из P и девятый символ из T . Так будет сэкономлено шесть сравнений из тех, которые делал наивный алгоритм.

Приведенный пример иллюстрирует характер идей, позволяющих пропустить некоторые сравнения, хотя еще не ясно, как их можно алгоритмизировать. Эффективная реализация была предложена в алгоритме Кнута–Морриса–Пратта, его модификации для реального времени, алгоритме Бойера–Мура и его варианте, разработанных Апостолико и Джанкарло. Все эти алгоритмы работают за линейное время (время $O(n + m)$). Подробности обсуждаются в двух следующих главах.

1.2. Препроцессная обработка

У многих алгоритмов сравнения и анализа строк эффективность сильно возрастает из-за пропусков сравнений. Эти пропуски получаются благодаря изучению внутренней структуры либо образца P , либо текста T . При этом другая строка может даже оставаться неизвестной алгоритму. Эта часть алгоритма называется *препроцессингом* фазой. За ней следует фаза *поиска*, на которой информация, полученная в препроцессинговой фазе, используется для сокращения работы по поиску вхождений P в T . В приведенном примере мы считали, что более разумный метод знает, что символ a не появляется до позиции 5, а еще более разумный метод знает, что образец abx повторяется, начиная с позиции 5. Это предполагаемое знание и вырабатывается на препроцессинговой фазе.

В задаче поиска точного совпадения все алгоритмы, упомянутые в предыдущем параграфе, обрабатывают образец P . (Альтернативный подход используется в других алгоритмах, где заранее обрабатывается текст T ; таковы подходы, основанные на суффиксных деревьях. К ним мы обратимся позднее.) Методы препроцессинговой обработки в своем первоначальном виде сходны по духу, но могут различаться деталями и концептуальной трудностью. В нашей книге мы выбрали иной подход и не хотим начинать с объяснения этих первоначальных методов. Напротив, мы подчеркиваем сходство этих *действий* в разных алгоритмах поиска, определяя сначала *основной препроцессинг* P , который не зависит от конкретных алгоритмов поиска. Затем мы покажем, как каждый конкретный алгоритм использует информацию, вычисленную при основном препроцессинге P . В результате получится более простое и единообразное представление препроцессингов, специфических для нескольких классических методов, и простой новый алгоритм с линейным временем, базирующийся исключительно на этом основном препроцессинге (см. п. 1.5). Этот подход к поиску за линейное время был развит в работе [202].

1.3. Основной препроцессинг образца

Основной препроцессинг будет описан для произвольной строки, обозначаемой через S . В конкретных приложениях роль S часто играет образец P , но здесь мы

используем S вместо P , так как основной препроцессинг будет применяться и в других ситуациях.

Следующее определение вводит основные величины, вычисляемые в ходе основного препроцессинга строки.

Определение. Для данной строки S и позиции $i > 1$ определим $Z_i(S)$ как длину наибольшей подстроки S , которая начинается в i и совпадает с префиксом S .

Другими словами, $Z_i(S)$ — это длина наибольшего префикса $S[i..|S|]$, совпадающего с префиксом S . Например, если $S = aabcaabxaaz$, то

$$\begin{aligned} Z_5(S) &= 3 \quad (aab\ldots aabx\ldots), \\ Z_6(S) &= 1 \quad (aa\ldots ab\ldots), \\ Z_7(S) &= Z_8(S) = 0, \\ Z_9(S) &= 2 \quad (aab\ldots aaz). \end{aligned}$$

Когда аргумент S ясен из контекста, мы будем вместо $Z_i(S)$ писать Z_i .

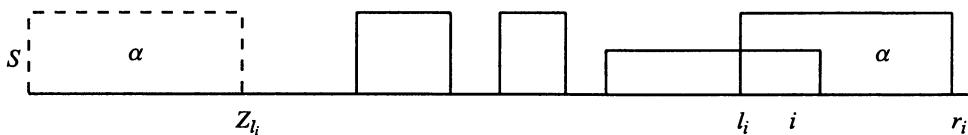


Рис. 1.2. Каждый “сплошной” блок соответствует подстроке S , которая совпадает с префиксом S и начинается между позициями 2 и i . Каждый блок называется Z -блоком. r_i обозначает самый правый конец Z -блока, начинающегося в позиции i или левее; α обозначает подстроку в том Z -блоке, который кончается в r_i . l_i обозначает левый конец α . На рисунке пунктиром показана также копия α , являющаяся префиксом для S

Чтобы ввести следующее понятие, рассмотрим блоки, изображенные на рис. 1.2. Каждый блок представляет собой подстроку S максимальной длины, которая совпадает с префиксом S и начинается не с первой позиции. Блок начинается в некоторой позиции $j > 1$, такой что $Z_j > 0$. Его длина равна Z_j . Такой блок называется Z -блоком. Более формально:

Определение. Для любой позиции $i > 1$, в которой $Z_i > 0$, определим Z -блок в i как интервал, начинающийся в i и кончающийся в позиции $i + Z_i - 1$.

Определение. Для любого $i > 1$ пусть r_i — крайний правый конец Z -блоков, начинающихся не позднее позиции i . По-другому r_i можно определить как наибольшее значение $j + Z_j - 1$ по всем $1 < j \leq i$, для которых $Z_j > 0$ (см. рис. 1.2).

Обозначим значение j , на котором достигается этот максимум, через l_i . Таким образом, l_i — это позиция левого конца Z -блока, кончающегося в r_i . В случае если таких Z -блоков, кончающихся в r_i , больше одного, в качестве l_i берется левый конец любого из них. Например, если $S = aabaabcxaabaabcy$, то $Z_{10} = 7$, $r_{15} = 16$ и $l_{15} = 10$.

Вычисление значений Z для строки S за линейное время и есть *основной* препроцессинг, который мы будем использовать во всех классических алгоритмах поиска за линейное время, требующих обработки P . Но прежде покажем, как же выполнить его за линейное время.

1.4. Основной препроцессинг за линейное время

Цель этого параграфа — продемонстрировать, как все значения Z для S вычисляются за линейное время (т.е. за время $O(|S|)$). Прямое вычисление, основанное на определении, дает время $\Theta(|S|^2)$. Метод, обсуждаемый здесь, был разработан в [307] для другой цели.

Препроцессный алгоритм вычисляет Z_i , r_i и l_i последовательно для каждой позиции, начиная с $i = 2$. Все полученные значения Z сохраняются алгоритмом, но на каждой итерации i используются только r_{i-1} и l_{i-1} , а более ранних значений r и l не требуется. Следовательно, для хранения последних вычисленных значений алгоритму достаточно иметь простые переменные r и l . Таким образом, на каждой итерации i , если наш алгоритм находит новый Z -блок (начинающийся в i), величина переменной r просто возрастает до достижения конца этого Z -блока и тем сохраняет свой статус самой правой позиции.

Сначала алгоритм находит значение Z_2 , непосредственно просматривая слева направо и сравнивая символы $S[2..|S|]$ и $S[1..|S|]$ до несовпадения. Z_2 равно длине совпадающей части. Если $Z_2 > 0$, то $r = r_2$ полагается равным $Z_2 + 1$, а $l = l_2$ — равным 2. В противном случае обе величины принимаются равными 0. Теперь сделаем индуктивное предположение, что алгоритм корректно вычислил Z_i для i до $k - 1 > 1$, и допустим, что алгоритм знает текущие значения $r = r_{k-1}$ и $l = l_{k-1}$. Далее нужно вычислить Z_k , $r = r_k$ и $l = l_k$.

Идея в том, чтобы использовать для ускорения расчета Z_k уже вычисленные значения Z . В некоторых случаях Z_k можно найти из предыдущих значений Z без дополнительных сравнений. Например, предположим, что $k = 121$, все значения от Z_2 до Z_{120} уже вычислены, $r_{120} = 130$ и $l_{120} = 100$. Это означает, что подстрока длины 31, начинающаяся в позиции 100, совпадает с префиксом S (длины 31). Отсюда следует, что подстрока длины 10, начинающаяся в позиции 121, должна совпадать с подстрокой длины 10, начинающейся в позиции 22, так что Z_{22} может быть очень полезно при вычислении Z_{121} . Так, если, скажем, $Z_{22} = 3$, простое рассуждение показывает, что и $Z_{121} = 3$. Итак, значение Z_{121} может быть получено без дополнительных сравнений вообще. Этот пример, как и все другие, будет формализован и корректно доказан ниже.

Z-алгоритм

При заданных Z_i для всех $1 < i \leq k - 1$ и текущих значениях r и l величина Z_k и изменения для r и l вычисляются следующим образом:

begin

- Если $k > r$, то найти Z_k непосредственным сравнением до несовпадения подстрок, начинающихся с позиции k и с позиции 1. Длина совпадающей части и дает Z_k . Если $Z_k > 0$, положить $r = k + Z_k - 1$ и $l = k$.

2. Если $k \leq r$, то позиция k находится в Z -блоке, и следовательно, $S(k)$ содержится в подстроке $S[l..r]$ (назовем ее α), такой что $l > 1$ и α совпадает с префиксом S . Поэтому символ $S(k)$ стоит и в позиции $k' = k - l + 1$. По тем же причинам подстрока $S[k..r]$ (назовем ее β) должна совпадать с подстрокой $S[k'..Z_l]$. Отсюда следует, что подстрока, начинающаяся с позиции k , должна совпадать по меньшей мере с префиксом S длины $\min\{Z_k, |\beta|\} = r - k + 1$ (рис. 1.3).

Рассмотрим два отдельных случая, в зависимости от того, на чем достигается этот минимум.

- Если $Z_{k'} < |\beta|$, то $Z_k = Z_{k'}$ и r, l не изменяются (рис. 1.4).
- Если $Z_{k'} \geq |\beta|$, то вся подстрока $S[k..r]$ должна быть префиксом S и $Z_k \geq |\beta| = r - k + 1$. Однако Z_k может быть больше, чем β , так что нужно сравнить до несовпадения символы, начиная с позиции $r + 1$, с символами, начиная с позиции $|\beta| + 1$. Пусть несовпадение произошло на символе $q \geq r + 1$. Тогда Z_k полагается равным $q - k$, $r = q - 1$ и $l = k$ (рис. 1.5).

end.

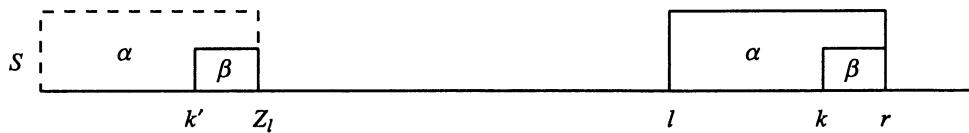


Рис. 1.3. Стока $S[k..r]$, помеченная буквой β , встречается также начиная с позиции k'

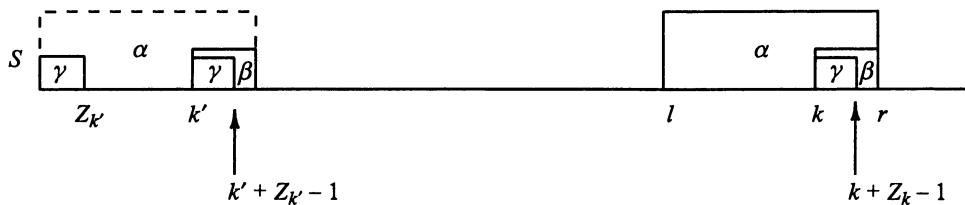


Рис. 1.4. Случай 2а. Самая длинная строка, начинающаяся с позиции k' и совпадающая с префиксом S , короче, чем $|\beta|$; при этом $Z_k = Z_{k'}$

Теорема 1.4.1. При использовании алгоритма Z значения Z_k корректно вычисляются, и переменные r и l корректно обновляются.

Доказательство. В случае 1 значение Z_k устанавливается правильно, так как вычисляется прямым сравнением. Кроме того (поскольку $k > r$), перед вычислением Z_k нет Z -блока, который начинался бы в позиции между 2 и $k - 1$ и кончался бы в позиции k или после нее. Поэтому в случае 1, когда $Z_k > 0$, алгоритм находит новый Z -блок, кончающийся в позиции k или после нее, и замена r на $k + Z_k - 1$ корректна. Итак, в случае 1 алгоритм работает верно.

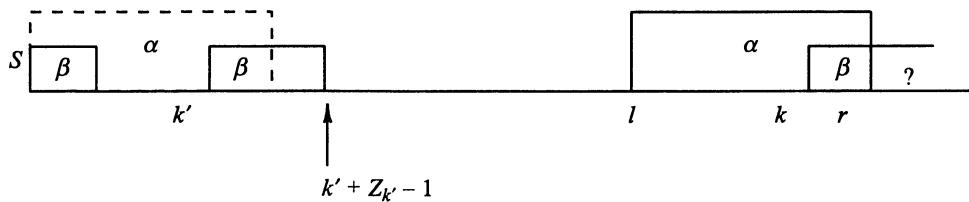


Рис. 1.5. Случай 2б. Самая длинная строка, начинающаяся с позиции k' и совпадающая с префиксом S , имеет длину не меньше, чем $|\beta|$

В случае 2а подстрока, начинающаяся в позиции k , может совпасть с префиксом S только на длину $Z_{k'} < |\beta|$. Если это не так, то следующий символ справа, символ $k + Z_{k'}$, должен совпасть с символом $1 + Z_{k'}$. Но символ $k + Z_{k'}$ совпадает с символом $k' + Z_{k'}$ (поскольку $Z_{k'} < |\beta|$), так что символ $k' + Z_{k'}$ должен тогда совпадать с символом $1 + Z_{k'}$. Однако это противоречило бы определению $Z_{k'}$, так как у нас получилась бы подстрока, которая начинается в k' и совпадает с префиксом S и длина которой больше $Z_{k'}$. Таким образом, $Z_k = Z_{k'}$. Далее, $k + Z_k - 1 < r$, так что r и l правильно остаются без изменений.

В случае 2б строка β должна быть префиксом S (как отмечено в тексте алгоритма), и так как любое расширение этого совпадения прямо проверяется сравнением символов после r с символами после префикса β , полное продолжение этого совпадения корректно вычисляется. Следовательно, в этом случае Z_k находится корректно. Далее, так как $k + Z_k - 1 \geq r$, алгоритм правильно изменяет r и l . \square

Следствие 1.4.1. Повторное выполнение Z-алгоритма для каждой позиции $i > 2$ корректно находит все значения Z_i .

Теорема 1.4.2. Все значения $Z_i(S)$ вычисляются приведенным выше алгоритмом за время $O(|S|)$.

Доказательство. Время пропорционально числу итераций $|S|$ плюс число сравнений символов. Каждое сравнение дает совпадение или несовпадение, так что дальше нам нужно оценить число и тех и других. Каждая итерация, которая выполняет сравнение, завершается сразу, как только находит несовпадение; следовательно, на протяжении всего алгоритма несовпадений не более $|S|$. Чтобы оценить число совпадений, отметим сначала, что $r_k \geq r_{k-1}$ для любой итерации k . Пусть теперь k — итерация, на которой обнаружилось $q > 0$ совпадений. Тогда r_k получит значение, не меньшее $r_{k-1} + q$. Наконец, $r_k \leq |S|$, так что полное число совпадений за все время работы алгоритма не превосходит $|S|$. \square

1.5. Простейший алгоритм поиска точного совпадения с линейным временем

Прежде чем обсуждать более сложные (классические) методы поиска точного совпадения, покажем, что уже основной препроцессинг дает алгоритм с линейным временем. Вот простейший из алгоритмов, который мы знаем.

Пусть $S = P\$T$ — строка, состоящая из образца P и текста T , между которыми стоит знак \$, отсутствующий в обеих строках. Напомним, что $|P| = n$, $|T| = m$ и $n \leq m$. Поэтому $|S| = n + m + 1 = O(m)$. Вычислим $Z_i(S)$ для i от 2 до $n + m + 1$. Так как \$ не встречается в P и T , то $Z_i \leq n$ для любого $i > 1$. Любое значение $i > n + 1$, для которого $Z_i(S) = n$, определяет вхождение P в T , начинающееся с позиции $i - (n + 1)$. И напротив, если P входит в T с позиции j , то $Z_{(n+1)+j} = n$ должно равняться n . Так как все $Z_i(S)$ можно вычислить за время $O(n + m) = O(m)$, такой алгоритм находит все вхождения P в T за время $O(m)$.

Этот метод можно реализовать, используя память $O(n)$ (кроме памяти, занятой под образец и текст) независимо от размера алфавита. Так как $Z_i \leq n$ для любого i , позиция k' (определенная в шаге 2) будет всегда попадать внутрь P . Поэтому нет нужды запоминать значения Z для символов из T . Достаточно запомнить их для n символов из P и поддерживать текущие l и r . Указанных данных достаточно для вычисления (без сохранения) значений Z для каждого символа из T и, значит, для распознания и вывода любой позиции i , где $Z_i = n$.

Есть еще одна особенность, которую стоит отметить: это *алфавитно-независимый* метод с линейным временем. Именно, нет необходимости предполагать, что алфавит конечен или мы знаем его заранее, — нужна лишь операция сравнения символов, которая определяет, совпадают два символа или нет. Мы покажем, что этим свойством обладают алгоритмы Кнута–Морриса–Пратта и Бойера–Мура, но не алгоритм Ахо–Корасика и не метод, основанный на суффиксных деревьях.

1.5.1. Зачем продолжать?

Так как функция Z_i вычисляется для образца за линейное время и может прямо использоваться для решения задачи точного совпадения с временем $O(m)$ (и дополнительной памятью только $O(n)$), то зачем продолжать? Почему нужно обращать внимание на более сложные методы (Кнут–Моррис–Пратт, Бойер–Мур, поиск в реальном времени, Апостолико–Джанкарло, Ахо–Корасик, суффиксные деревья и т. д.)?

Для задач точного совпадения алгоритм Кнута–Морриса–Пратта обладает лишь незначительным преимуществом перед прямым использованием Z_i . Однако он имеет историческое значение и был приспособлен в алгоритме Ахо–Корасика для поиска набора образцов в тексте за время, линейно зависящее от длины текста. Эту проблему нельзя решить хорошо, используя только значения Z_i . Обобщение метода Кнута–Морриса–Пратта на поиск в реальном времени имеет преимущество в ситуациях, когда текст последовательно вводится (в режиме on-line) и нужна уверенность, что алгоритм будет готов к приему каждого очередного символа. Метод Бойера–Мура важен потому, что (при удачной реализации) его время работы в худшем случае также линейно, но обычно требуется *сублинейное* время и проверяется только часть символов из T . Следовательно, в большинстве случаев этот метод предпочтителен. Метод Апостолико–Джанкарло обладает всеми преимуществами метода Бойера–Мура и допускает относительно простое доказательство линейности времени счета в наихудшем случае. Методы, основанные на суффиксных деревьях, обычно предварительно подготовливают не образец поиска, а текст и приводят к алгоритмам, в которых время поиска пропорционально размеру образца, а не текста. Это свойство чрезвычайно

желательно. Более того, суффиксные деревья могут применяться при решении задач, значительно более сложных, чем поиск точного совпадения, включая задачи, которые не решаются напрямую применением основного препроцессинга.

1.6. Упражнения

Первые четыре упражнения используют тот факт, что основной препроцессинг может быть выполнен за линейное время и все вхождения P в T также находятся за линейное время.

- Используйте существование алгоритма точного совпадения за линейное время для решения за линейное время следующей задачи. Пусть заданы две строки α и β . Определить, является ли α циклическим сдвигом β (это значит, что α и β имеют одинаковую длину и α состоит из суффикса β , склеенного с префиксом β). Например, $defabc$ является циклическим сдвигом $abcdef$. Это классическая задача с очень элегантным решением.
- Аналогично упражнению 1 разработайте алгоритм, который за линейное время определяет, не является ли строка α подстрокой циклической строки β . Циклическая строка длины n — это строка, в которой n -й символ считается предшествующим первому (рис. 1.6). Другой способ представить себе эту задачу таков. Пусть $\bar{\beta}$ — линейная строка, полученная из β , если начать с первого символа и кончить n -м. Тогда α есть подстрока циклической строки β в том и только том случае, когда α есть подстрока некоторого циклического сдвига β .

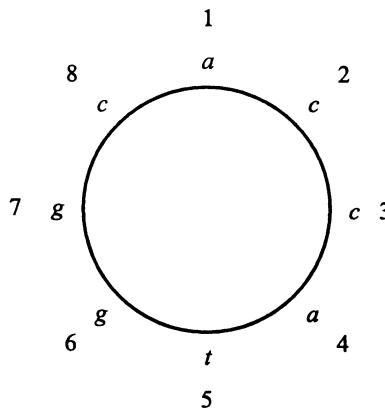


Рис. 1.6. Циклическая строка β . Полученная из нее линейная строка $\bar{\beta}$ — это $accatggc$

Отступление о циклических строках в ДНК. Две сформулированные задачи — это главным образом упражнения на существование алгоритмов с линейным временем. Мы не знаем каких-либо серьезных биологических задач, где бы они использовались. Однако нужно подчеркнуть, что *кольцевая ДНК* типична и важна. Бактериальные и митохондриальные ДНК обычно циклически, что справедливо и для их геномных ДНК, и для дополнительных малых двухспиральных кольцевых ДНК-молекул, называемых *плазмидами*. Даже некоторые эукариоты (высшие организмы, клетки которых содержат ядро), такие как дрожжи, имеют плазмиды дополнительно к их ядерной

ДНК. Следовательно, средства для работы с циклическими строками могут оказаться полезными для изучения этих организмов. Вирусная ДНК не всегда является кольцевой, но даже когда она линейна, некоторые вирусные геномы проявляют циклические свойства. Например, в некоторых вирусных популяциях линейный порядок одной ДНК получается циклическим сдвигом порядка в другой [450]. Нуклеотидные мутации в добавок к сдвигам часто происходят в вирусах, и возникает правдоподобная задача определения, не получились ли ДНК двух индивидуальных вирусов просто циклическим сдвигом, а не дополнительными мутациями.

Интересно отметить, что поставленные в этих упражнениях задачи реально “решены” в природе. Рассмотрим специальный случай упражнения 2, когда строка $|\alpha| = n$. Тогда задача ставится так: не является ли α циклическим сдвигом $\bar{\beta}$? Эта задача решается в упражнении 1 за линейное время. В точности эта задача совпадения встречается и “решается” в воспроизведении *E. coli* при некоторых опытных условиях, описанных в [475]. В этом опыте фермент (RecA) и молекулы АТФ (для энергии) добавляются к *E. coli*, содержащей простую спираль, одной из ее плазмид, называемой строкой β , и двухспиральную линейную молекулу ДНК, одна спираль которой называется строкой α . Если α — циклический сдвиг $\bar{\beta}$, то спираль, противоположная α (которая имеет последовательность ДНК, комплементарную к α), гибридизуется с β , создавая правильную двухспиральную плазмиду и оставляя α отдельной спиралью. Эта передача ДНК может быть шагом воспроизведения плазмиды. Таким образом, проблема определения, является ли α циклическим сдвигом $\bar{\beta}$, решается этой природной системой.

Другие опыты в [475] можно описать как задачи поиска совпадений *подстроки*, относящиеся к кольцевым и линейным ДНК в *E. coli*. Интересно, что природные системы решают свои задачи поиска совпадений быстрее, чем это может быть объяснено кинетическим анализом, и молекулярные механизмы, которые обеспечивают такое быстрое сопоставление, остаются неизвестными. Эти опыты показывают роль фермента RecA в воспроизведении *E. coli*, но не ставят прямо важных вычислительных задач. Однако они косвенно мотивируют развитие вычислительных средств, которые позволили бы работать с циклическими строками так же просто, как с линейными. Некоторые другие использования циклических строк будут обсуждаться в пп. 7.13 и 16.17.

3. **Суффиксно-префиксное совпадение.** Предложите алгоритм, который по двум строкам α и β длины n и m находит наибольший суффикс α , который является и префиксом β . Алгоритм должен работать за время $O(n + m)$.
4. **Тандемные ряды.** Подстрока α , содержащаяся в строке S , называется *тандемным рядом* (*tandem array*) строки β (называемой *основой*), если α состоит из более чем одной последовательной копии β . Например, если $S = xuzabcabcaabcabcprq$, то $\alpha = abcabcaabc$ — tandemный ряд строки $\beta = abc$. Заметим, что S содержит также tandemный ряд строки $abcaabc$ (т. е. tandemный ряд с более длинной основой). Тандемный ряд *максимальен*, если он не может быть продолжен ни влево, ни вправо. При фиксированной основе β tandemный ряд β в S может быть задан парой чисел (s, k) , определяющей его начальную позицию в S и число повторов β . Тандемный ряд — это пример повторяющейся подстроки (см. п. 7.11.1).

Пусть S имеет длину n . Постройте пример, показывающий, что два максимальных tandemных ряда с одной основой β могут перекрываться.

Далее предложите алгоритм с трудоемкостью $O(n)$, который по исходным данным S и β находит каждый максимальный tandemный ряд β и выводит для каждого вхождения пару (s, k) . Так как максимальные tandemные ряды с данной основой могут перекрываться, наивный алгоритм может дать только границу трудоемкости $O(n^2)$.

5. Если Z -алгоритм обнаруживает, что $Z_2 = q > 0$, то все значения $Z_3, \dots, Z_{q+1}, Z_{q+2}$ могут быть получены непосредственно, без дополнительных сравнений и без обращения к основному телу алгоритма Z . Детально обоснуйте это утверждение.
6. Z -алгоритм в случае 2b, когда $Z_{k'} \geq |\beta|$, выполняет прямые сравнения до несовпадения. Это хороший способ организации самого алгоритма, но в действительности в случае 2b можно избежать этого необязательного сравнения. Покажите, что если $Z_{k'} > |\beta|$, то $Z_k = |\beta|$ и, следовательно, сравнений не требуется. Так что прямое сравнение потребуется только при $Z_{k'} = |\beta|$.
7. Если разделить Z -алгоритм в случае 2b на два подслучаи, когда $Z_{k'}$ строго больше и когда равно $|\beta|$, ускорит ли это алгоритм? Вы должны считать все действия, а не только сравнение символов.
8. Бейкер [43] ввел следующую задачу поиска совпадения, которую применил к сопровождению программного обеспечения: “Приложение отслеживает дублирования в большой программной системе. Мы хотим находить не только точные совпадения участков кода, но и параметризуемые совпадения, в которых один участок переходит в другой при однозначной замене имен параметров (т. е. идентификаторов и констант)”.

Теперь дадим формальную постановку задачи. Пусть Σ и Π — два алфавита, не имеющие общих символов. Символы из Σ называются *элементами* (*tokens*), а символы из Π — *параметрами*. Стока может состоять из любой комбинации элементов и параметров. Например, если Σ состоит из прописных английских букв, а Π — из строчных букв, то $XYabCaCXZddW$ — законная строка над Σ и Π . Две строки S_1 и S_2 называются *p-совпадающими* в том и только том случае, если:

- a. Каждый элемент из S_1 (или S_2) сопоставлен такому же элементу из S_2 (или S_1).
- b. Каждый параметр из S_1 (или S_2) сопоставлен параметру из S_2 (или S_1).
- c. Если для любого параметра x одно вхождение x в S_1 (S_2) сопоставлено параметру y в S_2 (S_1), то и каждое вхождение x в S_1 (S_2) должно сопоставляться y в S_2 (S_1). Другими словами, выравнивание параметров в S_1 и S_2 определяет одно-однозначное соответствие между именами параметров в S_1 и именами параметров в S_2 .

Строки $S_1 = XYabCaCXZddbW$ и $S_2 = XYdxCdCXZccxW$ могут рассматриваться как *p*-совпадающие. Отметим, что параметр a в S_1 отображается в параметр d в S_2 , а параметр d в S_1 — в параметр c в S_2 . Это не противоречит определению *p*-совпадения.

В приложении Бэйкера элемент представляет часть программы, которую изменять нельзя, а параметр представляет программную переменную, которую можно переименовать, если все вхождения этой переменной заменяются согласованно. Таким образом, если S_1 и S_2 *p*-согласованы, то замена имен переменных в S_1 на соответствующие имена в S_2 делает эти две программы идентичными. Если обе эти программы входят в большую программу, то их можно заменить вызовом одной и той же подпрограммы.

Самая важная задача *p*-совпадения такова. Заданы текст T и образец P , соответственно, над алфавитами Σ и Π . Найти все подстроки T , которые *p*-совпадают с P . Конечно, хотелось бы найти все вхождения за время $O(|P| + |T|)$. Пусть функция Z_i^p от строки S вырабатывает длину самой длинной строки, начинающейся в позиции i и *p*-совпадающей с префиксом $S[1..i]$. Покажите, как модифицировать алгоритм Z , чтобы вычислить все значения Z_i^p за время $O(|S|)$ (детали реализации немного хитроумнее, чем для функции Z_i , но

все же не очень трудны). Затем покажите, как использовать этот модифицированный алгоритм для нахождения всех подстрок T , которые p -совпадают с P , за время $O(|P| + |T|)$. В [43] и [239] более хитроумные варианты задачи p -совпадения решаются более сложными методами.

Три следующие задачи можно решить без Z -алгоритма или других специальных средств. Нужно только подумать.

9. Заданы две строки по n символов каждая и дополнительный параметр k . В каждой строке $n - k + 1$ подстрок длины k , так что есть $\Theta(n^2)$ пар подстрок, в которых одна подстрока берется из первой строки, а другая — из второй. Для пары подстрок определим *счетчик совпадений* как число соответствующих символов, которые совпадают при выравнивании этих двух подстрок длины k . Требуется вычислить счетчики совпадений для всех $O(n^2)$ пар подстрок. Ясно, что это можно сделать за $O(kn^2)$ операций (сравнения символов и арифметических действий). Но разумная организация вычислений позволяет свести трудоемкость до $O(n^2)$ операций. (Задача Пола Хортона.)
10. Молекулу ДНК можно рассматривать как строку над алфавитом из четырех символов $\{a, t, c, g\}$ (нуклеотидов), а белок — как строку над алфавитом из двадцати символов (аминокислот). Ген, который физически вложен в молекулу ДНК, обычно кодирует последовательность аминокислот для конкретного белка. Это делается следующим образом. Начиная с определенной точки строки ДНК, каждые три последовательных символа ДНК перекодируются в один символ аминокислоты в строке белка. Получается, что три нуклеотида ДНК задают одну аминокислоту. Такая кодовая тройка называется *кодоном*, а полное соответствие кодонов аминокислотам называется *генетическим кодом*. Например, кодон *ttt* задает аминокислоту фенинапанин (сокращаемую в единичный символ аминокислотного алфавита F), а кодон *gtt* — аминокислоту валин (сокращаемую в V). Так как возможных троек $4^3 = 64$, а аминокислот только двадцать, есть возможность, что две или более троек образуют кодоны для одной и той же аминокислоты, а некоторые тройки не образуют кодонов. Так и есть на самом деле. Например, аминокислота лейцин кодируется шестью разными кодонами.

Пусть задана строка ДНК из n нуклеотидов, но вы не знаете правильной “рамки считываания”. То есть неизвестно, с какого нуклеотида этой строки начинается правильное разложение на кодоны: с первого, со второго или с третьего. Каждый такой “сдвиг рамки” потенциально порождает свою строку аминокислот. (Действительно, известны гены, где каждый из трех вариантов чтения не только кодирует строку аминокислотного алфавита, но и каждой строке соответствует функциональный белок, в каждом случае свой.) Необходимо получить для каждого из окон соответствующую аминокислотную строку. Например, рассмотрим строку *atggacgga*. В первом варианте есть три полных кодона, *atg*, *gac* и *gga*, которые в генетическом коде задают аминокислоты *Met*, *Asp* и *Gly*. Второму варианту соответствуют два полных кодона, *tgg* и *acg*, кодирующие аминокислоты *Trp* и *Thr*. Третий вариант обеспечивает два полных кодона, *gga* и *cgg*, кодирующих аминокислоты *Gly* и *Arg*.

Задача в том, чтобы выполнить эти три перевода с наименьшим числом проверок символов ДНК строки и наименьшим числом индексных действий (когда аминокислота находится по кодону в таблице, содержащей генетический код). Ясно, что три перевода можно сделать за $3n$ проверок символов ДНК и $3n$ индексных действий. Предположите метод, позволяющий найти решение не более чем за n проверок символов и n индексных действий.

Подсказка. Здесь может быть полезно, хотя и не необходимо, понятие конечного автомата, если вы знакомы с этой терминологией.

11. Пусть T — текстовая строка длины m , а \mathcal{S} — набор из n символов. Задача в том, чтобы найти все подстроки T длины n , составленные из символов \mathcal{S} . Например, пусть $\mathcal{S} = \{a, a, b, c\}$ и $T = abahgcabah$. Тогда $caba$ — подстрока T , составленная из символов \mathcal{S} .

Предложите решение этой задачи, требующее времени $O(m)$. Метод должен уметь также находить для каждой позиции i длину наибольшей подстроки в T , начинающейся в i и составленной из символов из \mathcal{S} .

Фантазия на тему расшифровки белка. Приведенная задача может оказаться полезной при расшифровке белка какого-либо организма после того, как значительная часть его генома уже расшифрована. Это проще всего объяснить на примере прокариотов, где ДНК не прерывается инtronами. У прокариотов последовательность аминокислот для данного белка кодируется сплошным сегментом ДНК — один кodon на каждую аминокислоту белка. Предположим, что мы имеем молекулу белка, но не знаем ее последовательности или размещения гена, который кодирует этот белок. В настоящее время химические способы определения последовательности аминокислот в белке очень медленны, дороги и не вполне надежны. Вместе с тем нахождение набора аминокислот, составляющих этот белок, относительно просто. Предположим, что известна вся последовательность ДНК для генома организма. Можно использовать эту длинную последовательность ДНК для определения последовательности аминокислот в интересующем нас белке. Прежде всего, перекодируем каждый кodon последовательности ДНК в аминокислотный алфавит (может понадобиться три прохода для установления нужной рамки считывания) и получим строку T ; затем химически определим набор \mathcal{S} аминокислот в белке, далее найдем все подстроки в T длины $|\mathcal{S}|$, которые составлены из аминокислот \mathcal{S} . Такие подстроки — это кандидаты на последовательность аминокислот белка, хотя кажется неправдоподобным, что кандидатов будет больше одного. Это совпадение, кроме того, устанавливает место гена для белка в длинной строке ДНК.

12. Рассмотрим двумерный вариант предыдущей задачи. Исходные данные состоят из двумерного текста (скажем, заполненного кроссворда) и набора символов. Задача заключается в нахождении в тексте двумерной подструктуры, в которой были бы использованы символы из набора. Как это можно сделать? Простой вариант задачи — когда структура прямоугольная.
13. Как отмечено в упражнении 10, существуют организмы (например, некоторые вирусы), содержащие интервалы кода ДНК не с одним белком, а с тремя, получающимися при разных установках рамки считывания. Таким образом, если каждый белок содержит n аминокислот, то строка ДНК, кодирующая эти три белка, имеет длину в $n + 2$ нуклеотидов (символов). Это очень компактное кодирование.

Иследовательская задача. Предложите алгоритм решения следующей задачи. Имеется белковая строка S_1 (в аминокислотном алфавите) длины n и другая белковая строка S_2 длины $m > n$. Определить, может ли строка ДНК, кодирующая S_2 , содержать подстроку ДНК, кодирующую S_1 . Кодирование для S_1 может начинаться в любой точке строки ДНК для S_2 (т.е. при любой рамке считывания этой строки). Эта задача трудна из-за вырожденности генетического кода и возможности использовать любую рамку считывания.

Точное совпадение: классические методы

2.1. Введение

В этой главе рассматривается ряд классических алгоритмов для задачи точного совпадения, основанных на сравнениях. С некоторыми улучшениями все эти алгоритмы могут быть реализованы с линейной оценкой времени работы, такая производительность достигается за счет препроцессной обработки образца P . (Методы, где обрабатывается T , мы рассмотрим во второй части книги.) Как уже упоминалось, оригинальные методы препроцессинга для этих алгоритмов родственны по духу, но очень различны по концептуальной сложности. Некоторые из них очень трудны.*¹) Мы не следуем исходным методам подготовки, а используем основной препроцессинг, описанный в предыдущей главе, и с его помощью выполняем препроцессинг для каждого конкретного алгоритма поиска совпадений.

В отличие от предшествовавших обзоров мы предпочитаем метод Бойера–Мура, а не метод Кнута–Морриса–Пратта, так как первый достаточно практичен. Второй тем не менее обсуждается тоже, отчасти по историческим причинам, но в основном потому, что он переносится на другие постановки задачи, такие как поиск в реальном времени и по набору образцов, гораздо легче, чем метод Бойера–Мура. Эти два аспекта будут описаны и в этой, и в следующей главе.

*¹) Седжвик [401] пишет: “Оба алгоритма, Кнута Морриса–Пратта и Бойера–Мура, требуют настолько сложной препроцессной обработки образца, что их трудно понять, и это ограничивает использование их по назначению”. В полном соответствии с Седжвиком, я так и не понял исходного препроцессинга Бойера–Мура для *сильного* правила хорошего суффикса.

2.2. Алгоритм Бойера–Мура

Как и наивный алгоритм, алгоритм Бойера–Мура последовательно прикладывает образец P к тексту T и проверяет совпадение символов P с прилежащими символами T . Когда проверка завершается, P сдвигается вправо по T точно так же, как в наивном алгоритме. Однако алгоритм Бойера–Мура использует три здравые идеи, которых нет в наивном алгоритме: просмотр справа налево, правило сдвига по плохому символу и правило сдвига по хорошему суффиксу. Совместный эффект этих трех идей позволяет методу обычно проверять меньше чем $m + n$ символов (метод с ожидаемым сублинейным временем) и (после некоторого улучшения) работать за линейное время в худшем случае. Наше изложение алгоритма Бойера–Мура и его улучшений концентрируется на *доказательных* аспектах его поведения. Широкие экспериментальные и практические исследования метода Бойера–Мура и его вариантов изложены в [229, 237, 409, 410, 425].

2.2.1. Просмотр справа налево

При любом прикладывании P к T алгоритм Бойера–Мура проверяет совпадение сканированием символов *справа налево*, а не слева направо, как в наивном алгоритме. Например, пусть P приложено к T так, как показано ниже:

	1	2	
	12345678901234567		
T:	x	p	b
	c	t	b
	x	a	b
	a	b	p
P:	t	r	a
	p	a	b
	a	b	a

Проверяя вхождение P в T в этой позиции, алгоритм Бойера–Мура начинает с *правого* конца P и сравнивает $T(9)$ с $P(7)$. Обнаружив совпадение, он сравнивает $T(8)$ с $P(6)$ и т.д., перемещаясь справа налево, пока не встретит несовпадение $T(5)$ и $P(3)$. После этого P перемещается *направо* по T (величина смещения будет обсуждаться ниже), и сравнения возобновляются с правого конца P .

Ясно, что если бы образец P сдвигался на одно место вправо после каждого несовпадения или обнаружения полного вхождения P в текст, то оценка времени счета по худшему случаю была бы $O(nm)$, точно как в наивном алгоритме. Так что пока не ясно, почему сравнение символов справа налево лучше, чем слева направо. Однако два дополнительных правила (*плохого символа* и *хорошего суффикса*) часто разрешают сдвигаться больше чем на одну позицию, и обычными становятся именно большие сдвиги. Рассмотрим эти правила.

2.2.2. Правило плохого символа

Чтобы воспринять саму идею правила плохого символа, предположим, что последний (крайний правый) символ P есть y , а символ в T , с которым он сопоставляется, — это $x \neq y$. Когда случается это начальное несовпадение, то, зная в P крайнюю правую позицию символа x , мы можем смело сдвигать P направо до совмещения крайнего правого x в P с найденным x в T , потому что более короткий сдвиг немедленно даст несовпадение. Так что этот большой сдвиг корректен (в том смысле, что при

нем не теряется вхождений P в T). Если x вообще не встречается в P , то мы можем сдвинуть P полностью за точку несовпадения в T . В этих случаях некоторые символы из T вообще не будут проверяться, и метод будет работать за “сублинейное” время. Это наблюдение формализуется ниже.

Определение. Для каждого символа алфавита x пусть $R(x)$ — позиция крайнего правого вхождения x в P . Если x в P не входит, $R(x)$ считается нулем.

Легко просмотреть P за время $O(n)$ и подготовить значения $R(x)$, мы оставляем это как упражнение. Отметим, что такая обработка не требует основного препроцессинга, рассмотренного в главе 1 (который понадобится для более сложного правила сдвига — правила хорошего суффикса).

Мы используем значения R в следующем правиле *сдвига по плохому символу*.

Предположим, что при некотором сопоставлении P с T крайние правые $n - i$ символов P совпадают со своими парами в T , но следующий слева символ $P(i)$ не совпадает со своей парой, скажем, в позиции k строки T . *Правило плохого символа* гласит, что P следует сдвинуть вправо на $\max\{1, i - R(T(k))\}$ мест. Таким образом, если крайнее правое вхождение в P символа $T(k)$ занимает позицию $j < i$ (включая возможность $j = 0$), то P сдвигается так, чтобы символ j в P поравнялся с символом k в T . В противном случае P сдвигается на одну позицию.

Цель этого правила — сдвиг P , когда это возможно, больше чем на одно место. В рассмотренном примере $T(5) = t$ не совпадает с $P(3)$ и $R(t) = 1$, так что P можно сдвинуть вправо на две позиции. После сдвига сравнение P и T начинается заново с правого конца P .

Расширенное правило плохого символа

Правило плохого символа полезно при несовпадениях, близких к правому концу P , но эффект от него мал, если несовпадающий символ из T встречается в P справа от точки несовпадения. Это бывает, когда алфавит мал и текст содержит много похожих, но не совпадающих подстрок. Такая ситуация типична для ДНК, алфавит которой содержит четыре символа, и даже белок, с алфавитом в двадцать символов, часто содержит разные области высокого сходства. В таких случаях нижеследующее *расширенное правило плохого символа* более устойчиво:

Когда несовпадение случилось в позиции i образца P и x — несовпадающий символ в T , нужно сдвинуть P вправо, совместив с этим x ближайшее вхождение x в P слева от позиции i .

Так как расширенное правило дает большие сдвиги, единственный резон в пользу более простого правила — это дополнительные издержки на реализацию расширенного правила. Простое правило использует только $O(|\Sigma|)$ памяти (Σ — это алфавит) для массива R и одно обращение к таблице при несовпадении. Как мы увидим, расширенное правило можно реализовать так, что будет использовано $O(n)$ памяти и не более одного дополнительного шага на каждое сравнение символов. Дополнительная память часто не критична, а вопрос о том, окупает ли увеличение сдвигов дополнительные затраты времени, решается по результатам практических наблюдений. Исходный алгоритм Бойера–Мура использовал только простое правило плохого символа.

Реализация расширенного правила плохого символа

Обработаем образец P так, чтобы расширенное правило плохого символа реализовалось эффективно по затратам и времени, и памяти. Препроцессинг должен найти для каждой позиции i в P и каждого символа алфавита x позицию ближайшего появления x в P слева от i . Очевидный подход состоит в создании для этой информации двумерного массива размера $n \times |\Sigma|$. При несовпадении в позиции i и несовпадающем символе x в тексте T мы выбираем из массива элемент (i, x) . Выборка из массива производится быстро, но размер массива и время на его построение могут быть чрезмерны. Однако есть приемлемый компромисс.

Во время препроцессинга просмотрим P справа налево, накапливая для каждого символа алфавита x список позиций, где x входит в P . Так как просмотр идет справа налево, каждый список будет расположен по убыванию. Например, если $P = abacbabc$, то для символа a получится список 6, 3, 1. Эти списки накапливаются за время $O(n)$ и, конечно, занимают память только $O(n)$. На стадии поиска алгоритма Бойера–Мура, когда нашлось несовпадение в позиции i и несовпадающий символ из T — это x , будем просматривать список этого символа, пока не достигнем первого числа, меньшего чем i , или не установим, что такого числа нет. Если его нет, то x до позиции i не появляется, и весь образец P сдвигается за позицию x в T . В противном случае найденный элемент списка дает требуемую позицию x .

При несовпадении в позиции i время просмотра не превосходит $n - i$, т. е. грубо говоря, числа сравниваемых символов. Так что этот подход увеличивает время счета по алгоритму Бойера–Мура не более чем вдвое в наихудшем случае. Однако в большинстве постановок задач увеличение времени значительно меньше указанного. Для сокращения времени, если понадобится, можно применить и двоичный поиск.

2.2.3. (Сильное) правило хорошего суффикса

Само по себе правило плохого символа имеет репутацию высокоеффективного в практических условиях, в частности для английского текста [229], но оно оказалось менее эффективным для маленьких алфавитов и не дает линейного времени в худшем случае. Поэтому мы введем еще одно правило, называемое *сильным правилом хорошего суффикса*. Исходный препроцессинг [278] для сильного правила хорошего суффикса обычно считается очень трудным и даже таинственным (хотя его слабая версия легка для понимания). В действительности препроцессинг для сильного правила был изложен в [278] некорректно и исправлен, без пристальных пояснений, в [384]. Код, основанный на [384], приведен без доступного объяснения в работе Баасе [32]. Печатных источников, где бы пытались растолковать этот метод, нет^{*)}). Программа на Паскале, выполняющая подготовку для сильного правила, основанная на наброске Ричарда Коула [107], приводится в упражнении 24 в конце этой главы.

^{*)} Недавно в Интернете в группе новостей `comp.theory` появилась мольба: “Я ищу элегантное (легко понимаемое) доказательство корректности части алгоритма Бойера–Мура для поиска совпадений. Эта трудная для доказательства часть — алгоритм, который вычисляет таблицу (хороших суффиксов) dd_2 . Я не смог найти понятного доказательства и буду признателен за любую помощь!”



Рис. 2.1. Правило сдвига по хорошему суффиксу, где символ x из T не совпал с символом y из P . Символы y и z из P гарантированно различны по правилу хорошего суффикса, поэтому y и z есть шанс совпасть с x

Напротив, основной препроцессинг P , рассмотренный в главе 1, делает необходимую обработку очень простой. Наше *сильное правило хорошего суффикса* таково:

Пусть строка P приложена к T и подстрока t из T совпадает с суффиксом P , но следующий левый символ уже не совпадает. Найдем, если она существует, крайнюю правую копию t' строки t в P , такую что t' не является суффиксом P и *символ слева от t' в P отличается от символа слева от t в P* . Сдвигнем P вправо, приложив подстроку t' в P к подстроке t в T (рис. 2.1). Если t' не существует, то сдвинем левый конец P за левый конец t в T на наименьший сдвиг, при котором префикс сдвинутого образца совпал бы с суффиксом t в T . Если такого сдвига не существует, то сдвинем P на n позиций вправо. Если найдено вхождение P , то сдвинем P на наименьший сдвиг, при котором *собственный* префикс сдвинутого P совпадает с суффиксом вхождения P в T . Если такой сдвиг невозможен, нужно сдвинуть P на n мест, т. е. сдвинуть P за t в T .

Рассмотрим конкретный пример:

0	1
123456789012345678	
$T:$ prstabstubabvqxrst	
*	
$P:$ qcabdabdac	
1234567890	

Когда нашлось несовпадение в позиции 8 строки P и позиции 10 строки T , мы получили $t = ab$ и строку t' в P , начинающуюся с позиции 3. Следовательно, P сдвигается вправо на шесть позиций, создавая такое соответствие:

0	1
123456789012345678	
$T:$ prstabstubabvqxrst	
$P:$ qcabdabdac	

Отметим, что расширенное правило плохого символа сдвинуло бы P в этом примере только на один символ.

Теорема 2.2.1. Использование правила хорошего суффикса никогда не сдвинет P за его вхождение в T .

Доказательство. Предположим, что правый конец P до сдвига стоял у символа k строки T , а правило хорошего суффикса сдвинуло P так, что его правый конец встал у символа $k' > k$. Любое вхождение P , кончающееся в позиции l строго между k и k' , противоречит правилу выбора k' , поскольку тогда в P нашлась бы более близкая копия t или более длинный префикс P совпадал бы с суффиксом t . \square

Опубликованный первоначально алгоритм Бойера–Мура [75] использует простой, слабый вариант правила хорошего суффикса. Этот вариант требует только, чтобы смешенный образец P прилегал к тексту по строке t , и не требует, чтобы следующий символ влево от вхождения t был другим. Явная формулировка слабого правила может быть получена удалением курсивной фразы в формулировке сильного правила хорошего суффикса. В предыдущем примере слабое правило сдвинет P на три позиции вместо шести. Когда нам потребуется различать эти два правила, мы будем называть более простое правило *слабым* правилом хорошего суффикса, а правило, сформулированное выше, *сильным* правилом хорошего суффикса. Для целей доказательства того, что поисковая часть метода Бойера–Мура работает в худшем случае за линейное время, слабого правила недостаточно, и в этой книге всюду, где не оговаривается противное, мы используем сильный вариант правила.

2.2.4. Препроцессинг для правила хорошего суффикса

Формализуем теперь препроцессную обработку, требуемую для алгоритма Бойера–Мура.

Определение. Для каждого i пусть $L(i)$ — наибольшая позиция, меньшая n и такая, что строка $P[i..n]$ совпадает с суффиксом строки $P[1..L(i)]$. Если такой позиции нет, $L(i)$ считается равным 0. Для каждого i пусть $L'(i)$ — наибольшая позиция, меньшая чем n и такая, что $P[i..n]$ совпадает с суффиксом $P[1..L'(i)]$, а символ, предшествующий этому суффиксу, не равен $P(i - 1)$. Если такой позиции нет, $L'(i)$ считается равным 0.

Например, если $P = cabdab dab$, то $L(8) = 6$ и $L'(8) = 3$.

$L(i)$ определяет позицию правого конца крайней правой копии $P[i..n]$, которая сама не является суффиксом P , а $L'(i)$ — такую же позицию с дополнительным усиливающим условием, что предшествующий символ не равен $P(i - 1)$. Поэтому в варианте сильного сдвига алгоритма Бойера–Мура если символ $i - 1$ образца P участвовал в обнаружении несовпадения и $L'(i) > 0$, то P сдвигается вправо на $n - L'(i)$ позиций. В результате если правый конец P до сдвига стоял на уровне позиции k в T , то на уровне k теперь нужно выровнить позицию $L'(i)$.

Препроцессинг для алгоритма Бойера–Мура вычисляет $L'(i)$ (и $L(i)$, если нужно) для каждой позиции i в P . Это делается за время $O(n)$ благодаря следующим определению и теореме.

Определение. Пусть $N_j(P)$ — длина наибольшего суффикса подстроки $P[1..j]$, который является также суффиксом полной строки P .

Например, если $P = cabdab dab$, то $N_3(P) = 2$ и $N_6(P) = 5$.

Напомним, что $Z_i(S)$ — длина наибольшей подстроки S , начинающейся в i и совпадающей с префиксом S . Ясно, что функция N является обращенной к Z в том смысле, что если P^r обозначает строку, полученную переворотом P , то $N_j(P) = Z_{n-j+1}(P^r)$. Следовательно, значения $N_j(P)$ можно получить за время $O(n)$, используя алгоритм Z для P^r . Немедленно формулируется следующая теорема.

Теорема 2.2.2. $L(i)$ — наибольший индекс j , меньший n и такой, что $N_j(P) \geq |P[i..n]| = n - i + 1$. $L'(i)$ — наибольший индекс j , меньший n и такой, что $N_j(P) = |P[i..n]| = (n - i + 1)$.

Из теоремы 2.2.2 прямо следует, что все значения $L'(i)$ могут быть получены за линейное время из значений N использованием следующего алгоритма:

Метод Бойера–Мура, основанный на Z -алгоритме

```
for  $i := 1$  to  $n$  do  $L'(i) := 0$ ;
for  $j := 1$  to  $n - 1$  do begin
     $i := n - N(P) + 1$ ;
     $L'(i) := j$ ;
end;
```

Значения $L(i)$ (если потребуется) могут быть получены добавлением к этому псевдокоду следующих строк:

```
 $L(2) := L'(2)$ ;
for  $i := 3$  to  $n$  do  $L(i) := \max\{L(i - 1), L'(i)\}$ ;
```

Теорема 2.2.3. Описанный метод корректно вычисляет значения L .

Доказательство. $L(i)$ указывает позицию правого конца крайней правой подстроки P , совпадающей с $P[i..n]$ и не являющейся суффиксом $P[1..n]$. Значит, эта подстрока начинается с позиции $L(i) - n + i$, которую мы обозначим через j . Докажем, что $L(i) = \max\{L(i - 1), L'(i)\}$, а для этого посмотрим, каким может быть символ $j - 1$. Прежде всего, если $j = 1$, то символ $j - 1$ отсутствует, так что $L(i - 1) = 0$ и $L'(i) = 1$. Поэтому предположим, что $j > 1$. Если символ $j - 1$ равен символу $i - 1$, то $L(i) = L(i - 1)$. Если же символ $j - 1$ не равен символу $i - 1$, то $L(i) = L'(i)$. Таким образом, во всех случаях $L(i)$ должно совпадать с $L'(i)$ или с $L(i - 1)$.

Однако, конечно, $L(i)$ должно быть не меньше этих значений. Так как $L(i)$ равно либо $L'(i)$, либо $L(i - 1)$ и не меньше их обоих, то $L(i)$ должно равняться максимуму из $L'(i)$ и $L(i - 1)$. \square

Окончательные подробности препроцессинга

Предварительный этап должен подготовить данные и к случаю, когда $L'(i) = 0$ или когда найдено вхождение P . Этую обработку обеспечивают следующие определение и теорема.

Определение. Пусть $l'(i)$ обозначает длину наибольшего суффикса $P[i..n]$, который является префиксом P , если такой существует. Если же не существует, то $l'(i)$ равно нулю.

Теорема 2.2.4. $l'(i)$ равно наибольшему $j \leq |P[i..n]| = n - i + 1$, для которого $N_j(P) = j$.

Мы оставляем читателю доказательство, как и задачу о вычислении значений $l'(i)$ за линейное время в качестве простого упражнения (упражнение 9 к этой главе).

2.2.5. Правило хорошего суффикса на стадии поиска в алгоритме Бойера–Мура

Вычисленные заранее значения $L'(i)$ и $l'(i)$ для каждой позиции i в P используются на стадии поиска для увеличения сдвигов. Если при поиске обнаружилось несовпадение в позиции $i - 1$ строки P и $L'(i) > 0$, то правило хорошего суффикса сдвигает P на $n - L'(i)$ мест вправо, так что префикс сдвинутого P длины $L'(i)$ пристраивается к суффиксу длины $L'(i)$ несдвинутого P . В случае $L'(i) = 0$ правило хорошего суффикса смещает P на $n - l'(i)$ позиций. Когда обнаруживается вхождение P , это правило сдвигает P на $n - l'(2)$ мест. Заметьте, что правила работают корректно и при $l'(i) = 0$.

Остался один специальный случай. Когда первое сравнение дает несовпадение (т.е. не совпал символ $P(n)$), то P надо сдвинуть на одно место вправо.

2.2.6. Полный алгоритм Бойера–Мура

Мы увидели, что ни правило хорошего суффикса, ни правило плохого символа не пропускают при сдвиге P его вхождений. Поэтому алгоритм Бойера–Мура сдвигает образец на наибольшее из расстояний, предлагаемых обоими правилами. Мы можем теперь представить весь алгоритм.

Алгоритм Бойера–Мура

{Препроцессинг}

Задан образец P .

Вычислить $L'(i)$ и $l'(i)$ для каждой позиции i из P ,
а также $R(x)$ для каждого символа x из Σ .

{Стадия поиска}

$k := n;$

while $k \leq m$ do begin

$i := n;$

$h := k;$

 while $i > 0$ и $P(i) = T(h)$ do begin

$i := i - 1;$

$h := h - 1;$

 end;

 if $i = 0$ then begin

 зарегистрировать вхождение P в T с последней позицией k .

$k := k + n - l'(2);$

 end

 else

 сдвинуть P (увеличить k) на максимальную из величин,

 задаваемых (расширенным) правилом плохого символа
 и правилом хорошего суффикса.

end;

Заметим, что мы всегда говорим о “сдвиге P ” и даем правила для определения того, насколько P должно быть “сдвинуто”, физически ничто не сдвигается. Просто индекс k увеличивается до значения, которое соответствует правому концу “сдвинутого” P . Следовательно, каждое действие сдвига P выполняется за константное время.

Мы покажем далее, в п. 3.2, что использование только сильного правила хорошего суффикса в методе Бойера–Мура дает в наихудшем случае время счета $O(n)$ в предположении, что образец в тексте не встречается. Это было впервые доказано Кнутом, Моррисом и Праттом [278], а еще одно доказательство было дано Гьюбасом и Одлыжко [196]. Оба доказательства были очень трудны и давали оценку не лучше, чем в $5m$ сравнений. Позднее Ричард Коул дал значительно более простое доказательство [108] с границей в $4m$ сравнений и трудное доказательство с жесткой оценкой в $3m$ сравнений. Мы представим доказательство Коула про $4m$ сравнений в п. 3.2.

Когда образец в тексте не встречается, исходный метод Бойера–Мура работает в худшем случае за время $O(nm)$. Однако некоторые простые модификации метода справляются с этой трудностью, обеспечивая границу времени $O(m)$ для всех случаев. Первая из этих модификаций была предложена Галилом [168]. После обсуждения доказательства Коула в п. 3.2 для случая, когда P не входит в T , мы используем вариант идеи Галила, чтобы получить линейную оценку для всех случаев.

С другой стороны, если мы используем только правило сдвига по плохому символу, то наихудшее время счета будет $O(nm)$, но при случайно генерируемых строках ожидаемое время сублинейно. Более того, в типичных приложениях поиска совпадений строк, включая тексты на обычном языке, на практике почти всегда наблюдается сублинейное время. Мы не будем обсуждать анализ случайных строк и отсылаем читателя к книге [184].

Хотя принадлежащее Коулу доказательство линейности времени существенно проще предыдущих и важно для завершения полного изложения метода Бойера–Мура, оно не тривиально. Однако совсем простое развитие алгоритма Бойера–Мура, предложенное Апостолико и Джанкарло [26], дает алгоритм, “подобный Бойеру–Муру”, с прямым доказательством оценки сравнений $2m$ в худшем случае. Этот вариант алгоритма рассматривается в п. 3.1.

2.3. Алгоритм Кнута–Морриса–Пратта

Самый известный алгоритм с линейным временем для задачи точного совпадения предложен Кнутом, Моррисом и Праттом [278]. Хотя этот метод редко используется и часто на практике уступает методу Бойера–Мура (и другим), он может быть просто объяснен, и его линейная оценка времени легко обосновывается. Кроме того, он создает основу для известного алгоритма Ахо–Корасика, который эффективно находит все вхождения в текст любого образца из заданного набора образцов *).

*) Несколько решений этой задачи с набором образцов, включая метод Ахо Корасика, излагаются в п. 3.4. По этим причинам и учитывая его историческое значение для данного вопроса, мы полностью приводим здесь метод Кнута–Морриса–Пратта.

2.3.1. Идея сдвига Кнута–Морриса–Пратта

Предположим, что при некотором выравнивании P около T наивный алгоритм обнаружил совпадение первых i символов из P с их парами из T , а при следующем сравнении было несовпадение. В этом случае наивный алгоритм сдвинет P на одно место и начнет сравнение заново с левого конца P . Но часто можно сдвинуть образец дальше. Например, если $P = abcxa\overline{bcde}$ и при текущем расположении P и T несовпадение нашлось в позиции 8 строки P , то есть возможность (и мы докажем это ниже) сдвинуть P на четыре места без пропуска вхождений P в T . Отметим, что это можно увидеть, ничего не зная о тексте T и расположении P относительно T . Требуется только место несовпадения в P . Алгоритм Кнута–Морриса–Пратта, основываясь на таком способе рассуждений, и делает сдвиг больше, чем наивный алгоритм. Теперь формализуем эту идею.

Определение. Для каждой позиции i образца P определим $sp_i(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P .

Иначе говоря, $sp_i(P)$ — это длина наибольшей собственной подстроки $P[1..i]$, которая кончается в i и совпадает с префиксом P . Когда аргумент P ясен из контекста, мы будем вместо полного обозначения использовать sp_i .

Например, если $P = abca\overline{eabcabd}$, то $sp_2 = sp_3 = 0$, $sp_4 = 1$, $sp_8 = 3$ и $sp_{10} = 2$. Заметьте, что по определению для любой строки $sp_1 = 0$.

Оптимизированный вариант алгоритма Кнута–Морриса–Пратта использует такие значения.

Определение. Для каждой позиции i образца P определим $sp'_i(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P , с дополнительным условием, что символы $P(i + 1)$ и $P(sp'_i + 1)$ не равны.

Ясно, что $sp'_i(P) \leq sp_i(P)$ для всех позиций i и любой строки P . Для примера, если $P = bbccaa\overline{ebbcabd}$, то $sp_8 = 2$, поскольку строка bb встречается и как собственный префикс $P[1..8]$, и как суффикс $P[1..8]$. Однако обе копии этой строки продолжаются одним и тем же символом c , так что $sp'_8 < 2$. На самом деле $sp'_8 = 1$, так как символ b является и первым, и последним символом $P[1..8]$ и продолжается символом b в позиции 2 и символом c в позиции 9.

Правило сдвига Кнута–Морриса–Пратта

Мы опишем алгоритм в терминах значений sp' и предоставим читателю его модификацию на случай, когда используются только слабые значения sp .*) Алгоритм Кнута–Морриса–Пратта прикладывает P к T и затем сравнивает соответствующие пары символов *слева направо*, как в наивном алгоритме.

Если для любого расположения P и T первое несовпадение (при ходе слева направо) отмечается в позиции $i + 1$ образца P и позиции k текста T , то сдвиг P вправо

*) Нужно предупредить читателя, что традиционно алгоритм Кнута–Морриса–Пратта описывается в терминах функций *неудач*, которые родственны значениям sp_i . Функции неудач явно вводятся в п. 2.3.3.

(относительно T) таков, что $P[1..sp'_i]$ пристроится к $T[k - sp'_i..k - 1]$. Другими словами, P сдвигается вправо на $i + 1 - (sp'_i + 1) = i - sp'_i$ мест, так что символ $sp'_i + 1$ из P поравняется с символом k из T . В случае если вхождение P найдено (несовпадений нет), P сдвигается на $n - sp'_n$ мест.

Это правило сдвига гарантирует, что после сдвига P префикс $P[1..sp'_i]$ совпадет с прилегающей подстрокой T . В следующем сравнении будут участвовать символы $T(k)$ и $P[sp'_i + 1]$. Сильное правило сдвига, использующее sp'_i , гарантирует, что то же несовпадение не встретится при новом расположении, но не гарантирует, что $T(k) = P[sp'_i + 1]$.

В приведенном примере, где $P = abcxabcde$ и $sp'_7 = 3$, если 8-й символ из P не дает совпадения, то P будет сдвинуто на $7 - 3 = 4$ места. Это утверждение верно независимо от того, каков текст T и каково взаиморасположение P и T .

Обсуждаемое правило сдвига имеет два преимущества. Во-первых, P часто сдвигается больше чем на один символ. Во-вторых, после сдвига гарантировано, что левые sp'_i символов P совпадают со своими парами в T . Таким образом, для проверки полного совпадения P с приложенным текстом T надо начинать сравнивать P и T с позиции $sp'_i + 1$ в P (и позиции k в T). Например, пусть, как раньше, $P = abcxabcde$, $T = xuabcxabcxadcqdqfeg$ и левый конец P приложен к 3-му символу в T . Тогда P и T совпадут по семи символам, на 8-м символе не совпадут, и P будет сдвинуто на четыре места, как это показано ниже:

1	2
123456789012345678	
xuabcxabcxadcqdqfeg	
abcxabcde	
abcxabcde	

Как и гарантировалось, первые 3 символа сдвинутого P совпадают со своими парами в T (и их парами в несдвинутом P).

Подводя итог, мы получаем следующую теорему.

Теорема 2.3.1. *После несовпадения в позиции $i + 1$ образца P и сдвига на $i - sp'_i$ места вправо левые sp'_i символов P гарантированно совпадут со своими парами в T .*

Теорема 2.3.1 частично обеспечивает корректность алгоритма Кнута–Морриса–Пратта, но для того, чтобы он выглядел безукоризненно, нам нужно показать, что правило не сдвигает образец слишком далеко, т. е. мы не пропустим ни одного вхождения P .

Теорема 2.3.2. *Для любого выравнивания P с T если символы от 1 до i в P равны своим парам в T , но символ $i + 1$ не совпадает с $T(k)$, то P может быть сдвинуто на $i - sp'_i$ места вправо без пропуска вхождений P в T .*

Доказательство. Предположим, напротив, что есть вхождение P , начинающееся строго слева от сдвинутого P , и пусть α и β — подстроки, представленные на рис. 2.2. В частности, β — префикс P длины sp'_i , показанный для P в сдвинутой позиции. Несдвинутый P совпадает с T вплоть до позиции i в P и позиции $k - 1$ в T , а все символы в (предполагаемом) пропущенном вхождении P совпадают со

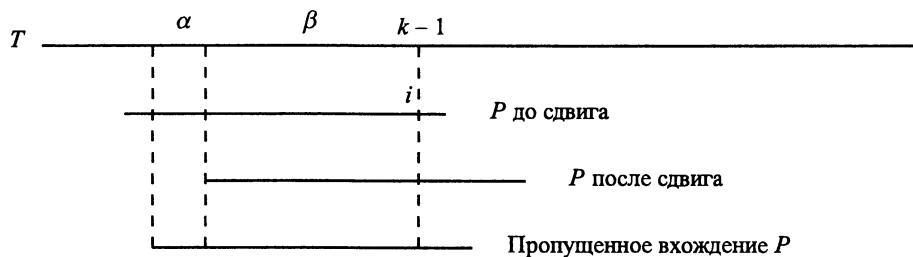


Рис. 2.2. Предполагаемый пропуск вхождения, используемый при доказательстве корректности метода Кнута–Морриса–Пратта

своими парами в T . Обе эти совпадающие области содержат подстроки α и β , так что несмещеннное P и предполагаемое вхождение P совпадают по всей подстроке $\alpha\beta$. Следовательно, $\alpha\beta$ является суффиксом $P[1..i]$, который совпадает с собственным префиксом P . Теперь пусть $l = |\alpha\beta| + 1$, так что позиция l в “пропущенном вхождении” P лежит напротив позиции k в T . Символ $P(l)$ не может равняться $P(i+1)$, так как по предположению $P(l)$ совпадает с $T(k)$, а $P(i+1)$ не совпадает с $T(k)$. Таким образом, $\alpha\beta$ — собственный суффикс $P[l..i]$, который совпадает с префиксом P , и следующий символ не равен $P(i+1)$. Но $|\alpha| > 0$ в силу предположения, что вхождение P начинается строго до смещенного P , так что $|\alpha\beta| > |\beta| = sp'_i$, вопреки определению sp'_i . Следовательно, теорема доказана. \square

Теорема 2.3.2 утверждает, что правило сдвига Кнута–Морриса–Пратта не пропускает вхождений P в T , и значит, алгоритм Кнута–Морриса–Пратта корректно найдет все вхождения P в T . Временной анализ так же прост.

Теорема 2.3.3. В методе Кнута–Морриса–Пратта число сравнений символов не превосходит $2m$.

Доказательство. Разделим алгоритм на фазы сравнения/сдвига, так что каждая фаза состоит из сравнений, выполняемых между последовательными сдвигами. После очередного сдвига сравнения начинаяющейся фазы выполняются слева направо и начинаются либо в последнем символе T , который сравнивался в предыдущей фазе, либо еще правее. Так как P никогда не сдвигается влево, в каждой фазе не более одного сравнения выполняется с символом T , который уже сравнивался. Таким образом, общее число сравнений символов не превосходит $m + s$, где s — число сдвигов, сделанных алгоритмом. Но $s < m$, так как после m сдвигов правый край P окажется справа от правого конца T , так что число сравнений ограничено числом $2m$. \square

2.3.2. Препроцессинг для метода Кнута–Морриса–Пратта

Ключом к ускорению алгоритма Кнута–Морриса–Пратта по сравнению с наивным алгоритмом является использование значений sp' (или sp). Легко увидеть, как они вычисляются из значений Z , полученных при основном препроцессинге P . Проверим это.

Определение. Позиция $j > 1$ отображается в i , если $i = j + Z_j(P) - 1$. Это значит, что j отображается в i , если i есть правый конец Z -блока, начинаящегося в j .

Теорема 2.3.4. Для любого $i > 1$ имеем $sp'_i(P) = Z_j = i - j + 1$, где $j > 1$ — наименьший индекс, отображаемый в i . Если ни одного такого j нет, то $sp'_i(P) = 0$. Для любого $i > 1$ имеем $sp_i(P) = i - j + 1$, где j — наименьший индекс в диапазоне $1 < j \leq i$, отображаемый в i или дальше. Если такого j нет, то $sp_i(P) = 0$.

Доказательство. Если $sp'_i(P)$ положительно, то существует собственный суффикс α строки $P[1..i]$, совпадающий с префиксом P и такой, что $P[i+1]$ не совпадает с $P[|\alpha|+1]$. Поэтому выбор такого j соответствует началу α с $Z_j = |\alpha| = sp'_i(P)$ и j отображается в i . Следовательно, если в диапазоне $1 < j \leq i$ нет j , отображаемого в i , то $sp'_i(P) = 0$.

Теперь предположим, что $sp'_i(P) > 0$ и j вычислено, как сказано выше. Мы утверждаем, что j — наименьшее место в диапазоне от 2 до i , отображающееся в i . Допустим, что это не так, и пусть j^* — место в диапазоне $1 < j^* < j$, отображающееся в i . Тогда $P[j^*..i]$ будет собственным суффиксом $P[1..i]$, который совпадает с префиксом P (назовем его β). Более того, по определению отображения, $P(i+1) \neq P(|\beta|)$, так что $sp'_i(P) \geq |\beta| > |\alpha|$, вопреки предположению, что $sp'_i = \alpha$.

Доказательство утверждений для $sp_i(P)$ аналогично и оставляется на упражнения. \square

По теореме 2.3.4 все значения sp' и sp можно вычислить, используя значения Z_i за линейное время следующим образом:

Метод Кнута–Морриса–Пратта с использованием Z

```

for i := 1 to n do
    sp'_i := 0;
    for j := n downto 2 do begin
        i := j + Z_j(P) - 1;
        sp'_i := Z_j;
    end;

```

Значения sp можно получить, добавив следующее:

```

sp_n(P) := sp'_n(P);
for i := n - 1 downto 2 do
    sp_i(P) := max{sp_{i+1}(P) - 1, sp'_i(P)};

```

2.3.3. Полная реализация метода Кнута–Морриса–Пратта

Мы описали алгоритм Кнута–Морриса–Пратта в терминах сдвигов P , но время, требуемое на сами сдвиги, никогда не учитывали. Дело в том, что сдвиг — это только принцип, а явным образом P никуда не сдвигается. Просто, как и в методе Бойера–Мура, увеличиваются указатели на P и T . Мы используем указатель p на место в P и один указатель c (от “current” — текущий символ) на место в T .

Определение. Для каждого места i от 1 до $n + 1$ определим функцию неудачи $F'(i)$ как $sp'_{i-1} + 1$ (и аналогично, $F(i) = sp_{i-1} + 1$), причем sp'_0 и sp_0 приняты равными 0.

Сейчас мы будем использовать только (сильную) функцию неудачи $F'(i)$, но дальше будем ссылаться и на $F(i)$.

После несовпадения в позиции $i + 1 > 1$ строки P алгоритм Кнута–Морриса–Пратта “сдвигает” P так, что следующими будут сравниваться символы в позиции c строки T и в позиции $sp'_i + 1$ строки P . Но $sp'_i + 1 = F'(i + 1)$, так что общий “сдвиг” можно выполнить за константное время, просто полагая p равным $F'(i + 1)$. Осталось два особых случая. Когда несовпадение нашлось в позиции 1 из P , то p полагается равным $F'(1) = 1$, а c увеличивается на 1. Когда находится вхождение P , то P сдвигается вправо на $n - sp'_n$ мест. Этот сдвиг реализуется тем, что $F'(n + 1)$ полагается равным $sp'_n + 1$.

Соединение всего вместе дает полный алгоритм Кнута–Морриса–Пратта.

Алгоритм Кнута–Морриса–Пратта

begin

Обработать P , найдя $F'(k) = sp'_{k-1} + 1$ для k от 1 до $n + 1$.

$c := 1$;

$p := 1$;

while $c + (n - p) \leq m$ do begin

 while $P(p) = T(c)$ и $p \leq n$ do begin

$p := p + 1$;

$c := c + 1$;

 end;

 if $p = n + 1$ then

 зарегистрировать вхождение P в T , начиная с позиции $c - n$.

 if $p = 1$ then $c := c + 1$

$p := F'(P)$;

end;

end.

2.4. Поиск строк в реальном времени

На стадии поиска в алгоритме Кнута–Морриса–Пратта образец P располагается около подстроки T , и эти две строки сравниваются слева направо до исчерпания всего P (это значит, что нашлось вхождение P в T) либо до несовпадения в каких-то позициях $i + 1$ в P и k в T . В последнем случае если $sp'_i > 0$, то P сдвигается вправо на $i - sp'_i$ мест, гарантируя, что префикс $P[1..sp'_i]$ сдвинутого образца совпадет с прилегающей подстрокой в T . Никаких явных сравнений этих подстрок не требуется, и следующими будут сравниваться символы $T(k)$ и $P(sp'_i + 1)$. Хотя сдвиг, основанный на sp'_i , обеспечивает, что $P(i + 1)$ отличается от $P(sp'_i + 1)$, он не гарантирует, что $T(k) = P(sp'_i + 1)$. Следовательно, $T(k)$ может сравниваться несколько раз (возможно, $\Omega(|P|)$ раз) с разными символами из P . По этой причине метод Кнута–Морриса–Пратта не является методом *реального времени*.

Чтобы быть методом реального времени, он должен делать не более чем *контактную* работу между первой и последней проверками любой позиции в T . В методе Кнута–Морриса–Пратта если символ из T совпал с чем-то, то он больше не проверяется (в этом легко убедиться), но, как отмечено выше, это не так, если он не совпал с символом из образца. Заметим, что определение реального времени имеет отношение только к поисковой стадии алгоритма. К предварительной обработке P требования реального времени не относятся. Укажем также, что если стадия поиска работает в реальном времени, то она работает за линейное время.

Польза от “сравнителя за реальное время” (real-time matcher) двоякая. Во-первых, в некоторых приложениях, например, когда символы текста засыпаются в машину с маленькой памятью, может потребоваться полная обработка каждого символа до прибытия следующего. Задача решается, если время обработки константное, не зависящее от длины строки. Во-вторых, в таком сравнителе за реальное время сдвиги P могут быть длиннее, но никогда не станут короче, чем в исходном алгоритме Кнута–Морриса–Пратта. Следовательно, в некоторых случаях он сможет работать быстрее.

По правде сказать, аргументы в пользу алгоритмов сравнения за реальное время перед методами линейного времени несколько вымученные, и сравнение в реальном времени имеет скорее теоретическое, чем практическое значение. Тем не менее стоит потратить немного времени на его обсуждение.

2.4.1. Преобразование метода Кнута–Морриса–Пратта в метод реального времени

Для преобразования метода Кнута–Морриса–Пратта в метод реального времени мы воспользуемся значениями Z , полученными при основном препроцессинге P . Требуемый здесь препроцессинг P совершенно аналогичен обработке в п. 2.3.2 для алгоритма Кнута–Морриса–Пратта. По историческим причинам получающийся метод реального времени часто называется *детерминированным автоматом для сравнения строк* (a deterministic finite-state string matcher) и представляется диаграммой автомата. Здесь мы не будем использовать эту терминологию, изобразив метод в псевдокоде.

Определение. Пусть x — какой-нибудь символ алфавита. Для каждой позиции i образца P определим $sp'_{(i,x)}(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P , *при дополнительном условии, что символ $P(sp'_i + 1)$ равен x* .

Знание значений $sp'_{(i,x)}$ для всех символов алфавита x позволяет сформулировать правило сдвига, которое преобразует метод Кнута–Морриса–Пратта в алгоритм реального времени. Предположим, что образец P сравнивался с подстрокой T и налось несовпадение символов $T(k) = x$ и $P(i+1)$. В этом случае P нужно сдвинуть вправо на $i - sp'_{(i,x)}$ мест. Этот сдвиг обеспечивает, что префикс $P[1..sp'_{(i,x)}]$ совпадает с прилегающей подстрокой T , а $T(k)$ — со следующим символом P . Следовательно, сравнение $T(k)$ и $P(sp'_{(i,x)} + 1)$ можно опустить. Следующее необходимое сравнение относится к символам $P(sp'_{(i,x)} + 2)$ и $T(k+1)$. С таким правилом сдвига метод становится методом реального времени, так как он не анализирует повторно символ

из T , совпадший с символом образца (черта, унаследованная от алгоритма Кнута–Морриса–Пратта), и не анализирует также символ, который не совпал. Итак, на стадии поиска в этом алгоритме никакой символ T не проверяется больше одного раза. Отсюда следует, что поиск идет в реальном времени. Ниже мы покажем, как за линейное время найти все значения $sp'_{(i,x)}$. Все вместе дает нам алгоритм с линейной обработкой P и поиском в T за реальное время.

Легко убедиться, что этот алгоритм находит все вхождения P в T , — мы оставляем это как упражнение.

2.4.2. Препроцессинг для поиска в реальном времени

Теорема 2.4.1. Для $P[i + 1] \neq x$ имеет место равенство $sp'_{(i,x)}(P) = i - j + 1$, где j — наименьшая позиция, такая что j отображается в i и $P(Z_j + 1) = x$. Если такого j нет, то $sp'_{(i,x)}(P) = 0$.

Доказательство этой теоремы почти идентично доказательству теоремы 2.3.4 (с. 50) и оставляется читателю. В предположении (обычном), что алфавит конечен, следующая небольшая модификация обработки для метода Кнута–Морриса–Пратта (п. 2.3.2) дает необходимые значения $sp'_{(i,x)}$ за линейное время.

Поиск за реальное время, основанный на Z -значениях

```
for  $i := 1$  to  $n$  do
     $sp'_{(i,x)} := 0$  для каждого символа  $x$ ;
    for  $j := n$  downto 2 do begin
         $i := j + sp'_{(i,x)}(P) - 1$ ;
         $x := P(Z_j + 1)$ ;
         $sp'_{(i,x)} := Z_j$ ;
    end;
```

Заметим, что линейная граница времени (и памяти) требует, чтобы алфавит Σ был конечен. Это позволяет нам делать $|\Sigma|$ сравнений за константное время. Если размер алфавита явно включать в оценки, то время и память, необходимые при обработке, имеют оценку $O(|\Sigma|n)$.

2.5. Упражнения

- В “типовых” приложениях точного поиска, таких как поиск английского слова в книге, простое правило плохого символа выглядит столь же эффективным, как расширенное. Объясните это “на пальцах”.
- При поиске отдельного слова или маленькой фразы в большом тексте грубая сила (наивный алгоритм), как сообщается в [184], работает быстрее, чем многие другие методы. Объясните это “на пальцах”. Как вы думаете, сохранится ли результат в случае маленьких алфавитов (скажем, в ДНК с алфавитом из четырех букв), когда размер образца растет и когда текст имеет много секций со схожими, но не совпадающими подстроками?

3. “Здравый смысл” и оценка порядка $\Theta(nt)$ для наихудшего случая в алгоритме Бойера–Мура (при использовании только правила плохого символа) дают основания думать, что эмпирическое время счета растет с ростом длины образца (при фиксированном тексте). Но при поиске в реальных английских текстах алгоритм Бойера–Мура с удлинением образца считает быстрее. Так, в английских текстах порядка 300 000 символов слово *Inter* ищется в пять раз дольше, чем *Interactively*.
- Объясните это “на пальцах”. Предположим, что длина образца растет неограниченно. В какой момент вы ожидаете прекращения убывания времени поиска? Не кажется ли вам, что в какой-то момент время начнет расти?
4. Оцените эмпирически выгоду от применения расширенного правила плохого символа по сравнению с простым. Проведите оценку в сочетании с разными вариантами правила хорошего суффикса. Насколько увеличится средний сдвиг при использовании расширенного правила? Окупает ли увеличение сдвига дополнительные затраты в счете?
5. Оцените эмпирически, делая различные предположения о размерах P и T , числе вхождений P в T и размере алфавита, следующую идею ускорения метода Бойера–Мура. Предположим, что фаза алгоритма кончается несовпадением и что правило хорошего суффикса сдвигает P дальше, чем расширенное правило плохого символа. Пусть x и y обозначают несовпадающие символы в T и P соответственно, а z — символ в сдвигнутом P под x . По правилу суффикса z не равен y , но нет гарантии, что он равен x . Почему бы вместо того, чтобы начинать сравнения сдвигнутого P справа, как полагается в методе Бойера–Мура, не сравнить сначала x и z ? Если они равны, то сравнение справа налево начнется с правого конца P , но если они не равны, то мы можем применить расширенное правило плохого символа от z в P . Это сдвинет P снова. В этот момент мы должны будем начать сравнение P с T справа налево.
6. Идея правила плохого символа в алгоритме Бойера–Мура может быть обобщена таким образом, чтобы вместо проверки символов из P справа налево алгоритм сравнивал символы P в порядке их встречаемости в T (начиная с самых редких). То есть предлагается сначала смотреть на те символы из P , наличие которых наименее вероятно в T . При несовпадении применяется, как и раньше, простое или расширенное правило плохого символа. Оцените практичность этого подхода, эмпирически на реальных данных или с помощью анализа, предполагая строки случайными.
7. Постройте пример, когда при использовании только правила плохого символа делается меньше сравнений, чем при комбинировании его с правилом хорошего суффикса.
8. Оцените эмпирически эффективность правила сильного хорошего суффикса по сравнению со слабым в методе Бойера–Мура.
9. Докажите теорему 2.2.4. Покажите, как за линейное время найти все значения $I'(i)$.
10. Если мы используем в методе Бойера–Мура слабое правило хорошего суффикса, которое сдвигает ближайшую копию t до совпадения с суффиксом t , но не требует, чтобы следующий символ отличался, то препроцессинг может использовать непосредственно значения sp_i , а не значения Z . Объясните это.
11. Докажите, что правила сдвига в методе Кнута–Морриса–Пратта (основанные как на sp , так и на sp') не пропускают вхождений P в T .
12. Можно присоединить правило сдвига по плохому символу из метода Бойера–Мура к методу Кнута–Морриса–Пратта или прямо к наивному методу. Покажите, как это сделать. Оцените эффективность этого правила и объясните, почему оно более действенно на своем месте, в алгоритме Бойера–Мура.

13. Вспомним определение l_i на с. 28. Естественно предположить, что $sp_i = i - l_i$ для любого индекса i , где $i \geq l_i$. Покажите на примере, что это предположение неверно.
14. Докажите утверждения теоремы 2.3.4 относительно $sp'_i(P)$.
15. Верно ли, что при заданных значениях sp для строки P величины sp' полностью определены? Можно ли получить значения sp только по $sp'?$

Использование значений sp для вычисления значений Z . В п. 2.3.2 мы показали, что все значения sp можно вычислить, зная только значения Z для строки S (т. е. не зная самой S). В следующих пяти упражнениях мы установим обратное и создадим алгоритм с линейным временем для вычисления всех значений Z , исходя только из значений sp . Первое упражнение предлагает естественный путь для такого вычисления, а следующее — обнаруживает в этом методе пробел. Три оставшихся упражнения развивают корректный алгоритм с линейным временем, описанный в [202]. Скажем, что sp_i отображается в k , если $k = i - sp_i + 1$.

16. Предположим, что существует такая позиция i , что sp_i отображается в k , и пусть i — наибольшая из таких позиций. Докажите, что $Z_k = i - k + 1 = sp_i$ и что $r_k = i$.
17. Зная ответ предыдущего упражнения, естественно предположить, что Z_k всегда равно sp_i , где i — наибольшая позиция, для которой sp_i отображается в k . Покажите, что это не так. Приведите пример, использующий не меньше трех различных символов.

Говоря иначе, постройте пример, показывающий, что Z_k может быть больше нуля, даже если *нет* позиции i , для которой sp_i отображается в k .

18. Напомним, что r_{k-1} известно к началу итерации k в Z-алгоритме (когда вычисляется Z_k), но r_k известно только в конце итерации k . Предположим, однако, что r_k известно (откуда-то) в начале итерации k . Покажите, как Z-алгоритм можно модифицировать, чтобы Z_k вычислялось без сравнения символов. Благодаря этому модифицированный алгоритм и не будет нуждаться в строке S .
19. Докажите, что если Z_k больше нуля, то r_k равно наибольшей позиции i , в которой $k \geq i - sp_i$. Убедитесь, что r_k может быть получено из значений sp для каждой позиции k , в которой $Z_k \neq 0$.
20. Скомбинируйте результаты двух предыдущих упражнений для создания алгоритма с линейным временем, который вычисляет все значения Z для строки S по заданным значениям sp и без самой строки S . Объясните, в каком смысле этот метод является “имитацией” Z-алгоритма.
21. Может показаться, что $l'(i)$ (необходимое для метода Бойера–Мура) должно быть равно sp_n для любого i . Покажите, почему это не так.
22. В п. 1.5 мы показали, что все вхождения P в T можно найти за линейное время вычислением значений Z для строки $S = P\$T$. Объясните, как нужно изменить этот метод, если мы используем $S = PT$, т. е. не ставим разделителя между строками P и T . Покажите теперь, как найти все вхождения P в T за линейное время, используя $S = PT$, но со значениями sp вместо Z . (Это не так просто, как кажется на первый взгляд.)
23. В методе Бойера–Мура и ему подобных алгоритмах поиск в образце идет справа налево, хотя сам образец движется по тексту слева направо. Это несколько затрудняет объяснение методов и комбинирование препроцессинга для методов Бойера–Мура и Кнута–Морриса–Пратта. Однако небольшое изменение метода Бойера–Мура позволило бы упростить изложение и унифицировать подготовку. Нужно сначала поставить образец в *правый* конец

текста и вести весь поиск в образце *слева направо*, сдвигая образец *влево* после несовпадения. Разработайте детали этого подхода и покажите, как он позволит унифицировать изложение препроцессинга, необходимого для этого метода и метода Кнута–Морриса–Пратта. Почему в среднем этот метод ведет себя так же, как исходный метод Бойера–Мура?

24. Ниже приводится код на Паскале (Турбо Паскале), реализующий препроцессинг по Ричарду Коуллу для сильного правила хорошего суффикса. Он отличается от препроцессинга, базирующегося на основном препроцессинге, и ближе к исходному методу, описанному в [278]. Разберитесь в коде, чтобы извлечь из программы алгоритм. Затем объясните идею алгоритма, докажите его корректность и проанализируйте время счета. Имейте в виду, что извлечь алгоритмический замысел из программы непросто.

```

program gsmatch(input,output);
{реализация метода обработки Ричарда Коула
для сильного правила хорошего суффикса}
type
tstring = string[200];
indexarray = array[1..100] of integer;

const
zero = 0;

var
p: tstring;
bmshift,matchshift: indexarray;
m,i: integer;

procedure readstring(var p: tstring; var m: integer);
begin
read(p);
m := Length(p);
writeln('длина строки равна ', m);
end;

procedure gsshift(p: tstring;
var gs_shift: indexarray; m: integer);

var
i,j,j_old,k: integer;
kmp_shift: indexarray;
go_on: boolean;
begin {1}
for j := 1 to m do
gs_shift[j] := m;
kmp_shift[m] := 1;
{stage 1}
j := m;

for k := m-1 downto 1 do begin {2}
go_on := true;
while (p[j] <> p[k]) and go_on do begin {3}

```

```

    if (gs_shift[j] > j-k) then gs_shift[j] := j-k;
    if (j < m) then j := j+kmp_shift[j+1]
    else go_on := false;
end; {3}
if (p[k] = p[j]) then begin {3}
    kmp_shift[k] := j-k;
    j := j-1;
end {3}
else
    kmp_shift[k] := j-k+1;
end; {2}

{stage 2}
j := j+1;
j_old := 1;

while (j <= m) do begin {2}
    for i := j_old to j-1 do
        if (gs_shift[i] > j-1) then gs_shift[i] := j-1;
    j_old := j;
    j := j+kmp_shift[j];
end; {2}
end; {1}

begin {main}

writeln('введите строку в виде одной строки текста');

readstring(p,m);
gsshift(p,matchshift,m);
writeln('Значение в ячейке i равно величине сдвига после');
writeln('несовпадения в позиции образца i');

for i := 1 to m do
    write(matchshift[i]:3);
writeln;
end. {main}

```

25. Докажите, что правило сдвига, используемое “сравнителем строк в реальном времени”, не пропускает вхождений P в T .
26. Докажите теорему 2.4.1.
27. В этой главе мы показали, как использовать значения Z для вычисления и значений sp'_i и sp_i в методе Кнута–Морриса–Пратта и значений $sp'_{(i,x)}$ в его модификации для реального времени. Покажите, как вместо использования значений Z для $sp'_{(i,x)}$ получить желаемое из sp_i и/или sp'_i за линейное время порядка $O(n|\Sigma|)$, где $n = |P|$ и $|\Sigma|$ — размер алфавита.
28. Мы не знаем, как переделать алгоритм Бойера–Мура в метод реального времени тем же путем, что и метод Кнута–Морриса–Пратта. Однако можно аналогично изменить сильное правило сдвига, чтобы повысить эффективность метода Бойера–Мура. Именно, когда нашлося несовпадение $P(i)$ и $T(h)$, мы можем поискать крайнюю правую копию $P[i + 1..n]$

в P (отличную от самой $P[i + 1..n]$), такую что ей предшествует символ $T(h)$. Покажите, как модифицировать препроцессинг Бойера–Мура таким образом, чтобы необходимая информация при фиксированном размере алфавита собиралась за линейное время.

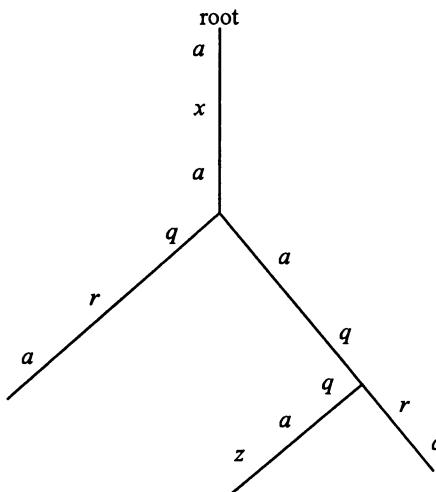


Рис. 2.3. Образец $P = aqra$ помечает два подпути путей, начинающихся в корне. Сами пути начинаются в корне, но подпути, содержащие $aqra$, — нет. В дереве есть другой подпуть, помеченный $aqra$ (он начинается выше символа z), который, однако, нарушает требование, чтобы это был подпуть пути, начинающегося в корне. Заметим, что метка ребра выписана сверху вниз. Так, на рисунке есть дуга, помеченная qra , и нет arg

29. Предположим, что задано дерево, каждое ребро которого помечено одним или более символом, и задан образец P . Метка подпути в дереве есть конкатенация меток ребер, входящих в подпуть. Задача состоит в том, чтобы у путей, начинающихся из корня, найти все подпути, которые помечены образцом P . Заметим, что, хотя подпуть может быть частью пути, направленного из корня, сам подпуть начинаться из корня не обязан (рис. 2.3). Предложите для этой задачи алгоритм со временем работы, пропорциональным полному числу символов на ребрах дерева плюс $|P|$.

Глава 3

Более глубокий взгляд

3.1. Вариант метода Бойера–Мура с “простой” линейной оценкой времени

Апостолико и Джанкарло [26] предложили вариант алгоритма Бойера–Мура, который допускает замечательно простое доказательство линейной оценки наихудшего времени счета. В этом варианте никакой символ из T не участвует в сравнениях после его первого совпадения с каким-нибудь символом из P . Отсюда немедленно следует, что число сравнений не превзойдет $2m$. Каждое сравнение дает либо совпадение, либо несовпадение; последних может быть только m , так как при каждом несовпадении происходит ненулевой сдвиг P , а совпадений — не больше m , так как никакой символ T не сравнивается после совпадения с символом из P . Мы покажем, что и остальная вычислительная работа (кроме сравнений) в этом методе линейно зависит от m .

Когда уже знаешь историю очень трудного и неполного анализа алгоритма Бойера–Мура, поражаешься тому, что близкий вариант этого алгоритма имеет простую линейную оценку времени. Мы представим здесь дальнейшее развитие идеи Апостолико–Джанкарло, в результате которой получился алгоритм, *точно* воспроизводящий сдвиги алгоритма Бойера–Мура. Поэтому предлагаемый метод сохраняет все преимущества быстрых сдвигов метода Бойера–Мура и допускает простой анализ времени счета с линейной оценкой для наихудшего случая.

3.1.1. Ключевые идеи

Наш вариант алгоритма Апостолико–Джанкарло имитирует алгоритм Бойера–Мура, находя в точности те же несовпадения и делая в точности те же сдвиги. Однако он предвидит результат многих сравнений, которые делает метод Бойера–Мура, и избегает их.

На высоком уровне алгоритм Бойера–Мура выглядит так. Мы делим его работу на *фазы сравнения/сдвига*, нумеруемые от 1 до $q \leq m$. В каждой фазе сравнения/сдвига правый конец P прикладывается к какому-то символу из T , и P сравнивается справа налево с выбранными символами T до обнаружения либо полного совпадения с P , либо несовпадения в каком-то символе. После этого P сдвигается вправо на расстояние, предписываемое правилами сдвига Бойера–Мура.

Напомним из п. 2.2.4, где обсуждалась препроцессная обработка для метода Бойера–Мура, что $N_i(P)$ — это длина наибольшего суффикса $P[1..i]$, совпадающего с суффиксом P . Там мы показали, как вычислять N_i для каждого i за время $O(n)$, где n — длина P . Будем считать сейчас, что вектор N уже получен в препроцессинге P .

В алгоритме Бойера–Мура нужно сделать две модификации. Во-первых, при поиске P в T (уже после препроцессинга), мы будем работать с вектором M длины m и на каждой фазе модифицировать не более одного его элемента. Рассмотрим фазу, в которой правый конец P расположен у позиции j строки T , и предположим, что P и T совпали в l местах (справа налево), но не далее. Положим $M(j)$ равным $k \leq l$ (правила выбора k уточняются ниже). $M(j)$ фиксирует тот факт, что суффикс P длины k (по меньшей мере) входит в T и кончается ровно в позиции j . По мере работы алгоритма значение $M(j)$ вычисляется для каждой позиции j строки T , которой сопоставлен правый конец P ; для остальных позиций T значение $M(j)$ не определено.

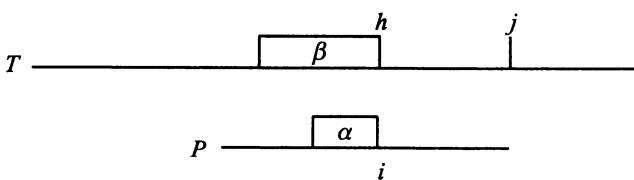


Рис. 3.1. Подстрока α имеет длину N_i , а подстрока β — длину $M(h) > N_i$. Правые концы строк совпадают на N_i символов, но следующие символы уже различаются

Вторая модификация, используя векторы N и M , ускоряет алгоритм Бойера–Мура за счет пропуска некоторых проверок. Чтобы прояснить саму идею, предположим, что алгоритм собирается сравнить символы $P(i)$ и $T(h)$, и допустим, что $M(h) > N_i$ (рис. 3.1). Это означает, что подстрока P длины N_i кончается в позиции i и совпадает с суффиксом P , тогда как подстрока T длины $M(h)$ кончается в позиции h и совпадает с суффиксом P . Значит, суффиксы этих подстрок длины N_i должны совпадать, и мы можем заключить, что следующие N_i сравнений (от $P(i)$ и $T(h)$ влево) алгоритма Бойера–Мура дадут совпадение. Далее, если $N_i = i$, то нашлось вхождение P в T , а если $N_i < i$, то мы можем быть уверены, что следующее сравнение (после N_i совпадений) даст несовпадение. Следовательно, в имитации Бойера–Мура, если

$M(h) > N_i$, мы можем избежать по меньшей мере N_i явных сравнений. Конечно, условие $M(h) > N_i$ не исчерпывает всех случаев, но они схожи, как будет показано ниже.

3.1.2. Одна фаза подробно

Как и в оригинальном алгоритме Бойера–Мура, когда правый конец P встанет у позиции j в T , P сравнивается с T справа налево. Когда несовпадение находится явно или устанавливается логически, а также когда находится вхождение P , образец сдвигается по исходным правилам сдвига Бойера–Мура (в сильном или слабом варианте) и фаза сравнения/сдвига на этом кончается. Здесь мы приведем детали только одной фазы. Она начинается при $h = j$ и $i = n$.

Фазовый алгоритм

- Если $M(h)$ не определено или $M(h) = N_i = 0$, то сравнить $T(h)$ и $P(i)$ следующим образом.

Если $T(h) = P(i)$ и $i = 1$, то сообщить о вхождении P , кончающемся позицией j в строке T , положить $M(j) = n$ и сдвинуть P , как в алгоритме Бойера–Мура (завершая эту фазу).

Если $T(h) = P(i)$ и $i > 1$, то уменьшить h и i на единицу и повторить фазовый алгоритм.

Если $T(h) \neq P(i)$, то положить $M(j) = j - h$ и сдвинуть P по правилам Бойера–Мура, основываясь на несовпадении, обнаруженному в позиции i строки P (это завершает фазу).

- Если $M(h) < N_i$, то P совпадает с приложенной частью T от позиции n влево до позиции $i - M(h) + 1$. По определению $M(h)$, P могло бы совпадать с большей частью T налево, так что нужно положить i равным $i - M(h)$, h равным $h - M(h)$ и повторить фазовый алгоритм.
- Если $M(h) \geq N_i$ и $N_i = i > 0$, то сообщить, что вхождение P найдено в T с концом в позиции j . $M(j)$ должно получить значение, не превосходящее n . Положить $M(j) = j - h$ и сдвинуть образец согласно правилам Бойера–Мура, применимым к случаю, когда нашлось вхождение P с концом в j (это завершает фазу).
- Если $M(h) > N_i$ и $N_i < i$, то P совпадает с T от своего правого конца до символа $i - N_i + 1$, но следующая пара символов уже не совпадает (т.е. $P(i - N_i) \neq T(h - N_i)$). Следовательно, P совпадает с T в $j - h + N_i$ символах и не совпадает в своей позиции $i - N_i$. Таким образом, значение $M(j)$ должно быть не больше $j - h + N_i$. Положим $M(j) = j - h$. Сдвинем P по правилам Бойера–Мура, относящимся к несовпадению в позиции $i - N_i$ строки P (это завершает фазу).
- Если $M(h) = N_i$ и $0 < N_i < i$, то P и T должны совпадать по меньшей мере на $M(h)$ символов влево, но левый конец P еще не достигнут, так что следует уменьшить i и h на $M(h)$ и повторить фазовый алгоритм.

3.1.3. Корректность и анализ линейности времени

Теорема 3.1.1. Используя M и N , как выше, вариант Апостолико–Джанкарло алгоритма Бойера–Мура правильно находит все вхождения P в T .

Доказательство. Мы докажем корректность, когда убедимся, что алгоритм имитирует исходный алгоритм Бойера–Мура. Действительно, при любом данном расположении P относительно T алгоритм корректен, когда он объявляет совпадение до данной позиции и несовпадение в следующей позиции. Остаток имитации корректен, так как правила сдвига те же, что и в алгоритме Бойера–Мура.

Сделаем индуктивное предположение, что значения $M(h)$ верны вплоть до некоторой позиции в T . Это значит, что там, где $M(h)$ определено, в T имеется подстрока длины $M(h)$, кончающаяся в позиции h , которая совпадает с суффиксом P . Первое такое значение, $M(n)$, правильно потому, что оно найдено прикладыванием P к левой части T и непосредственными сравнениями, соответствующими случаю 1 фазового алгоритма до обнаружения несовпадения или полного вхождения P . Рассмотрим теперь фазу, когда правый конец P выровнен по позиции j строки T . Фаза имитирует действия метода Бойера–Мура за исключением того, что в случаях 2, 3, 4 и 5 некоторые явные сравнения опускаются, а в случае 4 несовпадение устанавливается логически, а не наблюдается. Но, хотя сравнения и пропускаются, они должны давать совпадение по определению N и M и по предположению, что значения M до данного момента вычислялись корректно. Поэтому корректен и пропуск этих сравнений. В случае 4 несовпадение в позиции $i - N_i$ из P правильно устанавливается логически, так как N_i есть максимальная длина подстроки, кончающейся в позиции i и совпадающей с суффиксом P , тогда как $M(h)$ не превосходит максимальной длины подстроки, кончающейся в h и совпадающей с суффиксом P . Следовательно, эта фаза корректно имитирует метод Бойера–Мура и находит точно то же несовпадение (или вхождение P в T), которое нашел указанный алгоритм. Значение, присваиваемое $M(j)$, верно, так как во всех случаях оно не превосходит длины суффикса P , который должен совпадать с соответствующими символами подстроки $T[1..i]$. \square

Следующие определения и лемма будут полезны при уточнении работы алгоритма.

Определение. Если j — позиция, где $M(j)$ положительно, интервал $[j - M(j) + 1..j]$ называется *покрытым интервалом*, определенным позицией j .

Определение. Пусть $j' < j$ и покрытые интервалы определены и для j , и для j' . Будем говорить, что эти покрытые интервалы *пересекаются*, если $j - M(j) + 1 \leq j'$ и $j' - M(j') + 1 < j - M(j) + 1$ (рис. 3.2).

Лемма 3.1.1. Никакие два покрытых интервала, вычисленных алгоритмом, не пересекаются. Более того, если алгоритм проверяет позицию h из T в покрытом интервале, то h — правый конец этого интервала.

Доказательство. Доказательство проводится индукцией по числу созданных интервалов. Ясно, что утверждение верно, когда создан еще только первый интервал, который сам себя не пересекает. Предположим, что пересекающихся интервалов нет и рассматривается фаза, когда правый конец P выровнен по позиции j из T .

Так как в начале фазы $h = j$ и j лежит справа от любого интервала, процесс начинается с h , находящегося вне любого интервала. Рассмотрим, как h могло бы попасть

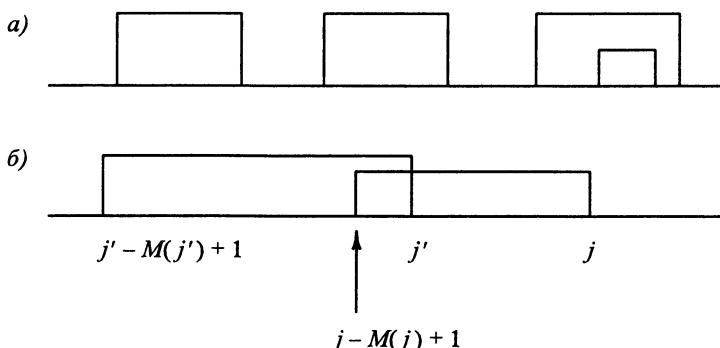


Рис. 3.2. Покрытые интервалы: *а* — диаграмма, показывающая покрытые интервалы, которые не пересекаются, хотя один интервал может содержать другой; *б* — два пересекающихся покрытых интервала

в позицию внутри интервала, отличную от его правого конца. Случай 1 никогда не имеет места, если h есть правый конец интервала (поскольку здесь $M(h)$ определено и положительно), а если он возникает, то либо фаза завершается, либо h увеличивается на единицу. Значит, действия случая 1 не вызовут перемещения h внутрь покрытого интервала. То же верно для случаев 3 и 4, так как фаза в каждом из этих случаев завершается. Таким образом, если h и сдвигается внутрь интервала, такой сдвиг может произойти только при появлении случаев 2 или 5, в любом из которых h передвигается от правого конца некоторого интервала $I = [k..h]$ к позиции $k - 1$, на одно место влево от I . Допустим, что $k - 1$ лежит в некотором интервале I' , но не в его правом конце, и что это первый случай в фазе, когда h (сейчас равное $k - 1$) находится в позиции интервала, отличной от его правого конца. Это означает, что правый конец I не может быть левее правого конца I' (так как тогда позиция $k - 1$ была бы строго внутри I'), и правые концы I и I' не могут совпадать (так как $M(h)$ имеет для каждого h не больше одного значения). Но из этих условий следует, что I и I' пересекаются вопреки предположению. Следовательно, если никакие интервалы не пересекаются в начале фазы, то в самой фазе проверяется только правый конец любого покрытого интервала.

Новый покрытый интервал создается только в результате реализации случаев 1, 3 и 4. В любом из них интервал $[h + 1..j]$ создается после того, как алгоритм проверит позицию h . Значение h в случае 1 не принадлежит никакому интервалу, а в случаях 3 и 4 является правым концом интервала, так что во всех случаях $h + 1$ — это либо непокрытый интервал, либо левый конец интервала. Так как j находится справа от интервала и $h + 1$ либо вне интервала, либо на его левом конце, новый интервал $[h + 1..j]$ не пересекает никакого существующего интервала. Ранее существовавшие интервалы не менялись, так что фаза не создала пересекающихся интервалов, и индукция завершена. \square

Теорема 3.1.2. *Модифицированный алгоритм Апостолико–Джанкарло выполняет не более $2m$ сравнений символов и не более $O(m)$ дополнительных действий.*

Доказательство. Каждая фаза завершается, когда сравнение находит несовпадение, и после каждой фазы, кроме последней, выполняется ненулевой сдвиг P . Таким

образом, алгоритм может найти не более m несовпадений. Чтобы оценить число совпадений, заметим, что символы явно сравниваются только в случае 1, и если сравнение, включающее $T(h)$, дает совпадение, то в конце фазы $M(j)$ получает значение не меньшее $j - h + 1$. Это означает, что все символы в T , которые совпадают с символом из P на этой фазе, содержатся в покрытом интервале $[j - M(j) + 1..j]$. Далее алгоритм проверяет только правый конец интервала, и если h — правый конец, то $M(h)$ определено и положительно, так что алгоритм никогда не сравнивает символ из покрытого интервала. Следовательно, никакой символ из T после удачного сравнения не будет сравниваться заново. Поэтому алгоритм находит не более m совпадений, и полное число сравнений не превосходит $2m$.

Чтобы оценить объем дополнительной работы, найдем число обращений к M при исполнении всех пяти случаев, так как интересующая нас величина пропорциональна этому числу. В случае 1 происходит сравнение символов. В случаях 3 и 4 образец P немедленно сдвигается. Так как мы имеем не более $O(m)$ сдвигов и сравнений, то случаи 1, 3 и 4 могут исполняться не более $O(m)$ раз. Однако случаи 2 и 5 могут выполняться без немедленного сдвига или сравнения, так как в них предусмотрена возможность итерирования до сравнения или сдвига. Например, случай 5 можно встретить подряд дважды (без сдвига или сравнения), если $N_i = M(h) > 0$ и $N_{i-N_i} = M(h - M(h))$. Но когда мы встречаем случай 2 или 5, то $j > h$, и $M(j)$ получит в конце фазы значение не меньше $j - h + 1$. Таким образом, позиция h окажется внутри покрытого интервала, определенного j . Поэтому h не будет проверяться снова, и нового обращения к $M(h)$ никогда не последует. В результате мы видим, что случаи 2 и 5 применимы не более одного раза к каждой позиции в T , так что число обращений, связанных с этими случаями, также имеет порядок $O(m)$. \square

3.2. Линейная оценка Коула для наихудшего случая в методе Бойера–Мура

Здесь мы представляем в окончательном виде анализ линейности наихудшего времени в *оригинальном* методе Бойера–Мура. Рассмотрим сначала отдельно использование (сильного) правила хорошего суффикса. Позднее мы покажем, как влияет на анализ правило плохого символа. Напомним, что правило хорошего суффикса заключается в следующем:

Предположим, что при данном расположении P и T подстрока t из T совпадает с суффиксом P , но при следующем сравнении слева возникает несовпадение. Найдем в P , если она существует, самую правую копию t' подстроки t , такую что t' не является суффиксом P и символ слева от t' отличается от несовпадающего символа в P . Сдвинем P вправо так, чтобы подстрока t' в P встала под подстрокой t в T (вспомните рис. 2.1). Если t' не существует, то сдвинем левый конец P далее левого конца t в T на наименьшее расстояние, при котором префикс сдвинутого образца совпадет с суффиксом t в T . Если такой сдвиг невозможен, то сдвинем P на n мест вправо.

Если вхождение P найдено, P сдвигается на наименьшее расстояние, при котором собственный префикс сдвинутого образца совпадет с суффиксом вхождения P в T . Если такой сдвиг невозможен, то нужно сдвинуть P на n позиций.

Мы покажем, что уже при использовании одного только правила хорошего суффикса метод Бойера–Мура имеет оценку времени $O(m)$ в наихудшем случае — в предположении, что образца в тексте нет. Позднее мы разовьем метод Бойера–Мура, чтобы обеспечить и случай появления образца в тексте.

Как и при анализе алгоритма Апостолико–Джанкарло, разделим алгоритм Бойера–Мура на фазы *сравнения/сдвига*, занумерованные от 1 до $q \leq m$. В каждой такой фазе i суффикс P сравнивается справа налево с символами T , пока либо не обнаружится P полностью, либо не найдется несовпадение. В последнем случае подстроку T , состоящую из совпавших символов, обозначим через t_i ; как раз слева от t_i появилось несовпадение. Образец сдвигается вправо на расстояние, определяемое правилом хорошего суффикса.

3.2.1. Доказательство Коула в случае, когда образца в тексте нет

Определение. Пусть s_i обозначает расстояние, на которое P сдвигается вправо в конце фазы i .

Предположим, что P не встречается в T , так что сравнение в каждой фазе кончается несовпадением. В каждой фазе сравнения/сдвига будем различать 1) сравнения, в которых сравниваемый символ из T уже участвовал в сравнениях раньше, и 2) сравнения, в которых он участвует впервые. Пусть g_i — число сравнений первого типа в фазе i , а g'_i — второго типа. Общее число сравнений в алгоритме равно $\sum_{i=1}^q (g_i + g'_i)$, и наша цель — показать, что эта сумма имеет порядок $O(m)$.

Ясно, что $\sum_{i=1}^q g'_i \leq m$, так как каждый символ может сравниваться первый раз лишь единожды. Покажем, что для каждой фазы i выполняется неравенство $s_i \geq g_i/3$. Отсюда, так как $\sum_{i=1}^q s_i \leq m$ (поскольку полная длина всех сдвигов не превосходит m), будет следовать, что $\sum_{i=1}^q g_i \leq 3m$. Таким образом, общее число сравнений в алгоритме $\sum_{i=1}^q (g_i + g'_i) \leq 4m$.

Первоначальная лемма

Начнем с определения и леммы, которые интересны и сами по себе.

Определение. Для любой строки β обозначим через β^i строку, полученную конкатенацией i экземпляров β .

Лемма 3.2.1. *Пусть γ и δ — две непустые строки, такие что $\gamma\delta = \delta\gamma$. Тогда $\delta = \rho^i$ и $\gamma = \rho^j$ для некоторой строки ρ и натуральных i и j .*

Эта лемма гласит, что если строка не изменяется при циклическом сдвиге (значит, ее можно записать как $\gamma\delta$ и как $\delta\gamma$ для некоторых строк γ и δ), то γ и δ сами могут быть представлены как конкатенации некоторой строки ρ .

Например, пусть $\delta = abab$, $\gamma = ababab$ и $\delta\gamma = ababababab = \gamma\delta$. Тогда $\rho = ab$, $\delta = \rho^2$ и $\gamma = \rho^3$.

Доказательство. Лемма доказывается индукцией по $|\delta| + |\gamma|$. В качестве базы возьмем случай $|\delta| + |\gamma| = 2$, в котором $\delta = \gamma = \rho$ и $i = j = 1$. Теперь рассмотрим большие длины. Если $|\delta| = |\gamma|$, то снова $\delta = \gamma = \rho$ и $i = j = 1$. Допустим, что $|\delta| < |\gamma|$. Так как $\delta\gamma = \gamma\delta$ и $|\delta| < |\gamma|$, то δ должна быть префиксом γ , и значит, $\gamma = \delta\delta'$ для некоторой строки δ' . Подстановка этого равенства в $\delta\gamma = \gamma\delta$ дает

$\delta\delta\delta' = \delta\delta'\delta$. Убрав левое вхождение δ в обеих частях, получаем $\delta\delta' = \delta'\delta$. Однако $|\delta| + |\delta'| = |\gamma| < |\delta| + |\gamma|$, и по индукционному предположению $\delta = \rho^i$ и $\delta' = \rho^j$. Таким образом, $\gamma = \delta\delta' = \rho^k$, где $k = i + j$. \square

Определение. Стока α называется *полупериодической* с *периодом* P , если она состоит из непустого суффикса строки β (возможно, из целой β), за которой следует одна или больше копий β . Стока называется *периодической* с *периодом* β , если она состоит из двух или более *полных* копий β . Мы называем строку α *периодической*, если она периодическая с каким-либо периодом β .

Например, строка $bocababc$ — полупериодическая с периодом abc , но не периодическая. Стока $abababab$ — периодическая с периодом $abab$. Периодическая строка по определению также и полупериодическая. Заметим, что строка не может быть своим периодом, хотя период сам может быть периодическим. Например, строка $ababababab$ — периодическая с периодом $abab$ и с более коротким периодом ab . Иногда полезно альтернативное определение полупериодической строки.

Определение. Стока α называется *префиксно-полупериодической* с периодом γ , если α состоит из одной или более копий строки γ , за которыми следует непустой префикс строки γ (возможно, вся строка).

Мы используем термин “*префиксно-полупериодическая*”, чтобы отличить последнее определение от определения “*полупериодической*” строки, но следующая лемма (доказательство ее простое и оставляется на упражнения) показывает, что две формулировки на самом деле по-разному отражают одну и ту же структуру.

Лемма 3.2.2. *Стока α — полупериодическая с периодом β в том и только том случае, если она префиксно-полупериодическая с периодом той же длины, что и β .*

Например, строка $abaabaabaabaab$ — полупериодическая с периодом aab и префиксно-полупериодическая с периодом aba .

Следующую полезную лемму легко проверить, и ее доказательство типично для стиля рассуждений, используемого при работе с перекрывающимися совпадениями.

Лемма 3.2.3. *Предположим, что образец R встречается в тексте T , начиная с позиций p и $p' > p$, где $p' - p \leq |n/2|$. Тогда R — полупериодическая строка с периодом $p' - p$.*

Следующая лемма, называемая ОНД-леммой, дает мощное средство для работы с периодами в строках. Она не потребуется нам в обсуждении доказательства Коула, но естественно привести ее здесь. Доказательство леммы, а также ее применение см. в п. 16.17.5.

Лемма 3.2.4. *Пусть строка α — полупериодическая с двумя периодами длины p и q и $|\alpha| \geq p + q$. Тогда α — полупериодическая также с периодом, длина которого есть ОНД (общий наибольший делитель) p и q .*

Возврат к доказательству Коула

Напомним, что ключевой пункт доказательства — это неравенство $s_i \geq g_i/3$, где i — номер фазы. Как отмечалось выше, из него легко получается, что полное число сравнений не превосходит $4m$.

Рассмотрим i -ю фазу сравнения/сдвига, когда находится подстрока t_i из T , совпадающая с суффиксом P , а затем P сдвигается на s_i мест вправо. Если $s_i \geq (|t_i| + 1)/3$, то $s_i \geq g_i/3$, даже если все символы T , которые сравниваются в фазе i , участвовали в сравнениях раньше. Поэтому легко работать с фазами, в которых сдвиг “относительно” велик по сравнению с общим числом символов, проверенных во время фазы. Соответственно, в нескольких следующих леммах мы рассмотрим случай, когда сдвиг относительно мал (т. е. $s_i < (|t_i| + 1)/3$ или, что эквивалентно, $|t_i| + 1 > 3s_i$).

В этот момент нам понадобится еще одно обозначение. Пусть α — суффикс P длины s_i , а β — наименьшая подстрока, такая что $\alpha = \beta^l$ с некоторым целым l (допустимо, что $\beta = \alpha$ и $l = 1$). Пусть $\bar{P} = P[n - |t_i|..n]$ — суффикс P длины $|t_i| + 1$; это та часть P (включая несовпадение), которая была проверена в фазе i (рис. 3.3).



Рис. 3.3. Стока α имеет длину s_i , \bar{P} — длину $|t_i| + 1$

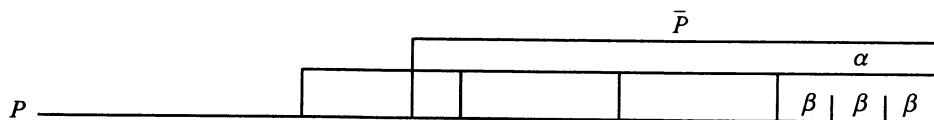


Рис. 3.4. Начиная справа, подстроки длины $|\alpha| = s_i$ помечаются в \bar{P}

Лемма 3.2.5. *Если $|t_i| + 1 > 3s_i$, то обе строки t_i и \bar{P} являются полупериодическими с периодом α и, следовательно, с периодом β .*

Доказательство. Начиная с правого конца \bar{P} , разметим подстроки длины s_i , пока слева не останется меньше чем s_i символов (рис. 3.4). Должно получиться не менее трех полных подстрок, поскольку $|\bar{P}| = |t_i| + 1 > 3s_i$. Фаза i кончается сдвигом P вправо на s_i позиций. Рассмотрим, как \bar{P} выровнено с T до и после этого сдвига (рис. 3.5). По определению s_i и α , строка α составляет часть сдвинутого \bar{P} справа от исходного \bar{P} . По правилу хорошего суффикса часть сдвинутого \bar{P} ниже t_i должна совпадать с частью несдвинутого \bar{P} ниже t_i , так что вторая размеченная подстрока от правого конца сдвинутого \bar{P} должна совпадать с первой подстрокой несдвинутого \bar{P} . Следовательно, они обе должны быть копиями строки α . Но вторая подстрока — одна и та же в обеих копиях \bar{P} . Продолжая это рассуждение, мы видим, что все размеченные подстроки длины s_i являются копиями строки α , а самая левая подстрока — суффикс строки α (если это не полная копия α). Таким образом, строка \bar{P} полупериодическая с периодом α . Крайние правые $|t_i|$ символов \bar{P} совпадают с t_i ,

так что и строка t_i полупериодическая с периодом α . Теперь $\alpha = \beta^l$, а \bar{P} и t_i должны также быть полупериодическими с периодом β . \square

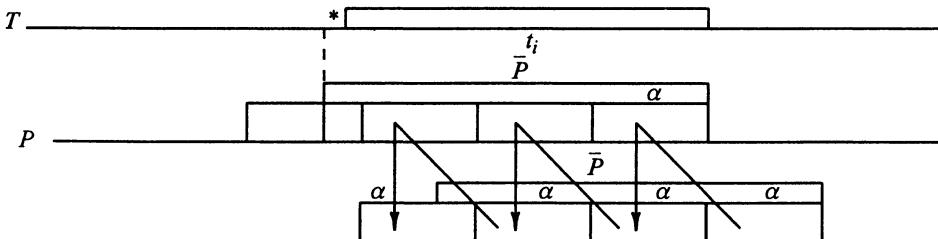


Рис. 3.5. Стрелки показывают равенства строк, описанные в доказательстве

Вспомним, что мы хотим найти границу для g_i — числа символов, сравниваемых на i -й фазе, которые уже сравнивались на предыдущих фазах. Все, кроме одного, символы, сравниваемые на фазе i , содержатся в t_i , и символ из t_i мог быть проверен раньше только в фазе, где образец P перекрывал t_i . Итак, для нахождения границы для g_i мы посмотрим, как P мог перекрывать t_i в предыдущих фазах.

Лемма 3.2.6. *Если $|t_i| + 1 > 3s_i$, то в любой фазе $h < i$ правый конец P не мог быть установлен напротив правого конца любой полной копии β в подстроке t_i из T .*

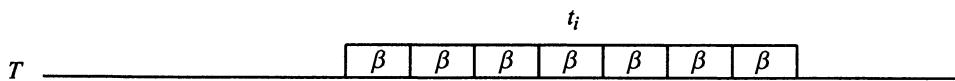


Рис. 3.6. Подстрока t_i строки T полупериодическая с периодом β

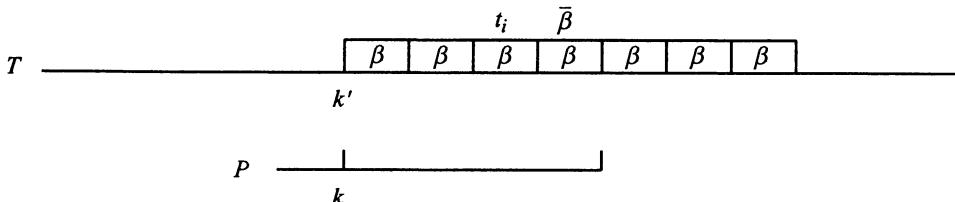


Рис. 3.7. Случай, когда правый конец P стоит вровень с правым концом $\bar{\beta}$ в фазе h . Здесь $q = 3$. Должно обнаружиться несовпадение $T(k')$ и $P(k)$

Доказательство. По лемме 3.2.5 строка t_i — полупериодическая с периодом β . На рис. 3.6 изображена строка t_i в виде конкатенации копий строки β . В фазе h правый конец P не может быть выровнен с правым концом t_i , поскольку таково расположение P и T в фазе $h > i$, а P должен между фазами h и i сдвигаться вправо. Итак, чтобы прийти к противоречию, предположим, что в фазе h правый конец P выровнен с правым концом другой полной копии β в t_i . Для определенности назовем

этую копию $\bar{\beta}$ и скажем, что ее правый конец находится в $q|\beta|$ позициях влево от правой части t_i , где $q \geq 1$ (рис. 3.7). Установим сначала, как должна завершиться фаза h , и используем это для доказательства леммы.

Пусть k' — позиция в T слева от t_i (так что $T(k')$ участвует в несовпадении, завершающем фазу i), а k — позиция в P напротив $T(k')$ в фазе h . Мы утверждаем, что в фазе h сравнение P и T найдет совпадения до левого конца t_i , а затем — несовпадение при сравнении $T(k')$ и $P(k)$. Предлагается следующая аргументация: строки \bar{P} и t_i полупериодические с периодом β , и в фазе h правый конец P выровнен с правым концом некоторого β . Поэтому в фазе h строки P и T определенно совпадут до левого конца строки t_i . Далее, \bar{P} — полупериодическая строка с периодом β , и в фазе h правый конец P находится на расстоянии $q|\beta|$ влево от правого конца t_i . Поэтому $\bar{P}(1) = \bar{P}(1 + |\beta|) = \dots = \bar{P}(1 + q|\beta|) = P(k)$. Но в фазе i несовпадение происходит при сравнении $T(k')$ с $\bar{P}(1)$, так что $P(k) = \bar{P}(1) \neq T(k')$. Следовательно, если в фазе h правый конец P выровнен по правому концу β , то фаза h должна закончиться несовпадением между $T(k')$ и $P(k)$. Этот факт будет ниже использован для доказательства леммы.*)

Посмотрим теперь, какие сдвиги P возможны в фазе h , и покажем, что любой такой сдвиг приводит к противоречию. Отсюда будет следовать, что предполагаемое расположение P и T в фазе h невозможно, что и докажет лемму.

Так как $h < i$, правый конец P не будет сдвигаться в фазе h за правый конец t_i ; поэтому после сдвига в фазе h символ из \bar{P} будет напротив символа $T(k')$ (того символа из T , который не совпадает в фазе i). Посмотрим, где будет правый конец P после сдвига в фазе h . Возможны два случая: 1) правый конец P будет напротив правого конца другой полной копии β (в t_i) или 2) правый конец P находится внутри полной копии β .

Случай 1. Если сдвиг в фазе h сопоставит правый конец P с правым концом полной копии β , то напротив $T(k')$ будет символ $P(k - r|\beta|)$ для какого-то r . Но так как строка \bar{P} полупериодическая с периодом β , $P(k)$ должно равняться $P(k - r|\beta|)$, что противоречит правилу хорошего суффикса.

Случай 2. Предположим, что сдвиг фазы h расположил P , поставив его правый конец вровень с каким-то символом внутри полной копии β . Это значит, что правый конец какой-то строки β в P встал рядом с символом внутри β . Более того, по правилу хорошего суффикса символы из сдвинутого P под $\bar{\beta}$ совпадают с $\bar{\beta}$ (рис. 3.8). Пусть $\gamma\delta$ — строка в сдвинутом P , расположенная напротив $\bar{\beta}$ в t_i , где γ — строка до конца β , а δ — остаток. Так как $\bar{\beta} = \beta$, то γ — суффикс β , δ — префикс β и $|\gamma| + |\delta| = |\bar{\beta}| = |\beta|$; таким образом, $\gamma\delta = \delta\gamma$. Однако по лемме 3.2.1 мы должны иметь $\beta = \rho^l$ и $t > 1$, а это противоречит предположению о том, что β — наименьшая строка, для которой $\alpha = \beta^l$ для некоторого l .

Начав с предположения о том, что в фазе h правый конец P выровнен с правым концом полной копии β , мы пришли к заключению, что в фазе h никакой сдвиг не возможен. Следовательно, предположение неверно, и лемма доказана. \square

*.) Позднее мы будем анализировать работу алгоритма Бойера–Мура в случае, когда P входит в T . Для этих целей отметим здесь, что если предположить, что фаза h заканчивается обнаружением вхождения P , то доказательство леммы 3.2.6 в этот момент завершается, так как устанавливается противоречие. Действительно, предположив, что правый конец P выровнен по правому концу β в фазе h , мы доказали, что фаза h завершается несовпадением, а это противоречит предположению, что h завершается обнаружением вхождения P в T . Итак, даже если фаза h завершается найденным вхождением P , правый конец P не мог бы быть выровнен с правым концом блоков β в фазе h .

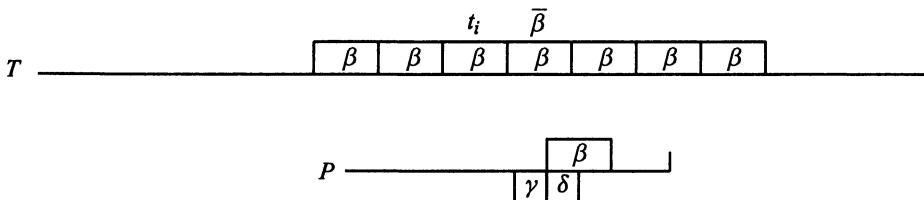


Рис. 3.8. Случай, когда правый конец P стоит вровень с символом внутри β . Тогда t_i должно иметь период меньше чем β , что противоречит определению β

Лемма 3.2.7. Если $|t_i| + 1 > 3s_i$, то в фазе $h < i$ образец P может совпасть с t_i из T не более чем в $|\beta| - 1$ символах.

Доказательство. Так как P не выровнен с концом какого-либо β в фазе h , то, если P совпадает с t_i в T не менее чем на $|\beta|$ символов, самые правые $|\beta|$ символов P совпадут со строкой, состоящей из суффикса (γ) и префикса (δ) строки β . Итак, мы получили бы опять $\beta = \gamma\delta = \delta\gamma$, и по лемме 3.2.1 это опять привело бы нас к противоречию с выбором β . \square

Отметим еще раз, что эта лемма верна и в предположении, что фаза h находит вхождение P . Действительно, нигде в доказательстве не предполагается, что фаза h кончается несовпадением, это предполагается только о фазе i . Мы используем это наблюдение в дальнейшем.

Лемма 3.2.8. Если $|t_i| + 1 > 3s_i$, то в фазе $h < i$ при выравнивании правого конца P по какому-то символу в t_i этим символом может быть один из $|\beta| - 1$ крайних левых или один из $|\beta|$ крайних правых символов t_i .

Доказательство. Предположим, что в фазе h правый конец P выровнен с символом t_i , отличным от $|\beta| - 1$ крайних левых и $|\beta|$ крайних правых символов. Скажем для определенности, что правый конец P выровнен с каким-то символом в копии β' строки β . Так как β' — не самая левая копия β , то правый конец P находится по меньшей мере в $|\beta|$ символах правее левого конца t_i , и значит, по лемме 3.2.7 в фазе h должно произойти несовпадение до того, как будет достигнут левый конец t_i . Пусть это несовпадение произошло в позиции k'' из T . После этого несовпадения P сдвигается вправо на некое расстояние, определяемое правилом хорошего суффикса. По лемме 3.2.6 сдвиг фазы h не может приблизить правый конец P к правому концу β' , и мы покажем также, что этот сдвиг не заведет конец P за правый конец β' .

Напомним, что правило хорошего суффикса сдвигает P (когда возможно) на наименьшее расстояние, при котором все символы T , совпавшие в фазе h , совпадут и в сдвинутом P , и при этом два символа P , выровненные по $T(k'')$ до и после сдвига, различны. Мы утверждаем, что эти условия выполняются, когда правый конец P выровняется с правым концом β' . Рассмотрим это выравнивание. Так как строка \bar{P} полуperiодическая с периодом β , то P и T совпадут по меньшей мере до левого конца t_i и должны поэтому совпасть и в позиции k'' из T . Следовательно, два символа P , выровненные с $T(k'')$ до и после сдвига, не могут быть равны. Так, если конец P выровнен с концом β' , то все символы T , которые совпали в фазе h , совпадут снова, и символы P , выровненные с $T(k'')$ до и после сдвига, будут различны. Значит, правило хорошего суффикса не сдвинет правый конец P за правый конец β' .

Поэтому, если правый конец P расположен внутри β' в фазе h , он должен оказаться внутри β' в фазе $h + 1$. Но h выбрано произвольно, так что сдвиг в фазе $h + 1$ также не должен задвигать правый конец P за β' . Итак, если правый конец P находится внутри β' в фазе h , он все время остается внутри. Но это невозможно, так как в фазе $i > h$ правый конец P встанет вровень с правым концом t_i , справа от β' . Следовательно, правый конец P не находится внутри β' , и лемма доказана. \square

Снова заметим, что лемма 3.2.8 верна, даже если предположить, что фаза h заканчивается обнаружением вхождения P в T . Действительно, в доказательстве используется только предположение, что фаза i заканчивается несовпадением, а от фазы h это не требуется. На самом деле, если фаза h находит вхождение P в T , то в доказательстве леммы используются только рассуждения из первых двух абзацев приведенного доказательства.

Теорема 3.2.1. *В предположении, что P не входит в T , для всех фаз i выполняется неравенство $s_i \geq g_i/3$.*

Доказательство. Это очевидно в случае $s_i \geq (|t_i| + 1)/3$, так что предположим, что $|t_i| + 1 > 3s_i$. По лемме 3.2.8 в любой фазе $h < i$ правый конец P лежит напротив либо одного из крайних слева $|\beta| - 1$ символов t_i , либо одного из крайних справа $|\beta|$ символов t_i (исключая крайний правый символ). По лемме 3.2.7 в фазе $h < i$ делается не больше $|\beta|$ сравнений. Следовательно, те символы, которые сравниваются в фазе i и могли сравниваться раньше, — это крайние слева $|\beta| - 1$ символов t_i , крайние справа $2|\beta|$ символов t_i и символ слева от t_i . Итак, $g_i \leq 3|\beta| = 3s_i$, когда $|t_i| + 1 > 3s_i$. В обоих случаях $s_i \geq g_i/3$. \square

Теорема 3.2.2. *В предположении, что P не встречается в T , число сравнений, выполняемых алгоритмом Бойера–Мура, в худшем случае не превосходит $4m$ [108].*

Доказательство. Как отмечалось ранее, $\sum_{i=1}^q g'_i \leq m$ и $\sum_{i=1}^q s'_i \leq m$, так что полное число сравнений, выполняемых алгоритмом, равно $\sum_{i=1}^q (g_i + g'_i) \leq (\sum_i 3s_i) + m \leq 4m$. \square

3.2.2. Случай, когда вхождения образца в текст существуют

Рассмотрим образец P , состоящий из n копий одного символа, и текст T — из m копий того же символа. Тогда P входит в T , начиная с любой позиции, кроме $n - 1$ последних, и число сравнений, выполняемых алгоритмом Бойера–Мура, равно $\Theta(mn)$. Доказанная в предыдущем пункте временная оценка $O(m)$ здесь не работает, так как она получена с помощью неравенства $g_i \leq 3s_i$ и предполагает, что фаза i кончается несовпадением. Итак, если P входит в T (и фазы не обязаны завершаться несовпадением), то, если мы хотим обеспечить линейное время, необходимо модифицировать алгоритм Бойера–Мура. Галил [168] дал первую такую модификацию. Ниже мы представим вариант его идеи.

Предлагаемый подход опирается на следующее наблюдение. Пусть в фазе i правый конец P выровнен по символу k строки T и P уже сравнилось с T влево до символа s из T . (Мы не уточняем, что именно обнаружилось в конце фазы — несовпадение или вхождение P в T .) Если сдвиг фазы i расположил P так, что левый конец P зашел за символ s из T , то в фазе $i + 1$ префикс P заведомо совпадает

с символами T вплоть до $T(k)$. Таким образом, в фазе $i + 1$ если сравнение справа налево дошло до позиции k из T , то алгоритм может обнаружить вхождение P даже без явных проверок символов влево от $T(k + 1)$. Эту модификацию алгоритма легко реализовать, и в оставшейся части этого параграфа мы предполагаем, что метод Бойера–Мура включает в себя это правило, которое мы называем *правилом Галила*.

Теорема 3.2.3. *При использовании правила Галила алгоритм Бойера–Мура никогда не делает больше $O(m)$ сравнений, независимо от того, как много вхождений P имеется в T .*

Доказательство. Разделим все фазы на те, которые находят вхождения P , и те, которые не находят. Пусть Q — множество фаз первого типа и d_i — число сравнений, выполненных в фазе i , если $i \in Q$. Тогда $\sum_{i \in Q} d_i + \sum_{i \notin Q} (|t_i| + 1)$ дает границу для общего числа сравнений, выполненных алгоритмом.

Величина $\sum_{i \notin Q} (|t_i| + 1)$ также имеет порядок $O(m)$. Чтобы убедиться в этом, вспомним, что леммы предыдущего пункта, где доказывалось неравенство $g_i \leq 3s_i$, использовали только предположение, что фаза i завершается несовпадением и $h < i$. В частности, анализ взаимного расположения P в фазах h и i , $h < i$, не требует, чтобы фаза h завершалась несовпадением. Эти доказательства охватывают оба случая завершения фазы h — и несовпадение, и обнаружение вхождения P . Следовательно, неравенство $g_i \leq 3s_i$ верно, когда фаза i завершается несовпадением, даже если в предыдущих фазах могло найтись вхождение.

Для фаз из Q мы также игнорируем случай $s_i \geq (n + 1)/3 \geq (d_i + 1)/3$, так как общее число сравнений в таких фазах не должно превосходить $\sum 3s_i \leq 3m$. Пусть в результате фазы i обнаружено вхождение P в T и сделан сдвиг меньше $n/3$. Из доказательства — в основном такого же, как для леммы 3.2.5, — следует, что строка P полупериодическая; пусть β обозначает кратчайший период в P . Следовательно, сдвиг в фазе i перемещает P вправо ровно на $|\beta|$ позиций, и, благодаря использованию правила Галила в алгоритме Бойера–Мура, ни один символ из T в фазе $i + 1$ не требует сравнения. Повторяя это рассуждение, видим, что если фаза $i + 1$ завершается находкой вхождения P , то P будет снова сдвинуто ровно на $|\beta|$ мест и ни в одном сравнении в фазе $i + 2$ не будет проверяться символ из T , участвовавший в проверке ранее. Этот цикл сдвигов P ровно на $|\beta|$ позиций с последующей идентификацией еще одного вхождения P и проверкой только $|\beta|$ новых символов T может повторяться много раз. Такая последовательность перекрывающихся вхождений P состоит из конкатенации копий β (каждая из копий P начинается ровно через $|\beta|$ мест вправо от предыдущего вхождения) и называется *полосой* (*run*). Вследствие использования правила Галила немедленно получается, что в любой отдельной полосе число сравнений, требуемых для распознания содержащихся в ней вхождений P , равно в точности длине полосы. Поэтому за все время работы алгоритма число сравнений для обнаружения этих вхождений равно $O(m)$. Если бы при обработке полосы не требовалось никаких других сравнений символов, анализ был бы полным. Однако дополнительные сравнения возможны, и мы должны их подсчитать.

Полоса заканчивается в некоторой фазе $k > i$, когда находится несовпадение (или алгоритм завершает работу). Может случиться, что символы из T в полосе будут проверены заново в фазах после k . Каждая такая фаза, перепроверяющая символы из полосы, завершается либо несовпадением, либо обнаружением вхождения P , которое перекрываетяется, хотя и не полностью, полосой, встретившейся ранее. Однако все сравнения в фазах, закончившихся несовпадением, уже подсчитаны (при подсчете фаз вне Q) и здесь игнорируются.

Пусть $k' > k > i$ — фаза, в которой найдено вхождение P , перекрывающееся, причем не полностью, полосой, встретившейся ранее. В качестве примера такого перекрытия возьмем $P = \alpha\alpha\alpha\alpha$ и текст T , содержащий подстроку $\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha\alpha$. Здесь полоса начинается в начале этой подстроки и кончается на ее двенадцатом символе, и с этого символа начинается перекрывающееся вхождение P (не входящее в полосу). Даже с правилом Галила символы в полосе будут проверяться заново в фазе k' , и, так как фаза k' не кончается несовпадением, эти сравнения должны быть подсчитаны.

В фазе k' если левый конец нового вхождения P в T начинается в левом конце копии β в полосе, то последовательные копии β продолжаются за правый конец полосы. Но никакое несовпадение невозможно в фазе k , так как в этой фазе образец выровнен на $|\beta|$ мест вправо от его позиции в фазе $k - 1$ (где было найдено вхождение P). Итак, в фазе k' левый конец нового P в T должен быть во внутреннем символе некоторой копии β . Но тогда, если P перекрывается с полосой больше, чем на $|\beta|$ символов, из леммы 3.2.1 следует, что строка β периодическая, а это противоречит выбору β . Следовательно, P может перекрывать полосу только по части крайней левой копии β . Далее, так как фаза k' завершается нахождением вхождения P , то образец сдвигается вправо на $s_{k'} = |\beta|$ позиций. Таким образом, любая фаза, которая найдет вхождение P с перекрытием более ранней полосы, сдвинет P на число позиций, большее чем размер перекрытия (и следовательно, число сравнений). Отсюда следует, что за все время работы алгоритма полное число таких дополнительных сравнений в перекрывающихся областях равно $O(m)$.

Теперь все сравнения подсчитаны, и следовательно, $\sum_{i \in Q} d_i = O(m)$, что завершает доказательство леммы. \square

3.2.3. Добавление правила плохого символа

Напомним, что при вычислении сдвига после несовпадения алгоритм Бойера–Мура делает наибольший из сдвигов, обеспечиваемых (расширенным) правилом плохого символа и (сильным) правилом хорошего суффикса. Кажется интуитивно ясным, что если временная граница равна $O(m)$, когда используется только правило хорошего суффикса, ее порядок должен быть тем же и при использовании обоих правил. Однако здесь можно ожидать некоторой “интерференции”, так что интуиция нуждается в доказательстве.

Теорема 3.2.4. *Когда оба правила сдвига используются совместно, оценка времени счета для модифицированного алгоритма Бойера–Мура в наихудшем случае остается $O(m)$.*

Доказательство. Анализируя алгоритм, использующий только правило суффикса, мы обращали внимание на сравнения, выполняемые в произвольной фазе i . В этой фазе правый конец P установлен около некоторого символа из T . Однако мы никогда не делали никаких предположений о том, как P добрался до этого места. Напротив, при произвольном расположении P в фазе, кончающейся несовпадением, мы устанавливали границы того, сколько символов, сравниваемых в этой фазе, могло сравниться в более ранних фазах. Следовательно, все леммы и весь анализ остаются справедливы, если P перемещается вправо произвольно на любом шаге алгоритма. (Расширенное) правило плохого символа лишь перемещает P дальше вправо, так что все леммы и рассуждения об оценке $O(m)$ остаются корректными и в этом случае. \square

3.3. Исходный препроцессинг по методу Кнута–Морриса–Пратта

3.3.1. Этот метод не использует основного препроцессинга

В п. 1.3 мы показали, как получить все значения sp_i из значений Z_i , вычисленных в основном препроцессинге P . Использование значений Z_i концептуально просто и открывает возможность для единообразной трактовки различных задач препроцессинга. Однако классический метод препроцессинга Кнута–Морриса–Пратта [278] не базируется на основном препроцессинге. Предложенный этими авторами подход хорошо известен и используется прямо или с модификацией в некоторых других методах (таких, как метод Ахо–Корасика, обсуждаемый ниже). По этим причинам серьезный студент, занимающийся строковыми алгоритмами, должен освоить и классический алгоритм препроцессинга в методе Кнута–Морриса–Пратта.

Этот алгоритм вычисляет $sp_i(P)$ для каждой позиции i от 2 до n (sp_1 равно нулю). Чтобы понять его работу, рассмотрим, как вычисляется sp_{k+1} в предположении, что sp_i известно для каждого $i \leq k$. Ситуация изображена на рис. 3.9, где строка α обозначает префикс P длины sp_k ; α' — самая длинная строка, которая появляется в P и как собственный префикс, и как подстрока, заканчивающаяся в позиции k . Пусть, для ясности, α' — копия α , заканчивающаяся в позиции k .



Рис. 3.9. Ситуация после нахождения sp_k

Пусть x обозначает символ $k + 1$ из P , а $\beta = \bar{\beta}x$ — это префикс P длины sp_{k+1} (т. е. префикс, который алгоритм будет дальше вычислять). Нахождение sp_{k+1} эквивалентно нахождению строки $\bar{\beta}$. Очевидно,

- ♦ $\bar{\beta}$ — наибольший собственный префикс $P[1..k]$, который совпадает с суффиксом $P[1..k]$ и за которым следует символ x в позиции $|\bar{\beta}| + 1$ строки P (рис. 3.10).

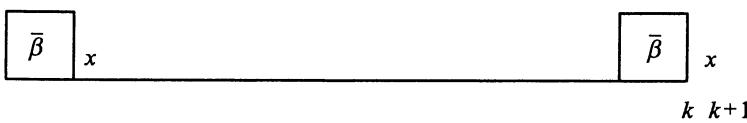


Рис. 3.10. sp_{k+1} найдено вычислением $\bar{\beta}$

Наша цель — найти sp_{k+1} или, что эквивалентно, $\bar{\beta}$.

3.3.2. Простой случай

Предположим, что сразу после α идет символ x (т.е. $P(sp_k + 1) = x$). Тогда строка αx является префиксом P и собственным префиксом $P[1..k + 1]$, а значит, $sp_{k+1} \geq |\alpha x| = sp_k + 1$. Можем ли мы теперь закончить наше вычисление sp_{k+1} заключением, что sp_{k+1} равно $sp_k + 1$? Или sp_{k+1} может быть строго больше чем $sp_k + 1$? Ответ на это дает следующая лемма.

Лемма 3.3.1. Для любого k имеем $sp_{k+1} \leq sp_k + 1$. При этом $sp_{k+1} = sp_k + 1$ в том и только том случае, если символ после α есть x . То есть $sp_{k+1} = sp_k + 1$ в том и только том случае, если $P(sp_k + 1) = P(k + 1)$.

Доказательство. Пусть $\beta = \bar{\beta}x$ обозначает префикс P длины sp_{k-1} . Это значит, что $\beta = \bar{\beta}x$ — наибольший собственный суффикс $P[1..k + 1]$, который является префиксом P . Если бы sp_{k+1} было строго больше чем $sp_k + 1 = |\alpha| + 1$, то $\bar{\beta}$ было бы префиксом P , который длиннее, чем α . Но $\bar{\beta}$ является и собственным суффиксом $P[1..k]$ (так как $\beta = \bar{\beta}x$ есть собственный суффикс $P[1..k + 1]$). Эти два факта противоречат определению sp_k (и выбору α). Следовательно, $sp_{k+1} \leq sp_k + 1$.

Теперь ясно, что $sp_{k+1} = sp_k + 1$, если символ справа от α есть x , так как αx будет тогда префиксом P , который совпадает также с собственным суффиксом $P[1..k + 1]$. И напротив, если $sp_{k+1} = sp_k + 1$, то символом после α должен быть x . \square

Лемма 3.3.1 распознает наибольшее значение, “претендующее” на sp_{k+1} , и определяет первоначальный подход к этому значению (и к строке β). Нам следует сначала проверить символ $P(sp_k + 1)$, ближайший справа от α . Если он равен $P(sp_k + 1)$, то мы заключаем, что $\bar{\beta}$ равно α , β равно αx и sp_{k+1} равно $sp_k + 1$. А что нам делать, если эти два символа не равны?

3.3.3. Общий случай

Если $P(k + 1) \neq P(sp_k + 1)$, то $sp_{k+1} < sp_k + 1$ (по лемме 3.3.1), так что $sp_{k+1} \leq sp_k$. Отсюда следует, что β должно быть префиксом α , а $\bar{\beta}$ — собственным префиксом α . Далее, подстрока $\beta = \bar{\beta}x$ заканчивается в позиции $k + 1$ и имеет длину не более sp_k , тогда как α' является подстрокой, кончающейся в позиции k и имеющей длину sp_k . Таким образом, $\bar{\beta}$ — суффикс α' , как показано на рис. 3.11. Но так как α' — копия α , то $\bar{\beta}$ является и суффиксом α .

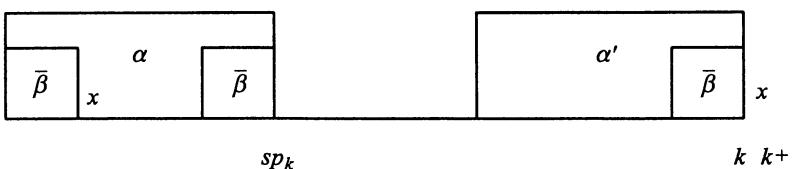


Рис. 3.11. Стока β должна быть суффиксом α

В итоге при $P(k + 1) \neq P(sp_k + 1)$ строка $\bar{\beta}$ является суффиксом α , а также собственным префиксом α , за которым следует символ x . Таким образом, при $P(k + 1) \neq P(sp_k + 1)$ строка $\bar{\beta}$ — это самый длинный собственный префикс α ,

совпадающий с суффиксом α и такой, что за ним следует символ x в позиции $|\bar{\beta}| + 1$ из P (см. рис. 3.11).

Однако, поскольку $\alpha = P[1..sp_k]$, мы можем сформулировать результат таким образом:

♦♦ $\bar{\beta}$ — это самый длинный собственный префикс $P[1..sp_k]$, совпадающий с суффиксом $P[1..k]$, за которым следует символ x в позиции $|\bar{\beta}| + 1$ из P .

Общая редукция

Утверждения ♦ и ♦♦ отличаются только заменой $P[1..k]$ на $P[1..sp_k]$, а в остальном совпадают. Следовательно, при $P(sp_k + 1) \neq P(k + 1)$ задача нахождения $\bar{\beta}$ приводится к той же исходной задаче, но с меньшей строкой $P[1..sp_k]$ вместо $P[1..k]$. Поэтому нам следует поступить, как раньше. То есть для нахождения $\bar{\beta}$ алгоритм должен найти самый длинный собственный префикс $P[1..sp_k]$, который совпадает с суффиксом $P[1..sp_k]$, и затем проверить, будет ли символ справа от этого префикса равен x . По определению sp_k искомый префикс кончается в символе sp_{sp_k} . Таким образом, если $P(sp_{sp_k} + 1) = x$, то мы нашли $\bar{\beta}$, а в противном случае снова пользуемся рекурсией, ограничивая поиск еще меньшими префиксами P . В результате либо найдется правильный префикс, либо мы дойдем до начала P . В последнем случае при $P(1) = P(k + 1)$ имеем $sp_{k+1} = 1$, а при нарушении условия $sp_{k+1} = 0$.

Препроцессинг полностью

Соединяя все части вместе, получаем алгоритм нахождения $\bar{\beta}$ и sp_{k+1} :

Как найти sp_{k+1}

```

 $x := P(k + 1);$ 
 $v := sp_k;$ 
 $while P(v + 1) \neq x \text{ and } v \neq 0 \text{ do}$ 
     $v := sp_v;$ 
 $if P(v + 1) = x \text{ then}$ 
     $sp_{k+1} := v + 1$ 
 $else$ 
     $sp_{k+1} := 0;$ 

```

См. пример на рис. 3.12.

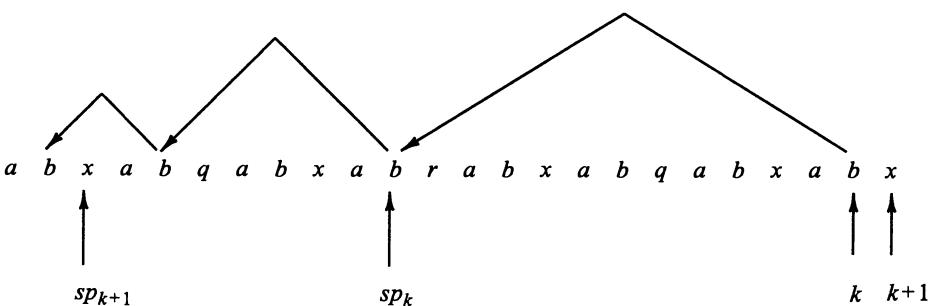


Рис. 3.12. Схема “прыгающего мяча” для исходного препроцессинга Кнута–Морриса–Пратта. Стрелки показывают последовательные присваивания переменной v .

Весь набор значений sp вычисляется следующим образом:

Алгоритм $SP(P)$

```

 $sp_1 := 0;$ 
for  $k := 1$  to  $n - 1$  do begin
     $x := P(k + 1);$ 
     $v := sp_k;$ 
    while  $P(v + 1) \neq x$  and  $v \neq 0$  do
         $v := sp_v;$ 
        if  $P(v + 1) = x$  then
             $sp_{k+1} := v + 1$ 
        else
             $sp_{k+1} := 0;$ 
end;

```

Теорема 3.3.1. Алгоритм SP находит все значения $sp_i(P)$ за время $O(n)$, где n — длина P .

Доказательство. Прежде всего отметим, что алгоритм состоит из двух вложенных циклов: цикла *for* и цикла *while*. Цикл *for* делает ровно $n - 1$ шагов, наращивая каждый раз значение k . Количество шагов в цикле *while* различно.

Работа алгоритма пропорциональна числу присваиваний переменной v . Рассмотрим те места, где выполняется такое присваивание, и проследим, как изменяется v в ходе работы алгоритма. Значение v присваивается один раз на каждом шаге цикла *for* и переменное число раз внутри цикла *while*, по одному присваиванию на каждом шаге. Следовательно, число присваиваний v равно $n - 1$ плюс число присваиваний внутри цикла *while*. Главный вопрос — сколько таких присваиваний.

Каждое присваивание v внутри цикла *while* должно уменьшать значение v , а в каждом из $n - 1$ присваиваний v в цикле *for* это значение либо увеличивается на единицу, либо сохраняется неизменным (равным нулю). Первоначально v равно нулю, так что оно может увеличиваться за все время работы алгоритма (внутри цикла *for*) не более $n - 1$ раз. Но так как v начинает с нуля и остается неотрицательным, то и полное число убываний ограничено сверху числом $n - 1$, число уменьшений не превосходит числа увеличений. Следовательно, v может изменяться внутри цикла *while* не более $n - 1$ раз, а полное число присваиваний v ограничивается сверху величиной $2(n - 1) = O(n)$, и теорема доказана. \square

3.3.4. Как вычислить оптимальные сдвиги

Значения sp'_i (более сильные) легко получить из значений sp_i за время $O(n)$ с помощью приведенного ниже алгоритма. В нем используется символ $P(n + 1)$, не существующий на самом деле и определенный как символ, отличный от любого другого символа в P .

Алгоритм $SP'(P)$

```

 $sp'_1 := 0;$ 
for  $i := 2$  to  $n$  do begin
     $v := sp_i;$ 

```

```

if  $P(v + 1) \neq P(i + 1)$  then
     $sp'_i := v$ 
else
     $sp'_i := sp_v;$ 
end;

```

Теорема 3.3.2. Алгоритм $SP'(P)$ корректно вычисляет все значения sp'_i за время $O(n)$.

Доказательство. Доказательство ведется индукцией по i . Так как $sp_1 = 0$ и $sp'_i \leq sp_i$ для всех i , то $sp'_1 = 0$, и алгоритм для $i = 1$ корректен. Теперь предположим, что значение sp'_i , найденное алгоритмом, корректно для всех $i < k$, и рассмотрим $i = k$. Если $P(sp_k + 1) \neq P(k + 1)$, то очевидно, что $sp'_k = sp_k$, так как префикс $P[1..k]$ длины sp_k удовлетворяет всем требованиям. Следовательно, в этом случае алгоритм находит sp'_k правильно.

Если $P(sp_k + 1) = P(k + 1)$, тогда $sp'_k < sp_k$, и, поскольку $P[1..sp_k]$ — суффикс $P[1..k]$, то sp'_k может быть истолковано как длина наибольшего собственного префикса $P[1..sp_k]$, который совпадает с суффиксом $P[1..sp_k]$, при том, что $P(k + 1) \neq P(sp'_k + 1)$. Но так как $P(k + 1) = P(sp_k + 1)$, условие можно переписать как $P(sp_k + 1) \neq P(sp'_k + 1)$. По индукционному предположению упомянутая длина уже вычислена как sp_{sp_k} . Итак, при $P(sp_k + 1) = P(k + 1)$ алгоритм правильно полагает sp'_k равным sp_{sp_k} .

Поскольку для каждой позиции трудоемкость алгоритма константная, полное время его работы имеет порядок $O(n)$. \square

Интересно сравнить этот классический метод вычисления sp и sp' с методом, использующим основной препроцессинг (т. е. Z -значения). В классическом методе сначала вычисляются (слабые) значения sp , а затем из них получаются искомые значения sp' , тогда как в основном препроцессинге порядок прямо противоположен.

3.4. Точный поиск набора образцов

Рассмотрим непосредственное (и важное) обобщение задачи точного поиска на случай множества образцов $\mathcal{P} = \{P_1, P_2, \dots, P_z\}$, когда требуется обнаружить все вхождения в текст T любого образца из \mathcal{P} . Это обобщение называется *множественной задачей точного поиска* (the exact set matching problem). Здесь n будет обозначать суммарную длину всех образцов в \mathcal{P} , а m , как и ранее, длину T . Тогда множественную задачу поиска можно решить за время $O(n + zm)$, применяя любой метод с линейным временем отдельно для каждого из z образцов.

Однако эту задачу можно решить быстрее, чем за $O(n + zm)$, а именно за время $O(n + m + k)$, где k — число вхождений в T образцов из \mathcal{P} . Первый метод с такой границей был предложен Ахо и Корасиком [9] *). В этом параграфе мы представим алгоритм Ахо–Корасика; некоторые из доказательств будут оставлены читателю. Столь же эффективный, но более робастный метод для множественной задачи поиска основывается на суффиксных деревьях, он будет рассмотрен в п. 7.2.

*) Имеется более свежее изложение метода Ахо–Корасика в [8], где алгоритм используется точно как “приемник”, решающий, обнаружилось или нет вхождение в T по меньшей мере одного образца из \mathcal{P} . Так как нам захочется явно найти все вхождения, эта версия алгоритма слишком ограничена для ее использования здесь.

Определение. Деревом ключей (keyword tree) для множества \mathcal{P} называется ориентированное дерево с корнем \mathcal{K} , удовлетворяющее трем условиям: 1) каждая дуга помечена ровно одним символом; 2) любые две дуги, выходящие из одной и той же вершины, имеют разные пометки; и 3) каждый образец P_i в \mathcal{P} отображается в некоторую вершину v из \mathcal{K} , такую что символы на пути из корня \mathcal{K} в v в точности составляют P_i , и каждый лист из \mathcal{K} соответствует какому-либо образцу из \mathcal{P} .

Например, на рис. 3.13 показано дерево ключей для множества образцов $\{potato, poetry, pottery, science, school\}$.

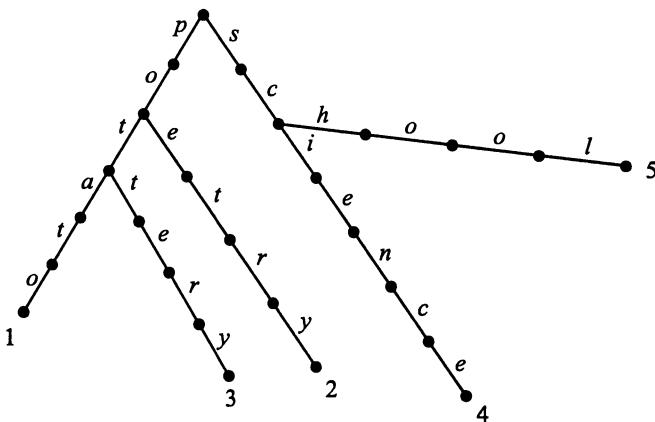


Рис. 3.13. Дерево ключей \mathcal{K} с пятью образцами

Ясно, что каждая вершина в дереве ключей соответствует префиксу какого-либо образца из \mathcal{P} и разные префиксы одного образца отображаются в разные вершины дерева.

В предположении, что размер алфавита конечен, легко построить дерево ключей для \mathcal{P} за время $O(n)$. Определим \mathcal{K}_i как (частичное) дерево ключей, кодирующее образцы P_1, \dots, P_i из набора \mathcal{P} . Дерево \mathcal{K}_1 состоит из простого пути с $|P_1|$ дугами, начинающегося в корне. Каждая дуга этого пути помечена символом из P_1 , и, если читать от корня, эти символы составят строку P_1 . Около конечной вершины этого пути написано число 1. Чтобы сделать \mathcal{K}_2 из \mathcal{K}_1 , найдем самый длинный путь из r , в котором символы совпадают по порядку с P_2 . То есть найдем наибольший префикс P_2 , который совпадает с символами какого-либо пути, идущего из корня. Этот путь кончается либо исчерпанием P_2 , либо тем, что в какой-нибудь вершине дерева v совпадение текста пути и образца заканчивается. В первом случае P_2 уже есть в дереве, и мы пишем номер 2 в вершине, где путь кончается. Во втором случае мы создаем первый путь, ведущий из v , помеченный оставшимися (несовпадающими) символами из P_2 , и пишем число 2 в конце этого пути. Пример этих двух возможностей показан на рис. 3.14.

В любом из этих двух случаев в \mathcal{K}_2 прибавится не больше одной ветвящейся вершины (вершины, у которой детей больше одного), и символы на двух дугах,

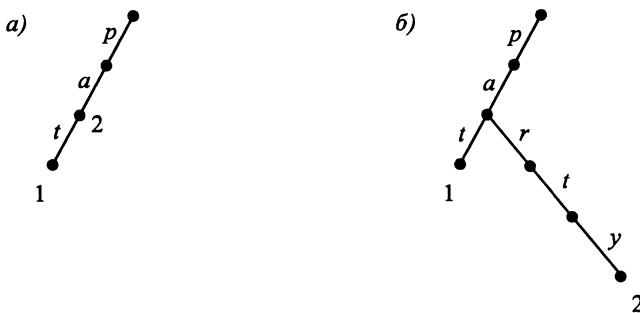


Рис. 3.14. Образец P_1 — это строка pat ;
 a — включение образца $P_2 = pa$; б — включение, когда $P_2 = party$

выходящих из этой ветвящейся вершины, будут различны. Мы увидим, что последнее свойство для любого дерева \mathcal{K}_i выполняется индуктивно. В том смысле, что в любой ветвящейся вершине v дерева \mathcal{K}_i все дуги, выходящие из v , имеют разные метки.

В общем, для создания \mathcal{K}_{i+1} из \mathcal{K}_i нужно стартовать из корня \mathcal{K}_i и двигаться так далеко, как возможно, по тому (единственному) пути в \mathcal{K}_i , на котором метки совпадают по порядку с символами из P_{i+1} . Этот путь единственен потому, что в каждой ветвящейся вершине v из \mathcal{K}_i все символы выходящих дуг различны. Если образец P_{i+1} исчерпался (полностью совпал), то вершине, где совпадение закончилось, приписывается номер $i + 1$. Если процесс пришел в вершину v , где дальнейшее совпадение невозможно и для очередного символа P_{i+1} подходящей дуги не нашлось, то создается новый путь из v , помечаемый оставшейся частью P_{i+1} , и конечная вершина этого пути получает номер $i + 1$.

Во время включения в дерево образца P_{i+1} работа для каждой вершины ограничивается константой, так как алфавит конечен и никакие две дуги, выходящие из вершины, не помечены одним и тем же символом. Следовательно, для любого i включение образца P_{i+1} в \mathcal{K}_i занимает времени $O(|P_{i+1}|)$, а создание всего дерева ключей — времени $O(n)$.

3.4.1. Наивное использование дерева ключей для множественного поиска

Поскольку никакие две дуги, выходящие из одной вершины, не помечены одинаковым символом, дерево ключей можно использовать для поиска всех вхождений в T образцов из \mathcal{P} . Для начала посмотрим, как искать вхождения образцов из \mathcal{P} , которые начинаются с первого символа строки T . Нужно следовать единственному пути в \mathcal{K} , который совпадает с префиксом T , так долго, как возможно. Если на пути найдется вершина, имеющая номер i , это значит, что строка P_i входит в T , начиная с позиции 1. Больше одной нумерованной вершины можно встретить при условии, что некоторые образцы из \mathcal{P} являются префиксами других образцов из \mathcal{P} .

В общем случае, чтобы найти все образцы, которые входят в T , мы стартуем с каждой позиции l строки T и идем единственным путем из корня r дерева \mathcal{K} , который совпадает с подстрокой T , начинающейся с символа позиции l . Нумерованные вершины вдоль пути отмечают образцы из P , которые входят в T , начиная

с позиции l . Для фиксированного l прокладка пути в \mathcal{K} требует времени, пропорционального минимуму из m и n , так что последовательно наращивая l от 1 до m и проходя через \mathcal{K} для каждого l , можно решить точную множественную задачу поиска за время $O(nm)$. Ниже мы уменьшим это время до $O(n + m + k)$, где k — число вхождений.

Задача о словаре

Этот простой алгоритм построения дерева ключей без каких-либо дальнейших улучшений эффективно решает специальный случай задачи множественного поиска, называемый *задачей о словаре*. В ней изначально известно и подготовлено множество строк (образующих словарь). Потом появляется последовательность отдельных строк; для каждой из них требуется определить, содержится ли она в словаре. В этом контексте вполне очевидно удобство дерева ключей. Строки словаря закодированы в дереве ключей \mathcal{K} , и, когда появляется отдельная строка, проход от корня в дереве \mathcal{K} определяет, есть ли указанная строка в словаре. В этом специальном случае точного множественного поиска необходимо дать ответ, совпадает ли весь текст T (отдельная строка) с какой-нибудь строкой в \mathcal{P} .

Вернемся теперь к общей проблеме множественного поиска, где устанавливается, какие строки из \mathcal{P} содержатся в тексте T .

3.4.2. Ускорение: обобщенный метод Кнута–Морриса–Пратта

Изложенный наивный подход к точной задаче множественного поиска аналогичен наивному поиску, который мы обсуждали до того, как ввели метод Кнута–Морриса–Пратта. Последовательное наращивание l на единицу и старт каждый раз от корня r аналогичен наивному точному поиску для отдельного образца, где после каждого несовпадения образец сдвигается только на одну позицию, и сравнения всегда начинаются с левого конца образца. Алгоритм Кнута–Морриса–Пратта улучшает этот наивный алгоритм сдвигом образца больше чем на одну позицию, когда это возможно, и отказом от сравнения символов слева от текущего символа в T . Алгоритм Ахо–Корасика использует улучшения того же типа, наращивая l больше чем на единицу и перепрыгивая через начальные части путей в \mathcal{K} , когда это возможно. Ключевая идея — такое обобщение функции sp_i (определенной на с. 47 для отдельного образца), которое позволяет оперировать с набором образцов. Имеется в виду непосредственное обобщение с одной только тонкостью, которая возникает, если какой-то образец в \mathcal{P} является собственной подстрокой другого образца в \mathcal{P} . Поэтому полезно (временно) сделать следующее предположение:

Предположение о подстроках. Никакой образец в \mathcal{P} не является собственной подстрокой любого другого образца в \mathcal{P} .

3.4.3. Функции неудач для дерева ключей

Определение. Каждая вершина v в \mathcal{K} помечена строкой, полученной конкатнацией символов на пути от корня \mathcal{K} до вершины v в порядке их появления. Для этой пометки используется обозначение $\mathcal{L}(v)$. Так что конкатенация символов пути от корня до v произносит строку $\mathcal{L}(v)$.

Например, на рис. 3.15 вершина, на которую указывает стрелка, помечена строкой $pott$.

Определение. Для любой вершины v дерева \mathcal{K} определим $lp(v)$ как длину наибольшего собственного суффикса строки $\mathcal{L}(v)$, которая является префиксом некоторого образца из \mathcal{P} .

Например, рассмотрим набор образцов $\mathcal{P} = \{potato, tattoo, theater, other\}$ и его дерево ключей, показанное на рис. 3.16. Пусть v — вершина, помеченная

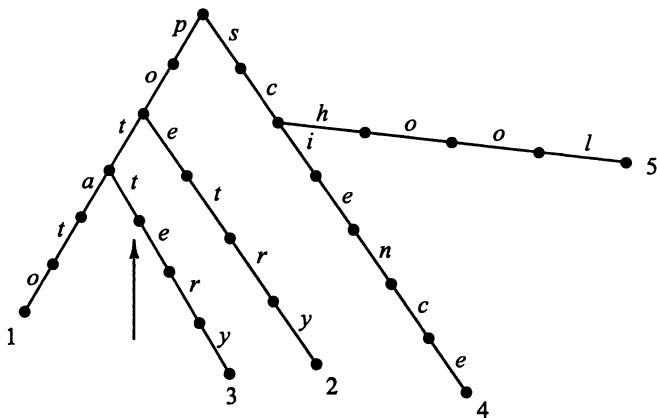


Рис. 3.15. Дерево ключей, показывающее пометку узла v

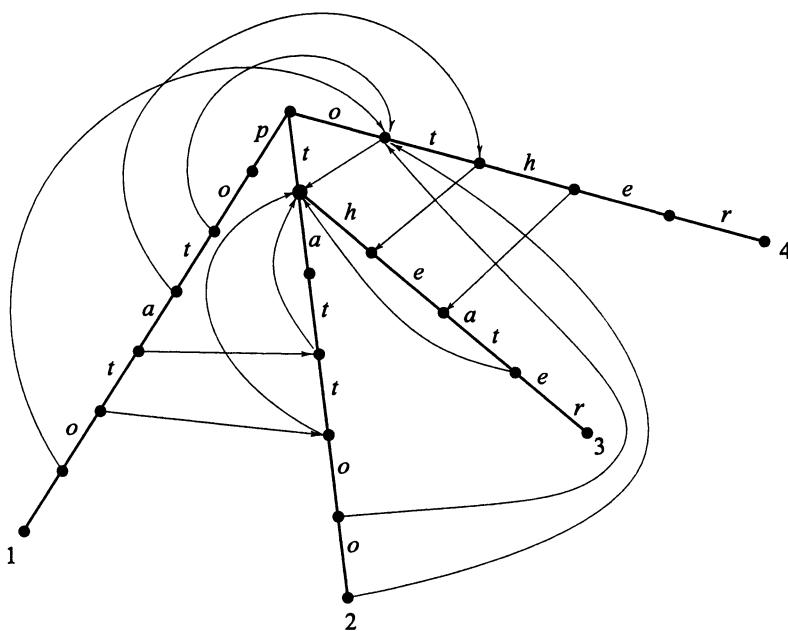


Рис. 3.16. Дерево ключей, показывающее связи неудач

строкой $potat$. Так как tat — это префикс $tattoo$ и наибольший собственный суффикс $potat$, который является префиксом какого-либо образца в \mathcal{P} , то $lp(v) = 3$.

Лемма 3.4.1. *Пусть α — суффикс строки $\mathcal{L}(v)$ длины $lp(v)$. Тогда существует единственная вершина в дереве ключей, помеченная строкой α .*

Доказательство. \mathcal{K} кодирует все образцы из \mathcal{P} , и по определению суффикс из $\mathcal{L}(v)$ длины $lp(v)$ есть префикс некоторого образца в \mathcal{P} . Так что должен быть путь из корня дерева \mathcal{K} , который произносит строку α . По построению \mathcal{K} никакие два пути не произносят одну и ту же строку, так что путь единственен, и лемма доказана. \square

Определение. Для вершины v дерева \mathcal{K} пусть n_v — единственная вершина в \mathcal{K} , помеченная суффиксом $\mathcal{L}(v)$ длины $lp(v)$. Если $lp(v) = 0$, то n_v — корень \mathcal{K} .

Определение. Назовем упорядоченную пару (v, n_v) связью неудачи.

На рис. 3.16 представлено дерево ключей для $\mathcal{P} = \{potato, tatoo, theater, other\}$. Связи неудач изображены как указатели от каждой вершины v к вершине n_v , где $lp(v) > 0$. Другие связи неудач указывают на корень и в рисунок не включены.

3.4.4. Связи неудач ускоряют поиск

Предположим, что мы знаем связь неудачи $v \mapsto n_v$ для каждой вершины v из \mathcal{K} . (Позднее мы покажем, как эффективно найти эти связи.) Каким образом связи неудач помогают ускорить поиск? Функция $v \mapsto n_v$ используется в алгоритме Ахо–Корасика (AC) аналогично функции $i \mapsto sp_i$ в алгоритме Кнута–Морриса–Пратта. Здесь l , как и раньше, будет отмечать стартовую позицию в T разыскиваемых образцов, а указатель c в T — “текущий символ”, который нужно сравнивать с символом из \mathcal{K} . Следующий алгоритм использует связи неудач для поиска вхождения в T образца из \mathcal{P} :

Алгоритм поиска AC

```

 $l := 1;$ 
 $c := 1;$ 
 $w :=$  корень  $\mathcal{K}$ 
repeat
    while есть дуга  $(w, w')$ , помеченная символом  $T(c)$  begin
        if  $w'$  занумерована образцом  $i$  then
            сообщить, что  $P_i$  встретилось в  $t$ , начиная с позиции  $l$ ;
             $w := w'; c := c + 1$ ;
        end;
         $w := n_w$ ;  $l := c - lp(w)$ ;
    until  $c > m$ ;
```

Чтобы понять назначение функции $v \mapsto n_v$, предположим, что мы прошли по дереву до вершины v и остановились (т. е. символ $T(c)$ не приписан никакой дуге, выходящей из v). Мы знаем, что строка $\mathcal{L}(v)$ встречается в T , начиная с позиции l и кончая позицией $c - 1$. По определению функции $v \mapsto n_v$ нам гарантировано, что строка $\mathcal{L}(n_v)$ совпадает со строкой $T[c - lp(v)..c - 1]$. Поэтому-то алгоритм и мог

проходит \mathcal{K} от корня до вершины n_v , и есть уверенность в совпадении всех символов на этом пути с символами T , начиная с позиции $c - lp(v)$. Итак, когда $lp(v) \geq 0$, позицию l можно увеличить до $c - lp(v)$, c можно оставить неизмененным и не нужно фактически делать сравнения на пути от корня до вершины n_v . На самом деле сравнения следует начать в вершине n_v , сопоставляя символ c из T с символами дуг, выходящих из n_v .

Например, рассмотрим текст $t = xxpotattooxx$ и дерево ключей, изображенное на рис. 3.16. Когда $l = 3$, текст совпадает со строкой *potat*, а в следующем символе — уже несовпадение. В этой точке $c = 8$, связь неудачи от вершины v , помеченной строкой *potat*, указывает на вершину n_v , помеченную *tat*, и $lp(v) = 3$. Таким образом, l возрастает до $5 = 8 - 3$, и следующее сравнение будет между символом $T(8)$ и символом t на дуге ниже *tat*.

В этом алгоритме, когда дальнейшие совпадения уже невозможны, l может возрасти больше чем на единицу, избавляя тем самым от повторной проверки символы из T слева от c , и при этом мы сохраняем уверенность, что каждое вхождение образца из \mathcal{P} , которое начинается с символа $c - lp(v)$ строки T , будет правильно распознано. Конечно (точно как у Кнута–Морриса–Пратта), мы должны обосновать, что никакое вхождение образца из \mathcal{P} не начинается между старым l и $c - lp(v)$ в T , и поэтому l можно увеличить до $c - lp(v)$ без пропуска вхождений. При сделанном предположении, что ни один образец из \mathcal{P} не является собственной подстрокой другого, аргументация почти идентична доказательству теоремы 2.3.2 в анализе метода Кнута–Морриса–Пратта и оставляется как упражнение.

При $lp(v) = 0$ нужно увеличить l до c и начать сравнение с корня \mathcal{K} . Остается только случай, когда несовпадение обнаружилось прямо в корне. Тогда c нужно увеличить на 1 и опять начать сравнение с корнем.

Поэтому использование функции $v \mapsto n_v$ заведомо ускоряет наивный поиск образца из \mathcal{P} . Но улучшает ли оно время счета для наихудшего случая? Аргументами того же типа, как при анализе времени поиска (а не препроцессинга) в методе Кнута–Морриса–Пратта (теорема 2.3.3), легко установить, что время поиска для метода Ахо–Корасика имеет порядок $O(m)$. Мы оставляем это как упражнение. Однако мы должны еще показать, как вычислить за линейное время функцию $v \mapsto n_v$.

3.4.5. Линейный препроцессинг для функции неудач

Напомним, что для любой вершины v из \mathcal{K} вершина n_v — единственная из \mathcal{K} , помеченная суффиксом $\mathcal{L}(v)$ длины $lp(v)$. Следующий алгоритм находит n_v для каждой вершины v из \mathcal{K} с полным временем $O(n)$. Ясно, что если v — корень r или отстоит от корня на один символ, то $n_v = r$. Предположим для некоторого k , что n_v вычислена для всех вершин, отстоящих от корня на не более k символов (дуг). Задача в том, чтобы вычислить n_v для вершины v , отстоящей от корня на $k + 1$ символов. Пусть v' — отец v в \mathcal{K} , а x — символ на дуге от v' к v , как показано на рис. 3.17.

Мы ищем вершину n_v и (неизвестную) строку $\mathcal{L}(n_v)$, помечающую путь от корня до этой вершины; мы знаем вершину $n_{v'}$, так как v' отстоит от r на k . Точно так же, как в изложении классической препроцессской обработки для метода Кнута–Морриса–Пратта, мы убеждаемся, что строка $\mathcal{L}(n_v)$ должна быть суффиксом $\mathcal{L}(n_{v'})$ (не обязательно собственным), за которым следует символ x . Таким образом, первое, что нужно проверить, — существует ли дуга $(n_{v'}, w')$, выходящая из вершины $n_{v'}$

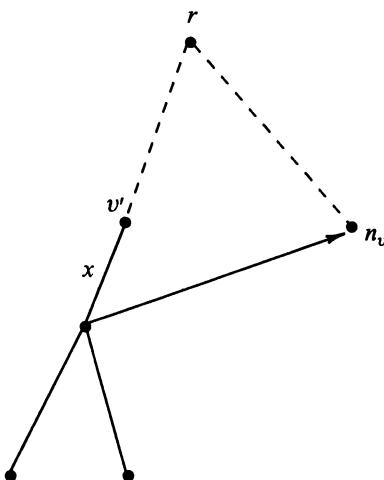


Рис. 3.17. Дерево ключей, используемое для вычисления функции неудач для узла v

и помеченная символом x . Если такая дуга существует, то $n_v = w'$, и все сделано. В противном случае $\mathcal{L}(n_v)$ есть собственный суффикс $\mathcal{L}(n_{v'})$, за которым следует x . Тогда проверим следующей вершину $n_{v'}$, чтобы проверить, не найдется ли выходящая из нее дуга, помеченная x . (Вершина $n_{v'}$ известна, так как $n_{v'}$ отстоит от корня не больше чем на k дуг.) Продолжая тем же способом с точно тем же обоснованием, как в классическом препроцессинге для метода Кнута–Морриса–Пратта, мы приходим к следующему алгоритму вычисления n_v для вершины v :

Алгоритм n_v

```

 $v'$  — отец  $v$  в  $\mathcal{K}$ ;
 $x$  — символ на дуге  $(v', v)$ ;
 $w := n_{v'}$ ;
while нет дуги, выходящей из  $w$ , помеченной  $x$ , и  $w \neq r$ 
    do  $w := n_w$ ;
if есть дуга  $(w, w')$ , выходящая из  $w$  и помеченная  $x$ , then
     $n_v := w'$ ;
else
     $n_v := r$ ;
```

Отметим важность предположения, что значение n_u уже известно для каждой вершины u , отстоящей не более чем на k символов от r .

Чтобы найти n_v для каждой вершины v , повторно применим предложенный выше алгоритм ко всем вершинам из \mathcal{K} . Будем обходить дерево в ширину, начиная от корня.

Теорема 3.4.1. Пусть n — полная длина всех образцов из \mathcal{P} . Полное время, затрачиваемое алгориттом n_v в применении его ко всем вершинам из \mathcal{K} , равно $O(n)$.

Доказательство. Аргументация этого доказательства прямо обобщает аргументацию из анализа времени при классическом препроцессинге в методе Кнута–

Морриса–Пратта. Рассмотрим единичный образец P из \mathcal{P} , имеющий длину t , и путь в \mathcal{K} для образца P . Будем анализировать время, затраченное алгоритмом для нахождения связей неудач для вершин этого пути так, как будто через проходимые им вершины не пролегают пути для других образцов из \mathcal{P} . Такой анализ начислит больше работы, чем ее выполняется на самом деле, но оценка все равно будет линейной.

Главное — проанализировать, как меняется $lp(v)$ при выполнении алгоритма для каждой очередной вершины v вниз по пути для P . Когда v отстоит от корня на одну дугу, $lp(v)$ равно 0. Пусть теперь v — произвольная вершина пути и v' — ее отец. Очевидно, $lp(v) \leq lp(v') + 1$, так что во всех исполнениях алгоритма n_v для вершины пути для P , $lp()$ увеличится не более чем на t . Посмотрим далее, насколько $lp()$ может уменьшиться. При вычислении n_v для любой вершины v , w стартует с $n_{v'}$, и таким образом начальная глубина вершины равна $lp(v')$. Однако затем глубина вершины w уменьшается при каждом новом присваивании w (внутри цикла *while*). Когда n_v будет найдено, $lp(v)$ становится равным текущей глубине w , так что если w присваивается k раз, то $lp(v) \leq lp(v') - k$, и $lp()$ уменьшилось по меньшей мере на k . Значит, $lp()$ не может быть отрицательным и вырастает во время вычислений по пути P не больше, чем на t . Отсюда следует, что во время всех вычислений для вершин на пути для P , число присваиваний внутри цикла *while* будет не больше t . Полное затраченное время пропорционально числу присваиваний внутри цикла, и следовательно, все связи неудач по пути для P находятся за время $O(t)$.

Применение этого рассуждения к каждому образцу из \mathcal{P} приводит к выводу, что все связи неудач получаются за время, пропорциональное сумме длин образцов в \mathcal{P} (т. е. за время $O(n)$). \square

3.4.6. Полный алгоритм Ахо–Корасика: освобождение от предположения о подстроках

До сих пор мы предполагали, что никакой образец из \mathcal{P} не является подстрокой другого. Снимем теперь это условие. Если один образец входит в другой и алгоритм АС (с. 83) использует то же дерево ключей, что и раньше, то l может достигнуть слишком больших значений. Рассмотрим случай, когда $\mathcal{P} = \{acatt, ca\}$ и $T = acatg$. В том виде, как он есть, алгоритм находит совпадение T по пути в \mathcal{K} , пока текущим не станет символ g . Путь заканчивается в вершине v с $\mathcal{L}(v) = acat$. Ни одна дуга, выходящая из v , не помечена g , и так как никакой собственный суффикс $acat$ не является префиксом $acatt$ или ac , то n_v есть корень \mathcal{K} . Таким образом, если вершина v останавливает алгоритм, то он возвращается к корню с g в качестве текущего символа, и l получает значение 5. Затем после одного дополнительного сравнения указатель текущего символа дойдет до $t + 1$, и алгоритм завершает работу, не найдя вхождения ca в T . Это случилось потому, что алгоритм сдвигает (увеличивает l) так, чтобы найти совпадение самого длинного суффикса $\mathcal{L}(v)$ с префиксом какого-либо образца в \mathcal{P} . Вложенные вхождения образцов в $\mathcal{L}(v)$, которые не являются суффиксами $\mathcal{L}(v)$, не влияют на увеличение l .

С такой трудностью легко справиться благодаря следующим наблюдениям, доказательства которых мы оставляем читателю.

Лемма 3.4.2. *Пусть в дереве ключей \mathcal{K} существует путь из связей неудач (возможно, пустой) от вершины v к вершине, занумерованной образцом i . Тогда в T*

должен обнаружиться образец P_i , который оканчивается в позиции c (текущий символ), как только во время фазы поиска алгоритма Ахо–Корасика будет достигнута вершина v .

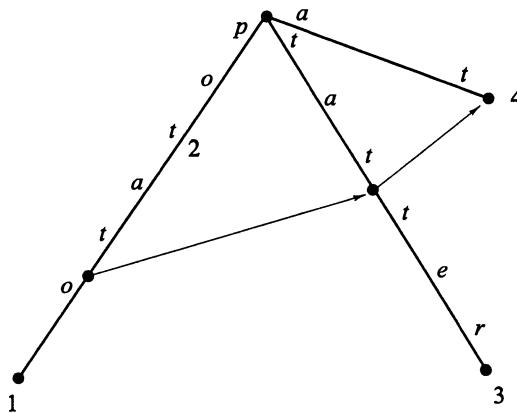


Рис. 3.18. Дерево ключей с путем от *rotat* до *at* через *tat*

Например, на рис. 3.18 изображено дерево ключей для $\mathcal{P} = \{\text{potato}, \text{pot}, \text{tatter}, \text{at}\}$ с некоторыми из связей неудач. Эти связи образуют путь от вершины v с пометкой *potat* к нумерованной вершине с пометкой *at*. Если проход по \mathcal{K} достигает v , то T наверняка содержит образцы *tat* и *at*, заканчивающиеся в текущем символе c .

И напротив:

Лемма 3.4.3. Пусть в ходе работы алгоритма достигнута вершина v . Тогда образец P_i появляется в T , заканчиваясь в позиции c , только если v имеет номер i или существует путь из связей неудач из v в вершину с номером i .

Итак, полный алгоритм поиска таков.

Алгоритм полного АС-поиска

```

 $l := 1;$ 
 $c := 1;$ 
 $w := \text{корень};$ 
repeat
    while существует дуга  $(w, w')$  с пометкой  $T(c)$  begin
        if  $w'$  занумерована образцом  $i$  или существует путь из связей неудач
            из  $w'$  в вершину с номером  $i$ 
            then сообщить о вхождении  $P_i$  в  $T$ , начиная с позиции  $c$ ;
         $w := w'; c := c + 1;$ 
    end;
     $w := n_w; l := c - lp(w);$ 
until  $c > n$ ;
```

Реализация

Леммы 3.4.2 и 3.4.3 указывают на высоком уровне, как найти все вхождения образцов в текст, однако нужно уточнить детали реализации. Наша цель — научиться строить

дерево ключей, определять функцию $v \mapsto n_v$ и иметь возможность выполнять алгоритм полного АС поиска за время $O(m + k)$. Нам понадобится дополнительный указатель в каждой вершине \mathcal{K} , который мы назовем *связью выхода* (output link).

Связь выхода (если она существует) в вершине v указывает на нумерованную вершину (вершину, соответствующую концу образца из \mathcal{P}), отличную от v и достижимую из v за наименьшее число связей неудачи. Связи выхода можно получить за время $O(n)$ при выполнении препроцессинга для n_v . Когда определено значение n_v , возможная связь выхода из вершины v определяется следующим образом. Если n_v — нумерованная вершина, то связь выхода из v указывает на n_v ; если n_v не занумерована, но имеет связь выхода на вершину w , то связь выхода из v указывает на w ; в противном случае v не имеет связи выхода. Здесь видно, что связь выхода может указывать только на нумерованную вершину и путь из связей выхода от любой вершины v проходит через все нумерованные вершины, достижимые из v путем, составленным из связей неудач. Например, на рис. 3.18 вершины для *tat* и *potat* получат свои связи выхода на вершину для *at*. Работа по добавлению связей выхода требует лишь константного времени на вершину, так что полное время для алгоритма n_v остается $O(n)$.

Располагая связями выхода, мы можем определить все вхождения в T образцов из \mathcal{P} за время $O(m + k)$. Как раньше, поскольку нумерованная вершина встречается во время полного АС-поиска, вхождение распознается и регистрируется. Вдобавок, если встречается вершина v , у которой есть связь выхода, алгоритм должен пройти путь связей выхода из v , регистрируя вхождение, кончающееся в позиции c из T для каждой связи этого пути. Когда прохождение по этому пути достигнет вершины без связи выхода, мы возвращаемся вдоль пути к вершине v и продолжаем выполнять алгоритм полного АС-поиска. Так как во время этого прохода не происходит сравнений символов, в обеих фазах, создания и поиска, число сравнений по-прежнему имеет границу $O(n + m)$. Далее, хотя число проходов по связям выхода может превысить линейную границу, каждый проход связи выхода обнаруживает вхождение образца, так что полное время алгоритма имеет оценку $O(n + m + k)$, где k — полное число вхождений. В результате мы имеем:

Теорема 3.4.2. *Если \mathcal{P} — набор образцов с полной длиной n и полная длина текста T равна m , то все вхождения в T образцов из \mathcal{P} можно найти за время $O(n)$ на препроцессинг плюс $O(m + k)$ на поиск k вхождений. Это верно даже без предположения о подстроках.*

В дальнейшем (п. 6.5) мы обсудим другие аспекты реализации, которые влияют на практическое осуществление как метода Ахо–Корасика, так и метода суффиксного дерева.

3.5. Три приложения точного множественного поиска

3.5.1. Сравнение с ДНК или библиотекой идентифицированных белков

В молекулярной биологии есть ряд прикладных направлений, при развитии которых созданы относительно устойчивые библиотеки интересных или распознанных подстрок ДНК или белков. Наша первая важная иллюстрация будет связана

с локализациями, маркованными последовательностями (sequence-tagged sites — STS) и экспрессируемыми маркерными последовательностями (expressed sequence tags — EST).

Маркованные локализации

Концепция маркованных локализаций (STS) — один из наиболее мощных побочных продуктов реализации международного проекта “Геном человека” (Human Genome Project) [111, 234, 399]. Не углубляясь в биологические подробности, можно на интуитивном уровне представить STS как строку ДНК длиной 200–300 нуклеотидов, у которой правый и левый концы, длиной 20–30 нуклеотидов каждый, встречаются во всем геноме только единожды [111, 317]. Таким образом, каждая STS может появиться в изучаемой ДНК лишь один раз. Хотя это определение не вполне корректно, для наших задач оно вполне подходит. Сначала целью проекта “Геном человека” были отбор и картирование (поиск в геноме) такого набора STS, чтобы любая подстрока генома длины 100 000 и более содержала по меньшей мере одну из этих STS. Уточненная цель — сделать карту, содержащую EST, под которыми понимают STS, соответствующие генам, а не их междугенным участкам ДНК. EST получаются из иРНК и кДНК (подробности о кДНК см. в п. 11.8.3) и типично отражают части белковых кодов в генной последовательности.

Имея карту STS, можно располагать на ней любую достаточно длинную строку неизвестных, но секвенированных ДНК — здесь задачей как раз и является то, какие STS содержатся в этой неизвестной ДНК. Таким образом, при наличии STS нахождение на карте места для неизвестной секвенированной ДНК становится строковой задачей — точного множественного поиска. STS и EST обеспечиваются компьютеризированным набором справочников, к которым могут обращаться при идентификации новых последовательностей ДНК. В настоящее время сотни тысяч STS и десятки тысяч EST уже найдены и помещены в компьютерные базы данных [234]. Отметим, что общая длина всех STS и EST очень велика по сравнению с типичным размером новых отрезков ДНК. Следовательно, дерево ключей и метод Ахо–Корасика (с временем поиска, пропорциональным длине безымянной ДНК) могут прямо использоваться в этой задаче, так как позволяют очень быстро идентифицировать STS или EST, встречающиеся в секвенируемой ДНК.

Конечно, при таком подходе могут случаться и ошибки как в карте STS, так и в секвенируемой ДНК, заставляющие усомниться в нем (см. обсуждение карт STS в п. 16.5). Но в рассматриваемом методе доля ошибок должна составлять малый процент по сравнению с длиной STS, а это в дальнейшем откроет дорогу для более изощренных точных (и неточных) методов поиска совпадений. Некоторые из них мы опишем в пп. 7.8.3, 9.4 и 12.2.

Еще одно родственное приложение исходит из предложенного метода “ВАС–РАС” [442] для секвенирования человеческого генома (см. с. 506). В этом методе сначала надо получить и ввести в компьютер 600 000 строк (образцов) длиной по 500 символов. Затем тысячи раз нужно будет просмотреть вхождения каждого из этих 600 000 образцов в текстовые строки длиной 150 000. Отметим, что полная длина этих образцов составляет 300 миллионов символов — это в две тысячи раз больше обычного текста, в котором идет поиск.

3.5.2. Точный поиск с джокером

В качестве приложения метода точного множественного поиска рассмотрим задачу точного поиска для одного образца, но немного усложним ее. Модификация заключается в введении специального символа ϕ , именуемого *джокером* (wild card), который “совпадает” с любым символом. По заданному содержащему шаблоны образцу P мы хотим найти все вхождения P в текст T . Например, образец $ab\phi\phi c\phi$ встречается дважды в тексте $hab\phi\phi cbabab\phi\phi$. Отметим, что в этом варианте задачи символ ϕ не входит в T , и каждый джокер соответствует одному символу, а не подстроке неопределенной длины.

Проблему совпадения с шаблонами не надо долго обосновывать, так как придумать правдоподобные случаи, когда образец содержит джокеры, нетрудно. Один очень важный случай, где встречаются джокеры, — это *факторы транскрипции* ДНК. Фактор транскрипции — белок, который закрепляется в определенных местах ДНК и регулирует, усиливая или подавляя, транскрипцию ДНК в РНК. Так регулируется синтез белка, для которого, собственно, и предназначены коды ДНК. Это научное направление бурно развилось за последнее десятилетие; сейчас известны многие факторы транскрипции, их можно разделить на семейства, характеризуемые специфическими подстроками с джокерами. Например, *цинковый палец* — это общий фактор транскрипции, записываемый так:

$$\text{CYS}\phi\phi\text{CYS}\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\text{HIS}\phi\phi\text{HIS},$$

где CYS — аминокислота цистein, а HIS — аминокислота гистидин. Другой важный фактор транскрипции — *лейциновая застежка*, который включает в себя от четырех до семи остатков лейцина, разделенных группами по шесть джокеров аминокислот.

Если число разрешенных джокеров не ограничено, то неизвестно, можно ли решить задачу за линейное время. Однако если число джокеров ограничено фиксированной постоянной (не зависящей от размера P), то следующий метод, основанный на точном множественном поиске образцов, работает за линейное время.

Точное совпадение с джокерами

0. Пусть C — вектор длины $|T|$, инициализированный нулями.
1. Пусть $\mathcal{P} = \{P_1, P_2, \dots, P_k\}$ — набор максимальных подстрок P , которые не содержат джокеров. Пусть l_1, l_2, \dots, l_k — начальные позиции этих подстрок в P .
(Например, если $P = ab\phi\phi c\phi ab\phi\phi$, то $\mathcal{P} = \{ab, c, ab\}$ и $l_1 = 1, l_2 = 5, l_3 = 7$.)
2. Используя алгоритм Ахо–Корасика (или метод суффиксного дерева, который мы опишем в дальнейшем), найти для каждой строки P_i из \mathcal{P} начальные позиции всех вхождений P_i в текст T . Для каждого такого начала j строки P_i в T увеличить счетчик в ячейке $j - l_i + 1$ вектора C на единицу.
(Например, если при поиске второго вхождения строки ab оно нашлось в T , начиная с позиции 18, то на единицу увеличивается содержимое ячейки 12 вектора C .)
3. Просмотреть вектор C в поисках ячеек со значением k . Вхождение P в T , начинающееся с позиции p , имеется в том и только том случае, если $C(p) = k$.

Корректность и сложность метода

Корректность. Ясно, что вхождение P в T , начинающееся с позиции p , существует в том и только том случае, если для каждого i подобразец $P_i \in \mathcal{P}$ находится в позиции $j = p + l_i - 1$ текста T . Приведенный метод использует обращение этой идеи. Если вхождение образца $P_i \in \mathcal{P}$ найдено, начиная с позиции j из T , а образец P_i начинается в позиции l_i в P , то это дает одно “свидетельство”, что P входит в T , начиная с позиции $p = j - l_i + 1$. Следовательно, P входит в T , начиная с p в том и только том случае, если такие показания для позиции p получены от всех k строк \mathcal{P} . Алгоритм считает в позиции p количество показаний в пользу вхождения P , начиная с p . Это корректно определяет, есть ли такое вхождение, так как каждая строка из \mathcal{P} может вызвать приращение не больше чем на один в ячейке p вектора C .

Сложность. Время, требуемое в алгоритме Ахо–Корасика на построение дерева ключей для \mathcal{P} , равно $O(n)$. Время поиска вхождений в T образцов из \mathcal{P} равно $O(m + z)$, где $|T| = m$ и z — число вхождений. Мы считаем каждый образец в \mathcal{P} отличным от других, даже если есть кратные вхождения. Поэтому когда обнаруживается вхождение образца из \mathcal{P} в T , то значение увеличивается только в одной ячейке C ; более того, значение в ячейке увеличивается не более k раз. Таким образом, z не превосходит km , и алгоритм работает за время $O(km)$. Хотя число используемых сравнений символов имеет порядок $O(m)$, порядок km не обязан быть $O(m)$, и следовательно, число изменений в C может расти быстрее, чем $O(m)$, приводя к нелинейной оценке $O(km)$. Однако если k полагается ограниченным (независимо от $|P|$), то метод работает за линейное время. В результате:

Теорема 3.5.1. *Если число джокеров в образце P ограничено константой, то задача точного поиска с джокерами в образце может быть решена за время $O(n + m)$.*

Ниже, в п. 9.3, мы вернемся к проблеме джокеров, когда они будут встречаться и в образце, и в тексте, или и там и там.

3.5.3. Двумерное точное совпадение

Второе классическое применение точного множественного поиска встречается в обобщении строкового поиска на двумерный точный поиск. Пусть задана прямоугольная дигитализированная картинка T , в которой каждая точка задается числом, описывающим цвет и яркость. Задана также меньшая прямоугольная картинка P , также дигитализированная, и требуется найти все вхождения (возможно, перекрывающиеся) меньшей картинки в большую. Мы предполагаем, что края большего и меньшего прямоугольников параллельны. Это и называется двумерным обобщением задачи точного строчечного поиска.

Надо признаться, что эта задача немного надуманная. В отличие от одномерной задачи точного поиска, которая действительно встречается в многочисленных практических приложениях, безупречные приложения двумерного точного поиска найти трудно. Неточный двумерный поиск, допускающий некоторое число ошибок, — такая задача реалистичнее, но ее решение требует более сложной техники, вроде той, которую мы будем рассматривать в части III. Так что сейчас мы взглянем на двумерный точный поиск как на иллюстрацию возможного использования точного множественного поиска в более сложных постановках задачи и как на введение в более

реалистичные двумерные задачи. Представленный ниже метод следует основному подходу, предложенному в [44, 66]. С того времени было предложено много разных улучшений исходного метода. Однако, поскольку поставленная задача не вполне реалистична, мы не будем их обсуждать. Более сложную трактовку двумерного поиска см. в [22, 169].

Пусть m — полное число точек в T , n — число точек в P , а n' — число строк в P . Так же, как при точном строчечном поиске, мы хотим найти меньшую картинку в большей за время $O(n + m)$, тогда как время очевидного подхода — $O(nm)$. Сейчас мы предполагаем, что все строки P различны; позднее мы это предположение снимем.

Метод разделен на две фазы. В первой фазе ищутся все вхождения каждой строки P в строках T . Для этого в конце каждой строки из T добавляется специальный маркер (символ, не принадлежащий алфавиту) и все строки соединяются в одну текстовую строку T' длины $O(m)$. Затем, трактуя каждую строку P как отдельный образец, используем алгоритм Ахо–Корасика для поиска всех вхождений в T' каждой строки из P . Так как картинка P прямоугольна, все строки имеют одну и ту же ширину, ни одна строка не входит в другую как собственная подстрока, и мы можем использовать более простой вариант Ахо–Корасика, изложенный в п. 3.4.2. Итак, первая фаза находит все вхождения полных строк P в полные строки T за время $O(n + m)$.

Когда обнаруживается вхождение строки i из P , начинающееся в позиции (p, q) из T , запишем число i в позицию (p, q) другого массива M , той же размерности, что T . Так как все строки P предполагаются различными и картинка P прямоугольна, в каждую ячейку M будет вписано не больше одного числа.

Во второй фазе каждый столбец M сканируется в поисках вхождения строки 1, 2, …, n' в последовательных ячейках одного столбца. Например, если эта строка найдена в столбце 6, начиная со строки 12 и кончая строкой $n' + 11$, то P входит в T с левым верхним углом в позиции (6, 12). Фаза два может быть реализована за время $O(n' + m) = O(n + m)$ применением любого алгоритма точного поиска с линейным временем к каждой позиции M .

Получаем решение за время $O(n + m)$ для двумерной множественной задачи поиска. Отметим сходство между этим решением и решением задачи точного поиска образца с джокерами, рассмотренной в предыдущем пункте. Различие будет обсуждаться в упражнениях.

Теперь предположим, что среди строк в P есть совпадающие. Тогда сначала найдем все одинаковые строки и дадим им общую пометку (это легко сделать при создании дерева ключей для строк-образцов). Например, если строки 3, 6 и 10 одинаковы, мы могли бы дать им всем метку 3. То же самое делается и для других совпадающих строк. Тогда в первой фазе следует обеспечить поиск только вхождения строки 3, а строки 6 и 10 — не нужно. В результате в ячейку массива M будет записано не более одного числа. В фазе 2 нужно искать в столбцах M не строку 1, 2, 3, …, n' , а строку, в которой 6 и 10 заменены на 3 и т.д. Легко убедиться, что этот подход корректен и занимает время $O(n + m)$. В итоге:

Теорема 3.5.2. *Если T и P — прямоугольные изображения, соответственно, из m и n ячеек, то все точные вхождения P в T могут быть найдены за время $O(n + m)$, улучшая тем самым наивный метод с его временем $O(nm)$.*

3.6. Поиск образца, заданного регулярным выражением

Регулярное выражение — это способ описать множество близких строк, которое иногда называют *образцом**). Многие важные совокупности подстрок (образцов), найденные в биопоследовательностях, особенно в белках, могут быть описаны как регулярные выражения, и было создано несколько баз данных, поддерживающих работу с такими образцами. База данных PROSITE, созданная Эймосом Бейрохом [41, 42], — основная база данных с регулярными выражениями для важнейших образцов в белках (дальнейшие подробности о PROSITE см. в п. 15.8).

В этом параграфе мы рассмотрим задачу нахождения в текстовой строке подстрок, совпадающих с одной из строк, описываемых данным регулярным выражением. Такие сравнения проводятся утилитой *grep* в Unix. Несколько специальных программ было создано специально для сравнений с регулярными выражениями в биологических последовательностях [279, 416, 422].

Полезно начать с простого примера. Формальное определение регулярного выражения будет дано позже. Следующее выражение из PROSITE представляет совокупность подстрок, некоторые из которых появляются в конкретном семействе граниновых белков:

$$[ED] - [EN] - L - [SAN] - x - x - [DE] - x - E - L.$$

Всякая строка, описываемая этим регулярным выражением, имеет десять позиций, которые разделены черточками. Каждая прописная буква задает одну аминокислоту, а группа аминокислот, заключенная в квадратные скобки, означает, что должна быть выбрана одна из этих аминокислот. Строчное *x* показывает, что в этой позиции может стоять любая из двадцати аминокислот белкового алфавита. Это регулярное выражение описывает 192 000 аминокислотных строк, но только некоторые из них действительно встречаются в каких-либо известных белках. Например, строка *ENLSSEDEEL*, задаваемая этим регулярным выражением, найдена в граниновых белках человека.

3.6.1. Формальные определения

Дадим формальное рекурсивное определение регулярного выражения, образованного из алфавита Σ . Для простоты и вопреки приведенному примеру предположим, что алфавит Σ не содержит символов из следующего списка: *, +, (,), ϵ .

Определение. Простой символ из Σ есть регулярное выражение. Символ ϵ есть регулярное выражение. Регулярное выражение, за которым следует еще одно регулярное выражение, есть регулярное выражение. Два регулярных выражения, разделенных символом +, образуют регулярное выражение. Регулярное выражение, заключенное в круглые скобки, есть регулярное выражение. Регулярное выражение, заключенное в круглые скобки, за которыми записан символ *, есть регулярное выражение. Символ * называется замыканием Клини.

* Заметим, что в контексте регулярных выражений смысл слова “образец” отличается от предыдущей трактовки, основной для этой книги.

Эти рекурсивные правила просты для понимания, но все же нуждаются в некотором объяснении. Символ ϵ представляет пустую строку (т. е. строку длины нуль). Если R — заключенное в скобки регулярное выражение, то R^* означает, что выражение R может быть повторено любое число раз (даже нуль раз). Включение скобок в регулярное выражение (вне Σ) не стандартно, но ближе к тому определению регулярных выражений, которое используется во многих приложениях. Отметим, что приведенный выше пример в формате PROSITE не соответствует данному определению, но легко к нему приводится.

Для примера пусть Σ — множество строчных латинских символов. Тогда $R = (a + c + t)ukk(p + q) * vdt(l + z + \epsilon)(pq)$ — это регулярное выражение над Σ , и $S = aukkprqppvdtprq$ — строка, задаваемая R . Чтобы определить S , подвыражение $(p + q)$ из R повторяется четыре раза, а выбором в подвыражении $(l + z + \epsilon)$ является пустая строка ϵ .

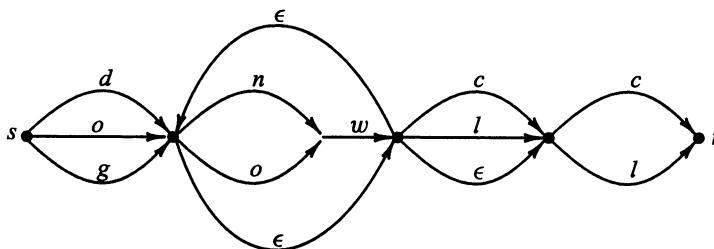


Рис. 3.19. Ориентированный граф для регулярного выражения $(d + o + g)((n + o)w) * (c + l + \epsilon)(c + l)$

Полезно представить регулярное выражение R ориентированным графом $G(R)$ (обычно называемым недетерминированным конечным автоматом). Пример показан на рис. 3.19. Граф имеет начальную вершину s и конечную вершину t , а каждая дуга помечена простым символом из $\Sigma \cup \epsilon$. Всякий путь от s до t в графе $G(R)$ определяет строку, которая получается конкатенацией символов из Σ , помечающих дуги пути. Множество строк, установленных такими путями, в точности совпадает с множеством строк, определенных регулярным выражением R . Правила построения $G(R)$ из R очень просты и оставляются как упражнение. Легко показать, что если регулярное выражение R имеет n символов, то $G(R)$ можно построить, используя не более $2n$ дуг. Подробности оставляются как упражнение и могут быть найдены в [10, 8].

Определение. Подстрока T' строки T соответствует регулярному выражению R , если в $G(R)$ существует путь от s до t , определяющий T' .

Поиск соответствий

Для поиска в тексте T подстроки, которая соответствует регулярному выражению R , рассмотрим сначала более простую задачу: соответствует ли R некоторый (неуточненный) префикс T . Пусть $N(0)$ — множество вершин $G(R)$, состоящее из s и всех вершин, достижимых из s проходом по дугам с пометкой ϵ . В общем случае вершина v принадлежит множеству $N(i)$ для $i > 0$, если v достижима из какой-либо вершины, принадлежащей $N(i-1)$, по дуге с пометкой $T(i)$, за которой могут еще следовать дуги с пометкой ϵ . Это дает конструктивное правило нахождения множества $N(i)$ по

множеству $N(i-1)$ и символу $T(i)$. Индукцией по i легко установить, что вершина v принадлежит $N(i)$ в том и только том случае, когда существует путь в $G(R)$ из s в v , генерирующий строку $T[1..i]$. Поэтому префикс $T[1..i]$ соответствует R в том и только том случае, если $N(i)$ содержит вершину t .

Исходя из этого рассуждения, мы видим, что для нахождения всех префиксов T , соответствующих R , нужно вычислить множества $N(i)$ для i от 0 до $m = |T|$. Если $G(R)$ содержит e дуг, то время работы этого алгоритма есть $O(me)$, потому что каждая итерация i (находящая $N(i)$ по $N(i-1)$ и символу $T(i)$) может быть реализована так, чтобы она выполнялась за время $O(e)$ (см. упражнение 29).

Чтобы найти *непрефиксную* подстроку T , соответствующую R , надо просто искать префикс T , который соответствует регулярному выражению Σ^*R . Σ^* обозначает любое число повторений (возможно, нулевое) любых символов из Σ . После этого уточнения мы получаем следующую теорему.

Теорема 3.6.1. *Если $|T| = m$ и регулярное выражение R содержит n символов, то определить, содержит ли T подстроку, соответствующую R , можно за время $O(nm)$.*

3.7. Упражнения

1. Оцените эмпирически сравнительную скорость методов Бойера–Мура и Апостолико–Джанкарло при разных предположениях относительно текста и образца. Эти предположения должны включать размер алфавита, “случайность” текста или образца, уровень периодичности текста или образца и т. п.
2. В методе Апостолико–Джанкарло массив M имеет размер m и может быть большим. Покажите, как модифицировать этот метод, чтобы он считал за такое же время, но вместо M использовался массив размера n .
3. Возможно, в методе Апостолико–Джанкарло лучше сначала сравнивать символы, а затем проверять M и N , если два символа совпали. Оцените эту идею теоретически и практически.
4. В методе Апостолико–Джанкарло $M(j)$ полагается равным числу, которое не меньше длины совпадения P и T при просмотре справа налево, начиная с позиции j в T . Найдите примеры, где алгоритм устанавливает значение, которое строго меньше, чем размер совпадения. Теперь, поскольку алгоритм находит во всех случаях точное положение несовпадения, $M(j)$ следовало бы полагать равным полной длине совпадающей части, и казалось бы, что это хорошо. Покажите, что это изменение привело бы к корректной имитации метода Бойера–Мура. Затем объясните, почему это не сделано в алгоритме.

Подсказка. Из-за временной оценки.

5. Докажите лемму 3.2.2, показав эквивалентность двух определений полупериодических строк.
6. Для каждого из n префиксов P мы хотим знать, не является ли префикс $P[1..i]$ периодической строкой. То есть для каждого i надо найти наибольшее $k > 1$ (если такое существует), для которого $P[1..i]$ можно записать в виде α^k с некоторой строкой α . Конечно, мы хотим знать также и период. Покажите, как это сделать для всех n префиксов P за время, линейное по n .

Подсказка. Z-алгоритм.

- Решите предыдущую задачу с некоторой модификацией: требуется определять, не является ли каждый префикс полупериодическим и с каким периодом. Снова время должно быть линейным.
 - С помощью более тщательного учета получите константу в оценке $O(m)$ в проведенном Коулом анализе времени для алгоритма Бойера–Мура.
 - Покажите, в каком месте не проходят рассуждения для оценки наихудшего случая Коула, если использовать только слабое правило сдвига Бойера–Мура. Можно ли этот аргумент исправить или линейная временная граница становится неправильной, когда используется только слабое правило? Рассмотрите пример с $T = ababababababababab$ и $P = xaaaaaaa$, отказавшись также от правила плохого символа.
 - Аналогично тому, что мы делали в п. 1.5, покажите, что применение классического препроцессинга Кнута–Морриса–Пратта к строке $P\$T$ дает метод с линейным временем для нахождения всех вхождений P в T . В действительности поисковая часть алгоритма Кнута–Морриса–Пратта (после окончания препроцессинга P) может рассматриваться как слегка оптимизированный вариант препроцессинга Кнута–Морриса–Пратта, примененный к части T строки $P\$T$. Уточните это утверждение и оцените выгоду, привнесенную оптимизацией.
 - Используя предположение, что \mathcal{P} свободно от подстрок (т.е. что никакой образец $P_i \in \mathcal{P}$ не входит подстрокой в другой образец $P_j \in \mathcal{P}$), завершите доказательство корректности алгоритма Ахо–Корасика. То есть докажите, что если в вершине v невозможны дальнейшие совпадения, то l можно положить равным $c - lp(v)$, и сравнения завершатся в вершине n_v без пропуска вхождений в T образцов из \mathcal{P} .
 - Докажите, что поисковая фаза алгоритма Ахо–Корасика выполнится за время $O(m)$, если никакой образец в P не является собственной подстрокой другого, а в противном случае — за время $O(m + k)$, где k — полное число вхождений.
 - Алгоритм Ахо–Корасика может встретиться с той же трудностью, что и алгоритм Кнута–Морриса–Пратта, когда он использует значения sp , а не sp' . Это видно, например, на рис. 3.16, где дуга ниже символа a в *potato* ведет к символу a в *tattoo*. Улучшенная функция λ неудач помогла бы обойти эту трудность. Уточните детали вычисления улучшенной функции неудач.
 - Постройте пример, показывающий, что k , число вхождений в T образцов из множества \mathcal{P} , может расти быстрее, чем $O(n + m)$. Убедитесь, что вы правильно понимаете размер ввода n . Постарайтесь сделать рост максимально возможным.
 - Докажите леммы 3.4.2 и 3.4.3, относящиеся к случаю образцов, не удовлетворяющих предположению о подстроках.
 - Временной анализ в доказательстве теоремы 3.4.1 отдельно рассматривает путь в \mathcal{K} для каждого образца P из \mathcal{P} . Это завышает время счета по сравнению с реально используемым в алгоритме. Выполните более тщательный анализ, чтобы связать время счета в алгоритме с числом вершин в \mathcal{K} .
 - Обсудите задачу (и решение, если вы можете его предложить) использования алгоритма Ахо–Корасика, когда:
 - джокеры разрешено использовать в тексте, но не в образце;
 - джокеры можно использовать и в тексте, и в образце.
 - Поскольку нелинейное время в работе алгоритма с джокерами получается из-за дублирования копий строк в \mathcal{P} , а такие дубликаты могут быть найдены и удалены за линейное

время, кажется заманчивым “исправить” метод, удалив сначала из \mathcal{P} все дубликаты. Этот подход аналогичен тому, который был применен в задаче поиска для двумерной строки, когда находились и помечались совпадающие строчки картинки. Рассмотрите этот подход и постараитесь использовать его для построения метода с линейным временем в задаче с шаблонами. Будет ли он работать, и если нет, то в чем трудности?

19. Покажите, как модифицировать метод джокеров, заменив массив C (длины $m > n$) на список длины n , сохранив такое же время счета.
 20. В задаче о джокерах мы предположили вначале, что никакой образец в \mathcal{P} не является подстрокой другого, а затем расширили алгоритм на случай, когда это предположение не выполняется. Не можем ли мы вместо этого просто свести случай, когда подстроки образцов допустимы, к случаю, когда они недопустимы? Например, можно добавить в конце каждой строки из \mathcal{P} новый символ, не входящий в сами образцы. Будет ли это работать? Рассмотрите вопросы корректности и сложности.
 21. Предположим, что джокер может соответствовать подстроке любой длины, а не только отдельному символу. Что вы можете сказать о точном поиске с джокерами такого типа в образце, в тексте, или и там и там?
 22. Другой подход к работе с джокерами в образце — это модификация алгоритмов Кнута–Морриса–Пратта и Бойера–Мура, при которой меняются правила сдвига и методы пре-процессинга. Можно ли ожидать успеха? Попробуйте и обсудите трудности (и решения, если вы их найдете).
 23. Дайте полное доказательство корректности и временной границы $O(n + m)$ для метода поиска двумерного совпадения, описанного в тексте (п. 3.5.3).
 24. Предположим, что в задаче поиска двумерного совпадения для каждого образца из \mathcal{P} используется метод Кнута–Морриса–Пратта, а не метод Ахо–Корасика. Какой будет временная оценка?
 25. Покажите, как изменить метод поиска двумерного совпадения в случае, если основание прямоугольного образца не параллельно основанию большой картинки, но ориентация этих оснований известна. Что случится, если образец не прямоугольный?
 26. Попробуем опустить фазу 2 в методе поиска двумерного совпадения следующим образом. Заведем счетчик в каждой ячейке большой картинки. Когда мы обнаруживаем, что строчка i малой картинки встретилась в строчке j большой картинки, начиная с позиции (i', j) , увеличиваем счетчик в ячейке $(i', j - i + 1)$. Заявим теперь, что P входит в T с верхним левым углом в любой ячейке, счетчик которой дошел до n' (числа строчек в P). Будет ли это работать?
- Подсказка.** Нет.
- Почему нет? Можете ли вы это исправить и заставить считать за время $O(n + m)$?
27. Предположим, что нам задано $q > 1$ малых (различных) прямоугольных картинок и мы хотим найти все вхождения любой из q маленьких картинок в большую прямоугольную картинку. Пусть n — полное число точек во всех малых картинках, а m — число точек в большой картинке. Подумайте, как решить эту задачу эффективно. Для упрощения предположите, что все малые картинки имеют одну и ту же ширину. Покажите, что время $O(n + m)$ достаточно.
 28. Покажите, как построить требуемый ориентированный граф $G(R)$ для регулярного выражения R . Конструкция должна обладать тем свойством, что если R содержит n символов, то $G(R)$ содержит не более $O(n)$ дуг.

29. Так как ориентированный граф $G(R)$ включает в себя $O(n)$ дуг, когда R содержит n символов, то $|N(i)| = O(n)$ для любого i . Есть ощущение, что множество $N(i)$ можно наивно найти по $N(i - 1)$ и $T(i)$ за время $O(ne)$. Однако в тексте для этой задачи указано время $O(e)$. Подумайте, как этого достигнуть. Объясните, почему уменьшение тривиально, если $G(R)$ не содержит дуг ϵ .
30. Растолкуйте, зачем в графе $G(R)$ нужны дуги ϵ . Если R не содержит символа замыкания $*$, можно ли всегда обойтись без дуг ϵ ? Биологические строки конечны, следовательно, можно совсем обойтись без $*$. Объясните, как это упрощает алгоритм поиска.
31. Шаблоны легко закодировать в регулярное выражение, как оно определено в тексте. Однако может оказаться более эффективным модифицировать определение регулярного выражения, явно включив в него символ-джокер. Развейте эту идею и объясните, как можно эффективно работать с джокерами, модифицировав алгоритм поиска по регулярному выражению.
32. В образцах PROSITE число возможных повторов подстроки часто задается в виде конечного диапазона чисел. Например, $CD(2\text{--}4)$ обозначает, что CD может быть повторено два, три или четыре раза. Формальное определение регулярного выражения не включает таких лаконичных спецификаций диапазона, но конечные спецификации диапазона могут быть отражены в регулярном выражении. Объясните, каким образом. Насколько такие спецификации могут увеличить длину выражения по сравнению с более компактным выражением PROSITE? Покажите, как такие диапазонные спецификации отражаются в ориентированном графе для регулярного выражения (разрешаются дуги ϵ). Покажите, что с его помощью можно искать в T подстроку, соответствующую регулярному выражению, за время $O(me)$, где m — длина T , а e — число дуг графа.
33. Теорема 3.6.1 задает границу времени, за которое можно определить, содержит ли T подстроку, отвечающую регулярному выражению R . Распространите рассуждения и теорему на задачу обнаружения и вывода всех таких вхождений. Представьте эту временную границу в виде суммы величины, не зависящей от числа вхождений, и величины, которая зависит от этого числа.

Глава 4

Получисленное сравнение строк

4.1. Что использовать: сравнения или арифметику?

Все методы точного поиска из первых трех глав, как и большинство других методов, которые еще будут обсуждаться, относятся к методам, *основанным на сравнении*. Главной элементарной операцией в каждом из них является сравнение двух символов. Есть, однако, методы поиска в строках, основанные на *битовых и арифметических операциях*. Они имеют иной оттенок, чем методы, основанные на сравнениях, даже несмотря на то, что иногда можно увидеть сравнения символов, спрятанные внутрь этих “получисленных” методов. Мы обсудим три примера этого подхода: метод *Shift-And* и его развитие в программе *agrep* для неточного поиска; использование быстрого преобразования Фурье в строковом поиске и метод случайных дактилограмм (отпечатков пальцев) Карпа и Рабина.

4.2. Метод *Shift-And*

Р. Беза-Йетс и Дж. Гоннет [35] изобрели простой битовый метод, который очень эффективно решает задачу точного поиска для относительно малых образцов (например, длиной в типичное английское слово). Они назвали его методом *Shift-Or*, хотя кажется более естественным назвать методом *Shift-And*. Напомним, что образец P имеет размер n , а текст T — размер m .

Определение. Пусть M — двоичный массив размером $n \times (m + 1)$, в котором индекс i пробегает значения от 1 до n , а индекс j — от 0 до m . Элемент

$M(i, j)$ равен 1 в том и только том случае, если первые i символов P точно совпадают с i символами T , кончаясь на позиции j . В противном случае элемент равен нулю.

Другими словами, $M(i, j) = 1$ тогда и только тогда, когда $P[1..i]$ точно совпадает с $T[j - i + 1..j]$. Например, если $T = \text{california}$ и $P = \text{for}$, то $M(1, 5) = M(2, 6) = M(3, 7) = 1$, тогда как для остальных комбинаций индексов $M(i, j) = 0$. Существенно, что элементы, равные 1, в строчке i показывают все места в T , где заканчиваются копии $P[1..i]$, а столбец j показывает все префиксы P , которые заканчиваются в позиции j строки T .

Ясно, что $M(n, j) = 1$ в том и только том случае, если вхождение P заканчивается в позиции j строки T ; следовательно, вычисление последней строчки M решает задачу точного совпадения. Чтобы вычислить M , алгоритм сначала создает для каждого символа алфавита x двоичный вектор $U(x)$ длины n . $U(x)$ полагается равным 1 в тех позициях P , где стоит символ x . Например, если $P = abacdeab$, то $U(a) = 10100010$.

Определение. Определим $\text{Bit-Shift}(j)$ как вектор, полученный сдвигом вектора для столбца j вниз на одну позицию и записью 1 в первой позиции. Старое значение в позиции n теряется. Иначе говоря, $\text{Bit-Shift}(j)$ состоит из единицы, к которой приписаны первые $n - 1$ битов столбца j .

На рис. 4.1 показан столбец j до и после этой операции.

0	1
0	0
1	0
0	1
1	0
1	1
0	1
1	0

Рис. 4.1. Столбец j до и после операции $\text{Bit-Shift}(j)$

4.2.1. Как построить массив M

Массив M вычисляется по столбцам следующим образом. Нулевой столбец M весь состоит из нулей. Элементы любого другого столбца $j > 0$ получаются из столбца $j - 1$ и вектора U для символа $T(j)$. В частности, вектор для столбца j получается операцией побитового логического умножения AND вектора $\text{Bit-Shift}(j - 1)$ и вектора $U(T(j))$. Более формально, если мы обозначим через $M(j)$ столбец j матрицы M , то $M(j) = \text{Bit-Shift}(j - 1) \text{ AND } U(T(j))$. Например, если $P = abaac$ и $T = xabxabaaxa$, то восьмой столбец M равен

1
0
1
0
0

так как префиксы P длины 1 и 3 кончаются в позиции 7 строки T . Восьмой символ T — это a , у которой вектор U таков:

1
0
1
1
0

Когда восьмой столбец M сдвигается вниз и логически умножается на $U(a)$, мы получаем

1
0
0
1
0

что и является правильным девятым столбцом M .

Чтобы понять в общем случае, почему метод *Shift-And* правильно вычисляет элементы массива, заметим, что для любого $i > 1$ элемент (i, j) должен равняться 1 в том и только том случае, если первые $i - 1$ символов P совпадают с $i - 1$ символами T , кончающимися на $j - 1$, а символ $P(i)$ совпадает с символом $T(j)$. Первое условие выполнено, когда элемент массива $(i - 1, j - 1)$ равен 1, а второе — когда i -й бит вектора U для символа $T(j)$ равен 1. После сдвига столбца $j - 1$ алгоритм логически умножает элемент $(i - 1, j - 1)$ столбца $j - 1$ на элемент i вектора $U(T(j))$. Следовательно, алгоритм вычисляет элементы массива M правильно.

4.2.2. Метод *Shift-And* эффективен для небольших образцов

Хотя метод *Shift-And* очень прост и, очевидно, в наихудшем случае количество битовых операций равно $\Theta(mn)$, он очень эффективен, если n меньше, чем размер машинного слова. В этом случае каждый столбец M и каждый вектор U могут быть закодированы одним словом, а операции *Bit-Shift* и AND выполняются каждая в одну команду. В большинстве компьютеров эти команды работают очень быстро, и в языках типа Си к ним можно непосредственно обращаться. Даже если n в несколько раз превосходит длину машинного слова, требуется лишь пара-другая действий со словами. Кроме того, в любой момент алгоритм нуждается только в двух столбцах массива M . Столбец j зависит единственно от столбца $j - 1$, так что предыдущие столбцы можно забывать. Следовательно, для образцов умеренного размера, таких как обычные английские слова, метод *Shift-And* очень эффективен и по времени, и по занимаемой памяти независимо от размера текста. С чисто теоретической точки зрения он не относится к методам с линейным временем, но заведомо практичен и может быть применен во многих ситуациях.

4.2.3. *agrep*: метод *Shift-And* с ошибками

С. Ву и Ю. Манбер [482] изобрели метод, который распространяет метод *Shift-And* на нахождение неточного вхождения образца в текст. Этот метод заложен в программу *agrep*. Говоря “неточный”, мы имеем в виду, что образец входит в текст либо точно,

либо с “малым” числом несовпадений, вставок или пропусков символов. Например, образец *atcga* входит в текст *aatatccasaa* с двумя несовпадениями, начиная с позиции четыре; он входит также и с четырьмя несовпадениями, начиная с позиции два. В этом пункте мы объясним программу *agrep* и то, как она справляется с несовпадениями. Случай, когда допустимы вставки и пропуски, будет оставлен на упражнения. При малом числе ошибок и небольших образцах программа *agrep* очень эффективна и может использоваться в ядре более изощренных методов текстового поиска. Неточное совпадение будет основной темой части III, но идеи, стоящие за *agrep*, настолько близки к методу *Shift-And*, что *agrep* вполне подходит для изложения и здесь.

Определение. Для двух строк, P и T , с длинами n и m соответственно, пусть M^k — двоичный массив, в котором $M^k(i, j)$ равно 1 в том и только том случае, если не менее $i - k$ из первых i символов P совпадают с i символами в отрезке T , кончающимся позицией j .

Таким образом, $M^k(i, j)$ — это естественное обобщение определения $M(i, j)$, допускающее k несовпадений. Здесь M^0 — это массив M , используемый в методе *Shift-And*. Если $M^k(n, j) = 1$, то существует вхождение P в T , кончающееся в позиции j и имеющее не более k несовпадений. Пусть $M^k(j)$ обозначает j -й столбец M^k .

В *agrep* пользователь выбирает значение k , после чего вычисляются массивы M , M^1, M^2, \dots, M^k . Эффективность метода зависит от размера k — чем больше k , тем медленнее метод. Для многих приложений достаточно малых значений k , таких как 3 или 4, и метод предельно быстр.

4.2.4. Как вычислить M^k

Пусть k — заданный максимум разрешенного числа несовпадений, выбранный пользователем. Метод вычисляет M^l для всех l от 0 до k . Есть несколько способов организации вычисления и его описания, но для простоты мы будем вычислять столбец j каждого массива M^l до того, как столбцы после j будут вычислены в каком-нибудь из массивов. Далее, для каждого j мы будем вычислять столбец j в массивах M^l в порядке увеличения l . В частности, нулевой столбец каждого массива снова инициализируется нулями. Затем j -й столбец матрицы M^l вычисляется так:

$$M^l(j) = M^{l-1}(j) \text{ OR } [\text{Bit-Shift}(M^l(j-1)) \text{ AND } U(T(j))] \text{ OR } M^{l-1}(j-1).$$

Интуитивно ясно, что здесь просто говорится, что первые l символов P совпадут с подстрокой T , кончающейся в позиции j , с не более чем l несовпадениями в том и только том случае, если выполнится одно из следующих трех условий:

- Первые i символов P совпадают с подстрокой T , кончающейся в j , с не более чем $l - 1$ несовпадениями.
- Первые $i - 1$ символов P совпадают с подстрокой T , кончающейся в $j - 1$, с не более чем l несовпадениями, и следующая пара символов в P и T совпадает.
- Первые $i - 1$ символов P совпадают с подстрокой T , кончающейся в $j - 1$, с не более чем $l - 1$ несовпадениями.

Легко проверить, что эти рекурсии корректны, и по всему алгоритму число битовых операций равно $O(knm)$. Как и в методе *Shift-And*, практическая эффективность

метода определяется тем, что мы рассматриваем битовые векторы (снова длины n) и операции очень просты — сдвиг на одну позицию и логическое умножение битовых векторов. Таким образом, когда образец относительно мал, так что столбец любой из матриц M^l умещается в несколько машинных слов, и k также мало, то *agrep* работает предельно быстро.

4.3. Задача о счете совпадений и быстрое преобразование Фурье

Если мы позволим себе не только битовые операции и разрешим, чтобы элементы матрицы M содержали числа между 0 и n , то можно легко адаптировать метод *Shift-And* к расчету для каждой пары i, j числа символов в $P[1..i]$, совпадающих с $T[j - i + 1..j]$. Это вычисление также представляет собой форму неточного совпадения — главной темы части III. Однако, как и в случае *agrep*, решение настолько связано с методом *Shift-And*, что мы рассматриваем его здесь. Вдобавок это естественное введение в следующую тему — счет совпадений (match-count). Для удобства изложения определим новую матрицу MC .

Определение. Матрица MC — это целочисленная матрица n на $m + 1$, каждый элемент которой $MC(i, j)$ равен числу символов в $P[1..i]$, совпавших при выравнивании с $T[j - i + 1..j]$.

Простой алгоритм для вычисления матрицы MC обобщает метод *Shift-And*, заменив операцию логического умножения на операцию *прибавления единицы*. Нулевой столбец MC стартует с нулевых значений, но каждый элемент $MC(i, j)$ теперь полагается равным $MC(i - 1, j - 1)$, если $P(i) \neq T(j)$, и $MC(i - 1, j - 1) + 1$ — в противном случае. Любой элемент в последней строчке со значением n снова указывает на вхождение P в T , но значения меньшие чем n подсчитывают *точное число совпадающих символов* при каждом возможном расположении P относительно T . Это обобщение использует $\Theta(nm)$ сложений и сравнений, хотя каждое сложение предельно просто: приращение на единицу.

Если мы хотим вычислить весь массив MC , то необходимо время $\Theta(nm)$, но наиболее важная информация содержится в последней строчке MC . Для каждой позиции $j \geq n$ из T последняя строчка показывает число символов, которые совпадают, когда правый конец P ставится вровень с символом j из T . Задача нахождения последней строчки MC называется задачей *счета совпадений*. Счет совпадений полезен в ряде задач, обсуждаемых ниже.

4.3.1. Быстрый метод для задачи счета совпадений?

Можно ли какой-нибудь из линейных методов поиска точного совпадения, изложенных в предыдущих главах, приспособить для решения за линейное время задачи счета совпадений? Это открытый вопрос. Обсуждавшееся выше обобщение метода *Shift-And* решает задачу счета совпадений, но требует во всех случаях $\Theta(nm)$ арифметических операций.

Удивительно, но задачу счета совпадений можно решить за $O(m \log m)$ арифметических операций, если использовать умножение и деление комплексных чисел.

Числа достаточно малы, так что допустима модель вычислений с единичным временем (т. е. никакое число не требует больше чем $O(\log m)$ битов), но операции остаются более сложными, чем простое приращение числа на единицу. Метод трудоемкости $O(m \log m)$ основывается на *быстрым преобразованием Фурье* (fast Fourier transform — FFT). Этот подход был разработан Фишером и Патерсоном [157] и независимо в биологической литературе Фельсенштейном, Сойером и Коином [152]. Другие исследования, основанные на этом методе, можно найти в [3, 58, 59, 99]. Мы сведем задачу счета совпадений к задаче, которая может быть эффективно решена с помощью FFT, но само FFT мы рассматриваем как черный ящик и предоставляем интересующемуся читателю изучать детали FFT самостоятельно.

4.3.2. Использование быстрого преобразования Фурье для счета совпадений

Суть задачи счета совпадений в нахождении последней строчки матрицы MC . Однако мы не будем работать непосредственно с MC , а займемся более общей задачей, решение которой содержит исходную информацию. Для этого удобно обозначить две строки, о которых пойдет речь, через α и β , а не через P и T , поскольку их роль в решении будет совершенно симметричной. Мы предположим, однако, что $|\alpha| = n \leq m = |\beta|$.

Определение. Определим $V(\alpha, \beta, i)$ как число символов α и β , которые совпадут, если левый конец строки α встанет напротив позиции i строки β . Обозначим через $V(\alpha, \beta)$ вектор, i -й элемент которого есть $V(\alpha, \beta, i)$.

Ясно, что при $\alpha = P$ и $\beta = T$ вектор $V(\alpha, \beta)$ включает в себя информацию, необходимую для последней строчки MC . Но он содержит больше информации, потому что мы разрешаем левому концу α заходить влево от левого конца β , а правому концу α — за правый конец β . Отрицательные числа определяют позиции влево от левого конца β , а положительные — все остальные позиции. Например, когда α выровнено с β так

21123456789
B: accctgtcc
A: aactgccg

то левый конец α выровнен по позиции -2 строки β .

Индекс i меняется от $-n + 1$ до m . Отметим, что когда $i > m - n$, правый конец α находится правее правого конца β . Для любого фиксированного i значение $V(\alpha, \beta, i)$ можно рассчитать за время $O(n)$ (просто подсчитав число совпадений и несовпадений), так что $V(\alpha, \beta)$ вычисляется за полное время $O(nm)$.

Покажем теперь, как получить $V(\alpha, \beta)$ за время $O(m \log m)$, используя быстрое преобразование Фурье. Для большинства интересных задач $\log m \ll n$, так что эта техника дает значительное ускорение. Более того, имеется специализированное “железо” для FFT, дающее возможность ускорения через оборудование. Это решение может работать с любым алфавитом, но давать объяснения легче на примере маленького алфавита. Для конкретности используем четырехбуквенный алфавит ДНК: a, t, c, g .

Общая схема подхода

Разобьем проблему счета совпадений на четыре задачи, по одной на каждую букву алфавита.

Определение. Определим $V_a(\alpha, \beta, i)$ как число совпадений буквы a , получающихся при размещении начала строки α около позиции i строки β . $V_a(\alpha, \beta)$ — вектор длины $(n + m)$, содержащий эти значения.

Аналогичные определения вводятся для трех остальных букв. Используя их, получаем

$$V(\alpha, \beta, i) = V_a(\alpha, \beta, i) + V_t(\alpha, \beta, i) + V_c(\alpha, \beta, i) + V_g(\alpha, \beta, i)$$

и

$$V(\alpha, \beta) = V_a(\alpha, \beta) + V_t(\alpha, \beta) + V_c(\alpha, \beta) + V_g(\alpha, \beta).$$

Осталось определить, как вычислить $V_a(\alpha, \beta, i)$ для каждого i . Преобразуем наши строки в двоичные строки $\bar{\alpha}_a$ и $\bar{\beta}_a$ соответственно, где каждое вхождение символа α заменяется единицей, а остальные символы — нулями. Например, пусть $\alpha = accaaccggaggtat$ и $\beta = accacsgaag$. Тогда двоичные строки $\bar{\alpha}_a$ и $\bar{\beta}_a$ будут 1011000100010 и 100100110. Чтобы вычислить $V_a(\alpha, \beta, i)$, расположим начало $\bar{\beta}_a$ у позиции i строки $\bar{\alpha}_a$ и подсчитаем число столбцов, где оба бита равны 1. Например, если $i = 3$, то мы получаем

$$\begin{array}{r} 1011000100010 \\ 10010110 \end{array}$$

и $V_a(\alpha, \beta, 3) = 2$. Если $i = 9$, то

$$\begin{array}{r} 1011000100010 \\ 10010110 \end{array}$$

и $V_a(\alpha, \beta, 9) = 1$.

Другой способ получить ответ — заполнить все незанятое пространство нулями (чтобы оба двоичных вектора имели одну и ту же длину), логически умножить строки и сложить биты результата.

Чтобы formalизовать эту идею, дополним правый конец $\bar{\beta}$ (большой строки) n нулями, а правый конец $\bar{\alpha}$ — m нулями. Обе получившиеся строки будут иметь длину $n + m$ каждая. Далее, для удобства перенумеруем индексы в обеих строках, чтобы они изменялись от 0 до $n + m - 1$. Тогда

$$V_a(\alpha, \beta, i) = \sum_{j=0}^{j=n+m-1} \bar{\alpha}_a(j) \times \bar{\beta}_a(i + j),$$

где индексы вычисляются по модулю $n + m$. Дополнительные нули позволяют справиться со случаями, когда левый конец α находится левее левого конца β или, наоборот, когда правый конец α — правее правого конца β . Нулей приписано достаточно, так что когда правый конец α стоит правее правого конца β , то соответствующие биты в добавленной $\bar{\alpha}_a$ все стоят против нулей. Следовательно, “незаконного перекрытия” α и β не получится и $V_a(\alpha, \beta, i)$ вычисляется правильно.

Пока все, чего мы достигли, — это изменение формулировки задачи о счете совпадений, которое еще не предлагает способа вычисления $V_a(\alpha, \beta)$ более эффективного, чем при выполнявшемся двоичном кодировании и присваивании нулей. Но здесь уже сходятся вместе корреляция и FFT.

Циклическая корреляция

Определение. Пусть X и Y — два вещественных вектора длины z с компонентами, индексированными от 0 до $z - 1$. Циклическая корреляция X и Y — это вещественный вектор длины z , определяемый формулой $W(i) = \sum_{j=0}^{i=z-1} X(j) \times Y(i + j)$, где индексы берутся по модулю z .

Ясно, что задача вычисления вектора $V_a(\alpha, \beta)$ совпадает с задачей вычисления циклической корреляции дополненных строк $\bar{\alpha}_a$ и $\bar{\beta}_a$. Точнее, $X = \bar{\alpha}_a$, $Y = \bar{\beta}_a$, $z = n + m$ и $W = V_a(\alpha, \beta)$.

Алгоритм, основанный на расчете циклической корреляции, потребует $\Theta(z^2)$ операций, так что никакого прогресса пока не наблюдается. Но циклическая корреляция — это классическая задача, о которой известно, что она решается за время $O(z \log z)$ с помощью быстрого преобразования Фурье. (FFT чаще ассоциируется с задачей свертки двух векторов, но циклическая корреляция и свертка очень похожи. На самом деле, циклическая корреляция получается из свертки, если перевернуть один из векторов.)

Метод FFT и его использование в задаче циклической корреляции выходят за рамки этой книги, но главное в том, что он решает эту задачу для двух векторов, длиной z каждый, за $O(z \log z)$ арифметических операций. Следовательно, он решает задачу счета совпадений, используя только $O(m \log m)$ арифметических операций. Метод FFT поразительно эффективен и определенно улучшает оценку $\Theta(nm)$, которую дает обобщенный подход *Shift-And*. Однако он требует операций с комплексными числами, и поэтому каждый арифметический шаг более сложен (и возможно, дороже стоит), чем в более непосредственном методе *Shift-And*.^{*}

Счет совпадений при работе с джокерами

Вспомним обсуждение джокеров, начатое в п. 3.5.2, где символ-джокер ϕ совпадал с любым другим одиночным символом. Например, если $\alpha = ag\phi\phi\phi tfa$ и $\beta = agctcgt$, то $V(\alpha, \beta, 1) = 7$ (т.е. все позиции, кроме последней, считаются совпадающими). Можно ли использовать символы-джокеры в счете совпадений с помощью FFT? Если джокеры встречаются только в одной из двух строк, скажем, в β , то все решается просто. При вычислении $V_x(\alpha, \beta, i)$ для каждой позиции i и символа x нужно заменить каждый символ-джокер в β символом x . Это работает, так как для любой фиксированной начальной точки i и любой позиции j в β j -я позиция добавит единицу в $V_x(\alpha, \beta, i)$ не больше чем для одного символа x , в зависимости от того, какой символ находится в позиции $i + j - 1$ строки α (т.е. от того, какой символ в α стоит около j -й позиции в β).

Но если джокеры встречаются и в α и β , то этот прямой путь не работает. Предположим, что два джокера стоят напротив друг друга, и когда α начинается в позиции i , значение $V_x(\alpha, \beta, i)$ будет завышено, так как эти два символа будут засчитываться как совпадение при вычислении $V_x(\alpha, \beta, i)$ для каждого $x = a, t, c$ и g . Так что если при фиксированном i имеется k мест, где два джокера стоят

*¹) Один родственный подход [58] пытается решить задачу счета совпадений за $O(m \log m)$ операций цепочисленной арифметики (не комплексных), реализуя FFT над конечным полем. На практике он, вероятно, превосходит подход, основанный на комплексных числах, хотя в терминах теории сложности объявленная граница $O(m \log m)$ не вполне кошерна, так как она использует предварительно вычисленную таблицу чисел, и оценка справедлива только для m до определенного предела.

напротив друг друга, то вычисленное значение $\sum_x V_x(\alpha, \beta, i)$ будет на $3k$ больше, чем правильное значение $V(\alpha, \beta, i)$. Как можно избежать этого завышения?

Ответ заключается в том, что нужно найти k и потом исправить сумму. Идея такова: рассматривать ϕ как настоящий символ и вычислить $V_\phi(\alpha, \beta, i)$ для каждого i . Тогда

$$V(\alpha, \beta, i) = \sum_{x \neq \phi} V_x(\alpha, \beta, i) - 3V_\phi(\alpha, \beta, i),$$

где $V_x(\alpha, \beta, i)$ считается для каждого символа x с заменой каждого джокера на x . В результате имеем:

Теорема 4.3.1. Задача счета совпадений может быть решена за время $O(m \log m)$, даже если в P и T включено неограниченное число джокеров.

Позднее, после обсуждения суффиксных деревьев и общих предшественников, мы предложим в п. 9.3 другой подход к работе с джокерами, появляющимися в обеих строках. Этот подход будет основан на сравнениях.

4.4. “Дактилоскопические” методы Карпа–Рабина для поиска точного совпадения

Метод *Shift-And* предполагает, что мы можем эффективно выполнять сдвиг вектора битов, а обобщенный метод *Shift-And* — что мы можем эффективно прибавлять к целому числу единицу. Если мы рассматриваем (строковый) битовый вектор как целое число, то сдвиг влево на один бит приводит к удвоению числа (в предположении, что бит не потерялся на левом конце). Так что будет не очень существенным добавлением предположение, что кроме прибавления единицы мы можем так же эффективно умножать целое число на два. Добавив эту элементарную операцию, мы можем превратить задачу поиска точных совпадений (снова без несовпадений) в арифметическую задачу. Первым результатом будет простой метод с линейным временем, в котором вероятность сделать ошибку очень мала. Этот метод будет преобразован в другой, который ошибок не делает, но у которого линейно только *ожидаемое* время работы. Мы объясним эти результаты, используя двоичную строку P и двоичный текст T . То есть мы предполагаем, что алфавит состоит из 0 и 1. Обобщение на большие алфавиты выполняется без трудностей и оставляется читателю.

4.4.1. Арифметика заменяет сравнения

Определение. Для текстовой строки T пусть T_r^n обозначает ее подстроку длины n , начинающуюся в символе r . Обычно n понятно из контекста, и в таких случаях мы T_r^n заменим на T_r .

Определение. Для двоичного образца P определим

$$H(P) = \sum_{i=1}^{i=n} 2^{n-i} P(i).$$

Аналогично определим

$$H(T_r) = \sum_{i=1}^{i=n} 2^{n-i} T(r+i-1).$$

Итак, мы рассматриваем P как n -битовое двоичное число. И каждое T_r^n тоже. Например, если $P = 0101$, то $n = 4$ и $H(P) = 2^3 \times 0 + 2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$. если $T = 101101010$, $n = 4$ и $r = 2$, то $H(T_r) = 6$.

Ясно, что если имеется вхождение P в T , начинающееся в r , то $H(P) = H(T_r)$. Обратное также верно.

Теорема 4.4.1. *Строка P входит в T , начиная с позиции r в том и только том случае, если $H(P) = H(T_r)$.*

Доказательство, которое мы оставляем читателю, является прямым следствием того, что каждое целое число можно единственным образом представить в виде суммы положительных степеней 2.

Теорема 4.4.1 сводит задачу поиска точных совпадений к вычислительной задаче, в которой сравниваются не символы, а два числа, $H(P)$ и $H(T_r)$. Но кроме тех случаев, когда образец достаточно мал, вычисление $H(P)$ и $H(T_r)$ не будет эффективным *). Проблема в том, что степени двойки, требуемые в определении $H(P)$ и $H(T_r)$, растут слишком быстро. (С точки зрения теории сложности, использование таких больших чисел нарушает предположения о базовой вычислительной машине с *памятью свободного доступа* с единичным временем (random access machine model — RAM). В этой модели наибольшие допустимые числа должны быть представлены числом битов порядка $O(\log(n+m))$, тогда как число 2^n требует n битов. Таким образом, требуемые числа экспоненциально велики.) Более того, когда алфавит не двоичный, а имеет, скажем, t символов, потребуются числа порядка t^n .

В 1987 г. Р. Карп и М. Рабин [266] опубликовали метод (изобретенный почти на десять лет раньше), названный методом *рандомизированной дактилограммы*, который сохраняет дух изложенного численного подхода и при этом предельно эффективен, используя числа, которые удовлетворяют модели памяти. Это *рандомизированный* метод, в котором часть “*только в том случае*” теоремы сохраняется, а часть “*в том случае*” не верна. Она оказывается верной только с *высокой вероятностью*. Это подробно объясняется в следующем пункте.

4.4.2. Дактилограммы P и T

Общая идея в том, чтобы вместо самих чисел $H(P)$ и $H(T_r)$, слишком больших, работать с остатками от их деления на относительно малое число p , т. е. работать с числами по *модулю* p . Тогда все расчеты будут проводиться в малом числе битов, и это будет эффективно. Но по-настоящему привлекательной чертой этого метода является доказательство того, что вероятность ошибки может быть сделана малой, если P выбирается случайно из некоторого множества. Следующие определения и леммы уточняют это утверждение.

*) Можно эффективно получить $H(T_{r+1})$ из $H(T_r)$, а не прямо из определения (и это нам позднее понадобится), но дело сейчас не во времени этого пересчета.

Определение. Для положительного числа p значение $H_p(P)$ определяется как $H(P) \bmod p$. Таким образом, $H_p(P)$ — это остаток от деления $H(P)$ на p . Аналогично, $H_p(T_r)$ определяется как $H(T_r) \bmod p$. Числа $H_p(P)$ и $H_p(T_r)$ называются *дактилограммами* P и T_r .

Несомненна выгода уже самого использования дактилограмм. Вычисление $H(P)$ и $H(T_r)$ по модулю p удерживает каждую дактилограмму в диапазоне от 0 до $P - 1$, так что размер дактилограммы никогда не нарушает требований модели памяти. Но если бы $H(P)$ и $H(T_r)$ нужно было вычислять до нахождения остатков от деления на p , мы столкнулись бы с теми же проблемами слишком больших промежуточных результатов. К счастью, модульная арифметика позволяет делать приведение в любое время (т. е. можно не приводить слишком сильно), так что имеет место следующее обобщение правила Горнера.

Лемма 4.4.1. $H_p(P) = (((\dots(((P(1) \times 2 \bmod p + P(2)) \times 2 \bmod p + P(3)) \times 2 \bmod p + \dots + P(4)) \dots) \times 2 \bmod p + P(n)) \bmod p$, и во время вычисления $H_p(P)$ ни одно число не превзойдет $2p$.

Например, если $P = 101111$ и $p = 7$, то $H(P) = 47$ и $H_p(P) = 47 \bmod 7 = 5$. Более того, это вычисление можно выполнить так:

$$\begin{aligned} 1 \times 2 \bmod 7 + 0 &= 2 \\ 2 \times 2 \bmod 7 + 1 &= 5 \\ 5 \times 2 \bmod 7 + 1 &= 4 \\ 4 \times 2 \bmod 7 + 1 &= 2 \\ 2 \times 2 \bmod 7 + 1 &= 5 \\ 5 \bmod 7 &= 2. \end{aligned}$$

Особенность схемы Горнера не только в том, что требуемое число сложений и умножений линейно, но и в том, что промежуточные результаты остаются малыми.

Промежуточные результаты сохраняются малыми и при расчете $H_p(T_r)$ для любого r , его вычисление аналогично $H_p(P)$. Однако эффективность подсчета можно повысить: для $r > 1$ значение $H_p(T_r)$ получается из $H_p(T_{r-1})$ использованием малого *постоянного* числа операций. Так как

$$H_p(T_r) = H(T_r) \bmod p$$

и

$$H(T_r) = 2 \times H(T_{r-1}) - 2^n T(r-1) + T(r+n-1),$$

мы имеем

$$H(T_r) = ((2 \times H(T_{r-1}) \bmod p) - (2^n \bmod p) \times T(r-1) + T(r+n-1)) \bmod p.$$

Далее,

$$(2^n \bmod p) = 2 \times (2^{n-1} \bmod p) \bmod p.$$

Поэтому каждая следующая степень двойки берется по модулю 2 и каждое следующее значение $H_p(T_r)$ вычисляется за константное время.

Простые основания ограничивают ложные совпадения

Очевидно, что если P входит в T , начиная с позиции r , то $H_p(P) = H_p(T_r)$, но обратное верно не для любого p . Так что мы не имеем права делать вывод, что P входит в T , начиная от r , просто потому, что $H_p(P) = H_p(T_r)$.

Определение. Если $H_p(P) = H_p(T_r)$, но P не входит в T , начиная с позиции r , мы скажем, что произошло *ложное совпадение* P и T в позиции r . Если существует *некоторая* позиция r , для которой есть ложное совпадение P и T в r , мы скажем, что есть ложное совпадение P и T .

Нужно выбрать основание p , достаточно малое, чтобы арифметика была эффективной, и достаточно большое, чтобы вероятность ложного совпадения P и T была малой. Ключ к решению — в том, чтобы выбрать p *простым* числом из подходящего диапазона и использовать свойства простых чисел. Мы приведем требуемые свойства простых чисел без доказательства.

Определение. Для положительного целого q значение $\pi(q)$ равно *количество* простых чисел, не превосходящих q .

Следующая теорема — вариант известной *теоремы о простых числах*.

Теорема 4.4.2. *Имеют место следующие неравенства*

$$\frac{q}{\ln q} \leq \pi(q) \leq 1.26 \frac{q}{\ln q},$$

где $\ln q$ — *натуральный логарифм* q [383].

Лемма 4.4.2. *Если $q \geq 29$, то произведение всех простых чисел, не превосходящих q , превосходит 2^q [383].*

Например, для $q = 29$ простые числа, не превосходящие 29, это 2, 5, 7, 11, 13, 17, 19, 23 и 29. Их произведение равно 2 156 564 410, тогда как $2^{29} = 536\,870\,912$.

Следствие 4.4.1. *Если $q \geq 29$ и x — любое число, не превосходящее 2^q , то x имеет меньше чем $\pi(q)$ (различных) простых делителей.*

Доказательство. Предположим, что x имеет $k > \pi(q)$ различных простых делителей q_1, q_2, \dots, q_k . Тогда $2^q \geq x \geq q_1 q_2 \dots q_k$ (первое неравенство следует из сделанного предположения, а второе — из того, что некоторые простые числа в разложении x на множители могут повторяться). Но $q_1 q_2 \dots q_k$ не меньше, чем произведение k наименьших простых чисел, которое больше, чем произведение первых $\pi(q)$ простых (по предположению о том, что $k > \pi(q)$). Однако произведение простых чисел, не превосходящих q , больше чем 2^q (по лемме 4.4.2). Поэтому предположение, что $k > \pi(q)$, ведет к противоречивому неравенству $2^q > 2^q$, и лемма доказана. \square

Центральная теорема

Мы готовы доказать центральную теорему подхода Карпа–Рабина.

Теорема 4.4.3. *Пусть P и T — некоторые строки, причем $nm \geq 29$, где $n = |P|$ и $m = |T|$. Пусть I — некоторое положительное число. Если p — случайно выбранное простое число, не превосходящее I , то вероятность ложного совпадения P и T не превосходит $\pi(nm)/\pi(I)$.*

Доказательство. Пусть R — множество позиций в T , где P не начинается. Это значит, что $s \in R$ в том и только том случае, если P не входит в T , начинаясь в s . Для каждого $s \in R$ имеем $H(P) \neq H(T_s)$. Рассмотрим $\prod_{s \in R} (|H(P) - H(T_s)|)$. Это произведение не должно быть больше 2^{nm} , так как для любого s верно неравенство $|H(P) - H(T_s)| \leq 2^n$ (напомним, что предполагается двоичный алфавит). В силу следствия 4.4.1 произведение $\prod_{s \in R} (|H(P) - H(T_s)|)$ имеет не больше $\pi(nm)$ различных простых делителей.

Предположим, что ложное совпадение P и T случилось в некоторой позиции r из T . Это значит, что $H(P) \bmod p = H(T_r) \bmod p$ и что p является делителем разности $H(P) - H(T_r)$. Тогда p является делителем и $\prod_{s \in R} (|H(P) - H(T_s)|)$, т. е. p принадлежит к простым делителям этого произведения. Если p допускает ложное совпадение P и T , то p должно быть одним из не более $\pi(nm)$ чисел. Но p выбрано случайно из множества $\pi(I)$ чисел, так что вероятность того, что p — простое число, допускающее ложное совпадение P и T , не превосходит $\pi(nm)/\pi(I)$. \square

Отметим, что теорема 4.4.3 верна при любом выборе образца P и текста T , таких что $nm > 29$. Вероятность в теореме относится не к случайному выбору P и T , а к случайному выбору простого p . Таким образом, эта теорема не делает каких-либо (сомнительных) предположений о том, что P или T случайны или генерируются марковским процессом и т. п. Она верна для любых P и T ! Более того, теорема не только ограничивает вероятность ложного совпадения в конкретной позиции r , — она ограничивает вероятность того, что такая позиция найдется. Замечательно, что анализ в доказательстве теоремы выглядит "слабым". То есть устанавливается только очень слабое свойство простого числа p , допускающего ложное совпадение, именно то, что оно должно быть одним из не более $\pi(nm)$ чисел, на которые делится $\prod_{s \in R} (|H(P) - H(T_s)|)$. Предполагается, что истинная вероятность ложного совпадения P и T значительно меньше границы, полученной в теореме.

Теорема 4.4.3 ведет к следующему алгоритму случайной дактилограммы для поиска всех вхождений P в T .

Алгоритм рандомизированной дактилограммы

1. Выбрать положительное целое I (подробно обсуждается ниже).
2. Случайно указать простое число, не превосходящее I , и вычислить $H_p(P)$. (Для нахождения простых чисел существуют эффективные рандомизированные алгоритмы [331].)
3. Для каждой позиции r в T вычислить $H_p(T_r)$ и сопоставить с $H_p(P)$. Если они равны, то либо объявить о вероятном совпадении, либо явно проверить совпадение P с T , начиная с позиции r .

Так как каждое $H_p(T_r)$ можно пересчитать за константное время из $H_p(T_{r-1})$, то алгоритм дактилограммы работает за время $O(m)$, исключая время явной проверки объявленного совпадения. Может, однако, оказаться разумным не беспокоиться о явной проверке объявленного совпадения, в зависимости от вероятности ошибки. Мы вернемся к вопросу проверки позднее. А сейчас, чтобы завершить анализ вероятности ошибки, нам нужно ответить на вопрос, каким должно быть I .

Как выбрать I

Полезность метода дактилограммы зависит от выбора хорошего значения для I . Когда I возрастает, вероятность ложного совпадения P и T убывает, но допустимый размер p увеличивается и вместе с ним растут затраты на вычисление $H_p(P)$ и $H_p(T_r)$. Существует ли компромисс? Есть несколько хороших способов выбора I в зависимости от n и m . Один из них — положить $I = nm^2$. При таком выборе наибольшее из чисел, используемых в алгоритме, требует не более $4(\log n + \log m)$ битов, что удовлетворяет требованиям модели памяти, согласно которым эти числа должны быть малыми в сравнении с размером ввода. А что можно сказать о вероятности ложного совпадения?

Следствие 4.4.2. *Если $I = nm^2$, то вероятность ложного совпадения не превосходит $2.53/m$.*

Доказательство. По теореме 4.4.3 и теореме о простых числах (теорема 4.4.2) вероятность ложного совпадения имеет следующие границы:

$$\frac{\pi(nm)}{\pi(nm^2)} \leq 1.26 \frac{nm}{nm^2} \frac{\ln(nm^2)}{\ln(nm)} = 1.26 \frac{1}{m} \left(\frac{\ln n + 2 \ln m}{\ln n + \ln m} \right) \leq \frac{2.53}{m}. \quad \square$$

Маленький пример из [266] иллюстрирует эту границу. Возьмем $n = 250$, $m = 4000$ и, следовательно, $I = 4 \times 10^9 < 2^{32}$. Тогда вероятность ложного совпадения не превзойдет $2.53/4000 < 10^{-3}$. Таким образом, с 32-битовой дактилограммой для любых P и T вероятность того, что хотя бы одно из сделанных алгориттом опознаний будет ложным, не превзойдет 0.001.

С другой стороны, если $I = n^2m$, то вероятность ложного совпадения имеет порядок $O(1/n)$, и так как требуется время $O(n)$ на определение, ложное это совпадение или настоящое, то *ожидалось* время проверки ограничено константой. В результате получается метод с *ожидалым* временем $O(m)$ и без ложных совпадений.

Обобщения

Если так хорошо получается с одним простым числом, то почему не попробовать несколько? Почему бы не взять случайно k простых чисел p_1, p_2, \dots, p_k и не вычислить k дактилограмм? Для любой позиции r вхождение P , начинающееся в r , будет распознаваться, только если $H_{p_i}(P) = H_{p_i}(T_r)$ для каждого из k выбранных простых чисел. Определим теперь ложное совпадение P и T , как ситуацию, в которой найдется такая позиция r , что хотя P не входит в T , начиная с r , но $H_{p_i}(P) = H_{p_i}(T_r)$ для каждого из k простых чисел. Какова теперь будет вероятность ложного совпадения P и T ? Наша граница строится совсем просто и понятно.

Теорема 4.4.4. *Когда k простых чисел выбраны случайно между 1 и I и составлены k дактилограмм, вероятность ложного совпадения P и T не превосходит $(\pi(nm)/\pi(I))^k$.*

Доказательство. Мы видели в доказательстве теоремы 4.4.3, что если p — простое число, при котором $H_p(P) = H_p(T_r)$ для некоторой позиции r , где P не появилось, то p выбрано из множества не более $\pi(nm)$ чисел. Когда используются k дактилограмм, то ложное совпадение может случиться, только если любое из k простых

находится в этом множестве, и так как эти простые выбираются случайно (независимо), граница из теоремы 4.4.3 верна для *каждого* из них. Следовательно, вероятность того, что все простые находятся в этом множестве, ограничена величиной $(\pi(nm)/\pi(I))^k$, и теорема доказана. \square

В частности, если $k = 4$, а n , m и I выбраны, как в предыдущем примере, то вероятность ложного совпадения P и T не превосходит 10^{-12} . Таким образом, эта вероятность значительно уменьшилась, с 10^{-3} до 10^{-12} , тогда как вычислительные затраты при использовании четырех простых чисел выросли в четыре раза. Для типичных значений n и m выбор небольшого значения k обеспечит такую вероятность ошибки из-за ложного совпадения, которая будет меньше, чем вероятность ошибки из-за сбоя оборудования.

Еще меньшие границы ошибки

Анализ в доказательстве теоремы 4.4.4 все еще очень слаб, поскольку в нем просто перемножаются вероятности того, что каждое из k простых чисел допускает ложное совпадение *где-то* в T . Однако, чтобы алгоритм действительно делал ошибку в некоторой конкретной позиции r , нужно, чтобы каждое из простых допускало ложное совпадение при одном и том же r . Такое событие еще менее правдоподобно. Учитывая это, мы можем уменьшить вероятность ложного совпадения следующим образом:

Теорема 4.4.5. *Когда k простых чисел выбираются случайно между 1 и I и используется k дактилограмм, вероятность ложного совпадения P и T не превосходит $m(\pi(n)/\pi(I))^k$.*

Доказательство. Допустим, что ложное совпадение произошло в некоторой позиции r . Это означает, что каждое простое p_i должно быть делителем $|H(P) - H(T_r)|$. Так как $|H(P) - H(T_r)| \leqslant 2^n$, то у этого числа есть не более $\pi(n)$ простых делителей. Таким образом, каждое p_i было выбрано случайно из множества $\pi(I)$ простых и оказалось частью подмножества из $\pi(n)$ простых. Вероятность этого для фиксированного r равна $(\pi(n)/\pi(I))^k$. Так как имеется m возможных выборов r , вероятность ложного совпадения P и T (т. е. вероятность того, что найдется такая позиция r) не превосходит $m(\pi(n)/\pi(I))^k$, и теорема доказана. \square

Полагая, как и раньше, что $I = nm^2$, с помощью несложных выкладок (которые оставляются читателю) убеждаемся в следующем.

Следствие 4.4.3. *Если k простых чисел выбираются случайно и используются в алгоритме дактилограмм, вероятность ложного совпадения P и T не превосходит $(1.26)^k m^{-(2k-1)}(1 + 0.6 \ln m)^k$.*

Применение этого результата к рассматриваемому примеру с $n = 250$, $m = 4000$ и $k = 4$ уменьшает границу вероятности ложного совпадения до 2×10^{-22} .

Упомянем еще одно улучшение, предложенное в [266]. Возвращаясь к случаю, когда используется только одно простое число, предположим, что алгоритм явно проверяет, что P входит в T , если $H_p(P) = H_p(T_r)$, и обнаруживает, что вхождения нет. Может быть, для продолжения вычислений лучше сменить простое число? Интуиция подсказывает, что это имеет смысл. Теорема 4.4.3 рандомизирует по выбору простых и ограничивает вероятность того, что случайно взятое простое допускает ложное

совпадение где-то в T . Но когда выбранное простое число уже привело к ложному совпадению, оно более не является случайным, и не исключено, что может допускать многочисленные ложные совпадения (*чертова семя*). Теорема 4.4.3 ничего не говорит о том, насколько “плохим” может быть конкретное простое число. Но выбирая новое простое после каждой ошибки, мы можем применить следствие 4.4.2 к каждому из них и получить следующую теорему.

Теорема 4.4.6. *Если при обнаружении ошибки новое простое число выбирается случайно, то для любых образца и текста вероятность t ошибок не превзойдет $(2.53/m)^k$.*

Эта вероятность убывает так быстро, что защиту от длинных серий ошибок можно считать обеспеченной. Дальнейший анализ вероятностных свойств метода Карпа–Рабина см. в [182].

Обнаружение ошибки за линейное время

Все предложенные выше варианты метода Карпа–Рабина обладали тем свойством, что находили *все настоящие* вхождения P в T , но могли найти также и *ложные совпадения* — места, где сообщается о вхождении P в T , хотя его там нет. Если проверять каждое объявленное вхождение P , на это потребуется в наихудшем случае время $\Theta(pt)$, хотя ожидаемое время можно сделать меньше. Мы представим здесь метод со временем $O(m)$, впервые указанный С. Матакришнаном [336]. Этот метод проверяет, не является ли какое-либо из объявленных совпадений ложным. То есть метод за время $O(m)$ либо подтверждает, что алгоритм Карпа–Рабина не нашел ложных совпадений, либо декларирует, что по меньшей мере одно из совпадений ложное (но не может указать всех ложных совпадений).

Метод связан с предложенным Галилом обобщением метода Бойера–Мура (п. 3.2.2), но читателю не нужно к нему возвращаться. Рассмотрим список \mathcal{L} (начальных) позиций в T , где алгоритм Карпа–Рабина объявил о вхождении P . Назовем *полосой* наибольший интервал из последовательных начальных позиций l_1, l_2, \dots, l_r в \mathcal{L} , такой что два последовательных числа в интервале отстоят не более чем на $n/2$ (т. е. $|l_{i+1} - l_i| \leq n/2$). Метод работает с каждой полосой отдельно, так что нам нужно обсудить, как проверить на ложные совпадения отдельную полосу.

В отдельной полосе метод явно проверяет вхождения P в первых двух позициях полосы, l_1 и l_2 . Если P не входит в каком-нибудь из этих мест, то метод уже нашел ложное совпадение и останавливается. В противном случае, когда P входит в обоих местах, l_1 и l_2 , метод устанавливает, что строка P полупериодическая с периодом $l_2 - l_1$ (см. лемму 3.2.3). Обозначим этот период через d и покажем, что это наименьший период P . Если d не наименьший период, то d должно быть кратно наименьшему периоду, скажем, d' . (Это непосредственно получается из теоремы об ОНД, сформулированной в п. 16.17.1, см. с. 521.) Но отсюда следовало бы, что существует вхождение P , начинающееся в позиции $l_1 + d' < l_2$, и так как метод Карпа–Рабина *никогда* не пропускает вхождений P , это противоречит выбору l_2 как второго вхождения P в интервале между l_1 и l_2 . Итак, d должно быть наименьшим периодом P , а отсюда вытекает, что если в полосе нет ложных совпадений, то $|l_{i+1} - l_i| = d$ для каждого i в полосе. Следовательно, в качестве первого контроля метод проверяет, что $|l_{i+1} - l_i| = d$ для каждого i ; если для какого-нибудь i это равенство не выполняется, то метод объявляет о ложном совпадении и останавливается.

В противном случае, как в методе Галила для проверки всех мест из \mathcal{L} , достаточно сравнить последние d символов в каждом объявленном вхождении P с последними d символами P . Значит, в позиции l_i метод проверяет d символов T , начиная с позиции $l_i + n - d$. Если какая-либо из этих последовательных проверок обнаруживает несовпадение, то метод нашел ложное совпадение и останавливается. В противном случае P действительно входит в T в каждом указанном месте полосы.

Для анализа времени прежде всего отметим, что во время проверки отдельной полосы никакой символ из T не проверяется более чем дважды. Более того, так как две полосы разделены по меньшей мере $n/2$ позициями и каждая полоса имеет длину не менее n позиций, никакой символ T не может быть проверен больше чем в двух смежных полосах. Отсюда следует, что полное время, затрачиваемое методом на все полосы, равно $O(m)$.

Обладая возможностью проверять за время $O(m)$ на ложные совпадения, алгоритм Карпа–Рабина преобразуется из метода с малой вероятностью ошибки, считающего за время $O(m)$ в наихудшем случае, в метод, который не делает ошибок, но считает за ожидаемое время $O(m)$ (преобразование алгоритма Монте–Карло в алгоритм Лас–Вегаса). Для этого нужно заново пропускать алгоритм Карпа–Рабина и заново делать проверку, пока не обнаружится, что ложных совпадений нет. Мы оставляем подробности на упражнение.

4.4.3. Почему дактилограммы?

Метод дактилограммы Карпа–Рабина работает в наихудшем случае за линейное время, но с ненулевой (хотя и предельно малой) возможностью ошибки. В качестве варианта он может рассматриваться как метод, который ошибок не делает, но имеет линейное ожидаемое время. Однако мы видели несколько методов со временем, линейным в наихудшем случае, и без ошибок. Зачем же нам изучать метод Карпа–Рабина?

На этот вопрос есть три ответа. Во-первых, с практической точки зрения метод прост и может быть распространен на другие задачи, такие как двумерное совпадение образца нерегулярной формы — задача, более трудная для других методов. Во-вторых, метод сопровождается доказательствами, гарантирующими определенные свойства функционирования метода. Методы, родственные по духу дактилограммам (или фильтрам), появились до метода Карпа–Рабина, но, в отличие от него, не сопровождались теоретическим исследованием, и об их работе мало что было доказано. Однако наибольшая его привлекательность в том, что метод основан на совершенно иных идеях, чем методы с линейным временем, работающие без ошибок. Так что этот метод предложен потому, что главная цель книги — представить коллекцию идей, разнообразных по технике, алгоритмам и доказательствам.

4.5. Упражнения

- Сравните эмпирически метод *Shift-And* с методами, рассмотренными ранее. Изменяйте размеры P и T .
- Предложите развитие метода *agrep* на решение задачи о поиске “вхождения” образца P в текст T , когда разрешается небольшое число включений и исключений символов, а также несовпадений. То есть символы в P можно включать и исключать.

3. Адаптируйте *Shift-And* и *agrep* к работе с множеством образцов. Сможете ли вы сделать что-нибудь получше, чем просто работа с каждым образцом отдельно?
4. Докажите корректность метода *agrep*.
5. Покажите, как эффективно работать с джокерами (и в образце и в тексте) в подходе *Shift-And*. Сделайте то же для *agrep*. Покажите, что ни у какого метода эффективность не определяется числом джокеров в строках.
6. Преобразуйте метод *Shift-And*, чтобы он эффективно обрабатывал регулярные выражения, не использующие замыкание Клини. Сделайте то же для *agrep*. Объясните полезность этих вариантов методов для коллекций образцов биопоследовательностей, таких как в PROSITE.
7. Мы упоминали в упражнении 32 главы 3, что образцы PROSITE часто задают число повторов подобразца. Диапазоны такого типа легко можно обрабатывать за время $O(nm)$ методом поиска совпадений с регулярным выражением, предложенным в п. 3.6. Можно ли эффективно обрабатывать такие спецификации диапазона в методе *Shift-And* или в *agrep*? Ответ частично зависит от числа таких спецификаций в выражении.
8. Открытая проблема. Разработайте метод, основанный только на сравнениях, для счета совпадений за время $O(m \log m)$. Возможно, здесь помог бы детальный анализ метода FFT, из которого было бы видно, как комплексный анализ можно в случае счета совпадений заменить сравнениями символов.
9. Завершите доказательство следствия 4.4.3.
10. Метод случайных дактилограмм может быть перенесен на задачу поиска двумерного образца, рассмотренную в п. 3.5.3. Сделайте это.
11. Завершите детали и анализ превращения метода Карпа–Рабина из рандомизированного алгоритма в стиле метода Монте–Карло в рандомизированный алгоритм в стиле метода Лас–Вегас.
12. В методе проверки на ложные совпадения для метода Карпа–Рабина возможны улучшения. Например, этот метод может найти за время $O(m)$ все полосы, содержащие ложные совпадения. Объясните как. Кроме того, в некоторых случаях метод требует явной проверки вхождения P только для l_1 , а не для l_2 . Объясните когда и почему.

Часть II

Суффиксные деревья и их использование

Глава 5

Введение в суффиксные деревья

Суффиксное дерево — это структура данных, которая выявляет внутреннее строение строки более глубоко, чем основной препроцессинг, рассмотренный в п. 1.3. Суффиксные деревья можно использовать для решения задачи о точных совпадениях за линейное время (достигая той же границы для наихудшего случая, как в алгоритмах Кнута–Морриса–Пратта и Бойера–Мура), но их подлинное превосходство становится ясным при решении за линейное время многих строковых задач, более сложных, чем точные совпадения. Более того (как мы уточним в главе 9), суффиксные деревья наводят мост между задачами *точного* совпадения — центральной темой части I — и *неточного* совпадения — темой части III.

Классическим приложением для суффиксных деревьев служит задача *о подстроке*. Пусть задан текст T длины m . За препроцессное время $O(m)$, т. е. линейное, нужно подготовиться к тому, чтобы, получив неизвестную строку S длины n , за время $O(n)$ либо найти вхождение S в T , либо определить, что S в T не входит. Это значит, что допустим препроцессинг со временем, пропорциональным длине текста, но после этого поиск строки S должен выполняться за время, пропорциональное длине S , *независимо* от длины T . Эти границы достигаются применением суффиксного дерева. Оно строится для текста за время $O(m)$ в препроцессной стадии; а затем, используя это суффиксное дерево, алгоритм, получив строку длины n на входе, ищет ее за время $O(n)$.

Время $O(m)$ на препроцессинг и $O(n)$ на поиск в задаче о подстроке — это удивительный и крайне полезный результат. В обычных приложениях, после того как будет построено суффиксное дерево, вводится длинная последовательность строк-запросов, так что линейная оценка времени для каждого поиска очень важна. Такая оценка не достигается методами Кнута–Морриса–Пратта и Бойера–Мура; они предварительно обрабатывают каждую строку на вводе, а затем затрачивают в худшем

случае время $\Theta(m)$ на поиск строки в тексте. Так как m может быть огромно, по сравнению с n , эти алгоритмы будут непрактичны для любых текстов нетривиального размера.

Часто текст представляет собой фиксированное множество строк, например коллекцию STS или EST (см. пп. 3.5.1 и 7.10), так что задача о подстроке превращается в проверку того, не является ли введенная строка подстрокой одной из фиксированных строк. Суффиксные деревья прекрасно справляются и с этой задачей. При поверхностном рассмотрении случай многочисленных текстовых строк напоминает задачу о *словаре*, обсуждавшуюся в связи с алгоритмом Ахо–Корасика. И естественно ожидать, что здесь может использоваться этот алгоритм. Но он не решает задачу о подстроке за желаемое время, потому что определяет только, не является ли новая строка *полной* строкой из словаря, а здесь не исключается и подстрока словарной строки.

После описания алгоритмов некоторые их приложения и обобщения будут обсуждаться в главе 7. Затем в главе 8 будет представлен замечательный результат, *метод поиска наименьшего общего предка за константное время*. Этот метод сразу показывает полезность суффиксных деревьев, которая далее будет иллюстрироваться различными приложениями в главе 9. Некоторые из них, как уже говорилось, наводят мост на пути к неточным совпадениям; список приложений суффиксных деревьев продолжится в части III, в которой неточные совпадения будут главной темой.

5.1. Краткая история

Первый алгоритм для конструирования суффиксных деревьев за линейное время был предложен Вайннером [473] в 1973 году, хотя он использовал термин *дерево позиций* (a position tree). Другой алгоритм построения суффиксных деревьев за линейное время, более экономный по памяти, был предложен Мак-Крейгом [318] несколько лет спустя. Недавно Укконен [438] разработал принципиально иной алгоритм построения суффиксных деревьев за линейное время, который обладает всеми преимуществами алгоритма Мак-Крейга (и с правильной точки зрения может рассматриваться как его вариант), но допускает более простое истолкование.

Несмотря на то что прошло уже больше двадцати лет со времени первоначального результата Вайнера (который Кнута призывают именовать “алгоритмом 1973” [24]), суффиксные деревья еще не вошли в основную программу обучения информатике, и на их долю достается меньше внимания и использования, чем можно было бы ожидать. Вероятно, это связано с тем, что две исходные работы 1970-х годов имеют репутацию крайне трудных для понимания. Такая репутация вполне заслужена, но об этом можно сожалеть, так как обсуждаемые алгоритмы, при всей их нетривиальности, не сложнее многих других широко применяемых методов. При хорошей реализации они очень практичны и разнообразны в применении. Мы не знаем ни одной другой структуры данных (если не говорить о структурах, эквивалентных суффиксным деревьям), которая допускала бы эффективное решение столь широкого класса трудных строковых задач.

В главе 6 полностью излагаются работающие за линейное время алгоритмы Укконена и Вайнера, а затем кратко упоминается в укрупненном виде организация алгоритма Мак-Крейга и его отношение к алгоритму Укконена. Мы стремимся

к описанию каждого алгоритма в общем виде, выбирая простые, *неэффективные* реализации. Затем эти реализации постепенно улучшаются для достижения линейного времени счета. Мы уверены, что наше изложение и анализ, особенно для алгоритма Вайнера, много проще и яснее, чем в исходных работах, и надеемся, что это приведет к более широкому практическому применению суффиксных деревьев.

5.2. Основные определения

Описывая, как строить суффиксное дерево для произвольной строки, будем работать с общей строкой S длины m . Мы не используем обозначений P и T (занятых под образец и текст), так как к суффиксным деревьям обращаются в широком диапазоне приложений, где исходная строка иногда играет роль образца, иногда текста, иногда их обоих, а иногда ни того, ни другого. Как обычно, алфавит предполагается конечным и известным. После обсуждения алгоритмов суффиксного дерева для отдельной строки S эта конструкция будет обобщена на случай наборов строк.

Определение. Суффиксное дерево \mathcal{T} для m -символьной строки S — это ориентированное дерево с корнем, имеющее ровно m листьев, занумерованных от 1 до m . Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей, а каждая дуга помечена непустой подстрокой строки S (дуговой меткой). Никакие две дуги, выходящие из одной и той же вершины, не могут иметь пометок, начинающихся с одного и того же символа. Главная особенность суффиксного дерева заключается в том, что для каждого листа i конкатенация меток дуг на пути от корня к листу i в точности составляет (произносит) суффикс строки S , который начинается в позиции i . То есть этот путь произносит $S[i..m]$.

Например, суффиксное дерево для строки $habxas$ показано на рис. 5.1. Путь из корня к листу 1 произносит полную строку $S = habxas$, тогда как путь до листа 5 произносит суффикс as , который начинается в позиции 5 строки S .

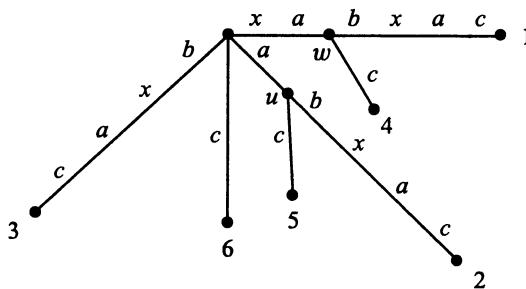


Рис. 5.1. Суффиксное дерево для строки $habxas$. Пометки u и w на двух внутренних вершинах понадобятся позднее

Как уже констатировалось, определение суффиксного дерева для S не гарантирует, что такое дерево действительно существует для любой строки S . Трудность в том, что если один суффикс совпадает с префиксом другого суффикса, то построить суффиксное дерево, удовлетворяющее данному выше определению, невозможно,

поскольку путь для первого суффикса не сможет закончиться в листе. Например, если удалить последний символ из строки $habxa$, образовав строку $habx$, то суффикс xa будет префиксом суффикса $habx$, так что путь, произносящий xa , не будет заканчиваться в листе.

Во избежание этой трудности предположим (это предположение выполняется на рис. 5.1), что последний символ S нигде больше в строку не входит. При таком условии никакой суффикс строки не сможет быть префиксом другого суффикса. Чтобы обеспечить это на практике, мы можем добавить в конце S какой-либо символ, не входящий в основной алфавит (из символов которого составлена строка). В этой книге в качестве такого “завершающего” символа мы используем $\$$. Когда важно подчеркнуть, что завершающий символ добавлен, будем писать его явно, например $S\$$. В большинстве же случаев это напоминание не будет необходимо, и, если противное явно не оговаривается, каждая строка S предполагается расширенной этим завершающим символом $\$$, даже если он явно и не выписан.

Суффиксное дерево сродни дереву ключей (без указателей назад), рассмотренному в п. 3.4. Пусть задана строка S . Если множество \mathcal{P} определено как множество из t суффиксов S , то суффиксное дерево для S может быть получено из дерева ключей для \mathcal{P} слиянием каждого пути из неветвящихся вершин в одну дугу. Простой алгоритм, предложенный в п. 3.4 для построения дерева ключей, можно использовать и для построения суффиксного дерева для S , но за время $O(m^2)$, а не за $O(m)$, которое мы получим ниже.

Определение. *Метка пути* от корня до некоторой вершины — это конкатенация подстрок, помечающих дуги пути в порядке их прохождения. *Путевая метка вершины* — это метка пути от корня \mathcal{T} до этой вершины.

Определение. Для любой вершины v суффиксного дерева *строковой глубиной* v называется число символов в путевой метке v .

Определение. Путь, который кончается в середине дуги (u, v) , делит метку (u, v) в своей точке назначения. Определим метку этого пути как путевую метку u с добавлением символов дуги (u, v) до делящей дуги точки назначения.

Например, на рис. 5.1 строка xa помечает внутреннюю вершину w (так что вершина w имеет путевую метку xa), строка a помечает вершину u , а строка $habx$ помечает путь, который кончается внутри дуги $(w, 1)$, т. е. внутри дуги, ведущей к листу 1.

5.3. Побуждающий пример

Прежде чем вдаваться в детали методов построения суффиксных деревьев, посмотрим, как суффиксное дерево для строки используется при решении задачи о точных совпадениях. Задан образец P длины n и текст T длины m , нужно найти все вхождения P в T за время $O(n + m)$. Мы уже видели несколько решений этой задачи. Суффиксные деревья дают другой подход.

Построим суффиксное дерево \mathcal{T} для текста T за время $O(m)$. Затем будем искать совпадения для символов из P вдоль единственного пути в \mathcal{T} до тех

пор, пока либо P не исчерпается либо очередное совпадение не станет невозможным. Во втором случае P в T не входит. В первом случае каждый лист в поддереве, идущем из точки последнего совпадения, имеет своим номером начальную позицию P в T , а каждая начальная позиция P в T нумерует такой лист.

Ключом к пониманию первого случая (когда все символы из P совпали с путем в T) служит такое наблюдение: P входит в T , начиная с позиции j , в том и только том случае, если P является префиксом $T[j..m]$. Но это происходит тогда и только тогда, когда P помечает начальную часть пути от корня до листа j . Именно по нему и проходит алгоритм сравнения.

Этот совпадающий путь единственен, так как никакие две дуги, выходящие из одной вершины, не имеют меток, начинающихся с одного и того же символа. И поскольку мы предположили, что алфавит конечен, работа в каждой вершине занимает константное время, а значит, время на проверку совпадения P с путем пропорционально длине P .

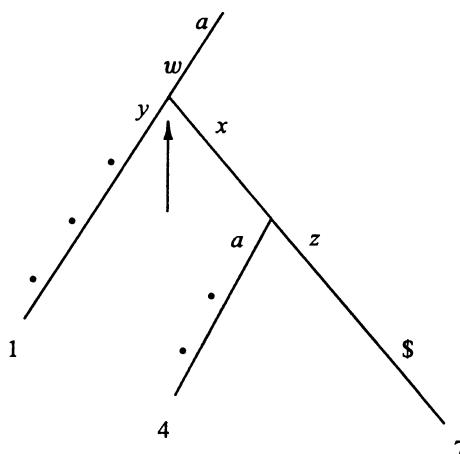


Рис. 5.2. Три вхождения aw в $awuareshawxz$. Их начальные позиции совпадают с номерами листьев в поддереве вершины с путевой меткой aw

На рис. 5.2 показан фрагмент суффиксного дерева для строки $T = awuareshawxz$. Образец $P = aw$ входит в T три раза, начинаясь в позициях 1, 4 и 7. Образец P совпадает с путем вниз до вершины, указанной стрелкой, и, как полагается, листья ниже этой точки имеют номера 1, 4 и 7.

Если P полностью совпадает с некоторым путем в дереве, алгоритм может найти все начальные позиции P в T проходом по поддереву вниз от конца совпавшего пути. При этом просмотре нужно просто собрать все номера встретившихся листьев. Таким образом, все вхождения P в T могут быть найдены за время $O(n + m)$. Такое же время достигается несколькими алгоритмами, рассмотренными в части I, но распределение работы между частями алгоритма здесь иное. Ранее предложенные алгоритмы тратили время $O(n)$ на препроцессинг P , а затем время $O(m)$ на поиск. Напротив, при использовании суффиксного дерева тратится время $O(m)$ на препроцессинг и $O(n + k)$ на поиск, где k — число вхождений P в T .

Чтобы собрать k начальных позиций P , обойдем поддерево из конца совпадающего пути, используя обход за линейное время (скажем, обход в глубину), и отметим встретившиеся номера листьев. Так как каждая внутренняя вершина имеет не меньше двух детей, число встретившихся листьев пропорционально числу пройденных дуг, так что время обхода равно $O(k)$, даже несмотря на то, что общая строковая глубина этих $O(k)$ дуг может быть сколь угодно больше k .

Если требуется найти только одно вхождение P , то можно чуть-чуть изменить предварительную обработку так, что поисковое время уменьшится с $O(n + k)$ до $O(n)$. Идея в том, чтобы записать в каждой вершине один номер (скажем, наименьший) листа в его поддереве. Это можно сделать на препроцессной фазе за время $O(m)$ обходом в глубину дерева \mathcal{T} . Детали очевидны и оставляются читателю. После такой подготовки в фазе поиска число, записанное в вершине или ниже конца совпадения, и дает начальную позицию P в T .

В п. 7.2.1 мы вернемся к рассмотрению относительных преимуществ методов, в которых предварительно готовится текст, перед методами, подготавливающими образец (или образцы). Позднее, в п. 7.8, мы покажем также, как использовать суффиксное дерево для решения задачи о точных совпадениях, затратив время $O(n)$ на подготовку и $O(m)$ на поиск и получив те же границы, что и в алгоритмах части I.

5.4. Наивный алгоритм построения суффиксного дерева

Чтобы еще подкрепить определение суффиксного дерева и развить у читателя интуицию, продемонстрируем алгоритм непосредственного построения суффиксного дерева для строки S . Этот наивный метод сначала помещает в дерево простую дугу для суффикса $S[1..m]\$$ (для всей строки); затем последовательно вводит в растущее дерево суффиксы $S[i..m]\$$ для i от 2 до m . Пусть N_i обозначает промежуточное дерево, в которое входят все суффиксы от 1 до i .

Подробнее, дерево N_1 состоит из одной дуги, идущей от корня дерева к вершине с номером 1. Эта дуга имеет метку $S\$$. Дерево N_{i+1} строится из дерева N_i следующим образом.

Начиная из корня N_i , найти самый длинный путь от корня, метка которого совпадает с префиксом строки $S[i + 1..m]\$$. Этот путь можно отыскать последовательным сравнением и поиском совпадений символов суффикса $S[i + 1..m]\$$ с символами вдоль единственного пути из корня до тех пор, пока не обнаруживается место, где очередного совпадения нет. Совпадающий путь единственен, так как никакие две дуги, выходящие из одной вершины, не могут иметь одинакового символа в начале их меток. В какой-то точке совпадения прекратятся, так как никакой суффикс $S\$$ не является префиксом другого суффикса $S\$$. Когда такая точка будет достигнута, то алгоритм останавливается либо в вершине, скажем, w , либо в середине какой-то дуги. Если в середине дуги, скажем, (u, v) , то он разбивает дугу (u, v) на две дуги, вводя новую вершину, которую мы обозначим через w , сразу после последнего совпадающего символа $S[i + 1..m]$ и перед первым несовпадающим символом. Новая дуга (u, w) имеет меткой часть метки (u, v) , которая совпадает с частью $S[i + 1..m]$, а новая дуга (w, v) помечена оставшейся частью метки (u, v) . Далее (независимо от того, создана вершина w заново или она уже

существовала) алгоритм создает новую дугу $(w, i + 1)$, идущую из w в новый лист с номером $i + 1$, и эта дуга помечается частью суффикса $S[i + 1..m]\$$, не нашедшей совпадения.

Теперь дерево содержит единственный путь из корня к листу $i + 1$, и этот путь имеет метку $S[i + 1..m]\$$. Заметим, что все дуги, выходящие из новой вершины w , имеют метки, отличающиеся первыми символами, и отсюда мы по индукции получаем, что никакие две дуги, выходящие из одной вершины, не имеют меток с одинаковыми первыми символами.

В обычном предположении ограниченного алфавита приведенный наивный метод строит суффиксное дерево для строки S длины m за время $O(m^2)$.

Глава 6

Построение суффиксных деревьев за линейное время

Мы подробно рассмотрим два метода построения суффиксных деревьев: метод Укконена и метод Вайнера. Вайнер первым показал возможность построения суффиксных деревьев за линейное время, и его метод излагается по причине его исторической значимости и ради некоторых оригинальных технических идей, которые в нем содержатся. Однако метод Укконена работает так же быстро, кроме того, при реализации он использует значительно меньше места (т. е. памяти), чем метод Вайнера. Следовательно, для большинства задач, где нужно строить суффиксное дерево, лучше выбрать метод Укконена. Мы уверены, что он также удобнее для изучения, и поэтому начнем с него. Читателю, который хочет освоить только один из методов, рекомендуется сосредоточиться на нем. Однако наше изложение метода Вайнера не зависит от понимания алгоритма Укконена, так что их можно изучать независимо (с одним небольшим общим пунктом, указанным при описании метода Вайнера).

6.1. Алгоритм Укконена для построения суффиксных деревьев за линейное время

Алгоритм, который изобрел Эско Укконен [438] для построения суффиксного дерева за линейное время, вероятно, самый простой из таких алгоритмов. По сравнению с алгоритмом Вайнера он имеет улучшение, экономящее память (которое впервые было введено при создании алгоритма Мак-Крейга), а кроме того, он обладает свойством “интерактивности”, которое иногда может оказаться полезным. Мы расскажем об этом свойстве тоже, но нужно подчеркнуть, что главное достоинство алгоритма Укконена — в простоте его описания, доказательства и анализа быстродействия. Эта

простота происходит оттого, что алгоритм можно представить сначала как простой, но не эффективный метод, который с помощью нескольких приемов реализации на уровне “здравого смысла” достигает уровня лучших алгоритмов по времени счета в наихудших условиях. Мы уверены, что этот не самый прямой путь изложения будет понятнее, потому что каждый шаг в нем легок для восприятия.

6.1.1. Неявные суффиксные деревья

Алгоритм Укконена строит последовательность *неявных* суффиксных деревьев, последнее из которых преобразуется в настоящее суффиксное дерево строки S .

Определение. *Неявное суффиксное дерево* строки S — это дерево, полученное из суффиксного дерева $S\$$ удалением всех вхождений терминального символа $\$$ из меток дуг дерева, удалением после этого дуг без меток и удалением затем вершин, имеющих меньше двух детей.

Неявное суффиксное дерево префикса $S[1..i]$ строки S аналогично получается из суффиксного дерева для $S[1..i]\$$ удалением символов $\$$, дуг и вершин, как описано выше.

Определение. Обозначим через \mathcal{T}_i неявное суффиксное дерево строки $S[1..i]$, где i меняется от 1 до m .

Неявное суффиксное дерево для любой строки S будет иметь меньше листьев, чем суффиксное дерево для строки $S\$$, в том и только том случае, если хотя бы один из суффиксов S является префиксом другого суффикса. Терминальный символ $\$$ был добавлен к концу S как раз во избежание этой ситуации. Однако если S заканчивается символом, который больше нигде в S не появляется, то неявное суффиксное дерево для S будет иметь лист для каждого суффикса и, следовательно, будет настоящим суффиксным деревом.

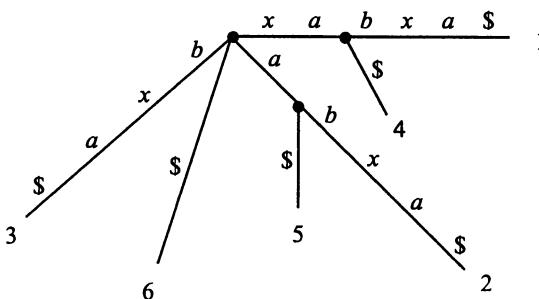


Рис. 6.1. Суффиксное дерево для строки $xabxa\$$

В качестве примера рассмотрим суффиксное дерево для строки $xabxa\$$, представленное на рис. 6.1. Суффикс xa является префиксом суффикса $xabxa$, и аналогично, строка a является префиксом $abxa$. Поэтому в суффиксном дереве для $xabxa$ дуги, ведущие к листьям 4 и 5, помечены только символом $\$$. Удаление этих дуг приводит к появлению двух вершин с одним потомком у каждой, и такие вершины

также должны быть удалены. Получающееся неявное суффиксное дерево для $habxa$ представлено на рис. 6.2. Другой пример демонстрируется на рис. 5.1 (с. 121), где построено дерево для строки $habxas$. Так как символ s появляется только в конце строки, дерево на рисунке является и суффиксным деревом, и неявным суффиксным деревом для этой строки.

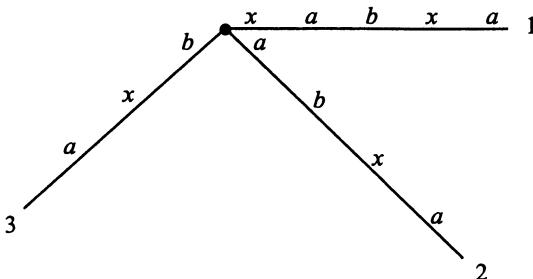


Рис. 6.2. Неявное суффиксное дерево для строки $habxa$

Хотя неявное суффиксное дерево может иметь листья не для всех суффиксов, в нем закодированы все суффиксы S — каждый произносится символами какого-либо пути от корня этого неявного суффиксного дерева. Однако если этот путь не кончается листом, то не будет маркера, обозначающего конец пути. Таким образом, неявные суффиксные деревья сами по себе несколько менее информативны, чем настоящие. Мы будем использовать их только как вспомогательное средство в алгоритме Укконена, чтобы получить настоящее суффиксное дерево для S .

6.1.2. Общее описание алгоритма Укконена

Алгоритм Укконена строит неявное суффиксное дерево \mathcal{T}_i для каждого префикса $S[1..i]$ строки S , начиная с \mathcal{T}_1 и увеличивая i на единицу, пока не будет построено \mathcal{T}_m . Настоящее суффиксное дерево для S получается из \mathcal{T}_m , и вся работа требует времени $O(m)$. Мы объясним алгоритм Укконена, представив сначала метод, с помощью которого все деревья \mathcal{T}_i строятся за время $O(m^3)$, а затем оптимизируем реализацию этого метода так, что будет достигнута заявленная скорость.

Общее описание алгоритма Укконена

Алгоритм Укконена делится на m фаз. В фазе $i + 1$ дерево \mathcal{T}_{i+1} строится из \mathcal{T}_i . Каждая фаза $i + 1$ сама делится на $i + 1$ продолжений (extensions), по одному на каждый из $i + 1$ суффиксов $S[1..i + 1]$. В продолжении j фазы $i + 1$ алгоритм сначала находит конец пути из корня, помеченного подстрокой $S[j..i]$. Затем он продолжает эту подстроку, добавляя к ее концу символ $S(i + 1)$, если только $S(i + 1)$ еще не существует. Итак, в фазе $i + 1$ сначала помещается в дерево строка $S[1..i + 1]$, а за ней строки $S[2..i + 1]$, $S[3..i + 1]$, ... (соответственно, в продолжениях 1, 2, 3, ...). Продолжение $i + 1$ фазы $i + 1$ продолжает пустой суффикс строки $S[1..i]$, т. е. включает в дерево строку из одного символа $S(i + 1)$ (если только ее там не было).

Дерево \mathcal{T}_1 состоит из одной дуги, помеченной символом $S(1)$. Формально алгоритм таков:

Общее описание алгоритма Укконена

Построить дерево \mathcal{T}_1 .

for i from 1 to $m - 1$ do begin {фаза $i + 1$ }

 for j from 1 to $i + 1$ begin {продолжение j }

 найти в текущем дереве конец пути из корня с меткой $S[j..i]$.

 Если нужно, продолжить путь, добавив символ $S(i + 1)$, обеспечив
появление строки $S[j..i + 1]$ в дереве.

 end;

end;

Правила продолжения суффиксов

Чтобы превратить это общее описание в алгоритм, мы должны точно указать, как выполнять *продолжение суффикса*. Пусть $S[j..i] = \beta$ — суффикс $S[1..i]$. В продолжении j , когда алгоритм находит конец β в текущем дереве, он продолжает β , чтобы обеспечить присутствие суффикса $\beta S(i + 1)$ в дереве. Алгоритм действует по одному из следующих трех правил.

Правило 1. В текущем дереве путь β кончается в листе. Это значит, что путь от корня с меткой β доходит до конца некоторой “листовой” дуги (дуги, входящей в лист). При изменении дерева нужно добавить к концу метки этой листовой дуги символ $S(i + 1)$.

Правило 2. Ни один путь из конца строки β не начинается символом $S(i + 1)$, но по крайней мере один начинающийся оттуда путь имеется.

В этом случае должна быть создана новая листовая дуга, начинающаяся в конце β и помеченная символом $S(i + 1)$. При этом, если β кончается внутри дуги, должна быть создана новая вершина. Листу в конце новой листовой дуги сопоставляется номер j .

Правило 3. Некоторый путь из конца строки β начинается символом $S(i + 1)$. В этом случае строка $\beta S(i + 1)$ уже имеется в текущем дереве, так что ничего не надо делать

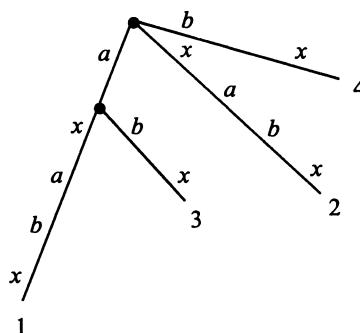


Рис. 6.3. Неявное суффиксное дерево для строки $axabx$ до того, как добавлен шестой символ b

(напомним, что в неявном суффиксном дереве конец суффикса не нужно помечать явно).

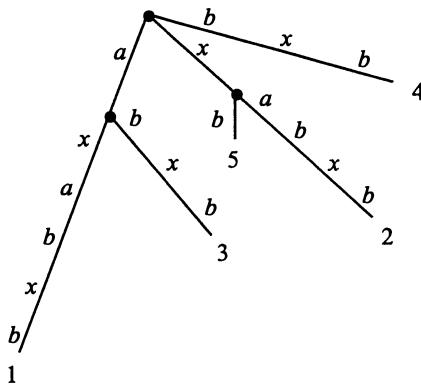


Рис. 6.4. Расширенное неявное суффиксное дерево после добавления символа b

В качестве примера рассмотрим на рис. 6.3 неявное суффиксное дерево для строки $S = axabx$. Первые четыре суффикса кончаются в листьях, а суффикс x , состоящий из одного символа, кончается внутри дуги. Когда к строке добавляется шестой символ, b , первые четыре суффикса получают продолжения по правилу 1, пятый — по правилу 2, а шестой — по правилу 3. Результат см. на рис. 6.4.

6.1.3. Реализация и ускорение

Будем считать, что используются алгоритмы, реализующие правила продолжения суффиксов. После того как конец суффикса β строки $S[1..i]$ будет найден в текущем дереве, на выполнение этих правил потребуется только константное время (чтобы обеспечить присутствие суффикса $\beta S(i+1)$ в дереве). Ключевым моментом в реализации алгоритма Укконена является определение концов всех $i+1$ суффиксов $S[1..i]$.

Действуя наивно, мы могли бы найти конец любого суффикса β за время $O(|\beta|)$, двигаясь от корня текущего дерева. При этом подходе продолжение j в фазе $i+1$ займет время $O(i+1-j)$, \mathcal{T}_{i+1} будет создано из \mathcal{T}_i за $O(i^2)$ и \mathcal{T}_m получится за время $O(m^3)$. Этот метод может показаться глуповатым, так как мы уже знаем алгоритм непосредственного построения суффиксного дерева за время $O(m^2)$ (и еще один рассматривается в упражнениях), но алгоритм Укконена со временем $O(m)$ легче описать как ускорение именно этого метода со временем $O(m^3)$.

Мы сведем $O(m^3)$ к $O(m)$ с помощью нескольких наблюдений и приемов реализации. Каждый прием сам по себе выглядит как полезная эвристика для ускорения алгоритма, и, действуя по отдельности, они не могут уменьшить оценку времени для наихудшего случая. Результат достигается только при совместном их применении. Наиболее существенный элемент ускорения — это использование *суффиксных связей*.

Суффиксные связи: первое ускорение реализации

Определение. Пусть $x\alpha$ обозначает произвольную строку, где x — ее первый символ, а α — оставшаяся подстрока (возможно, пустая). Если для внутренней вершины v с путевой меткой $x\alpha$ существует другая вершина $s(v)$ с путевой меткой α , то указатель из v в $s(v)$ называется *суффиксной связью*.

Мы будем иногда обозначать суффиксную связь из v в $s(v)$ парой $(v, s(v))$. Например, на рис. 6.1 (с. 127) пусть v — вершина с путевой меткой xa , а $s(v)$ — вершина с путевой меткой из одного символа a . Тогда существует суффиксная связь из v в $s(v)$. В этом примере строка α состоит из одного символа.

В особом случае, когда строка α пуста, суффиксная связь из внутренней вершины с путевой меткой $x\alpha$ идет в корневую вершину. Сама корневая вершина внутренней вершиной не является, и из нее суффиксные связи не выходят.

Хотя из определения суффиксных связей не следует, что из каждой внутренней вершины неявного суффиксного дерева выходит суффиксная связь, на самом деле так и будет. В действительности же из предложенных лемм и следствий получится даже более сильное утверждение.

Лемма 6.1.1. *Если новая внутренняя вершина v с путевой меткой $x\alpha$ добавляется к текущему дереву в продолжении j некоторой фазы $i + 1$, то либо путь, помеченный α , уже кончается во внутренней вершине текущего дерева, либо внутренняя вершина в конце строки α будет создана (по правилам продолжения) в продолжении $j + 1$ той же фазы $i + 1$.*

Доказательство. Новая внутренняя вершина v создается в продолжении j (фазы $i + 1$) только в случае, когда применяется правило продолжения 2. Это означает, что в продолжении j путь, помеченный $x\alpha$, продолжается каким-то символом, отличным от $S(i + 1)$, скажем, символом c . Таким образом, в продолжении $j + 1$ есть путь в дереве, помеченный α , и он, конечно, продолжается символом c (хотя, возможно, и другими символами тоже). Нужно рассмотреть два случая: либо путь, помеченный α , продолжается только символом c , либо помимо этого и каким-либо другим символом. В первом случае правило продолжения 2 создаст в конце пути α вершину $s(v)$. Если же α продолжается двумя различными символами, то в конце пути α уже должна быть вершина $s(v)$. Лемма доказана в любом случае. \square

Следствие 6.1.1. *В алгоритме Укконена любая вновь созданная внутренняя вершина будет иметь суффиксную связь с концом следующего продолжения.*

Доказательство. Доказательство ведется по индукции. Для дерева \mathcal{T}_1 утверждение верно, так как в нем внутренних вершин нет. Предположим, что утверждение оставалось верным вплоть до конца фазы i , и рассмотрим фазу $i + 1$. По лемме 6.1.1, когда новая вершина v создается в продолжении j , правильная вершина $s(v)$, кончавшая суффиксную связь из v , будет найдена или создана в продолжении $j - 1$. Но новая внутренняя вершина создается в последнем продолжении фазы (продолжении, которое вызывается суффиксом, состоящим из отдельного символа $S(i - 1)$). Поэтому все суффиксные связи из внутренних вершин, созданные на фазе $i - 1$, к концу фазы известны, и дерево \mathcal{T}_{i+1} содержит все свои суффиксные связи. \square

Следствие 6.1.1 аналогично теореме 6.2.5, которая появится при изучении алгоритма Вайнера и установит важный факт относительно неявных суффиксных деревьев и в конечном счете — относительно суффиксных деревьев. Чтобы подчеркнуть его, мы переформулируем следствие в несколько иных терминах.

Следствие 6.1.2. В любом неявном суффиксном дереве \mathcal{T}_i если внутренняя вершина v имеет путевую метку $x\alpha$, то найдется вершина $s(v)$ дерева \mathcal{T}_i с путевой меткой α .

По следствию 6.1.1 суффиксные связи будут исходить из всех внутренних вершин изменяющегося дерева, кроме внутренней вершины, введенной последней. Она получит свою суффиксную связь в конце следующего продолжения. Мы покажем, как суффиксные связи используются для ускорения вычислений.

Переходы по суффиксным связям при построении \mathcal{T}_{i+1}

Напомним, что в фазе $i + 1$ алгоритм находит место суффикса $S[j..i]$ строки $S[1..i]$ в продолжении j и j пробегает значения от 1 до $i + 1$. При наивном подходе строка $S[j..i]$ просматривается вдоль пути, идущего из корня в текущем дереве. Суффиксные связи могут сократить это движение по пути и каждое продолжение. Проще всего показать это на первых двух продолжениях (при $j = 1$ и $j = 2$) любой фазы $i + 1$.

Конец полной строки $S[1..i]$ должен быть в листе дерева \mathcal{T}_i , так как $S[1..i]$ — самая длинная строка в нем. Следовательно, конец этого суффикса найти легко (при конструировании деревьев мы можем все время поддерживать указатель на лист, соответствующий текущей полной строке $S[1..i]$), и его суффиксное продолжение строится по правилу 1. Таким образом, первое продолжение на любой фазе представляет собой особый случай, и на его построение достаточно константного времени, так как у алгоритма есть указатель на конец текущей полной строки.

Представим строку $S[1..i]$ в виде $x\alpha$, где x — один символ, а α — оставшаяся подстрока (возможно, пустая), и пусть $(v, 1)$ — дуга дерева, входящая в лист 1. Следующим действием алгоритма должно быть нахождение конца строки $S[2..i] = \alpha$ в текущем дереве, получающемся из \mathcal{T}_i . Здесь важно, что вершина v является либо корнем, либо внутренней вершиной \mathcal{T}_i . Если это корень, то для нахождения конца α алгоритм просто спускается в дереве по пути, помеченному α , как в наивном алгоритме. А если v — внутренняя вершина, то по следствию 6.1.2 (так как v входит в дерево \mathcal{T}_i) v имеет суффиксную связь с вершиной $s(v)$. Далее, поскольку $s(v)$ имеет путевой меткой префикс строки α , то конец строки α должен находиться в поддереве $s(v)$. Следовательно, при поиске конца α в текущем дереве алгоритм не должен проходить весь путь от корня, достаточно пройти путь от вершины $s(v)$. Это основная идея использования суффиксных связей в алгоритме.

Теперь опишем подробно второе продолжение. Пусть y обозначает дуговую метку дуги $(v, 1)$. Чтобы найти конец α , нужно пройти вверх от листа 1 до вершины v , далее по суффиксной связи из v в $s(v)$, а затем от $s(v)$ — вниз по пути (который может состоять больше чем из одной дуги) с меткой y . Конец этого пути и является концом α (рис. 6.5). В конце пути α дерево изменяется по правилам продолжения суффикса. Итак, первые два продолжения в фазе $i + 1$ описаны полностью.

При продолжении любой строки $S[j..i]$ до $S[j..i + 1]$ при $j > 2$ мы следуем той же общей идеи. Начиная в конце строки $S[j - 1..i]$ в текущем дереве, поднимаемся вверх не больше чем на одну вершину, попадая либо в корень, либо в вершину v , из которой выходит суффиксная связь. Далее, пусть y — дуговая метка этой дуги; в случае, если v не корень, проходим суффиксную связь из v в $s(v)$ и спускаемся

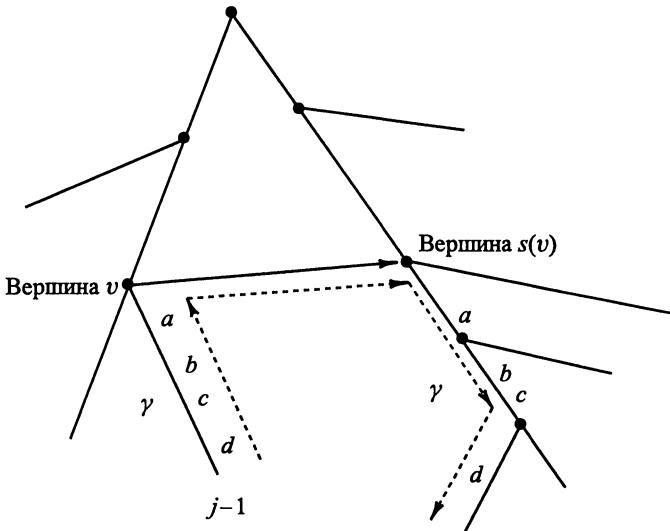


Рис. 6.5. Продолжение $j > 1$, в фазе $i + 1$. Поднимаемся вверх не более чем на одну дугу (с меткой γ) из конца пути с меткой $S[j - 1..i]$ к вершине v ; потом идем по суффиксной связи в $s(v)$; далее опускаемся по пути, определенному подстрокой γ ; затем используем соответствующее правило продолжения для включения суффикса $S[j..i + 1]$

по дереву из $s(v)$ по пути, помеченному γ , к концу $S[j..i]$. И наконец, продолжаем суффикс до $S[j..i + 1]$ по правилам продолжения.

Одно маленькое различие между продолжениями для $j > 2$ и первыми двумя все же есть. В общем случае конец $S[j - 1..i]$ может быть в вершине, из которой уже выходит суффиксная связь. Тогда алгоритм должен проходить эту связь. Отметим, что даже когда в продолжении $j - 1$ применяется правило продолжения 2 (так что концом $S[j - 1..i]$ является вновь созданная внутренняя вершина w), если родительская вершина для w не является корнем, то из нее уже выходит суффиксная связь, что обеспечивается леммой 6.1.1. Таким образом, в продолжении j алгоритм никогда не поднимается выше, чем на одну дугу.

Алгоритм отдельного продолжения (single extension algorithm)

Собирая вместе все эти фрагменты, реализующие использование суффиксных связей, можно записать продолжение $j \geq 2$ фазы $i + 1$.

Алгоритм отдельного продолжения: SEA

begin

- Найти в конце строки $S[j - 1..i]$ или выше его первую вершину v , которая либо имеет исходящую суффиксную связь, либо является корнем. Для этого нужно пройти вверх от конца $S[j - 1..i]$ в текущем дереве не более чем на одну дугу. Пусть γ обозначает строку между v и концом $S[j - 1..i]$ (возможно, пустую).

2. Если v не корень, пройти суффиксную связь из v в вершину $s(v)$ и спуститься из $s(v)$ по пути строки γ . Если v — корень, пройти по пути для $S[j..i]$ из корня (как в наивном алгоритме).
3. Обеспечить вхождение строки $S[j..i]S(i+1)$ в дерево, используя правила продолжения.
4. Если в продолжении $j-1$ была создана новая внутренняя вершина w (по правилу продолжения 2), то по лемме 6.1.1 строка α должна кончаться в вершине $s(w)$, конце суффиксной связи из w . Создать эту суффиксную связь $(w, s(w))$.

end.

В предположении, что алгоритм хранит указатель на текущую полную строку $S[1..i]$, первое продолжение фазы $i+1$ не требует переходов вверх или вниз. Более того, первое продолжение всегда использует правило продолжения суффикса 1.

Чего мы достигли?

Использование суффиксных связей дает очевидное практическое улучшение за счет сокращения передвижений от корня в каждом продолжении, которые выполняются в наивном алгоритме. Но улучшит ли это оценку времени в наихудшем случае?

Нет, как уже говорилось, еще не улучшит. Однако здесь мы введем особый прием, который уменьшит оценку для алгоритма до $O(m^2)$. Этот прием будет основным и в других алгоритмах построения и использования суффиксных деревьев.

Прием номер 1: скачок по счетчику

На шаге 2 продолжения $j+1$ алгоритм идет *вниз* из вершины $s(v)$ по пути с меткой γ . Напомним, что такой путь $s(v)$ обязательно должен быть. При буквальной реализации прохождение вдоль γ занимает время, пропорциональное $|\gamma|$, *числу символов* в этом пути. Но простой прием, именуемый *скакком по счетчику* (*skip/count*), уменьшит время перехода до приблизительно пропорционального *числу вершин* в пути. Отсюда будет следовать, что время на все проходы вниз за фазу не превзойдет $O(m)$.

Прием 1. Пусть g обозначает длину γ . Напомним, что никакие две метки дуг, выходящих из $s(u)$, не могут начинаться с одного и того же символа, так что первый символ γ должен быть начальным только у одной дуги, выходящей из $s(v)$. Пусть g' — число символов в этой дуге. Если $g' < g$, то алгоритму не нужно просматривать остальные символы дуги; он просто перепрыгивает к ее концу. Затем g полагается равным $g - g'$, $h = g' + 1$ и просматриваются выходящие дуги, чтобы найти среди них правильное продолжение (с начальным символом, равным символу h строки γ). Вообще, когда алгоритм идентифицирует следующую дугу пути, он сравнивает текущее значение g с числом символов на этой дуге g' . Когда g не меньше g' , алгоритм перескакивает к концу дуги, полагает g равным $g - g'$, h равным $h + g'$, а затем находит дугу, первый символ которой равен символу h строки γ , и этот шаг повторяется. По достижении дуги, в которой g не превосходит g' , алгоритм переходит к символу g на дуге и завершается, поскольку путь γ из $s(v)$ кончается на этой дуге, строго в g символах вниз по метке (рис. 6.6).

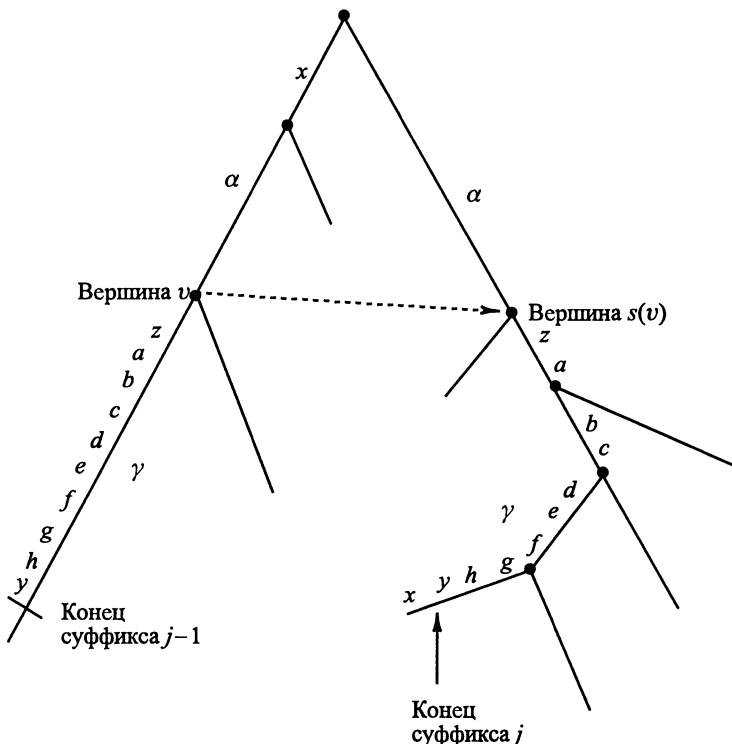


Рис. 6.6. Прием скачка по счетчику. В фазе $i + 1$ подстрока γ имеет длину десять. Из вершины $s(v)$ выходит копия подстроки γ ; ее конец найден в трех символах по последней дуге, после того как алгоритм проскочил четыре вершины

При простых и очевидных деталях реализации (таких, как знание числа символов на каждой дуге и возможность за констанное время извлечь символ строки S из любой заданной позиции) эффект от использования скачков по счетчику будет в обеспечении перехода от вершины пути γ к другой вершине за *константное время*^{*)}. Полное время прохода по пути становится при этом пропорциональным числу вершин, а не символов в нем.

Это полезная эвристика, но что с нее возьмешь в терминах оценок наихудшего случая? Следующая лемма прямо дает ответ.

Определение. Определим *вершинную глубину* вершины как число *вершин* на пути до нее от корня.

Лемма 6.1.2. Пусть $(v, s(v))$ — суффиксная связь, проходимая во время алгоритма Укконена. В этот момент вершинная глубина v не более чем на единицу превосходит вершинную глубину $s(v)$.

Доказательство. Когда проходится дуга $(v, s(v))$, любой внутренний предшественник v с путевой меткой, например $x\beta$, имеет суффиксную связь с вершиной,

^{*)} Мы по-прежнему предполагаем фиксированный размер алфавита.

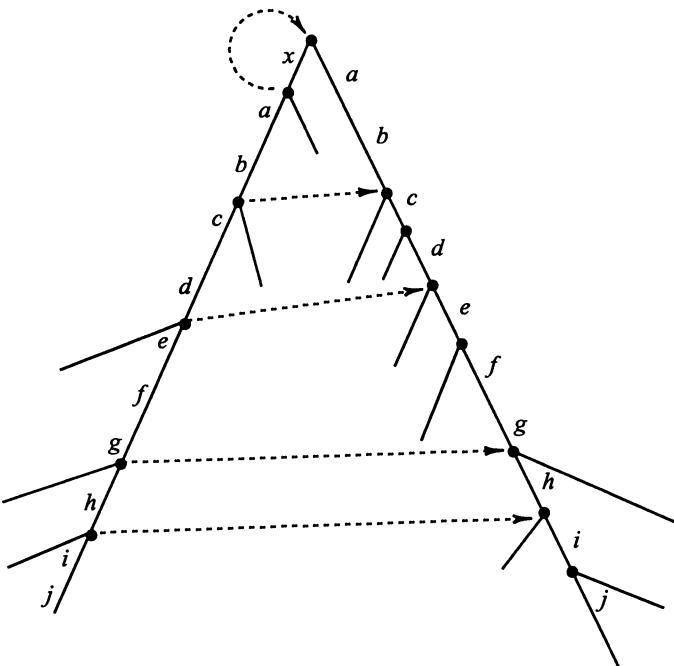


Рис. 6.7. Для каждой вершины v пути $x\alpha$ соответствующая вершина $s(v)$ лежит на пути α . Однако вершинная глубина $s(v)$ может быть на единицу меньше глубины v , может быть равна или быть больше. Например, вершина с меткой hab имеет глубину 2, а глубина ab равна 1. Глубина вершины с меткой $abcdefg$ равна 4, а вершины $abcdefg$ — 5

имеющей путевую метку β . Но $x\beta$ — это префикс пути в v , так что β — префикс пути в $s(v)$, и отсюда следует, что суффиксная связь из любого внутреннего предшественника v ведет к предшественнику $s(v)$. Более того, если строка β непустая, то вершина с меткой β — внутренняя. И так как вершинная глубина у двух различных предшественников v должна быть различной, каждый предшественник v имеет суффиксную связь с предшественником $s(v)$, и все предшественники различны. Отсюда следует, что вершинная глубина $s(v)$ не меньше чем 1 (корень) плюс число внутренних предшественников v , у которых путевые метки имеют длину больше 1. Единственный предшественник v , который может не иметь пары в предшественниках $s(v)$, — это внутренний предшественник с путевой меткой x длины 1. Поэтому вершинная глубина v не превосходит глубины $s(v)$ плюс 1 (рис. 6.7). \square

Определение. В ходе работы алгоритма за текущую вершинную глубину алгоритма принимается глубина последней по времени вершины, посещенной алгоритмом.

Теорема 6.1.1. При использовании скачков по счетчику любая фаза алгоритма Укконена занимает время $O(m)$.

Доказательство. В фазе i выполняется $i + 1 \leq m$ продолжений. В отдельном продолжении алгоритм поднимается не более чем на одну дугу, чтобы найти вершину с суффиксной связью, проходит одну суффиксную связь, опускается на некоторое

число вершин, применяет одно из правил продолжения суффиксов и, возможно, добавляет суффиксную связь. Мы уже установили, что все эти операции, кроме спуска по пути, имеют константное время, так что осталось проанализировать время на спуски. Для этого рассмотрим, как во время фазы меняется текущая вершинная глубина.

В каждом продолжении сначала подъем может уменьшить текущую глубину на единицу (если подъем произойдет), каждый проход по суффиксной связи может также уменьшить глубину на единицу (по лемме 6.1.2), а каждая дуга, проходимая при спуске, переводит алгоритм в вершину с большей глубиной. Таким образом, за всю фазу текущая глубина уменьшается не более $2m$ раз, и так как никакая вершина не имеет глубины больше m , то полное возможное приращение текущей глубины за всю фазу не превосходит $3m$. Отсюда следует, что за всю фазу общее число прохождений дуг во время спусков не превосходит $3m$. При использовании скачков по счетчику время спуска по дуге — константное, так что полное время фазы на спуски равно $O(m)$, и теорема доказана. \square

Всего у нас m фаз, так что верно следствие:

Следствие 6.1.3. *Суффиксные связи в алгоритме Укконена обеспечивают время работы $O(m^2)$.*

Заметим, что временная граница для алгоритма $O(m^2)$ получена умножением временной оценки отдельной фазы $O(m)$ на m (число фаз). Такое грубое умножение было необходимо, так как временной анализ проводился для отдельной фазы. Теперь нужны некоторые изменения в реализации, которые позволят провести временной анализ, выходящий за границы отдельных фаз. Вскоре изменения будут сделаны.

В этот момент читатель может несколько забеспокоиться, так как не увидит никакого прогресса: мы же начали с наивного метода трудоемкости $O(m^2)$. Зачем все эти усилия, чтобы прийти к той же временной оценке? Дело в том, что, хотя мы не достигли прогресса во временной оценке, наши концептуальные достижения очень велики, так что осталось еще чуть-чуть, чтобы время упало до $O(m)$. Нам понадобятся еще одна простая деталь реализации и два небольших приема.

6.1.4. Простая деталь реализации

Теперь мы получим временную оценку построения суффиксного дерева $O(m)$. Здесь немедленно возникает одна трудность: суффиксное дерево может занять в памяти место порядка $\Theta(m^2)$. Согласно тому описанию, которое мы до сих пор использовали, его пометки дуг могут содержать суммарно больше, чем $\Theta(m)$ символов. Поскольку время работы алгоритма не меньше, чем размер созданного им результата, такое число символов делает границу $O(m)$ невозможной. Рассмотрим строку $S = abcdefghijklmнопqrstuvхуз$. Каждый суффикс начинается со своего собственного символа; следовательно, из корня выходит 26 дуг, и каждая дуга имеет меткой весь суффикс, все эти метки занимают место $26 \times 27/2$ символов. Когда строки длиннее, чем алфавит, символы будут повторяться, но все равно можно построить строки произвольной длины m с суммарной длиной дуговых меток, превосходящей $O(m)$ символов. Таким образом, алгоритм построения суффиксных деревьев трудоемкости $O(m)$ требует иной схемы представления дуговых меток.

Сжатие дуговых меток

Есть простой вариант пометки дуг. Вместо явного выписывания подстроки достаточно записать только *пару индексов*, определяющих начальную и конечную позиции этой подстроки в S (рис. 6.8). Так как алгоритм располагает экземпляром строки S , он может определить каждый конкретный ее символ по позиции за константное время. Поэтому мы можем описать любой конкретный алгоритм суффиксного дерева, считая, что дуговые метки выписаны явно, но реализовывать его с постоянным числом символов, записанных на каждой дуге (индексную пару, показывающую начальную и конечную позиции подстроки).

$$S = abcde \mathbf{f} abcsiuw$$

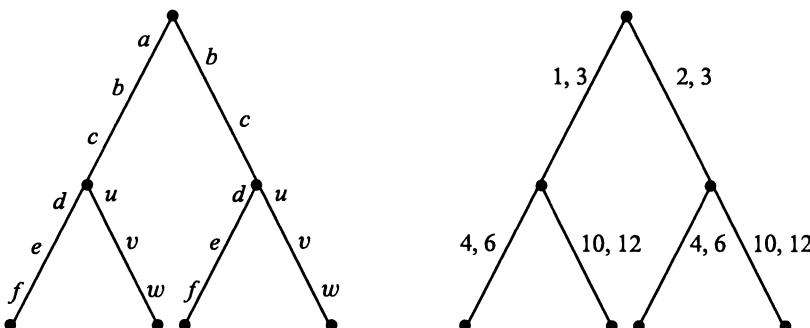


Рис. 6.8. Левое дерево — фрагмент суффиксного дерева для строки $S = abcdefabcsiuw$ с явно заданными дуговыми метками. Правое дерево показывает сжатые дуговые метки. Заметьте, что дуга с меткой 2, 3 могла бы иметь и метку 8, 9

Например, в алгоритме Укконена, когда идет сравнение вдоль дуги, индексная пара этой дуги используется для получения из S требуемых символов, и затем выполняются сравнения этих символов. Правила продолжения реализуются также с этой схемой пометок. Когда в фазе $i + 1$ применяется правило продолжения 2, то вновь создаваемая дуга помечается индексной парой $(i + 1, i + 1)$, а когда — правило продолжения 1 (на листовой дуге), индексная пара (p, q) заменяется на $(p, q + 1)$. Легко показать по индукции, что $q = i$ и, следовательно, новая метка $(p, i + 1)$ представляет правильную новую подстроку этой листовой дуги.

При наличии индексных пар каждой дуге сопоставляются только два числа, и так как число дуг не превосходит $2m - 1$, то суффиксное дерево использует память $O(m)$. Появляется надежда, что дерево можно построить за время $O(m)$. *) Хотя полностью реализованный алгоритм и не записывает подстроку на дуге, нам будет удобно говорить о “подстроке или метке дуги или пути” как о явно там записанных.

6.1.5. Еще два приема — и все сделано

Мы введем еще два приема реализации, которые вытекают из двух наблюдений взаимодействия правил продолжения в последовательных продолжениях и фазах. Эти приемы вместе с леммой 6.1.2 сразу дадут желаемую линейную оценку времени.

*) Мы делаем стандартное предположение о модели вычислений и считаем, что число с количеством битов до $\log m$ требует константного времени на чтение, запись и сравнение.

Наблюдение 1: правило 3 кончает дело. В любой фазе, если правило продолжения 3 применяется в продолжении j , оно будет реализовываться во всех дальнейших продолжениях (от $j + 1$ по $i + 1$) до конца фазы. Причина в том, что при использовании правила 3 путь, помеченный $S[j..i]$ в текущем дереве, должен продолжаться символом $S(i + 1)$, и точно так же продолжается путь, помеченный $S[j + 1..i]$, поэтому правило 3 применяется в продолжениях $j + 1, j + 2, \dots, i + 1$.

Когда используется правило продолжения 3, никакой работы делать не нужно, так как требуемый суффикс уже в дереве есть. Более того, новая суффиксная связь должна добавляться к дереву только после продолжения, в котором участвуется правило 2. Эти факты и наблюдение 1 приводят к следующему приему реализации.

Прием 2. Заканчивать каждую фазу $i + 1$ после первого же использования правила продолжения 3. Если это случится в продолжении j , то уже не требуется явно находить концов строк $S[k..i]$ с $k > j$.

Продолжения в фазе $i + 1$, которые “сделаны” после первого выполнения правила 3, будут называться *сделанными неявно*, в отличие от *явных* продолжений j , когда конец $S[j..i]$ находится впрямую.

Очевидно, что прием 2 сильно сокращает работу, но приведет ли он к уменьшению оценки времени, пока не понятно. Для выяснения нам понадобятся еще одно наблюдение и еще один прием.

Наблюдение 2: стал листом — листом и останешься. То есть если в какой-то момент работы алгоритма Укконена будет создан лист с меткой j (для суффикса, начинающегося в позиции j строки S), он останется листом во всех последовательных деревьях, созданных алгоритмом. Это верно потому, что у алгоритма нет механизма для продолжения листовой дуги дальше текущего листа. Более подробно, если есть лист с меткой j , правило продолжения 1 будет применяться для продолжения j на всех последующих фазах. Итак, стал листом — листом и останешься.

Далее, лист 1 создан в фазе 1, так что в каждой фазе i есть исходная последовательность продолжений (начинающаяся с продолжения 1), где применяются правила 1 и 2. Пусть j_i обозначает последнее продолжение в этой последовательности. Так как любое использование правила 2 создает новый лист, из наблюдения 2 следует, что $j_i \leq j_{i+1}$. Это значит, что исходная последовательность продолжений, в которых применяются правила 1 и 2, не может “усыхать” в последовательных фазах. Отсюда следует прием реализации, который в фазе $i + 1$ минует явные продолжения от 1 до j_i . На их неявную замену потребуется только константное время.

Чтобы описать этот прием, вспомним, что метку любой дуги неявного суффиксного дерева (и суффиксного дерева также) можно представить парой индексов (p, q) , задающей подстроку $S[p..q]$. Вспомним также, что для любой листовой дуги \mathcal{T}_i индекс q равен i и в фазе $i + 1$ индекс q возрастает до $i + 1$, отражая добавление символа $S(i + 1)$ к концу каждого суффикса.

Прием 3. В фазе $i + 1$, когда создается листовая дуга, которая нормально помечается подстрокой $S[p..i + 1]$, вместо записи индексов этой дуги $(p, i + 1)$ использовать запись (p, e) , где e — символ, обозначающий “текущий конец”. Символ e — глобальный индекс, которому присваивается значение $i + 1$ в начале фазы. Благодаря этому в фазе $i + 1$, когда алгоритм узнает, что в продолжениях с 1 до j_i будет применено правило 1, в первых j_i продолжениях уже не потребуется никакой дополнительной работы. Останутся только константная работа на увеличение переменной e и некоторая явная работа с продолжениями, начиная с продолжения $j_i + 1$.

Кульминация

При использовании приемов 2 и 3 явные продолжения в фазе $i + 1$ (применяющие алгоритм SEA) требуются только от продолжения $j_i + 1$ до первого продолжения, где используется правило 3 (или до продолжения $i + 1$). Все другие продолжения (до и после этих явных продолжений) выполняются неявно. В итоге фаза $i + 1$ реализуется так:

Алгоритм одной фазы (single phase algorithm): SPA

begin

1. Увеличить индекс e до $i + 1$. (С помощью приема 3 при этом увеличиваются все неявные продолжения от 1 до j_i .)
2. Явно вычислить последовательные продолжения (используя алгоритм SEA), начиная от $j_i + 1$ и до достижения первого продолжения j^* , где применяется правило 3, или до достижения конца этой фазы. (С помощью приема 2 правильно выполняются все продолжения от $j^* + 1$ до $i + 1$.)
3. Приготовляясь к следующей фазе, положить j_{i+1} равным $j^* - 1$.

end.

Шаг 3 устанавливает значение j_{i+1} корректно, так как исходная последовательность продолжений, в которых используются правила 1 и 2, должна заканчиваться в точке, где впервые применяется правило 3.

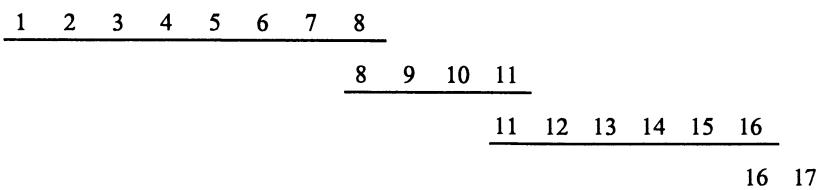


Рис. 6.9. Схема возможного исполнения алгоритма Укконена. Каждая линия представляет фазу алгоритма, а каждое число — исполняемое явно продолжение. Здесь показано четыре фазы и семнадцать явных продолжений. В любых двух последовательных фазах найдется не более одного продолжения, которое явно исполняется в обеих фазах

Ключевая особенность алгоритма SPA в том, что фаза $i + 2$ начнет вычислять явные продолжения с продолжения j^* , где j^* — последнее явное продолжение, рассчитанное в фазе $i + 1$. Поэтому две последовательные фазы имеют *не более одного общего индекса (j^*)* с явным вычислением продолжения (рис. 6.9). Более того, фаза $i + 1$ завершается, отмечая, где кончается строка $S[j^*..i + 1]$, так что повтор продолжения j^* в фазе $i + 2$ может выполнять правило продолжения суффикса для j^* без шага наверх, перехода по суффиксной связи и пропуска вершин. Это означает, что первое явное продолжение в любой фазе требует константного времени. Теперь легко доказать основной результат.

Теорема 6.1.2. Используя суффиксные связи и реализацию приемов 1, 2 и 3, алгоритм Укконена строит неявные суффиксные деревья от \mathcal{T}_1 до \mathcal{T}_m за полное время $O(m)$.

Доказательство. Время на все неявные продолжения каждой фазы константное и в пересчете на весь алгоритм оценивается как $O(m)$.

Рассмотрим индекс \bar{j} , соответствующий продолжению, которое алгоритм явно исполняет. За все время работы алгоритма \bar{j} никогда не убывает и остается неизменным при переходе от фазы к фазе. Так как фаз только m и $\bar{j} \leq m$, алгоритм делает явно не более $2m$ продолжений. Как установлено ранее, время на явное продолжение равно константе плюс время, пропорциональное числу прыжков через вершины при спуске по дереву во время этого расширения.

Чтобы ограничить общее число таких прыжков во всех спусках, посмотрим (аналогично доказательству теоремы 6.1.1), как в последовательных продолжениях изменяется вершинная глубина, в том числе при переходе от фазы к фазе. Дело в том, что первое явное продолжение в любой фазе (после первой) начинается с продолжения j^* , которое было последним явным продолжением предыдущей фазы. Поэтому текущая вершинная глубина не меняется при переходе от фазы к фазе. Но (как уточнялось в доказательстве теоремы 6.1.1) в каждом явном продолжении текущая вершинная глубина сначала уменьшается не более чем на два (подъем на одну дугу и переход по одной суффиксной связи), а затем спуск в этом продолжении увеличивает глубину на единицу при каждом скачке. Так как максимальная вершинная глубина равна m и есть только $2m$ явных продолжений, отсюда следует (как в доказательстве теоремы 6.1.1), что максимальное число скачков во время всех спусков (а не только отдельной фазы) имеет порядок $O(m)$. Все сосчитано, и теорема доказана. \square

6.1.6. Создание настоящего суффиксного дерева

Окончательное неявное суффиксное дерево \mathcal{T}_m можно преобразовать в истинное суффиксное дерево за время $O(m)$. Для этого нужно добавить терминальный символ $\$$ к концу S и выполнить шаг алгоритма Укконена с этим символом. В результате никакой суффикс не будет префиксом любого другого суффикса и получится неявное суффиксное дерево, в котором каждый суффикс будет кончаться листом и которое будет явно представлено. Единственная необходимая доработка — замена каждого индекса e у всякой листовой дуги числом m . Такая замена выполняется обходом дерева, требующим времени $O(m)$. После выполнения указанных модификаций получится настоящее суффиксное дерево. Итак:

Теорема 6.1.3. Алгоритм Укконена строит настоящее суффиксное дерево для S и все его суффиксные связи за время $O(m)$.

6.2. Алгоритм Вайнера для построения суффиксных деревьев за линейное время

В отличие от алгоритма Укконена, алгоритм Вайнера начинает со всей строки S . Однако, подобно алгоритму Укконена, он вводит в растущее дерево каждый раз по одному суффиксу, хотя в совсем ином порядке. Именно, сначала в дерево вводится строка $S(m)\$$, затем $S[m-1..m]\$$, ... и наконец вся строка $S\$$.

Определение. $Suff_i$ обозначает суффикс $S[i..m]$ строки S , начинающийся с позиции i .

Например, $Suff_1$ — это вся строка S , а $Suff_m$ — отдельный символ $S(m)$.

Определение. Пусть \mathcal{T}_i — дерево, которое включает в себя $m - i + 2$ листьев, перенумерованных от i до $m + 1$, и такое, что путь от корня до любого листа j ($i \leq j \leq m + 1$) имеет метку $Suff_j \$$. То есть \mathcal{T}_i — дерево, кодирующее все суффиксы строки $S[i..m]\$$ (и только их), так что это суффиксное дерево строки $S[i..m]\$$.

Алгоритм Вайнера строит деревья от \mathcal{T}_{m+1} вниз к \mathcal{T}_1 (т. е. по убыванию i). Сначала реализуем этот метод неэффективным непосредственным способом. Он поможет нам ввести важные определения и факты и проиллюстрировать их. Затем ускорим эту конструкцию и получим алгоритм Вайнера с линейным временем.

6.2.1. Непосредственное построение

Первое дерево \mathcal{T}_{m+1} состоит из одной дуги, выходящей из корня и помеченной терминальным символом $\$$. Поэтому для каждого i от m до 1 алгоритм строит дерево \mathcal{T}_i по дереву \mathcal{T}_{i+1} и символу $S(i)$. Идея метода в основном такая же, как при построении дерева ключей (п. 3.4), но для другого множества строк и без определения обратных ссылок. Во время работы алгоритма каждое дерево \mathcal{T}_i будет обладать тем свойством, что никакие две дуги, выходящие из одной вершины v в \mathcal{T}_{i+1} , не будут иметь меток, начинающихся с одной и той же буквы. Так как \mathcal{T}_{m+1} состоит только из одной дуги, для него это свойство выполняется. Для индукции предположим, что \mathcal{T}_{i+1} таким свойством обладает, и докажем, что и \mathcal{T}_i — тоже.

В общем случае, чтобы получить \mathcal{T}_i из \mathcal{T}_{i+1} , мы начинаем с корня \mathcal{T}_{i+1} и идем, сколько возможно, вниз по пути, метка которого совпадает с префиксом $Suff_i \$$. Обозначим этот путь через R . Если говорить подробнее, путь R находят, начиная с корня, явным последовательным сравнением символов $Suff_i \$$ с символами вдоль единственного пути в \mathcal{T}_{i+1} . Этот путь совпадения единственен из-за того, что в любой вершине v дерева \mathcal{T}_{i+1} метки никаких двух выходящих дуг не начинаются с одного символа. Поэтому совпадение может быть не более чем на одной выходящей дуге. В конце концов, так как никакой суффикс не является префиксом другого суффикса, процесс попадет в точку, где совпадений нет. Если в этой точке вершины нет, мы создадим новую вершину. Обозначим вершину остановки (старую или новую) через w . Наконец, добавим дугу, идущую из w в новый лист с меткой i , и пометим дугу (w, i) оставшейся (несовпавшей) частью $Suff_i \$$. Так как дальнейшее совпадение было невозможно, первый символ этой метки будет отличен от первого символа любой другой дуги, выходящей из w . Таким образом, объявленное индуктивное свойство сохраняется. Ясно, что путь из корня к листу i имеет метку $Suff_i \$$. Это значит, что путь произносит строку $Suff_i \$$, и дерево \mathcal{T}_i готово.

Например, рис. 6.10 показывает преобразование \mathcal{T}_2 в \mathcal{T}_1 для строки tat .

Определение. Для любой позиции i обозначим $Head(i)$ самый длинный префикс $S[i..m]$, который совпадает с подстрокой $S[i + 1..m]\$$.

Заметим, что $Head(i)$ может быть пустой строкой. Действительно, строка $Head(m)$ всегда пуста, так как строка $S[i + 1..m]$ пуста при $i + 1 > m$ и $S(m) \neq \$$.

Поскольку копия строки $Head(i)$ начинается с какой-то позиции между $i + 1$ и m , то $Head(i)$ является префиксом $Suff_k$ для некоторого $k > i$. Отсюда следует, что $Head(i)$ — это самый длинный префикс (возможно, пустой) строки $Suff_i$, являющейся меткой некоторого пути из корня в дереве \mathcal{T}_{i+1} .

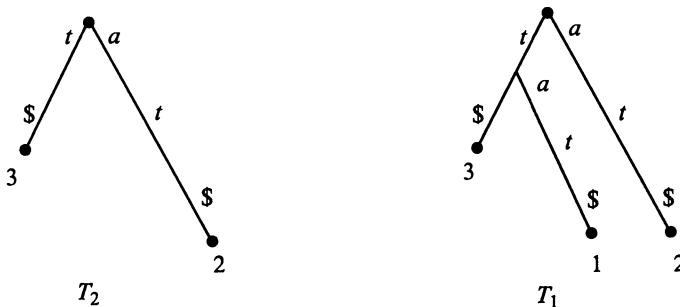


Рис. 6.10. Шаг наивного алгоритма Вайнера. Полная строка tat добавлена к суффиксному дереву для at . Дуга с меткой из одного символа $\$$ не показана, поскольку такая дуга входит в каждое суффиксное дерево

Приведенный алгоритм непосредственного построения \mathcal{T}_i из \mathcal{T}_{i+1} можно записать так:

Наивный алгоритм Вайнера

1. Найти конец пути с меткой $Head(i)$ в дереве \mathcal{T}_{i+1} .
2. Если в конце $Head(i)$ вершины нет, то создать ее. Пусть w обозначает эту вершину (новую или старую). Если w создана, то расщепить существовавшую дугу и ее метку так, чтобы w имела вершинную метку $Head(i)$. Создать новый лист с номером i и новую дугу (w, i) , помеченную оставшимися символами $Suff_i \$$. Это значит, что новая метка содержит последние $m - i + 1 - |Head(i)|$ символов $Suff_i$ и терминальный символ $\$$.

6.2.2. Путь к более эффективной реализации

Этот непосредственный алгоритм строит окончательное суффиксное дерево $\mathcal{T} = \mathcal{T}_i$ за время $O(m^2)$. Ясно, что трудоемкая часть этого алгоритма заключается в нахождении $Head(i)$, так как шаг 2 для любого i занимает константное время. Таким образом, для ускорения алгоритма нужно находить $Head(i)$ более эффективно. Но, как и в случае алгоритма Укконена, линейное время невозможно при явной записи дуговых меток, так что мы переходим к индексной записи, в которой каждая метка представляется парой позиций, помечающей подстроки, — начальной и конечной (см. п. 6.1.4).

Алгоритм Вайнера легко реализуется, если метки представлены на дугах индексными парами. При включении в дерево $Suff_i$ предположим, что алгоритм нашел совпадение вплоть до k -го символа дуги (u, z) , помеченной интервалом $[s, t]$, а следующий символ не совпал. Создаваемая новая вершина w делит (u, z) на две дуги, (u, w) и (w, z) . Строится и новая дуга, идущая из w в лист i . Дуга (u, w) получает метку $[s, s + k - 1]$, дуга (w, z) — метку $[s + k, t]$, а дуга (w, i) — метку $[i + d(w), m]\$$, где $d(w)$ — строковая глубина (число символов) пути от корня до вершины w . Эти строковые глубины легко находить и обеспечивать их правильные значения, так как $d(w) = d(u) + k$. Строковая глубина листа i равна $m - i + 1$.

Эффективное нахождение $Head(i)$

Обратимся теперь к основному вопросу — как эффективно находить $Head(i)$. Ключевой момент алгоритма Вайнера — использование двух векторов, сопоставляемых каждой нелистовой вершине (включая корень). Первый вектор называется *индикаторным вектором* I , а второй — *вектором связей* L (от link). Оба вектора имеют длину в размер алфавита и индексируются его символами. Например, для английского алфавита, дополненного символом \$, оба вектора имеют длину 27.

Вектор связей — конструкция, обратная суффиксной связи в алгоритме Укконена; эти два типа связей и используются сходным образом, чтобы ускорять переходы внутри дерева.

Индикаторный вектор — это битовый вектор, так что его элементы принимают значения 0 и 1, тогда как каждый элемент вектора связей — либо указатель на вершину дерева, либо null (*пустой указатель*). Пусть $I_v(x)$ — элемент индикаторного вектора, а $L_v(x)$ — элемент вектора связей в вершине v , соответствующие символу x (который рассматривается как индекс в этих массивах).

Векторы I и L имеют два решающих свойства, которые индуктивно соблюдаются во время работы алгоритма:

- $I_u(x) = 1$ для любого символа x и любой вершины u в \mathcal{T}_{i+1} в том и только том случае, если в дереве \mathcal{T}_{i+1} существует путь из корня с меткой xa , где a — путевая метка вершины u . Путь, помеченный xa , не обязан кончаться в вершине.
- $L_u(x)$ для любого символа x указывает в \mathcal{T}_{i+1} на (внутреннюю) вершину \bar{u} в том и только том случае, если \bar{u} имеет путевую метку xa , где u имеет путевую метку a . В противном случае $L_u(x)$ равно null.

Например, рассмотрим в дереве на рис. 5.1 (с. 121) две внутренние вершины u и w с путевыми метками a и xa соответственно. Тогда $I_u(x) = 1$ для указанного символа x и $L_u(x) = w$. Кроме того, $I_w(b) = 1$, но $L_w(b)$ равно null.

Очевидно, что $L_u(x)$ для любой вершины u и любого символа x имеет значение не null, только если $I_u(x) = 1$, но обратное неверно. Ясно также, что если $I_u(x) = 1$, то $I_v(x) = 1$ для любой вершины v , предшествующей u .

Дерево \mathcal{T}_m имеет только один нелист, именно, корень r . Положим в этой вершине $I_r(S(m)) = 1$, $I_r(x) = 0$ для всех остальных символов x , а все элементы вектора связей примем равными null. Тогда указанные свойства для \mathcal{T}_m будут выполняться. Алгоритм поддерживает изменение векторов при изменении дерева, и мы докажем по индукции, что этими свойствами обладают все остальные деревья.

6.2.3. Основная идея алгоритма Вайнера

Алгоритм Вайнера использует для более эффективного поиска $Head(i)$ и построения \mathcal{T}_i индикаторные векторы и векторы связей. Ему приходится иметь дело с двумя вырожденными случаями, которые, однако, не сильно отличаются от общего “хорошего” случая. Начнем с построения \mathcal{T}_i по \mathcal{T}_{i+1} в хорошем случае, а потом разберемся с вырожденными.

Алгоритм для хорошего случая

Предположим, что дерево \mathcal{T}_{i+1} уже построено, и мы начинаем строить \mathcal{T}_i . Алгоритм стартует с листа $i + 1$ (это лист для $Suff_{i+1}$) и идет к корню, разыскивая первую вершину v , для которой $I_v(S(i)) = 1$, если такая вершина существует. Если она найдена,

то путь продолжается от v наверх в поисках первой вершины v' с ненулевым значением $L_{v'}(S(i))$ (это может быть и v). По определению $L_{v'}(S(i))$ не равно null, только если $I_{v'}(S(i)) = 1$, так что если вершина v' нашлась, то это первая вершина на пути от листа $i + 1$, для которой $L_{v'}(S(i))$ не равно null. В общем случае может не найтись ни v , ни v' или v найдется, а v' нет. Отметим, однако, что v или v' может быть корнем.

“Хороший” случай — это когда *обе* вершины v и v' существуют.

Пусть l_i — число символов на пути между v' и v . При $l_i > 0$ обозначим через c первый из этих l_i символов.

Предполагая, что мы имеем дело с хорошим случаем, и значит, и v и v' существуют, докажем, что если вершина v имеет путевую метку α , то строка $Head(i)$ совпадает с $S(i)\alpha$. Далее мы докажем, что когда $L_{v'}(S(i))$ указывает на вершину v'' дерева \mathcal{T}_{i+1} , то $Head(i)$ либо кончается в v'' , если $l_i = 0$, либо кончается в l_i символах ниже v'' на дуге, выходящей из v'' . Таким образом, в любом случае после нахождения v' за константное время найдем и $Head(i)$.

Теорема 6.2.1. *Предположим, что алгоритм нашел вершину v и ее путевая метка равна α . Тогда строка $Head(i)$ в точности равна $S(i)\alpha$.*

Доказательство. $Head(i)$ — это самый длинный префикс $Suff_i$; он является также префиксом $Suff_k$ для некоторого $k > i$. Так как v было найдено с условием $I_v(S(i)) = 1$, то в \mathcal{T}_{i+1} имеется путь, который начинается с $S(i)$, и $Head(i)$ имеет длину по меньшей мере в один символ. Поэтому мы можем представить $Head(i)$ как $S(i)\beta$ для некоторой (возможно, пустой) строки β .

Обе строки $Suff_i$ и $Suff_k$ начинаются с $Head(i) = S(i)\beta$ и дальше различаются. Пусть, для конкретности, начало $Suff_i$ представлено как $S(i)\beta a$, а начало $Suff_k$ как $S(i)\beta b$. Но тогда $Suff_{i+1}$ начинается с βa , а $Suff_{k+1}$ — с βb . Как $i + 1$, так и $k + 1$ лежат между $i + 1$ и m , так что оба суффикса уже присутствуют в дереве \mathcal{T}_{i+1} . Поэтому в дереве \mathcal{T}_{i+1} должен быть путь из корня с меткой β (возможно, это пустая строка), который продолжается двумя способами: одно продолжение начинается с символа a , а другое — с символа b . Следовательно, в дереве \mathcal{T}_{i+1} существует вершина u с путевой меткой β , и $I_u(S(i)) = 1$, так как в \mathcal{T}_{i+1} есть путь (именно, начальная часть пути к листу k) с меткой $S(i)\beta$. Далее, вершина u должна лежать на пути к листу $i + 1$, поскольку β является префиксом для $Suff_{i+1}$.

Теперь, $I_v(S(i)) = 1$ и v имеет путевую метку α , поэтому $Head(i)$ должно начинаться с $S(i)\alpha$. Это означает, что α является префиксом β , так что вершина u с путевой меткой β должна совпадать с v либо лежать ниже v на пути к листу $i + 1$. Значит, если $u \neq v$, то u должна лежать ниже v на пути к листу $i + 1$, и $I_u(S(i)) = 1$. Налицо противоречие выбору вершины v , следовательно, $v = u$, $\alpha = \beta$, и теорема доказана. Итак, $Head(i)$ в точности равно строке $S(i)\alpha$. \square

Заметим, что в теореме 6.2.1 и ее доказательстве мы предполагали только существование вершины v . Никаких предположений относительно v' не делалось. Это обстоятельство пригодится в одном из вырожденных случаев, обсуждаемых ниже.

Теорема 6.2.2. *Предположим, что найдены u и v и v' , и $L_{v'}(S(i))$ указывает на вершину v'' . Если $l_i = 0$, то строка $Head(i)$ кончается в v'' , а в противном случае — после l_i символов на дуге, идущей из v'' .*

Доказательство. Поскольку v' лежит на пути к листу $i + 1$ и $L_{v'}(S(i))$ указывает на вершину v'' , то путь из корня с меткой $Head(i)$ должен включать в себя v'' . По теореме 6.2.1 $Head(i) = S(i)\alpha$, так что строка $Head(i)$ должна кончаться ровно через l_i символов после v'' . Таким образом, когда $l_i = 0$, строка $Head(i)$ кончается в v'' . Но когда $l_i > 0$, из v'' должна выходить дуга $e = (v'', z)$, метка которой начинается с символа c , первого из l_i символов на пути из v' в v в дереве \mathcal{T}_{i+1} .

Можно ли продолжить $Head(i)$ ниже до вершины z (т. е. до вершины ниже v'')? Вершина z должна быть ветвящейся, поскольку если бы она была листом, то какой-нибудь суффикс $Suff_k$, где $k > i$, был бы префиксом $Suff_i$, что невозможно. Пусть z имеет путевую метку $S(i)\gamma$. Если $Head(i)$ продолжается вниз до ветвящейся вершины z , то должны найтись две подстроки, начинающиеся в позиции $i + 1$ строки S или после нее, которые обе начинаются со строки γ . Поэтому в \mathcal{T}_{i+1} должна быть вершина z' с путевой меткой γ . Эта вершина должна располагаться ниже v' на пути к листу $i + 1$, что противоречит выбору v' . Таким образом, $Head(i)$ не должна достигать z и должна кончаться внутри дуги e . Конкретно, она кончается ровно в l_i символах от v'' на дуге e . \square

Следовательно, когда $l_i = 0$, мы знаем, что $Head(i)$ кончается в v'' , а когда $l_i > 0$, находим $Head(i)$ по v'' . Для этого мы проверяем выходящие из v'' дуги и обнаруживаем единственную дугу e , метка которой начинается с c . $Head(i)$ кончается точно в l_i символах вниз по e от v'' . Дерево \mathcal{T}_i строится делением дуги e , созданием в точке деления вершины w и добавлением новой дуги из w в лист i с меткой, равной остатку $Suff_i$. Поиск правильной дуги, выходящей из v'' , занимает только константное время, поскольку алфавит фиксирован.

В итоге, когда v и v' существуют, описанный метод правильно строит \mathcal{T}_i по \mathcal{T}_{i+1} , хотя мы еще должны обсудить, как обновлять векторы. Равным образом, еще не вполне ясно, почему он эффективнее, чем наивный алгоритм нахождения $Head(i)$. Это мы увидим ниже. Сначала посмотрим, как алгоритм справляется с вырожденными случаями, когда либо v , либо v' не существует.

Два вырожденных случая

Обсудим два вырожденных случая, когда не существует вершина v (а следовательно, и v') или v существует, а v' нет. Как в этих случаях можно эффективно найти $Head(i)$? Напомним, что r обозначает корневую вершину.

Случай 1. $I_r(S(i)) = 0$.

В этом случае спуск кончается в корне и вершины v нет. Отсюда следует, что символ $S(i)$ не появляется ни в какой позиции, большей чем i , так как если бы он там появлялся, то нашелся бы суффикс, начинающийся с $S(i)$, а тогда был бы путь из корня, начинающийся с $S(i)$, и $I_r(S(i))$ было бы равно 1. Так что при $I_r(S(i)) = 0$ строка $Head(i)$ пуста и кончается в корне.

Случай 2. $I_v(S(i)) = 1$ для некоторого v (возможно, для корня), но v' не существует.

В этом случае обход кончается в корне с пустым значением $L_r(S(i))$. Пусть t_i — число символов от корня до v . По теореме 6.2.1 строка $Head(i)$ кончается в $t_i + 1$ символах от корня. Так как v существует, то найдется дуга $e = (r, z)$, дуговая метка которой начинается с символа $S(i)$. Это верно независимо от того, будет $t_i = 0$ или $t_i > 0$.

Если $t_i = 0$, то $Head(i)$ кончается после первого символа, $S(i)$, на дуге e .

Аналогично, если $t_i > 0$, то $\text{Head}(i)$ кончается в $t_i + 1$ символах от корня на дуге e . Действительно, допустим, что $\text{Head}(i)$ продолжено любым путем до некоторого потомка z (или ниже). Тогда так же, как в доказательстве теоремы 6.2.2, вершина z должна быть ветвящейся и должна найтись вершина z' на пути от корня к листу $i + 1$, для которой $L_{z'}(S(i))$ не равно null, что даст противоречие. Поэтому если $t_i > 0$, то $\text{Head}(i)$ кончается в $t_i + 1$ символах от корня на дуге e , выходящей из корня. Этую дугу можно найти за константное время, поскольку она выходит из корня и известен ее первый символ — $S(i)$.

В любом из этих вырожденных случаев (как и в хорошем случае) $\text{Head}(i)$ находится за константное время после того, как обход достиг корня. Когда конец $\text{Head}(i)$ обнаружен и найдена или создана w , алгоритм работает так же, как в общем случае.

Отметим, что вырожденный случай 2 очень похож на хороший случай, но отличается маленькой деталью — тем, что конец $\text{Head}(i)$ находится в $t_i + 1$ символах вниз по e , а не в t_i символах (естественный аналог хорошего случая).

6.2.4. Полный алгоритм построения \mathcal{T}_i по \mathcal{T}_{i+1}

Соединение всех случаев вместе дает следующий алгоритм.

Продолжение дерева по Вайнеру

1. Начать с листа $i + 1$ в дереве \mathcal{T}_{i+1} (лист для Suff_{i+1}) и идти к корню в поиске первой вершины v , для которой $I_v(S(i)) = 1$.
2. Если достигнут корень и $I_r(S(i)) = 0$, то $\text{Head}(i)$ кончается в корне. Перейти к шагу 4.
3. Пусть v — найденная вершина (возможно, корень) и $I_v(S(i)) = 1$. Продолжать подниматься вверх до первой вершины v' (это может быть и v), у которой $L_{v'}(S(i))$ не null.
 - 3a. Если достигнут корень и $L_r(S(i))$ равно null, обозначить через t_i число символов в пути от корня до v . Найти выходящую из корня дугу e с дуговой меткой, начинающейся с $S(i)$. $\text{Head}(i)$ кончается в $t_i + 1$ символах от корня по дуге e .

В противном случае {когда условие в 3a не выполняется}:

- 3b. Если найденное v' такое, что $L_{v'}(S(i))$ равно не null, а, скажем, v'' , то надо идти по связи (для $S(i)$) в v'' . Пусть l_i — число символов на пути от v' к v и c — первый символ этого пути. Если $l_i = 0$, то $\text{Head}(i)$ кончается в v'' . В противном случае нужно искать дугу e , выходящую из v'' , первый символ которой равен c . Стока $\text{Head}(i)$ кончается точно в l_i символах ниже v'' на дуге e .
4. Если в конце $\text{Head}(i)$ уже существует вершина, то обозначить ее через w ; в противном случае создать вершину w в конце $\text{Head}(i)$. Создать новый лист с номером i и новую дугу (w, i) , метка которой равна оставшейся подстроке Suff_i (т.е. последним $m - i + 1 - |\text{Head}(i)|$ символам Suff_i), к которым добавляется терминальный символ $\$$. Теперь дерево \mathcal{T}_i построено.

Корректность

Из доказательства теорем 6.2.1 и 6.2.2 и обсуждения вырожденных случаев должно быть ясно, что алгоритм корректно строит дерево \mathcal{T}_i по \mathcal{T}_{i+1} , однако перед тем, как он сможет построить \mathcal{T}_{i-1} , должны быть исправлены векторы I и L .

Как исправлять векторы

После нахождения (или создания) вершины w мы должны исправить векторы I и L так, чтобы они соответствовали дереву \mathcal{T}_i . Если алгоритм нашел вершину v , для которой $I_v(S(i)) = 1$, то по теореме 6.2.1 вершина w имеет путевой меткой $S(i)\alpha$ в \mathcal{T}_i , где α — путевая метка v . В этом случае $L_v(S(i))$ должно указывать в \mathcal{T}_i на w . Это единственное исправление, которое требуется для векторов связи, так как только одна вершина может указывать на любую другую вершину через вектор связи и была создана только одна новая вершина. Далее, если вершина w создана заново, все элементы ее вектора связей в \mathcal{T}_i должны быть пустыми (null). Увидеть это просто, предполагая от противного, что есть вершина u дерева \mathcal{T}_i с путевой меткой $xHead(i)$, где w имеет путевую метку $Head(i)$. Вершина u не может быть листом, так как $Head(i)$ не содержит символа $\$$. Но тогда в \mathcal{T}_{i+1} должна существовать вершина с путевой меткой $Head(i)$, вопреки тому, что вершина w была включена в \mathcal{T}_{i+1} , чтобы создать \mathcal{T}_i . Следовательно, в \mathcal{T}_i нет вершины с путевой меткой вида $xHead(i)$ и все элементы вектора L для w должны быть пустыми.

Посмотрим теперь, какие необходимы исправления в индикаторных векторах для дерева \mathcal{T}_i . Для каждой вершины u пути от корня до листа $i + 1$ должно быть установлено $I_u(S(i)) = 1$, так как теперь для строки $Suff_i$ в \mathcal{T}_i есть путь. Легко установить по индукции, что если при движении из листа $i + 1$ найдена вершина v с $I_v(S(i)) = 1$, то $I_u(S(i)) = 1$ для каждой вершины u выше v на пути к корню. Поэтому должны быть установлены только индикаторные векторы для вершин ниже v на пути к листу $i + 1$. Если вершина v не найдена, то на пути вверх из листа $i + 1$ до корня будут проойдены все вершины, и у всех нужно исправлять индикаторные векторы. Необходимые исправления для вершин ниже v можно сделать во время поиска v (т. е. отдельного прохода не требуется). Во время движения от листа $i + 1$ для каждой встретившейся вершины u принимается $I_u((S(i)) = 1$. Время на исправление индикаторных векторов пропорционально времени обхода.

Единственное оставшееся исправление — это установка вектора I для вновь обработанной вершины w внутри дуги $e = (v'', z)$.

Теорема 6.2.3. *Когда новая вершина w создается внутри дуги (v'', z) , ее индикаторный вектор нужно скопировать из индикаторного вектора z .*

Доказательство. Сразу видно, что если $I_z(x) = 1$, то $I_w(x)$ также должно равняться 1 в дереве \mathcal{T}_i . Но не случится ли так, что $I_w(x) = 1$ и $I_z(x) = 0$ в момент создания w ? Покажем, что такого быть не может.

Пусть вершина z имеет путевой меткой y , и конечно, вершина w имеет путевой меткой $Head(i)$, некоторый префикс y . Тот факт, что между u и z в дереве \mathcal{T}_{i+1} нет вершин, означает, что каждый суффикс от $Suff_m$ до $Suff_{i+1}$, который начинается со строки $Head(i)$, должен в действительности начинаться с более длинной строки y . Следовательно, в \mathcal{T}_{i+1} путь с меткой $xHead(i)$ может быть найден, только если найдется путь с меткой $x y$, и это верно для любого символа x . Поэтому если в \mathcal{T}_i существует путь с меткой $xHead(i)$ (что нужно для выполнения условия $I_w(x) = 1$),

но не путь $x y$, то гипотетическая строка $x \text{Head}(i)$ должна начинаться с символа i строки S . Это означает, что Suff_{i+1} должно начинаться со строки $\text{Head}(i)$. Но так как w имеет путевой меткой $\text{Head}(i)$, то лист $i + 1$ должен быть ниже w в дереве \mathcal{T}_i , а также ниже z в \mathcal{T}_{i+1} . Значит, z лежит на пути от корня к $i + 1$. Однако алгоритм построения \mathcal{T}_i из \mathcal{T}_{i+1} начинается в листе $i + 1$ и идет к корню, а когда он находит вершину v или достигает корня, элемент индикаторного вектора с индексом x положен равным 1 для каждой вершины пути от листа $i + 1$. Движение кончается до того, как будет создана вершина w , поэтому невозможно, чтобы $I_z(x) = 0$ в момент создания w . Таким образом, если путь $x \text{Head}(i)$ существует в дереве \mathcal{T}_i , то $I_z(x) = 1$ в момент создания w , и теорема доказана. \square

6.2.5. Временной анализ алгоритма Вайнера

Время, требуемое на построение \mathcal{T}_i из \mathcal{T}_{i+1} и на исправление векторов, пропорционально времени на движение от листа $i + 1$ (кончающееся либо в v' , либо в корне). Движение направлено от каждой вершины к ее родителю, и в предположении, что имеются обычные ссылки на родителя, на каждый такой переход расходуется только константное время. Константное время требуется и на перемещение по указателю связи L , и на добавление w и дуги (w, i) . Следовательно, время на построение \mathcal{T}_i пропорционально числу вершин в пути из листа $i + 1$.

Напомним, что вершинная глубина вершины v определена как число вершин на пути от корня до v .

Представим себе, что в ходе работы алгоритма мы следим за тем, какая вершина просматривалась последней, и за ее вершинной глубиной. Назовем вершинную глубину последней посещенной вершины *текущей вершинной глубиной*. Пусть, когда алгоритм стартует, текущая вершинная глубина равна единице, а сразу после создания \mathcal{T}_m она равна двум. Ясно, что, когда алгоритм идет вверх по пути от листа, текущая вершинная глубина на каждом шаге уменьшается на единицу. Точно так же, когда алгоритм находится в вершине v'' (или в корне) и создает новую вершину w ниже v'' (или ниже корня), текущая вершинная глубина увеличивается на единицу. Осталось только выяснить, как изменяется текущая вершинная глубина при переходе по связи от вершины v' к v'' .

Лемма 6.2.1. *При переходе алгоритма по связи от вершины v' к вершине v'' в \mathcal{T}_{i+1} , текущая вершинная глубина увеличивается не больше чем на единицу.*

Доказательство. Пусть u — некорневая вершина в \mathcal{T}_{i+1} на пути от корня к v' и ее путевая метка имеет вид $S(i)\alpha$ с некоторой непустой строкой α . К этому типу относятся все вершины на пути от корня до v'' за исключением единственной вершины (если она существует) с путевой меткой $S(i)$. Далее, $S(i)\alpha$ является префиксом строк Suff_i и Suff_k для некоторого $k > i$, но имеет в них разные продолжения. Так как v' лежит на пути от корня к листу $i + 1$, то α является префиксом для Suff_{i+1} , и на пути к v' в \mathcal{T}_{i+1} должна быть вершина (возможно, корень) с путевой меткой α . Таким образом, путь к v' имеет вершину, соответствующую любой вершине из пути к v'' , кроме вершины с путевой меткой $S(i)$ (если такая существует). Следовательно, глубина v'' не может превышать глубину v' более чем на единицу, хотя может быть и меньше. \square

Мы можем теперь завершить временной анализ.

Теорема 6.2.4. В предположении конечности алфавита алгоритм Вайнера строит суффиксное дерево для строки длины t за время $O(t)$.

Доказательство. Текущая вершинная глубина может возрасти на единицу при создании новой вершины или при проходе дуги связи. Следовательно, полное число ее увеличений не превосходит $2t$. Отсюда получаем, что текущая вершинная глубина может и уменьшиться не более $2t$ раз, потому что имеет начальное значение 0 и неотрицательна. А уменьшается она при движении наверх, так что полное число вершин, посещаемых при подъемах, также не превосходит $2t$. Поскольку время работы алгоритма пропорционально числу вершин, посещаемых при подъемах, то теорема доказана. \square

6.2.6. Последние замечания относительно алгоритма Вайнера

Из нашего обсуждения алгоритма Вайнера следует важный факт о суффиксных деревьях, не зависящий от того, как они построены.

Теорема 6.2.5. Если v — вершина суффиксного дерева, помеченная строкой xa , где x — отдельный символ, то в дереве есть вершина, помеченная строкой a .

Этот факт был получен и как следствие 6.1.2 при обсуждении алгоритма Укконена.

6.3. Алгоритм Мак-Крейга для построения суффиксных деревьев

Через несколько лет после того, как Вайнер опубликовал свой алгоритм с линейным временем для построения суффиксного дерева для строки S , Мак-Крейг [318] предложил другой метод, также с линейным временем, но на практике более эффективный по затратам памяти.*) Неэффективность алгоритма Вайнера — в расходе памяти на индикаторные векторы и векторы связи, I и L , хранимые для каждой вершины. Для фиксированного алфавита размер этой памяти считается зависящим линейно от длины S , но практически используемое место может быть большим. Алгоритм Мак-Крейга не нуждается в этих векторах, и поэтому требуемое им место меньше.

Алгоритм Укконена также не использует векторы I и L из алгоритма Вайнера, и его затраты памяти такие же, как у алгоритма Мак-Крейга.**) На самом деле полностью реализованный вариант алгоритма Укконена может рассматриваться как несколько приукрашенный вариант алгоритма Мак-Крейга. Однако на высоком уровне организации эти алгоритмы совершенно различны, и соотношение между ними не очевидно. Предположение о их связи было высказано Укконеном [438] и подтверждено Гигерихом и Курцем [178]. Так как алгоритм Укконена обладает всеми преимуществами алгоритма Мак-Крейга и проще для описания, мы опишем алгоритм Мак-Крейга только в общих чертах.

*) Уже после выхода этой книги появились новые исследования, улучшающие схему Мак-Крейга (см., например, [286a]). — Прим. перев.

**) Требования к памяти алгоритмов Укконена и Мак-Крейга определяются необходимостью сохранить дерево и быстро по нему перемещаться. Вопросы памяти и другие практические детали мы уточним в п. 6.5.

Алгоритм Мак-Крейга в общих чертах

Алгоритм Мак-Крейга строит суффиксное дерево \mathcal{T} для строки S длины m включением в него суффиксов по порядку, по одному за раз, начиная с суффикса один (т. е. с полной строки S). (Этот порядок противоположен порядку, используемому алгоритмом Вайнера, и совершенно отличен от алгоритма Укконена.) Он строит дерево, кодируя все суффиксы S , начинающиеся в позициях от 1 до $i + 1$, из дерева, кодирующего все суффиксы S , начинающиеся в позициях от 1 до i .

Наивный вариант метода прост и работает за время $O(m^2)$. Используя суффиксные связи и прием скачка по счетчику, это время можно снизить до $O(m)$. Мы оставляем детали заинтересованному читателю.

6.4. Обобщенное суффиксное дерево для набора строк

До сих пор мы рассматривали методы построения за линейное время суффиксного дерева для отдельной строки. Они легко развиваются на случай представления суффиксов набора строк $\{S_1, S_2, \dots, S_z\}$ в виде дерева, которое называется *обобщенным суффиксным деревом*. Это дерево будет использовано во многих приложениях.

Концептуально простой путь построения обобщенного суффиксного дерева открывается, если дополнить конец каждой строки из набора уникальным маркером конца, а затем конкатенировать все строки в одну и построить суффиксное дерево для этой единой строки. Эти концевые маркеры должны быть различны между собой и в тексте больше не встречаться. Получающееся суффиксное дерево будет иметь по одному листу на каждый суффикс конкатенированной строки, и время на его построение будет пропорционально сумме длин всех строк. Номера листьев можно легко конвертировать в пару чисел, одно из которых идентифицирует строку S_i , а другое — стартовую позицию в S_i .

У этого метода построения обобщенного суффиксного дерева есть тот дефект, что дерево представляет подстроки (конкатенированной строки), которые простираются более чем на одну из исходных строк. Эти *синтетические* суффиксы интереса не представляют. Однако, так как все концевые маркеры различны и отсутствуют в исходных строках, метка любого пути от корня до внутренней вершины должна быть подстрокой одной из исходных строк. Следовательно, уменьшая второй индекс метки на листовых дугах, можно удалить все нежелательные синтетические суффиксы, не трогая при этом никаких других частей дерева.

При более тщательной проверке обнаруживается, что предложенный метод можно имитировать без конкатенации строк. Мы опишем эту имитацию, используя алгоритм Укконена и две строки S_1 и S_2 , предполагаемые различными. Сначала построим суффиксное дерево для S_1 (считая, что терминальный символ добавлен). Затем, начиная с корня этого дерева, проверим совпадение S_2 (снова предполагая добавление того же терминального символа) с путем в дереве до обнаружения несовпадения. Предположим, что совпали первые i символов из S_2 . Дерево в этот момент кодирует все суффиксы S_1 и неявно — каждый суффикс строки $S_2[1..i]$. Существенно, что первые i фаз алгоритма Укконена для S_2 были выполнены сверх построения дерева для S_1 . Итак, с текущим деревом нужно завершить алгоритм Укконена с S_2 в фазе $i + 1$. Значит, нужно подняться вверх не более чем на одну вершину от конца $S_2[1..i]$ и т. д. Когда строка S_2 будет

полностью обработана, в дереве окажутся закодированы все суффиксы S_1 и S_2 , а синтетических суффиксов не будет. Повторение этого действия для всех строк создаст обобщенное суффиксное дерево за время, пропорциональное сумме длин его строк.

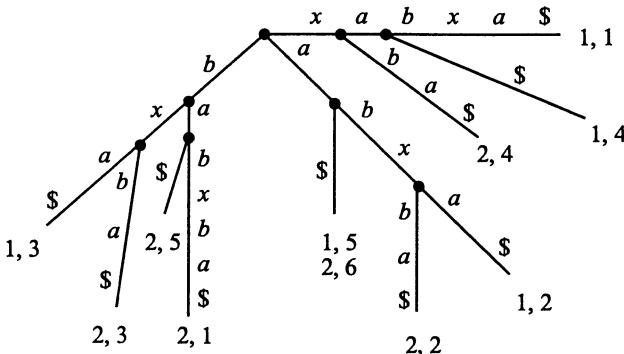


Рис. 6.11. Обобщенное суффиксное дерево для строк $S_1 = xabxa$ и $S_2 = babxba$. Первое число в листе указывает строку, второе — начальную позицию суффикса в строке

В этом втором подходе остались два незначительных нюанса. Во-первых, сжатые метки на разных дугах могут относиться к разным строкам. Следовательно, при каждой дуге нужно хранить не по два, а по три числа, но остальная проблема не вызывает. Во-вторых, суффиксы для двух строк могут совпадать, хотя по-прежнему никакой суффикс не будет префиксом другого суффикса. В этом случае лист должен указывать на все строки и начальные позиции этого суффикса.

Например, если мы добавим строку $babxba$ к дереву для $xabxa$ (представленному на рис. 6.1), результатом будет обобщенное суффиксное дерево, показанное на рис. 6.11.

6.5. Вопросы практической реализации

Уже обсуждавшиеся в этой главе детали реализации превращают наивные квадратические (или даже кубические) по времени алгоритмы в алгоритмы с временем наихудшего случая $O(m)$, в предположении о фиксированном алфавите Σ . Но чтобы сделать суффиксные деревья совсем практическими, требуется уделить еще больше внимания реализации, особенно если размер алфавита растет. Есть задачи, которые теоретически прекрасно решаются суффиксными деревьями и в которых типичные размеры строк измеряются сотнями тысяч и даже миллионами и/или размер алфавита — сотнями. Для этих задач “линейные” границы времени и памяти еще не дают достаточной гарантии практичности. Для больших деревьев серьезной проблемой является пейджинг (страничная организация памяти), поскольку деревья не имеют хороших свойств локальности. Действительно, по своему замыслу суффиксные связи позволяют алгоритму быстро переходить из одной части дерева в другую. Это хорошо для временных оценок наихудшего случая, но кошмарно для пейджинга, если дерево не помещается целиком в памяти. В силу этого задача о реализации

суффиксных деревьев с уменьшением практических затрат памяти очень серьезна.* Сделанные здесь замечания о суффиксных деревьях приложимы также к деревьям ключей, использованным в методе Ахо–Корасика.

Главные вопросы разработки программы в каждом из трех алгоритмов — это представление звезд выходящих дуг в вершинах дерева и поиск в них нужной дуги, а также представление индикаторного вектора и вектора связей в алгоритме Вайнера. Практический дизайн должен балансировать между ограничениями памяти и времени как при построении дерева, так и при дальнейшем его использовании. Мы обсудим представление дуг дерева, так как вопросы работы с векторами в алгоритме Вайнера аналогичны.

Для представления звезд есть четыре основные возможности. Простейшая — это использовать массив размера $\Theta(|\Sigma|)$ в каждой промежуточной вершине v . Этот массив индексируется символами алфавита; ячейка, индексированная буквой x , содержит указатель на потомка v , если существует выходящая из v дуга, дуговая метка которой начинается с буквы x , и null — в противном случае. Когда такая дуга существует, ячейка должна содержать также два индекса, представляющие ее дуговую метку. Этот массив обеспечивает доступ и исправления за константное время, но, хотя он прост для программирования, может использовать слишком много памяти, когда $|\Sigma|$ и m велики.

Альтернативой массиву является *цепной список* в каждой вершине v для символов, появившихся в начале меток дуг, выходящих из v . Когда к дереву добавляется новая дуга, выходящая из v , к списку добавляется новый символ (первый символ метки этой дуги). Переходы из вершины v реализуются сплошным поиском нужного символа в списке. Так как список просматривается последовательно, хранение его в *отсортированном порядке* обходится не дороже. Это несколько снижает среднее время поиска данного символа и таким образом ускоряет (на практике) построение дерева. Ключевой момент в том, что упорядочение позволяет быстрее закончить поиск символа, которого нет в списке. Поддержание списка отсортированным даст практическую пользу в некоторых приложениях суффиксных деревьев, которые обсуждаются ниже.

Работа с цепным списком в вершине v удобна, если число детей у v мало, но в худшем случае приводит к времени $|\Sigma|$ на каждой операции с вершиной. Оценка времени наихудшего случая $O(m)$ сохраняется, так как размер алфавита $|\Sigma|$ предполагается фиксированным, но если число детей у v велико, то по сравнению с массивом выигрышается немного места, а проигрыш во времени существенен.

Третья возможность, компромисс между местом и скоростью, — это реализация списка в вершине v как своего рода *сбалансированного дерева* [10]. Добавление дуги и поиск требуют $O(\log k)$ времени и $O(k)$ памяти, где k — число детей v . Из-за чрезмерных затрат памяти и сложности программирования эта альтернатива осмысlena только при действительно больших k .

Последняя возможность — это какая-либо *схема хеширования*. Опять же, нужно искать схему с правильным балансом места и скорости, но для больших деревьев и алфавитов хеширование очень привлекательно, по крайней мере для некоторых вершин. А использование совершенной техники хеширования [167] позволяет даже сохранить линейную временную оценку наихудшего случая.

* Совершенно другой подход к ограниченной памяти, основанный на замене суффиксного дерева на структуру данных, называемую *суффиксным массивом*, будет обсуждаться в п. 7.14

Когда m и Σ настолько велики, что реализация затрудняется, наилучшим вариантом будет, вероятно, смесь перечисленных возможностей. Вершины около корня дерева норовят иметь больше всего детей (корень имеет потомка на каждую букву, появляющуюся в S), и для этих вершин правильно использовать массивы. Вдобавок если дерево плотное на несколько слоев вниз по дереву от корня, то эти слои можно сжать и удалить из явного дерева. Например, есть 20^5 возможных подстрок длины пять для аминокислот. Каждая из этих подстрок уже существует в некоторых известных белковых последовательностях в базах данных. Поэтому при реализации суффиксного дерева для базы данных по белкам можно заменить первые пять уровней дерева на пятимерный массив (индексируемый строками длины пять), элементы которого являются указателями на место в оставшемся дереве, соответствующее продолжению данной пятерки. Та же идея была применена в [320] для данных по ДНК с глубиной семь. Вершины в суффиксном дереве около листьев обычно имеют немного детей, и предпочтение следует отдать спискам. В крайнем случае, если w — это лист, а v — его предок, информация относительно w может быть передана в v , что избавит от необходимости явного представления дуги (v, w) и вершины w . В зависимости от других выборов реализации это дает существенную экономию памяти, так как, по грубой оценке, половина вершин суффиксного дерева — это листья. Суффиксное дерево, листья которого удалены таким способом, называется *деревом позиций*. В нем задается однозначное соответствие между листьями и подстроками, встречающимися в S один раз.

Для вершин в середине суффиксного дерева наилучшим выбором могут быть хеширование или сбалансированные деревья. К счастью, самые большие суффиксные деревья используются в приложениях, где текст S фиксирован (словарь или база данных) на некоторое время, и суффиксное дерево будет использовано многократно. В этих приложениях можно найти и время, и повод для экспериментов с различными вариантами реализации. Более глубокий взгляд на проблемы реализации и другие предлагаемые варианты суффиксных деревьев см. в [23].

Какая бы реализация ни была выбрана, ясно, что суффиксное дерево для строки займет значительно больше места, чем само представление строки.*¹) Ниже в этой книге мы обсудим некоторые задачи, использующие по две (и более) строки P и T , в которых существует по два решения с временем $O(|P| + |T|)$: одно — использующее суффиксное дерево для P , а другое — суффиксное дерево для T . У нас появятся также примеры с одинаковыми эффективными по времени решения, из которых одно использует обобщенное суффиксное дерево для двух и более строк, а другое — просто суффиксное дерево для меньшей строки. По асимптотике наихудших оценок времени и памяти ни один из подходов не превосходит другой, и обычно концептуально проще тот подход, который строит большее дерево. Однако, когда ограничение по памяти более существенно (и во многих задачах, включая многие задачи молекулярной биологии, память ограничивает больше, чем время), размер суффиксного дерева для строки может заставить выбрать решение с меньшим суффиксным деревом. Поэтому, несмотря на лишнее концептуальное бремя, на протяжении всей книги мы будем обсуждать в деталях такие возможности экономии памяти.

*¹) Хотя мы строили суффиксные деревья для строк ДНК и аминокислот длиной более миллиона символов, которые могли полностью помещаться в оперативной памяти рабочей станции умеренного размера.

6.5.1. Независимость от алфавита: все линейные зависимости равны, но некоторые равнее других

Все рассмотренные выше ключевые проблемы реализации связаны с кратностью дуг (или связей) в вершинах. А на кратность влияет размер алфавита Σ — чем больше алфавит, тем больше проблема. По этой причине многие предпочитают явно отражать зависимость оценок времени и памяти в алгоритмах построения ключевых и суффиксных деревьев от размеров алфавита. При этом обычно записывают оценку времени построения ключевых и суффиксных деревьев в виде $O(m \log |\Sigma|)$, где m — размер всех образцов в дереве ключей или строки в суффиксном дереве. Алгоритмы Ахо–Корасика, Вайнера, Укконена и Мак–Крейга требуют либо памяти порядка $\Theta(m|\Sigma|)$, либо замены оценки времени $O(m)$ на минимум из $O(m \log m)$ и $O(m \log |\Sigma|)$. Аналогично, поиск образца P с использованием суффиксного дерева можно выполнить за $O(|P|)$ сравнений, только если мы используем память $\Theta(m|\Sigma|)$; в противном случае мы должны выполнить за время поиска P минимум из $O(|P| \log m)$ и $O(|P| \log |\Sigma|)$ сравнений.

Напротив, метод поиска точного совпадения, использующий значения Z , имеет оценки наихудшего случая для памяти и числа сравнений, которые *алфавитно-независимы* — на число сравнений (как символов, так и чисел), необходимых для вычисления значений Z , размер алфавита не влияет. Более того, при сравнении двух символов метод проверяет их только на равенство, их расположение в каком-либо упорядочении не существенно. Следовательно, никакой априорной информации относительно алфавита не требуется. Этими свойствами обладают также алгоритмы Кнута–Морриса–Пратта и Бойера–Мура. Алфавитная независимость этих алгоритмов ставит их линейные оценки времени и памяти выше в глазах некоторых исследователей, чем линейные оценки в алгоритмах с деревом ключей и суффиксным деревом: “все линейные зависимости равны, но некоторые равнее других”. Алфавитно-независимые алгоритмы развивались не только для поиска точного совпадения, но и для ряда других задач. Двумерный поиск точного совпадения — один из таких примеров. Метод, представленный в п. 3.5.3, основан на деревьях ключей и, следовательно, не является алфавитно-независимым. Тем не менее для этой задачи были разработаны алфавитно-независимые решения. Вообще, алфавитно-независимые методы сложнее, чем их более грубые собратья. В этой книге мы не рассматриваем далее независимость от алфавита, хотя и обсуждаем другие подходы к сокращению памяти, которые можно использовать, когда большие алфавиты вызывают чрезмерную потребность в памяти.

6.6. Упражнения

- Постройте бесконечное семейство строк над фиксированным алфавитом, такое, чтобы полная длина дуговых меток в их суффиксных деревьях росла быстрее, чем $O(m)$ (m — длина строки). То есть покажите, что линейные по времени алгоритмы для суффиксного дерева были бы невозможны, если бы дуговые метки явно записывались при дугах.
- В тексте мы ввели алгоритм Укконена сначала в укрупненном описании и отметили, что его можно реализовать за время $O(m^3)$. Это время было уменьшено до $O(m^2)$ без использования суффиксных связей и приема скачка по счетчику. Другой способ уменьшить

время с $O(m^3)$ до $O(m^2)$ (без суффиксных связей и скачков по счетчику) — поддерживать указатель на конец каждого суффикса $S[1..i]$. Тогда алгоритм Укконена в укрупненном описании посетит все эти концы и создаст \mathcal{T}_{i+1} из \mathcal{T}_i за время $O(i)$, так что весь алгоритм потребует времени $O(m^2)$. Объясните это подробно.

3. Взаимосвязь между суффиксным деревом для строки S и обратной строки S^r не очевидна. Однако существует важное соотношение между этими двумя деревьями. Найдите его, сформулируйте и докажите.

Подсказка. Помогут суффиксные связи.

4. Можно ли алгоритм Укконена реализовать так, чтобы он работал за линейное время без использования суффиксных связей? Идея заключается в том, чтобы хранить для каждого значения i указатель на вершину в текущем неявном суффиксном дереве, ближайшую к концу суффикса i .

5. В приеме 3 алгоритма Укконена символ e используется как второй индекс метки каждой листовой дуги, и в фазе $i + 1$ глобальная переменная e полагается равной $i + 1$. В качестве альтернативы для e можно было бы положить второй индекс каждой листовой дуги при создании этой дуги равным m (полной длине S). В этом случае не требуется никакой работы на обновление второго индекса. Объясните подробно, почему это предложение корректно, и обсудите неудобства, которые могут возникнуть по сравнению с использованием символа e .

6. Алгоритм Укконена строит все неявные суффиксные деревья от I_1 до I_m по порядку и в *режиме прямого исполнения* (on-line), все за время $O(m)$. Таким образом, он может быть назван онлайновым алгоритмом с линейным временем для построения неявных суффиксных деревьев.

Открытый вопрос. Найдите онлайновый алгоритм, работающий за полное время $O(m)$, который создавал бы все истинные суффиксные деревья. Так как время, затрачиваемое на явное сохранение этих деревьев, есть $\Theta(m^2)$, такой алгоритм должен (подобно алгоритму Укконена) только преобразовывать каждое дерево без его сохранения.

7. Алгоритм Укконена строит все неявные суффиксные деревья за время $O(m)$. Эта последовательность неявных суффиксных деревьев может дать больше информации об S , чем единственное окончательное суффиксное дерево. Найдите задачу, которую можно решить более эффективно с помощью этой последовательности неявных деревьев, чем с окончательным деревом. Заметьте, что алгоритм не может сохранить неявные деревья, и значит, задача должна решаться параллельно с построением деревьев.

8. Наивный алгоритм Вайнера для построения суффиксного дерева строки S (п. 6.2.1) можно описать в терминах алгоритма Ахо–Корасика из п. 3.4. Пусть задана строка S длины m , добавим к ней $\$$, и пусть \mathcal{P} будет множеством образцов, состоящим из $m + 1$ суффиксов строки $S\$$. Построим дерево ключей для множества \mathcal{P} , используя алгоритм Ахо–Корасика. Удаление обратных связей дает нам суффиксное дерево для S . На это построение требуется время $O(m^2)$. Вместе с тем при обсуждении алгоритма Ахо–Корасика этот метод рассматривался как метод с *линейным* временем. Разрешите возникшее противоречие.

9. Найдите явное соотношение между указателями связи в алгоритме Вайнера и суффиксными связями в алгоритме Укконена.

10. Временной анализ и алгоритма Укконена, и алгоритма Вайнера основывается на наблюдении изменений текущей вершинной глубины, и аргументы почти полностью симметричны. Сопоставьте эти алгоритмы и аргументы в их анализе, чтобы выяснить сходства

и различия. Нельзя ли дать такое укрупненное описание, из которого временные границы получались бы для обоих алгоритмов сразу?

11. Эмпирически оцените различные возможности представления звезд выходящих дуг и векторов, используемых в алгоритме Вайнера. Уделите особое внимание влиянию размера алфавита и длины строки и рассмотрите роль памяти и времени при построении суффиксного дерева и его последующем применении.
12. Используя приемы реализации вроде тех, которые были в алгоритме Укконена (в частности, суффиксные связи и скачки по счетчику), предложите реализацию алгоритма Мак-Крейга с линейным временем.
13. Разберитесь в подробностях соотношения между алгоритмами Мак-Крейга и Укконена, когда они оба реализуются так, чтобы исполняться за линейное время.
14. Предположим, что нужно работать с суффиксным деревом в динамике, когда строка растет или сокращается. Обсудите, как это эффективно сделать, если строка растет (сокращается) на левом конце, и как — если на правом.

Может ли алгоритм Вайнера или Укконена эффективно справиться с этими изменениями на левом и на правом концах? Что может помешать переворачиванию строки для того, чтобы изменения на левом конце “имитировались” изменениями на правом конце?
15. Рассмотрите предыдущую задачу в случае изменений в середине строки. Если вы не можете найти эффективное решение для изменений суффиксного дерева, объясните, в чем технические сложности и почему эта задача выглядит трудной.
16. Рассмотрим обобщенное суффиксное дерево, построенное для набора из k строк. К этому набору могут добавляться дополнительные строки или целые строки могут исключаться из набора. Это типичная ситуация при работе с обобщенным суффиксным деревом для биологических последовательностей [320]. Обсудите проблемы поддержания обобщенного суффиксного дерева в этой динамической постановке. Объясните, почему эта проблема решается гораздо проще, чем когда произвольные подстроки, представленные в суффиксном дереве, удаляются.

Первые приложения суффиксных деревьев

В этой книге мы увидим много примеров использования суффиксных деревьев. Большинство из них допускает неожиданно эффективные, линейные по времени решения сложных задач о строках. Некоторые из наиболее впечатляющих применений нуждаются в дополнительном средстве — алгоритме нахождения ближайшего общего предшественника, работающем за константное время, так что их рассмотрение откладывается до появления этого алгоритма (в главе 8). Другие приложения встречаются в контексте специальных задач, которые будут подробно обсуждаться дальше. Но есть много примеров, которые мы можем обсудить сейчас и которые показывают, насколько мощным и полезным средством являются суффиксные деревья. В этой главе и в упражнениях к ней развиваются некоторые из таких применений.

Возможно, наилучший способ для читателя оценить мощь суффиксных деревьев — потратить некоторое время на попытки решить обсуждаемые ниже задачи не применяя эту технику. Без такой попытки и без некоторой исторической перспективы доступность суффиксных деревьев может создать впечатление тривиальности отдельных задач, даже несмотря на то, что для части из них алгоритмы с линейным временем до появления суффиксных деревьев известны не были. Такова *задача о наибольшей общей подстроке*, рассматриваемая в п. 7.4. Кнут предположил, что в ней линейный по времени алгоритм невозможен [24, 278], а такой алгоритм прямо получается при использовании суффиксных деревьев. Другой классический пример — это рассмотренная в упражнениях *задача о наибольшем повторяющемся префикссе*, для которой обращение к суффиксным деревьям легко дает решение, линейное по времени, а лучший метод, известный до появления этой техники, имел трудоемкость $O(n \log n)$.

7.1. APL1: Точное совпадение строк

У этой задачи есть три важных варианта, отличающихся тем, какая из строк P и T считается известной заранее и сохраняется неизменной. Мы уже обсуждали (в п. 5.3) использование суффиксных деревьев в задаче о поиске точного совпадения, когда и образец и текст становятся известны алгоритму в одно и то же время. В этом случае использование суффиксного дерева дает ту же оценку наихудшего случая, $O(n + m)$, что алгоритмы Кнута–Морриса–Пратта и Бойера–Мура.

Но задача о точном совпадении часто встречается в ситуации, когда текст T известен заранее и на некоторое время фиксируется. После того как текст обработан, вводится длинная последовательность образцов, и для каждого образца P поиск всех его вхождений в T должен быть выполнен как можно скорее. Пусть n обозначает длину P , а k — число вхождений P в T . С использованием суффиксного дерева для T все вхождения можно найти за время $O(n + k)$ совершенно независимо от размера T . То, что любой образец (неизвестный на препроцессной стадии) может быть найден за время, пропорциональное только его длине, причем после препроцессской обработки T с линейным временем, восхищало и прямо побуждало к развитию суффиксных деревьев. Напротив, в алгоритмах с препроцессной подготовкой образца время на поиск каждого отдельного образца P будет равно $O(n + m)$.

Обратная ситуация — когда вначале известен образец и его можно обработать до появления текста; это классическая ситуация, в которой действуют методы Кнута–Морриса–Пратта и Бойера–Мура, а не суффиксные деревья. Эти алгоритмы расходуют на препроцессинг время $O(n)$, после чего поиск может быть выполнен за время $O(m)$. Можно ли при таком сценарии использовать суффиксные деревья, чтобы получить такие же временные оценки? Хотя это и не очевидно, ответ “да”. Обратное использование суффиксных деревьев будет обсуждаться вместе с более общей задачей в п. 7.8. Так, для задачи о точном совпадении (простой образец) суффиксные деревья можно применять для получения тех же оценок времени и памяти, как и в методах Кнута–Морриса–Пратта и Бойера–Мура, когда образец известен раньше или когда образец и текст известны одновременно; но они достигают полного превосходства в важном случае, когда текст известен заранее, а образцы меняются.

7.2. APL2: Суффиксные деревья и множественное точное совпадение

В п. 3.4 рассматривалась множественная задача точного совпадения — задача поиска всех вхождений набора строк \mathcal{P} в текст T , когда этот набор вводится весь сразу. Там мы предложили решение, линейное по времени, принадлежащее Ахо и Корасику. Напомним, что набор \mathcal{P} имеет полную длину n , а текст T — длину m . Метод Ахо–Корасика находит все вхождения в T любого образца из \mathcal{P} за время $O(n + m + k)$, где k — число вхождений. Эта же временная оценка легко получается при использовании суффиксного дерева \mathcal{T} для строки T . Действительно, мы видели в предыдущем пункте, что, когда текст T известен заранее, а образец P изменяется, все вхождения любого фиксированного P (длины n) в T могут быть найдены за время $O(n + k_P)$, где k_P — число вхождений P . Таким образом, множественная задача точного совпадения — это еще более простой случай, потому что набор \mathcal{P}

вводится, а текст уже известен. Для решения этой задачи мы строим суффиксное дерево \mathcal{T} для T за время $O(m)$, а затем используем это дерево для последовательного поиска всех вхождений каждого образца из \mathcal{P} . Полное время при этом подходе равно $O(n + m + k)$.

7.2.1. Сопоставление суффиксных и ключевых деревьев при множественном сравнении

Здесь мы сравним относительные преимущества дерева ключей и суффиксного дерева в множественной задаче точного совпадения. Хотя асимптотические оценки времени и памяти для этих двух методов совпадают, когда набор \mathcal{P} и строка T задаются одновременно, но один метод может быть предпочтительнее другого в зависимости от соотношения размеров \mathcal{P} и T и от того, какая строка подверглась препроцессингу. Метод Ахо–Корасика использует дерево ключей размера $O(n)$, построенное за время $O(n)$, а затем осуществляет поиск за время $O(m)$. Напротив, суффиксное дерево \mathcal{T} имеет размер $O(m)$, требует время $O(m)$ на построение и использует на поиск время $O(n)$. Постоянные в границах памяти и времени поиска зависят от способа, выбранного для представления деревьев (см. п. 6.5), но, определенно, они слишком велики для практического использования.

В случае, когда набор образцов больше, чем текст, суффиксное дерево использует меньше места, но требует больше времени на поиск. (Как отмечалось в п. 3.5.1, в молекулярной биологии имеются приложения, где библиотека образцов значительно больше, чем типичные тексты, появляющиеся после создания библиотеки.) Когда полный размер образцов меньше, чем текст, метод Ахо–Корасика использует меньше памяти, чем суффиксное дерево, но суффиксное дерево тратит меньше времени на поиск. Следовательно, нужно выбирать между временем и местом, и никакой метод не превосходит другого по обоим показателям. Определение относительных преимуществ метода Ахо–Корасика в сравнении с суффиксными деревьями, когда текст фиксирован, а набор образцов меняется, оставляется читателю.

Есть еще один подход, в котором суффиксные деревья лучше или более устойчивы, чем деревья ключей для множественной задачи точного совпадения (в добавок к другим задачам). Мы покажем в п. 7.8, как применить суффиксное дерево для решения этой задачи с теми же границами времени и места, как и в методе Ахо–Корасика: $O(n)$ — на подготовку и $O(m)$ — на поиск. Это обращение границ для суффиксных деревьев, показанных выше. Балансирование между временем и местом остается, но суффиксное дерево может быть использовано для любой выбранной комбинации время/место, тогда как с деревом ключей никакой выбор невозможен.

7.3. APL3: Задача о подстроке для базы образцов

Задача о подстроке была введена в главе 5 (с. 119). В наиболее ее интересном варианте первоначально задается и фиксируется набор строк, или база данных. Затем предоставляется последовательность строк, и для каждой строки S алгоритм должен найти в базе данных все строки, содержащие S как подстроку. Это обращение множественной задачи точного совпадения, где требуется найти, какой из фиксированных образцов является подстрокой вводимой строки.

В контексте баз данных для геномных ДНК [63, 320] задача нахождения подстроки вполне реальна, и она не может быть решена множественным точным совпадением. База данных по ДНК содержит коллекцию предварительно расшифрованных строк ДНК. Когда расшифровывается новая строка ДНК, она может входить в уже секвенированную строку, поэтому эффективный метод проверки такого предположения очень нужен. (Конечно, возможен и противоположный случай, когда новая строка содержит одну из строк базы данных, но это случай множественного точного совпадения.)

Одно, несколько патологическое, применение этой задачи о подстроке можно найти в упрощенном варианте процедуры, которая используется в США при идентификации останков военнослужащих. Собираются митохондриальные ДНК от живых военнослужащих, и по небольшому участку секвенируется ДНК от каждого человека. Расшифрованный участок имеет два ключевых свойства: он может быть надежно изолирован полимеразной цепной реакцией (см. словарь на с. 639) и строка ДНК в нем высоко варьируется (т. е., скорее всего, различна у двух индивидуумов). Этот интервал используется как “почти уникальный” идентификатор. Если требуется опознать убитого военнослужащего, митохондриальную ДНК извлекают из его останков. После выделения и расшифровки того же участка строку из останков можно сравнить с общей базой данных (или более узкой базой строк для пропавших военнослужащих). Вариант этой задачи с *подстрокой* встречается потому, что состояние останков не всегда позволяет провести полное выделение и расшифровку нужного интервала ДНК. В этом случае смотрят, не является ли выделенная и расшифрованная строка подстрокой одной из строк базы данных. Из-за возможных ошибок более реалистично пытаться вычислить длину наибольшей под строки, найденной и в новоизвлеченной ДНК, и в одной из строк базы данных. Эта наибольшая общая под строка сможет сузить возможности для идентификации личности. Задача о наибольшей общей под строке будет рассмотрена в п. 7.4.

Суммарная длина всех строк в базе данных, обозначаемая через m , предполагается большой. Что составляет хорошую структуру данных и алгоритм просмотра для задачи о подстроке? Есть два ограничения: база данных должна храниться в памяти малого размера, а каждый просмотр должен быть быстрым. Третье желательное свойство — начальная подготовка базы данных должна быть относительно быстрой.

Суффиксные деревья дают очень привлекательное решение этой задачи с базой данных. Обобщенное суффиксное дерево \mathcal{T} для строк в базе данных строится за время $O(m)$ и, что еще важнее, требует памяти только $O(m)$. Можно найти в базе данных любую отдельную строку S длины n , или установить ее отсутствие, за время $O(n)$. Как обычно, этот поиск сопровождается сравнением строки с путем в дереве, начиная с корня. Полная строка S присутствует в базе данных в том и только том случае, если путь сравнения достигает листа \mathcal{T} в точке, где проверяется последний символ S . Более того, если S — под строка строки из базы данных, то алгоритм может найти все строки в базе данных, содержащие S в качестве под строки. Это требует времени $O(n + k)$, где k — число вхождений под строки. Как и ожидалось, решение достигается просмотром под дерева ниже конца совпавшего пути для S . Если полная строка S не совпадает с путем в \mathcal{T} , то S не содержится ни в базе данных, ни в какой-либо из ее строк. Однако совпавший путь определяет самый длинный префикс S , который в базе данных содержится в качестве под строки.

Задача о подстроке — одно из классических применений суффиксных деревьев. Здесь результаты, достигаемые с помощью суффиксного дерева, потрясающи, и их не получить использованием алгоритмов Кнута–Морриса–Пратта, Бойера–Мура и даже Ахо–Корасика.

7.4. APL4: Наибольшая общая подстрока двух строк

Классическая задача анализа строк — найти наибольшую подстроку, общую для двух заданных строк S_1 и S_2 . Это задача о наибольшей общей подстроке (отличная от задачи о наибольшей общей подпоследовательности, которая будет рассматриваться в пп. 11.6.2 и 12.5 части III).

Например, если $S_1 = \text{superiorcalifornialives}$ и $S_2 = \text{sealiver}$, то наибольшей общей подстрокой S_1 и S_2 будет *alive*.

Эффективный и концептуально простой способ нахождения наибольшей общей подстроки открывается, если построить обобщенное суффиксное дерево для S_1 и S_2 . Каждый лист этого дерева представляет собой либо суффикс одной из этих двух строк, либо суффикс их обеих. Пометим каждую внутреннюю вершину v числом 1 (или 2), если в поддереве v существует лист, представляющий собой суффикс строки S_1 (или, соответственно, S_2). Путевая метка любой внутренней вершины с пометкой 1 или 2 есть подстрока, общая для S_1 и S_2 , и самая длинная такая строка и есть наибольшая общая подстрока. Так что алгоритм должен только найти вершину, имеющую наибольшую строковую глубину (число символов в пути до нее) среди вершин с пометками 1 и 2. Построение суффиксного дерева может быть выполнено за линейное время (пропорциональное суммарной длине S_1 и S_2), а пометка вершин и вычисления строковой глубины — с помощью стандартных методов обхода дерева, линейных по времени.

В итоге:

Теорема 7.4.1. *Наибольшая общая подстрока двух строк с использованием обобщенного суффиксного дерева может быть найдена за линейное время.*

Хотя задача о наибольшей общей подстроке выглядит теперь, благодаря нашим познаниям о суффиксных деревьях, тривиальной, интересно отметить, что в 1970 г. Дон Кнут предположил, что линейный по времени алгоритм для этой задачи невозможен [24, 278]. Мы вернемся к ней в п. 7.9, где предложим решение, более эффективное по памяти.

Вспомним теперь задачу об идентификации останков из п. 7.3. Она свелась к нахождению наибольшей подстроки в одной фиксированной строке, которая входит также в одну из строк базы данных. Решение этой задачи прямо обобщает решение задачи о наибольшей общей подстроке и оставляется читателю.

7.5. APL5: Распознание загрязнения ДНК

Часто различные лабораторные приемы, используемые для изоляции, очистки, клонирования, копирования, поддержания, апробирования или расшифровки строки ДНК, приводят к тому, что нежелательная ДНК включается в рассматриваемую строку или смешивается с коллекцией строк. Серьезной проблемой может быть и загрязнение белком в лаборатории. Во время клонирования загрязнение часто

вызывается фрагментом (подстрокой) *вектора* (строки ДНК), используемого для включения желательной ДНК в организм хозяина, или ДНК самого хозяина (например, бактерии или дрожжей). Загрязнения могут вызываться также очень малыми количествами нежелательной посторонней ДНК, которая физически встраивается в желательную ДНК и затем амплифицируется (умножается) с помощью полимеразной цепной реакции (polymerase chain reaction — PCR), используемой для изготовления копий нужной ДНК. Не углубляясь в эти и другие специфические способы загрязнения, мы ссылаемся на общий феномен *загрязнения ДНК*.

Загрязнение — крайне серьезная проблема, известны случаи крупномасштабных попыток секвенирования ДНК, где использование сильно загрязненных библиотек клонов привело к тому, что огромная работа оказалась напрасной. Появившееся несколько лет назад сообщение о том, что удалось успешно получить ДНК из кости динозавра, также рассматривается сейчас как в лучшем случае преждевременное. “Полученные” расшифровки ДНК оказались в результате поиска в базе данных ДНК больше похожими на ДНК млекопитающих (прежде всего, человека) [2], чем на ДНК птиц и крокодилов, заставляя думать, что большая часть анализируемой ДНК имеет человеческое происхождение, а не выделена из динозавров. Д-р С. Блайр Хеджес, один из критиков этого исследования, утверждал: “Просматривая ДНК динозавров, мы все сначала нашли материал, который выглядел как гены динозавров, но потом было доказано его человеческое происхождение, так что мы перешли к другим вопросам. Но этот случай был опубликован” [80].

Этих затруднений можно было бы избежать, если бы секвенированные последовательности сначала проверялись на признаки правдоподобных загрязнений, до использования их в серьезном анализе и до публикации результатов. Рассел Дулиттл пишет: “...менее радостно отметить, что далеко не единичны случаи, когда исследование было урезано из-за того, что предварительный поиск этой последовательности распознал ее как обычный контаминант... использованный в процессе очистки. Как правило, экспериментатор должен проводить поиск раньше и чаще” [129].

Очевидно, что важно знать, нет ли в изучаемой ДНК загрязнений. Кроме общего беспокойства о правильности получаемой расшифровки загрязнение может сильно усложнить задачу “дробовой сборки последовательности” (shotgun sequence assembly) (рассмотренную в пп. 16.14 и 16.15), в которой короткие строки секвенируемой ДНК, полученные после ее разрыва во многих местах, собираются в длинные строки просмотром перекрывающихся подстрок.

Часто последовательности ДНК для многих возможных контаминантов (загрязнителей) уже известны. Среди них — клонированные векторы (cloning vectors), праймеры (primers) для PCR, полные геномные последовательности организма-хозяина (например, дрожжей) и другие источники ДНК, с которыми работают в лаборатории. (История с динозаврами здесь не совсем подходит, так как надежной расшифровки ДНК человека еще нет.) Хорошей иллюстрацией может служить изучение нематоды *C. elegans*, одного из важнейших модельных организмов молекулярной биологии. При обсуждении необходимости использовать YAC (Yeast Artificial Chromosomes — искусственные хромосомы дрожжей) для расшифровки генома *C. elegans* проблема загрязнения ДНК и ее потенциальное решение были поставлены так:

“Главная трудность заключается в неизбежном загрязнении очищенных YAC значительными количествами ДНК дрожжей. Такое загрязнение вызывает потерю большого времени, затраченного на расшифровку и сборку ненужных дрожжевых последовательностей. Однако эту трудность нужно обходить (используя)... полную (дрожжевую)

последовательность... После этого можно будет исключить все расшифрованные части, распознанные как ДНК дрожжей, и сосредоточиться исключительно на ДНК *C. elegans*" [225].

Это побуждает к изучению следующей вычислительной задачи.

Задача загрязнения ДНК. Пусть задана строка S_1 (вновь выделенная и расшифрованная строка ДНК) и известная строка S_2 (комбинация источников возможного загрязнения). Требуется найти все подстроки S_2 , которые встречаются в S_1 и длины которых не меньше некоторого заданного l . Эти подстроки являются кандидатами ненежелательные места S_1 , которые загрязнили желательную строку ДНК.

Эта задача легко может быть решена за линейное время развитием подхода, обсуждавшегося выше в связи с задачей о наибольшей общей подстроке двух строк. Построим обобщенное суффиксное дерево для S_1 и S_2 . Затем пометим каждую внутреннюю вершину, которая имеет в своем поддереве лист, представляющий суффикс строки S_1 , и лист, представляющий суффикс S_2 . Наконец, впишем в результат все помеченные вершины, имеющие строковую глубину l и более. Если v — такая помеченная вершина, то путевая метка v есть подозрительная строка, которой может быть испорчена желательная строка ДНК. Если таких помеченных вершин со строковой глубиной выше порога l нет, тогда есть большая уверенность (но не определенность), что ДНК не загрязнена известными контаминалтами.

В более общей формулировке имеется целый набор известных строк ДНК, которые могут загрязнять желательную строку ДНК. Задача состоит в том, чтобы определить, входят ли в заданную строку ДНК достаточно длинные подстроки (скажем, длины не менее l) из известного набора возможных нарушителей. Подход заключается в том, чтобы строить обобщенное суффиксное дерево для набора \mathcal{P} возможных контаминалтков вместе с анализируемой строкой S_1 и затем маркировать каждую внутреннюю вершину, в поддереве которой содержатся листы, представляющие собой суффикс S_1 и суффикс какого-либо образца из \mathcal{P} . Все помеченные вершины строковой глубины l и более идентифицируют подозрительные подстроки.

Обобщенные суффиксные деревья можно строить за время, пропорциональное общей длине строк в дереве, а все остальные действия по маркировке и поиску, описанные выше, можно выполнить за линейное время стандартными методами обхода деревьев. Следовательно, суффиксные деревья можно использовать для решения задачи о загрязнении за линейное время. Напротив, может ли за линейное время решить эту задачу алгоритм Ахо–Корасика, неясно, так как он предназначен для поиска вхождений в S_1 полных образцов из \mathcal{P} , а не подстрок образцов.

Как и в задаче о наибольшей общей подстроке, имеется решение задачи о загрязнении, более эффективное по памяти, которое основано на материале п. 7.8. Мы оставляем это читателю.

7.6. APL6: Общие подстроки более чем двух строк

Один из наиболее важных вопросов, который задают относительно набора строк, таких: какие подстроки наиболее общие для большого числа различных строк? Он сильно отличается от важной задачи поиска подстрок, которые многократно встречаются в отдельной строке.

В биологических строках (ДНК, РНК или белковой молекуле) задача нахождения подстрок, общих для большого числа различных строк, встречается в самых разнообразных контекстах. Мы скажем об этом много больше, когда будем обсуждать поиск в базе данных в главе 15 и множественное сравнение строк в главе 14. Наиболее прямо задача нахождения общих подстрок приложима к поиску функционально значимых участков ДНК. Дело в том, что мутации, появляющиеся в ДНК при расхождении (дивергенции) двух видов, быстрее изменяют те участки ДНК или белка, которые менее важны функционально. Части ДНК или белка, критичные для правильного функционирования молекулы, сохраняются лучше, потому что мутации, появляющиеся в этих регионах, будут, скорее всего, летальны. Поэтому нахождение подстрок ДНК или белка, встречающихся у очень многих видов, позволяет указать участки или подобразцы, которые могут быть критичны для функции или структуры биологической строки.

Не так прямо задача нахождения (точного совпадения) общих подстрок в наборе различных строк встречается как составная часть многих эвристических методов, развитых в биологической литературе для *выравнивания* набора строк. Эта задача, именуемая множественным выравниванием, будет рассматриваться более подробно в п. 14.10.3.

Биологические приложения побуждают к анализу следующей задачи точного совпадения: по заданному набору строк найти подстроки, “общие” для большого числа этих строк. Слово “общие” здесь означает “встречающиеся с равенством”. Более сложна задача найти “похожие” подстроки во многих данных строках, где “похожесть” допускает небольшое число различий. Задачи этого типа будут обсуждаться в части III.

Формальная постановка задачи и первый метод

Пусть мы имеем K строк, сумма длин которых равна n .

Определение. Для каждого k между 2 и K определим $l(k)$ как длину самой длинной подстроки, общей для не меньше чем k строк.

Мы хотим вычислить таблицу из $K - 1$ элементов, в которой элемент k содержит $l(k)$ и указывает на одну из общих подстрок этой длины. Например, рассмотрим набор строк $\{sandollar, sandlot, handler, grand, pantry\}$. Для него значения $l(k)$ (без указателей на строки) таковы:

k	$l(k)$	Одна из подстрок
2	4	<i>s and</i>
3	3	<i>and</i>
4	3	<i>and</i>
5	2	<i>a n</i>

Удивительно, что эта задача может быть решена за линейное время $O(n)$ [236]. Действительно здорово, что так много информации о содержании и подструктуре строк можно извлечь за время, пропорциональное времени их чтения. Линейный по времени алгоритм будет полностью рассмотрен в главе 9, где строится метод с константным временем для нахождения наименьшего общего предшественника.

Чтобы подготовиться к результату $O(n)$, мы покажем здесь, как решать эту задачу за время $O(Kn)$. Такая граница тоже нетривиальна, но она достигается обобщением метода нахождения наибольшей общей подстроки для двух строк. Построим сначала обобщенное суффиксное дерево \mathcal{T} для K строк. Каждый лист этого дерева представляет суффикс одной из K строк и помечен одним из K уникальных строковых идентификаторов от 1 до K , указывающим, из которой строки взят этот суффикс. Каждой из K строк сопоставлен собственный терминальный символ, так что одинаковые суффиксы из разных строк заканчиваются в разных листьях дерева. Значит, каждый лист в \mathcal{T} имеет только один строковый идентификатор.

Определение. Для каждой внутренней вершины v из \mathcal{T} определим $C(v)$ как число различных строковых идентификаторов при листьях, входящих в поддерево v .

Когда известны числа $C(v)$ и строковые глубины всех вершин, искомые значения $l(k)$ можно получить за линейный по времени обход дерева. Этот обход строит вектор V , в котором $V(k)$ для k от 2 до K хранит строковую глубину (и положение, если хочется) самой глубокой (в смысле строковой глубины) вершины v , для которой $C(v) = k$. (Для нахождения вершины v с $C(v) = k$ нужно сравнить строковую глубину v с текущим значением $V(k)$, и если глубина вершины v больше, чем $V(k)$, то положить $V(k)$ равным глубине v .) В сущности, $V(k)$ показывает длину наибольшей строки, которая встречается ровно k раз. Поэтому $V(k) \leq l(k)$. Чтобы найти $l(k)$, нужно просто просмотреть V от наибольшего до наименьшего индекса, записывая в каждой позиции максимальное из просмотренных значений $V(k)$. То есть если $V(k)$ пусто или $V(k) < V(k+1)$, то $V(k)$ полагается равным $V(k+1)$. Получающийся вектор содержит искомые значения $l(k)$.

7.6.1. Вычисление значений $C(v)$

За линейное время легко вычислить для каждой внутренней вершины v число листьев в поддереве v . Но это число может быть больше, чем $C(v)$, так как два листа в поддереве могут иметь один и тот же идентификатор. Именно повтор идентификаторов затрудняет вычисление $C(v)$ за время $O(n)$. Поэтому вместо подсчета числа листьев ниже v алгоритм за время $O(Kn)$ явно считает, какие идентификаторы нашлись ниже каждой вершины. Для всякой внутренней вершины v создается битовый вектор длины K , в котором i -й бит установлен в 1, если в поддереве v присутствует идентификатор i . $C(v)$ равно числу единиц в этом векторе. Вектор для v получается логическим сложением векторов для потомков v . Для l детей это занимает время lK . Поэтому для всего дерева, так как в нем число дуг составляет $O(n)$, на построение всей таблицы уйдет время $O(Kn)$. Мы вернемся к этой задаче в п. 9.7, где будет дано решение со временем $O(n)$.

7.7. APL7: Построение меньшего ориентированного графа для точного совпадения

Как говорилось выше, во многих приложениях критическим ресурсом является память, и любое ее значительное уменьшение представляет интерес. В этом пункте мы посмотрим, как сжать суффиксное дерево в *ориентированный ациклический*

граф (directed acyclic graph — DAG), который можно использовать для решения задачи о точных совпадениях (и других задач) за линейное время и который требует меньше памяти, чем дерево. К методам сжатия можно прибегнуть также для построения *ориентированного ациклического графа слова* (directed acyclic word graph — DAWG); он является наименьшим конечным автоматом, распознающим суффиксы данной строки. Линейные по времени алгоритмы для построения DAWG описывались в [70, 71, 115]. Таким образом, метод, представленный здесь для сжатия суффиксных деревьев, может рассматриваться либо как приложение суффиксных деревьев для построения DAWG, либо просто как техника сокращения суффиксных деревьев.

Рассмотрим суффиксное дерево для строки $S = \text{хухахаха}$, представленное на рис. 7.1. Поддерево дуговых меток ниже вершины p изоморфно поддереву ниже вершины q , за исключением номеров листьев. То есть для каждого пути из p есть путь из q с теми же путевыми метками, и обратно. Если мы хотим только определить, действительно ли образец присутствует в большем тексте, а не искать места всех его вхождений, мы можем слить p с q , перенаправив помеченную дугу из родителя p в q , удаляя поддерево для p , как показано на рис. 7.2. Получающийся граф будет уже не деревом, а ориентированным ациклическим графом.

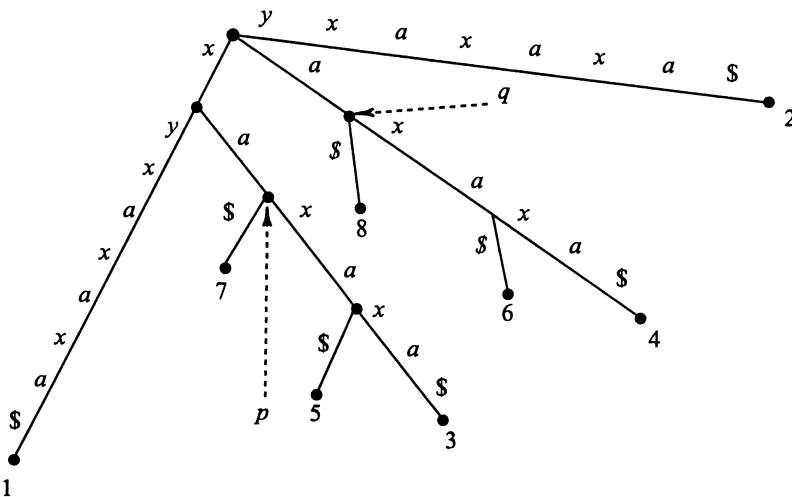


Рис. 7.1. Суффиксное дерево для строки хухахаха без изображения суффиксных связей

Ясно, что после слияния двух вершин суффиксного дерева ориентированный граф можно использовать для решения задачи о точном совпадении так же, как использовалось суффиксное дерево. Алгоритм сравнивает символы из образца с единственным путем из корня графа, и образец присутствует где-то в тексте в том и только том случае, если все символы из образца уложились вдоль пути. Однако номера листьев, достижимых из конца пути, уже могут не давать точных начальных позиций вхождений. Этому вопросу посвящено упражнение 10.

Так как после первого слияния мы уже имеем дело с ациклическим графом, алгоритм должен уметь сливать вершины и в ациклическом графе (DAG¹). Операция

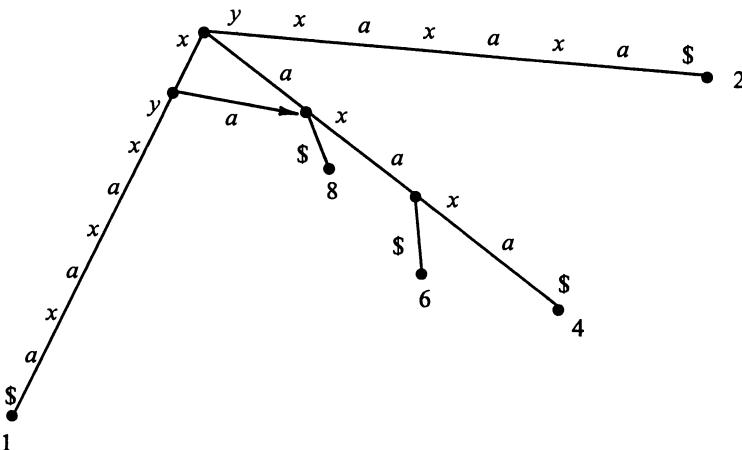


Рис. 7.2. Ориентированный ациклический граф, используемый для распознавания подстрок *хухахаха*

слияния, которая годится и для деревьев, и для DAG, формулируется следующим образом:

Присоединение вершины p к вершине q означает, что все дуги, выходящие из p , удаляются, а дуги, входящие в p , направляются в q , но сохраняют каждая свою метку и что любая часть графа, ставшая недостижимой из корня, удаляется.

Хотя эти присоединения вообще-то происходят в DAG, критерии, используемые для определения того, какие вершины присоединять, остаются привязанными к исходному суффиксному дереву — вершина p может быть присоединена к q , если поддерево p с его дуговыми метками изоморфно поддереву q с его метками в суффиксном дереве. Более того, p можно присоединить к q или q — к p , только если эти два дерева изоморфны. Так что главный алгоритмический вопрос — как найти изоморфные поддеревья суффиксного дерева. Существуют общие алгоритмы для изоморфизма поддеревьев, но суффиксные деревья имеют дополнительную структуру, делающую распознание изоморфизма много проще.

Теорема 7.7.1. В суффиксном дереве \mathcal{T} поддерево ниже вершины p с дуговыми метками изоморфно поддереву ниже вершины q в том и только том случае, если существует путь из суффиксных связей из одной вершины в другую и число листьев в этих двух поддеревьях одинаково.

Доказательство. Сначала предположим, что p обладает прямой суффиксной связью с q и эти две вершины имеют в своих поддеревьях по одному числу листьев. Поскольку существует суффиксная связь из p в q , то путевая метка вершины p представляется в виде $x\alpha$, где α — путевая метка q . Для каждого листа с номером i в поддереве p найдется лист с номером $i+1$ в поддереве q , так как суффикс T , начинающийся в i , имеет начало $x\alpha$, только если суффикс T , начинающийся в $i+1$, имеет начало α . Поэтому для каждого (помеченного) пути из p в лист его поддерева есть идентичный путь (с теми же помеченными дугами) из q в лист его поддерева.

Далее, количества листьев в поддеревьях p и q предположены равными, так что *каждый* путь из q идентичен какому-то пути из p , и следовательно, эти два поддерева изоморфны.

По тем же причинам если есть путь из суффиксных связей из p в q , идущий через вершину v , то число листьев в поддереве v должно быть не меньше, чем в поддереве p , и не больше, чем в поддереве q . Так как у p и q число листьев в поддеревьях одинаково, оно такое же и во всех промежуточных вершинах, и все их поддеревья изоморфны друг другу.

С другой стороны, пусть поддеревья p и q изоморфны. Ясно, что число листьев в них одинаково. Покажем, что существует путь из суффиксных связей между p и q . Пусть α — путевая метка p , а β — путевая метка q . Предположим, что $|\beta| \leq |\alpha|$.

Поскольку $\beta \neq \alpha$, то, если β — суффикс α , она должна быть собственным суффиксом. А если так, то по свойствам суффиксных связей должен быть путь из суффиксных связей из p в q , и теорема будет доказана. Так что будем доказывать от противного, что β должна быть суффиксом α .

Допустим, что β не является суффиксом α . Рассмотрим любое вхождение α в T , и пусть γ — суффикс T , расположенный точно вправо от этого вхождения α . Это означает, что $\alpha\gamma$ является суффиксом T и существует путь с меткой γ , идущий из вершины p в лист суффиксного дерева. Теперь, так как β не является суффиксом α , никакой суффикс T , начинающийся сразу после вхождения β , не может иметь длину $|\gamma|$, и поэтому нет пути длины $|\gamma|$ из q в какой-нибудь лист. Но из этого следует, что поддеревья с корнями p и q не изоморфны, что дает противоречие. \square

Определение. Пусть Q — множество всех таких пар (p, q) , что 1) существует суффиксная связь из p в q в дереве \mathcal{T} , и 2) p и q имеют по одинаковому числу листьев в их поддеревьях.

Теперь процедура сокращения суффиксного дерева может быть описана полностью.

Сокращение суффиксного дерева

begin

Задать множество Q таких пар (p, q) , что существует суффиксная связь из p в q и число листьев в соответствующих им поддеревьях совпадает.

while существует пара (p, q) из Q и обе вершины пары содержатся в текущем ациклическом дереве do присоединить вершину p к q .

end.

“Корректность” получающегося DAG формально декларируется в следующей теореме.

Теорема 7.7.2. Пусть \mathcal{T} — суффиксное дерево для вводимой строки S , и D — DAG, получающийся при выполнении алгоритма сокращения \mathcal{T} . Любой путь в D из корня просматривает подстроку S , и любая подстрока S просматривается каким-либо из таких путей. Поэтому задача определения того, является ли данная строка подструктурой S , может быть решена за линейное время с помощью D вместо \mathcal{T} .

Ациклический граф D можно использовать для определения того, входит ли образец в текст, но граф, кажется, должен потерять информацию о началах вхождений.

Однако можно добавить к графу простую (линейную по размеру памяти) информацию, так что при обходе графа будет устанавливаться и положение всех вхождений. Обсуждение этой возможности оставляется до упражнения 10.

Может удивить, что пары в алгоритме соединяются в произвольном порядке. Проверку корректности (доказательство необходимости в теореме 7.7.2) мы оставляем как упражнение. В практическом отношении разумно делать слияния сверху вниз, никогда не сливая двух вершин, имеющих предшественников по суффиксному дереву, которые сами могут быть слиты.

DAG или DAWG

Ациклический граф D , созданный алгоритмом, отличается от ациклического графа слова, определенного в [70, 71, 115]. DAWG представляет собой конечный автомат, и поэтому у него каждая дуговая метка может иметь только один символ. Более того, основное теоретическое свойство DAWG для строки S в том, что он является конечным автоматом с наименьшим числом состояний (вершин) среди автоматов, распознающих суффиксы S . Разумеется, D можно преобразовать в конечный автомат, разложив каждую дугу, метка которой состоит из k символов, в k дуг, помеченных каждая одним символом. Но получающийся конечный автомат не будет иметь непременно минимальное число состояний и, следовательно, не обязательно совпадает с DAWG для S .

Тем не менее DAG D для строки S имеет столько же (или меньше) вершин и дуг, как и соответствующий DAWG этой строки S , и, значит, так же компактен, хотя он и не является конечным автоматом. Поэтому конструирование DAWG для S имеет главным образом теоретический интерес. В упражнениях 16 и 17 мы посмотрим, как построить наименьший конечный автомат для распознания подстрок строки.

7.8. APL8: Обратная роль суффиксных деревьев и существенное уменьшение памяти

Мы уже показали ранее, как можно применить суффиксные деревья для решения задачи о точном совпадении, расходяя препроцессное время и память $O(m)$ (строя суффиксное дерево размера $O(m)$ для текста T) и поисковое время $O(n + k)$, где n — длина образца и k — число вхождений. Мы видели также, как используются суффиксные деревья для решения множественной задачи о точном совпадении с теми же границами времени и памяти (n теперь равно полному размеру всех образцов в наборе). Напротив, метод Кнута–Морриса–Пратта (или Бойера–Мура) предварительно обрабатывает образец, расходя время и место $O(n)$, и затем организует поиск за время $O(m)$. Метод Ахо–Корасика дает такие же границы для множественной задачи о совпадении.

Асимптотически методы, использующие суффиксное дерево и делающие препроцессинг текста, так же эффективны, как методы, в которых обрабатывается образец, — и те и другие считают за время $O(n + m)$ и требуют память $\Theta(n + m)$ (они должны представлять строки). Однако практические константы в границах времени и памяти для суффиксных деревьев часто делают их непривлекательными по сравнению с другими методами. Более того, иногда встречается ситуация, когда образец

(или образцы) задается первым и остается неизменным, тогда как текст варьируется. В таких случаях явно предпочтительнее подготовить образец. Так что возникает вопрос, не следует ли решать эти задачи, строя суффиксное дерево для *образца*, а не для текста. Этот подход обратен нормальному применению суффиксных деревьев. В пп. 5.3 и 7.2.1 мы упоминали, что такая обратная роль возможна и она позволяет использовать суффиксные деревья для достижения тех же оценок по времени и памяти (препроцессинг в сравнении с временем поиска и памятью), как и в методах Кнута–Морриса–Пратта и Ахо–Корасика. Чтобы объяснить это, мы обратимся к результату Чанга и Лаулера [94], которые решили несколько более общую задачу, называемую задачей о *статистике совпадений*.

7.8.1. Статистика совпадений: те же границы и меньшая память

Определение. Определим $ms(i)$ как длину наибольшей подстроки T , начинающейся с позиции i , которая совпадает *где-то* (но мы не знаем, где) с подстрокой P . Эти значения называются *статистикой совпадений*.

Например, если $T = abcxabcdex$ и $P = wyabcwzqabcdw$, то $ms(1) = 3$ и $ms(5) = 4$.

Ясно, что вхождение P , начинающееся с позиции i строки T , существует в том и только том случае, если $ms(i) = |P|$. Таким образом, задача нахождения статистики совпадений является обобщением задачи о точном совпадении.

Статистика совпадений приводит к уменьшению памяти

Статистику совпадений можно использовать для уменьшения размера суффиксного дерева, необходимого при решении задач, более сложных, чем точное совпадение. Это использование статистики совпадений будет, вероятно, более важным, чем при повторении оценок трудоемкости препроцессинга и поиска для методов Кнута–Морриса–Пратта и Ахо–Корасика. Первый пример уменьшения памяти благодаря статистике совпадений появится в п. 7.9.

Статистика совпадений применяется также и во множестве других приложений, описанных в этой книге. Не удержимся от рекламы: статистика совпадений будет центральной в методе поиска быстрого приближенного совпадения, разработанной для ускоренного поиска в базе данных. Уточнения см. в п. 12.3.3. Таким образом, статистика совпадений строит мост между методами точного совпадения и задачами приближенного совпадения строк.

Как вычислить статистику совпадений

Мы хотим вычислить $ms(i)$ для каждой позиции i в тексте T за время $O(m)$, обращаясь только к суффиксному дереву для P . Сначала построим для P суффиксное дерево \mathcal{T} , но не будем удалять суффиксные связи, использованные при построении дерева. (Суффиксные связи либо создаются алгоритмом Укконена, либо обратны указателям связи в алгоритме Вайнера.) Это суффиксное дерево и применяется для поиска всех $ms(i)$.

Наивный путь нахождения одного значения $ms(i)$ состоит в сравнении слева направо начальных символов $T[i..m]$ с деревом \mathcal{T} по единственному пути совпадений, продолжаемому, пока это возможно. Однако повтор этой процедуры для каждого i

не дал бы объявленной линейной границы для времени. Для ускорения вычислений используются суффиксные связи подобно тому, как это делалось при построении \mathcal{T} в алгоритме Укконена.

Чтобы найти $ms(1)$, мы сравниваем символы строки T с \mathcal{T} , следуя единственному пути совпадения для $T[1..m]$. Длина совпадающего пути равна $ms(1)$. Теперь предположим, что алгоритм уже прошел по совпадающему пути для нахождения $ms(i)$, где $i < |m|$. Это означает, что алгоритм нашел в \mathcal{T} такую точку b , что путь до нее точно совпал с префиксом $T[i..m]$, но дальнейшее совпадение уже невозможно (может быть, потому, что достигнут лист).

Имея $ms(i)$, получим $ms(i+1)$ следующим образом. Если b — внутренняя вершина v дерева \mathcal{T} , то алгоритм может следовать по ее суффиксной связи в вершину $s(v)$. Если вершина b не внутренняя, то алгоритм может подняться к вершине v , предшествующей b . Если v — корень, то поиск $ms(i+1)$ начинается с корня. Но если v не корень, то алгоритм переходит по суффиксной связи из v в $s(v)$. Путевая метка v , скажем, $x\alpha$, является префиксом $T[i..m]$, так что строка α должна быть префиксом $T[i+1..m]$. Но $s(v)$ имеет α в качестве путевой метки, и следовательно, путь из корня в $s(v)$ совпадает с префиксом $T[i+1..m]$. Поэтому поиск $ms(i+1)$ можно начинать вместо корня в вершине $s(v)$.

Пусть β обозначает строку между вершиной v и точкой b . Тогда $x\alpha\beta$ есть наибольшая подстрока в P , которая совпадает с подстрокой, начинающейся в позиции i из T . Следовательно, $\alpha\beta$ является строкой в P , которая совпадает с подстрокой T , начинающейся в позиции $i+1$. Так как $s(v)$ имеет путевую метку α , то должен быть путь с меткой β , выходящий из $s(v)$. Вместо прохода по этому пути с проверкой на нем каждого символа алгоритм использует прием скачка по счетчику (детализированный в алгоритме Укконена, п. 6.1.3) для прохода по нему за время, пропорциональное числу вершин в пути.

Когда достигается конец этого пути β , алгоритм продолжает сравнивать отдельные символы из T с символами из дерева, пока либо он не дойдет до листа, либо дальнейшие совпадения не окажутся невозможными. В любом случае $ms(i+1)$ будет строковой глубиной последней позиции. Заметим, что сравнения символов после достижения конца пути β начинаются либо с того же символа в T , на котором кончился поиск для $ms(i)$, либо со следующего символа в T — в зависимости от того, чем закончился этот поиск: несовпадением или листом.

Есть один особый случай, который может встретиться при вычислении $ms(i+1)$. Если $ms(i) = 1$ или $ms(i) = 0$ (так что алгоритм в корне) и $T(i+1)$ не в P , то $ms(i+1) = 0$.

7.8.2. Анализ корректности и времени для статистики совпадений

Корректность метода доказывается немедленно, так как он просто имитирует наивный метод нахождения каждого $ms(i)$. Рассмотрим теперь расход времени. Его анализ очень похож на анализ для алгоритма Укконена.

Теорема 7.8.1. Имея суффиксное дерево для P и копию T , полную статистику совпадений можно найти за время $O(m)$.

Доказательство. Поиск для любого $ms(i+1)$ начинается с подъема не более чем на одну дугу из позиции b в вершину v и перехода по одной суффиксной связи в вершину $s(v)$. Из $s(v)$ путь β проходится за время, пропорциональное числу

вершин в нем, а затем выполняется некоторое число дополнительных сравнений символов. Переходы назад и по дугам занимают для каждого i константное время, поэтому в пересчете на весь алгоритм они требуют времени $O(m)$. Для оценки полного времени на проход различных путей β вспомним понятие вершинной глубины из временного анализа алгоритма Укконена (с. 136). Было доказано, что переходы по связям уменьшают текущую глубину не больше чем на единицу (лемма 6.1.2), и поскольку каждый возврат уменьшает текущую глубину на единицу, ее полное уменьшение не может превзойти $2m$. Но так как текущая глубина не может превзойти m или стать отрицательной, то полный размер ее увеличений не превосходит $3m$. Эти увеличения происходят на шагах переходов β , и значит, общее число таких переходов также не превосходит $3m$. Осталось рассмотреть полное время, затраченное на все сравнения символов, сделанные в переходах “после β ”. Ключевой момент в том, что эти сравнения “после β ”, используемые в вычислении $ms(i+1)$ при $i > 1$, начинаются с того символа в T , на котором кончается расчет $ms(i)$, либо со следующего символа. Поэтому сравнения “после β ”, выполненные при вычислении $ms(i)$ и $ms(i+1)$, имеют не более одного общего символа. Отсюда следует, что сравнений “после β ” будет не больше $2m$. Этим исчерпывается вся работа по сбору статистики совпадений, и теорема доказана. \square

7.8.3. Маленькое, но важное обобщение

Число $ms(i)$ показывает длину наибольшей подстроки, начинающейся в позиции i из T , которая совпадает с подстрокой *где-то* в P , но не указывает места какого-либо из таких совпадений в P . Для некоторых приложений (таких, как в п. 9.1.2) мы должны также знать для каждого i место по крайней мере одной такой совпадающей подстроки. Модифицируем алгоритм нахождения статистики совпадений так, чтобы он давал и эту информацию.

Определение. Для каждой позиции i в T число $p(i)$ указывает начальную позицию в P , такую что подстрока, начинающаяся в $p(i)$, совпадает с подстрокой, начинающейся в позиции i в T на длину $ms(i)$.

Для того чтобы накопить значения $p(i)$, нужно сначала обойти в глубину дерево \mathcal{T} , пометив каждую вершину v каким-либо одним номером из номеров листьев ее поддерева; назовем этот номер суффиксным номером v . Время на это действие линейно зависит от размера \mathcal{T} . Затем при использовании \mathcal{T} для нахождения каждого $ms(i)$ если поиск останавливается в вершине u , то искомое $p(i)$ равно суффиксному номеру u , в противном случае (когда поиск останавливается на дуге (u, v)) $p(i)$ есть суффиксный номер v .

Назад к STS

Вспомним обсуждение STS (маркированных локализаций) в п. 3.5.1. Было упомянуто, что вследствие ошибок требование точного совпадения может быть не вполне подходящим способом для поиска STS в новых последовательностях ДНК. Но так как число ошибок расшифровки вообще-то мало, мы можем ожидать длинных зон соответствия между новой последовательностью ДНК и любой STS, которую она (в идеале) содержит. Эти зоны соответствия должны допускать корректную идентификацию содержащихся в них STS. Используя (предварительно подготовленное)

обобщенное суффиксное дерево для STS (которые играют роль P), можно вычислить статистику совпадений для новой последовательности ДНК (которая и есть T) и набора STS. В общем случае $p(i)$ будет указывать на подходящее STS в суффиксном дереве. Мы оставим читателю облачить все это в детали. Заметим, что при задании новой последовательности время вычислений всего лишь пропорционально ее длине.

7.9. APL9: Эффективный по памяти алгоритм нахождения наибольшей общей подстроки

В п. 7.4 мы решили задачу нахождения наибольшей общей подстроки строк S_1 и S_2 с помощью построения обобщенного суффиксного дерева для этих двух строк. Решение потребовало $O(|S_1| + |S_2|)$ времени и памяти. Но так как память, необходимая для построения и использования суффиксного дерева, превосходит практические возможности, решение, которое строит суффиксное дерево только для меньшей из этих строк, может быть предпочтительнее, даже если оценки памяти для наихудшего случая остаются теми же. Ясно, что наибольшая общая подстрока имеет длину, равную наибольшей статистике совпадений $ms(i)$. Нужная подстрока встречается в более длинной строке, начиная с позиции i , и в более короткой строке, начиная с позиции $p(i)$. Алгоритм из предыдущего пункта вычисляет все значения $ms(i)$ и $p(i)$, используя только суффиксное дерево для меньшей из двух строк и двигаясь по копии длинной строки. Следовательно, учет статистики совпадений уменьшает память, необходимую для решения задачи о наибольшей общей подстроке.

Задача о наибольшей общей подстроке иллюстрирует один из многих случаев экономии памяти в алгоритмах, использующих суффиксные деревья, за счет статистики совпадений. Некоторые их применения будут упомянуты ниже, но возможных приложений много больше, и мы просто не можем на все указать. Читатель приглашается проверить каждое использование суффиксных деревьев, в которое входит больше одной строки, чтобы найти самостоятельно те места, где возможно такое уменьшение памяти.

7.10. APL10: Проверка совпадения суффикса с префиксом во всех парах

Здесь мы представим более сложное использование суффиксных деревьев, которое интересно и само по себе, и как центральная конструкция в алгоритме приближенной надстроки с линейным временем, обсуждаемом в п. 16.17.

Определение. Пусть заданы две строки, S_i и S_j . Любой суффикс S_i , который совпадает с префиксом S_j , называется *суффиксно-префиксным совпадением* S_i , S_j .

Пусть задана коллекция строк $\mathcal{S} = S_1, S_2, \dots, S_k$ полной длины m . Задачей о *суффиксно-префиксных совпадениях всех пар* называется задача нахождения для всех упорядоченных пар S_i, S_j из \mathcal{S} наибольшего суффиксно-префиксного совпадения S_i, S_j .

Мотивация задачи

Основная мотивация к решению задачи о суффиксно-префиксных совпадениях всех пар идет из ее использования при реализации быстрых приближенных алгоритмов для задачи о кратчайшей надстроке (которая обсуждается в п. 16.17). Задача о надстроке сама мотивируется задачами расшифровки и картирования ДНК, рассмотренных в главе 16. Другая мотивировка для задачи о кратчайшей надстроке и, следовательно, для задачи о суффиксно-префиксных совпадениях всех пар появляется при сжатии данных, эта связь обсуждается в упражнениях к главе 16.

Другое, непосредственное, приложение задачи о суффиксно-префиксных совпадениях всех пар предложено в вычислениях, которые описаны в [190]. В этом исследовании набор, содержащий приблизительно 1400 EST (см. п. 3.5.1) из организма *C. elegans* (круглый червь), изучали на наличие подстрок высокой сохранности, называемых *древними консервативными участками* (*ancient conserved regions* — ACR). Одной из главных целей исследования была оценка числа ACR, появляющихся в генах *C. elegans*, с тем чтобы экстраполировать его на множество EST. Чтобы описать роль в этой экстраполяции суффиксно-префиксных совпадений, нам нужно вспомнить некоторые факты относительно EST.

Здесь мы можем считать EST расшифрованной подстрокой ДНК длиной около 300 нуклеотидов, происходящей из гена значительно большей длины. Если EST α происходит из гена β , то реальное расположение подстроки α в β можно считать случайным, и в одном и том же гене β может быть расположено много различных EST. Однако в общем методе, используемом для отбора EST, идентифицировать исходный ген не удается, поэтому сказать, происходят ли два EST из одного и того же гена, трудно. Более того, EST из одних генов выделяются чаще, чем из других. Обычно легче выделить EST из генов, которые экспрессируются (транскрибируются) с более высокой частотой. Таким образом, мы можем рассматривать EST как смещенную выборку из изучаемых генных последовательностей. Вернемся теперь к проблеме экстраполяции.

Задача заключается в том, чтобы использовать данные об ACR, наблюдаемые в EST, для оценки числа ACR во всем множестве генов. Простая экстраполяция была бы обоснована, если бы EST были случайными, выбранными равновероятно из всего множества генов *C. elegans*. Однако они выбраны не равновероятно, и простая экстраполяция была бы ошибочной, если распространенность ACR у часто и нечасто выраженных генов существенно различна. Как определить эту распространенность? По EST невозможно опознать ген, из которого она получена, и частоту появления такого гена. Как же отличить EST из часто выбираемых генов от остальных?

Подход, предложенный в [190], заключается в том, что вычисляется “перекрывание” пар EST. Так как длины всех EST сравнимы между собой, центральная часть вычисления состоит в решении задачи о суффиксно-префиксных совпадениях всех пар в наборе EST. Та EST, которая не имеет существенных перекрытий с другими EST, считается взятой из редко встречающегося (и экспрессируемого) гена, а EST, существенно перекрывающаяся с одной или более другими EST, — из часто встречающегося гена. (Так как возможны ошибки расшифровки и нарушения подстрок среди строк неравной длины, следует также решить для всех пар задачу о наибольшей общей подстроке.) После такой систематизации EST действительно было обнаружено, что ACR встречаются чаще в EST из высокоэкспрессируемых генов (более

точно — из EST, которые перекрываются с другими EST). Объясняя это, авторы заключают:

“По этим результатам можно предположить, что умеренно экспрессирующиеся белки в среднем меньше изменялись в течение длительных периодов эволюции, чем слабо экспрессирующиеся, и в частности, более вероятно, что их гены содержат ACR. Это, скорее всего, можно приписать более высокому селекционному давлению на оптимизацию деятельности и структур этих белков...” [190].

7.10.1. Решение задачи о суффиксно-префиксных совпадениях всех пар за линейное время

Для одной пары строк препроцессинг, рассмотренный в п. 2.2.4, найдет наибольшее суффиксно-префиксное совпадение за время, линейно зависящее от суммарной длины этих строк. Однако применение этого препроцессинга к каждой из k^2 пар строк отдельно дает полное время порядка $O(km)$. Использование суффиксных деревьев позволяет уменьшить время счета до $O(m + k^2)$, в (обычном) предположении, что алфавит фиксирован.

Определение. Назовем дугу *терминальной*, если ее метка состоит только из терминального символа. Ясно, что концом каждой такой дуги будет лист, но не все дуги, кончающиеся в листьях, терминальны.

Основной структурой данных при решении задачи о суффиксно-префиксных совпадениях всех пар будет обобщенное суффиксное дерево $\mathcal{T}(\mathcal{S})$ для k строк набора \mathcal{S} . При построении $\mathcal{T}(\mathcal{S})$ алгоритм строит также список $L(v)$ для каждой внутренней вершины v . Список $L(v)$ содержит индекс i в том и только том случае, если v инцидентна терминальной дуге, лист которой помечен суффиксом строки S_i . Это значит, что $L(v)$ содержит индекс i в том и только том случае, если путевая метка до v есть полный суффикс строки S_i . Например, рассмотрим обобщенное суффиксное дерево, представленное на рис. 6.11 (с. 152). Вершина с путевой меткой *ba* имеет L -список, состоящий из отдельного индекса 2, вершина с путевой меткой *a* — список, состоящий из индексов 1 и 2, а вершина с путевой меткой *xa* — список из индекса 1. Все другие списки в этом примере пусты. Ясно, что эти списки могут быть построены одновременно с деревом $\mathcal{T}(\mathcal{S})$ (или после этого) за линейное время.

Рассмотрим теперь фиксированную строку S_j . Будем изучать путь от корня дерева $\mathcal{T}(\mathcal{S})$ к листу j , представляющему всю строку S_j . Ключевое наблюдение таково: если v — вершина на этом пути и i принадлежит $L(v)$, то путевая метка v является суффиксом S_i , который совпадает с префиксом S_j . Таким образом, для каждого индекса i самая глубокая вершина v на пути к листу j , для которой $j \in L(v)$, определяет наибольшее совпадение между суффиксом S_i и префиксом S_j . Путевая метка v является наибольшим суффиксно-префиксным совпадением (S_i, S_j) . Легко видеть, что одним проходом от корня до листа j мы можем найти самые глубокие вершины для всех $1 \leq i \leq k$ ($i \neq j$).

Согласно сделанному наблюдению, алгоритм эффективно отбирает искомые суффиксно-префиксные совпадения просмотром $\mathcal{T}(\mathcal{S})$ в глубину. При этом поддерживается k стеков, по одному для каждой строки. Во время этого просмотра, когда вершина v достигается при проходе вниз по дуге, она загружается в i -й стек для каждого $i \in L(v)$. При достижении листа j (представляющего целую строку S_j) нужно

просмотреть все k стеков и записать для каждого индекса i текущую верхушку i -го стека. Нетрудно видеть, что верхушка стека i содержит вершину v , которая определяет суффиксно-префиксное совпадение (S_i, S_j) . Если стек i пуст, то перекрытие суффикса S_i и префикса S_j отсутствует. Когда проход в глубину делает шаг назад из вершины v , мы выталкиваем верхушки тех стеков, индексы которых лежат в $L(v)$.

Теорема 7.10.1. *Все k^2 самых длинных суффиксно-префиксных совпадений находятся описанным алгоритмом за время $O(m + k^2)$. Так как m — размер исходных данных, а k^2 — размер результата, алгоритм оптимален по времени.*

Доказательство. Полное число индексов во всех списках $L(v)$ равно $O(m)$. Число дуг в $\mathcal{T}(\mathcal{S})$ также $O(m)$. Каждая стековая операция связана с листом $\mathcal{T}(\mathcal{S})$, а каждый лист связан с не более чем одним включением в стек и одним исключением из стека; следовательно, просмотр $\mathcal{T}(\mathcal{S})$ и изменения стеков требуют времени $O(m)$. Запись каждого из $O(k^2)$ ответов требует константного времени на ответ. \square

Обобщения

Мы отметим два обобщения. Пусть $k' < k^2$ — число упорядоченных пар строк, которые имеют суффиксно-префиксное совпадение положительной длины. Используя двухсторонние ссылки, можно организовать цепной список *непустых стеков*. Тогда во время обхода дерева при достижении какого-либо листа нужно будет проверять только стеки из этого списка. Такой способ действий позволяет найти все непустые суффиксно-префиксные совпадения за время $O(m + k')$. Отметим, что позиция стеков в цепном списке может меняться, так как стек, который переходит от пустого состояния к непустому, должен быть прицеплен к одному из концов списка; следовательно, при каждом стеке должно содержаться имя связанной со стеком строки.

С другой стороны, предположим, что мы хотим отобрать для каждой пары не только самое длинное суффиксно-префиксное совпадение, а все суффиксно-префиксные совпадения, независимо от их длины. Можно модифицировать предложенное решение, чтобы при сканировании верхушек стеков прочитывалось полностью содержимое каждого сканируемого стека. Если размер результата имеет порядок k^* , то сложность этого решения будет $O(m + k^*)$.

7.11. Введение в повторяющиеся структуры в молекулярных строках

Несколько пунктов этой книги (7.12, 7.12.1, 9.2, 9.2.2, 9.5, 9.6, 9.6.1, 9.7 и 7.6), а также некоторые упражнения посвящены обсуждению эффективных алгоритмов поиска в строках различных типов *повторяющихся* структур. (На самом деле, некоторые аспекты одного типа повторяющихся структур, tandemные повторы, рассматривались уже в упражнениях к главе 1 и будут еще обсуждаться и дальше.) К общей теме повторяющихся структур в строках побуждают нас несколько источников, но наш основной интерес к ним идет от важных повторяющихся структур в биологических строках (ДНК, РНК и белок). Для конкретности мы коротко познакомим вас с некоторыми из этих повторяющихся структур. У нас нет намерения писать диссертацию о повторяющихся ДНК или белках, речь идет о мотивации для разрабатываемой алгоритмической техники.

7.11.1. Повторяющиеся структуры в биологических строках

Одна из наиболее поразительных черт ДНК (и в меньшей степени белка) — распространность повторяющихся подстрок в геноме. Это особенно верно для эукариот (высокоорганизованных организмов, ДНК которых располагается в ядре клетки). Например, большая часть Y-хромосомы человека состоит из повторяющихся подстрок, и более того:

“Семейства повторяющихся последовательностей составляют около одной трети человеческого генома” [317].

Имеется обширная литература по повторяющимся структурам в ДНК^{*)}, и даже в белках

“... сообщения о различных типах повторов так часто встречаются, что их трудно даже перечислить” [128].

При анализе 3.6 миллиона оснований ДНК *C. elegans* было идентифицировано более 7000 семейств повторяющихся последовательностей [5]. По контрасту с этим прокариоты (такие безъядерные, одноклеточные организмы, как бактерии, ДНК которых располагаются не в ядре) имеют в целом мало повторяющихся ДНК, хотя они еще и обладают некоторыми высоко структуризованными повторами малого размера.

Повторяющиеся последовательности ДНК нужно изучать не только потому, что их много, но также из-за разнообразия содержащихся в них повторяющихся структур, из-за возможных различных механизмов, объясняющих происхождение и поддержание повторов, и из-за биологических функций, которые могут обеспечивать некоторые из повторов (см. [394] по поводу одного аспекта генного дублирования). Во многих статьях (например, [317, 469, 315]) по генетике или молекулярной биологии можно найти развернутые дискуссии о повторяющихся строках и их предполагаемой функциональной и эволюционной роли. Начальные сведения о повторяющихся элементах в человеческой ДНК можно найти в [253, 255].

В следующем ниже обсуждении повторяющихся структур в ДНК и белках мы подразделяем такие структуры на три типа: локальные строки малого размера, функция или происхождение которых понятны по крайней мере частично; простые повторы, как локальные, так и разбросанные, функции которых менее ясны; более сложные разбросанные повторяющиеся строки, функции которых еще более сомнительны.

Определение. *Палиндром* — это строка, которая одинаково читается с начала и с конца.

Подчеркнем, что словарь Random House определяет “палиндром” как слово, предложение или стих, читающийся одинаково вперед и назад [441]. Например, строка *хуааух* по этому определению палиндром. Если игнорировать пробелы, то предложение *was it a cat i saw* дает еще один пример.

Определение. *Комплémentарный палиндром* — это строка ДНК или РНК, которая станет палиндромом, если каждый символ в одной половине строки заменить комплементарным для него символом (в ДНК

^{*)} В [192] сообщается, что поиск в базе данных MEDLINE по ключу (*repeat OR repetitive*) AND (*protein OR sequence*) показал, что за последние 20 лет на эту тему опубликовано 6 000 статей.

комплементарные пары *A-T* и *C-G*, в РНК — пары *A-U* и *C-G*). Например, *AGCTCGCGAGCT* — это комплементарный палиндром*).

Небольшие локальные повторы, функция и происхождение которых частично поняты, включают: *комплементарные палиндромы* в ДНК и РНК, которые участвуют в регулировании транскрипции ДНК (две части комплементарного палиндрома складываются и образуют структуру, напоминающую по форме “заколку для волос”); *вложенные комплементарные палиндромы* в тРНК (transfer RNA), которые позволяют молекуле свернуться в структуру типа клеверного листа за счет создания комплементарных пар; тандемные ряды повторяющихся РНК, которые окаймляют ретровирусы (вирусы, первичным генетическим материалом для которых служит РНК) и способствуют внедрению вирусной ДНК (получающейся из последовательности РНК в результате обратной транскрипции) в ДНК хозяина; единичные обращенные повторы, которые окаймляют “прыгающие” участки ДНК в различных организмах и этим способствуют их перемещению или смене ориентации ДНК; короткие повторяющиеся подстроки (палиндромные и непалиндромные) в ДНК, которые могут помочь хромосоме складываться в более компактную структуру; повторяющиеся подстроки на концах вирусной ДНК (в линейном состоянии), которые позволяют сцеплять много копий этой вирусной ДНК (молекулы такого типа называются *каптенанами*); копии генов, которые кодируют важные РНК (рРНК и тРНК), производимые в больших количествах; кластеры генов, кодирующих важные белки (такие, как гистон), которые регулируют структуру хромосомы и должны синтезироваться в больших количествах; семейства генов, кодирующие похожие белки (например, гемоглобины и миоглобины); похожие гены, возникающие, вероятно, в результате дупликации с последующей мутацией (включая *псевдогены*, которые “домутировались” до состояния, когда они уже не функционируют); общие экзоны эукариотической ДНК, которые могут быть основными строительными блоками для многих генов, и общие функциональные или структурные субъединицы в белке (мотивы и домены).



Рис. 7.3. Палиндром на языке молекулярной биологии. Двойная спираль совпадает сама с собой после отражения вокруг центра по горизонтали и вертикали. Каждая нить строки является комплементарным палиндромом в соответствии с определениями этой книги

Места рестрикции иллюстрируют другой тип малой структурированной повторяющейся подстроки, имеющий важное значение в молекулярной биологии. Рестриктаза — фермент, который распознает специфическую подстроку в ДНК прокариотов

*). Использование слова “палиндром” в молекулярной биологии не согласуется с обычным определением. Самый простой перевод “палиндрома” молекулярных биологов на нормальный язык — это “комплементарный палиндром”. Точка зрения, более близкая к молекулярному уровню, заключается в том, что палиндром — это сегмент двунитевой ДНК или РНК, такой что обе нити одинаковы, когда обе читаются в одном и том же направлении, скажем, от 5' к 3'. Альтернативный взгляд, что палиндром — это сегмент двунитевой ДНК, который симметричен (относительно отражения) вокруг горизонтальной оси, и средней точки сегмента (рис. 7.3). Так как эти две нити комплементарны друг другу, каждая нить является комплементарным палиндромом в смысле приведенного определения. В литературе по молекулярной биологии иногда для определения “палиндрома” в обычном смысле используется термин “зеркальный повтор”.

и эукариотов и разрезает (или рассекает) эту ДНК в каждом месте, где встречается этот образец (точнее, место рестрикции внутри образца зависит от образца). Известны сотни рестриктаз, и их использование имеет решающее значение почти во всех аспектах современной молекулярной биологии и технологии рекомбинирования ДНК. Например, удивительное открытие, что эукариотическая ДНК содержит *интраны* (подстроки ДНК, прерывающие кодовые области ДНК белка), за которое в 1993 г. была присуждена Нобелевская премия, тесно связано с открытием и использованием рестриктаз в конце 1970-х.

Места рестрикций — это любопытный пример повторов, поскольку они часто являются *комплémentарными палиндромными подстроками*. Например, рестриктаза *EcoRI* распознает комплементарный палиндром *GAATTTC* и разрезает его между *G* и смежным *A* (подстрока *TTC* после обращения и комплементации переходит в *GAA*). Другие рестриктазы распознают *отделенные* (или *прерванные*) комплементарные палиндромы. Например, рестриктаза *BglII* распознает *GCCNNNNNGGC*, где *N* обозначает любой нуклеотид. Энзим делает разрез между последними двумя *N*. Постулируется, что комплементарная палиндромная структура позволяет двум половинам комплементарного палиндрома (разделенным или нет) складываться и образовывать комплементарные пары. Это складывание затем само способствует либо распознанию, либо рестрикции ферментом. Ввиду палиндромической структуры мест рестрикций можно сканировать базы данных ДНК в поисках общих повторов этой формы для того, чтобы найти дополнительных кандидатов на неизвестные места рестрикций.

Простые повторы, которые хуже изучены, часто встречаются как *тандемные ряды* повторяющейся ДНК (последовательные повторяющиеся строки, называемые “прямыми повторами”). Например, строка *TTAGGG* появляется на концах каждой хромосомы человека в рядах, содержащих от одной до двух тысяч копий [332]. Считается, что некоторые тандемные ряды могут начинаться и продолжать расти в результате реализации постулируемого механизма *неравных кроссинговеров* в мейозе, хотя у этой теории есть серьезная оппозиция. С неравным кроссинговером в мейозе шансы на то, что в простом мейозе будет добавлено много копий, возрастают при росте числа существующих копий. Сейчас считается, что ряд генетических заболеваний (синдром ломкой хромосомы X, хорея Гентингтона, болезнь Кеннеди, миотоническая дистрофия, атаксия) вызывается возрастанием числа триплетных тандемных повторов ДНК. Триплетные повторы как-то нарушают синтез определенных белков. Более того, число триплетов в повторе растет от поколения к поколению, что объясняет, почему с каждым следующим поколением тяжесть болезни возрастает. Другие длинные тандемные ряды, состоящие из коротких строк, более обычны и широко распространены в геномах млекопитающих. Эти повторы называются *сателлитными ДНК* (различают микро- и минисателлитные ДНК), и их существование повсеместно используется в генетическом картировании и криминалистике. Часто встречаются распределенные по всему геному динуклеотидные повторы. Кроме триплетных в генетических заболеваниях человека могут играть роль и другие минисателлитные повторы [286].

Повторяющиеся ДНК, которые *рассеяны* по геномам млекопитающих и функции и происхождение которых менее ясны, подразделяются на SINE (*short interspersed nuclear sequences* — короткие диспергированные ядерные последовательности) и LINE (*long interspersed nuclear sequences* — длинные диспергированные ядерные

последовательности). Классическим примером SINE является семейство повторов *Alu*. *Alu* повторяется в человеческом геноме около 300 000 раз, занимая целых 5 % ДНК генома человека и других млекопитающих. Повторы *Alu* — это подстроки длиной около 300 нуклеотидов; они являются почти (но не точно) идентичными копиями, широко распространенными в геноме. Более того, внутренняя часть строки *Alu* сама состоит из повторяющихся подстрок длиной около 40, и последовательность *Alu* часто окаймляется (фланкируется) с каждой стороны tandemными повторами длиной 7–10. Участки окаймления являются обычно комплементарными палиндромными копиями друг друга. Таким образом, повторы *Alu* прекрасно иллюстрируют различные феномены, встречающиеся в повторяющихся ДНК. Описание повторов *Alu* см. в [254].

Одним из наиболее поразительных открытий в молекулярной генетике является феномен, называемый *геномным* (или *гаметным*) *импринтингом*, согласно которому конкретный аллель гена экспрессируется только тогда, когда наследуется от определенного родителя [48, 227, 391]. Иногда это может быть мать, а иногда — отец. Аллель не будет экспрессироваться или будет экспрессироваться неправильно, если он наследуется от “неправильного” родителя. Это противоречит классическому mendелевскому правилу *эквивалентности*, в соответствии с которым хромосомы (кроме Y-хромосомы) не имеют памяти о родителе, от которого они происходят, и тому, что один и тот же аллель, наследуемый от любого родителя, влияет на потомка одинаково. У мышей и людей найдено шестнадцать импринтных генных аллелей [48]. Пять из них должны наследоваться от матери, а остальные — от отца. Последовательности ДНК этих генов обладают тем свойством, что

“Они содержат область, богатую прямыми повторами, или тесно связаны с такой областью. Эти повторы, размер которых меняется от 25 до 120 пар оснований*), уникальны в соответствующих областях импринтинга, но не гомологичны друг другу или многократно повторяющимся последовательностям млекопитающих. Прямые повторы могут быть важной чертой гаметного импринтинга, поскольку они были найдены во всех импринтных генах, проанализированных к настоящему времени, и сохранились в процессе эволюции” [48].

Таким образом, вероятно, прямые повторы важны в генетическом импринтинге, но, подобно многим другим примерам повторяющихся ДНК, их функция и происхождение остаются тайной.

7.11.2. Использование повторяющихся структур в молекулярной биологии

Какое-то время большинство распространенных повторяющихся ДНК рассматривалось как помеха, не имеющая никакого функционального или экспериментального значения. Но сейчас во множестве методов существование повторяющихся ДНК реально используется. Для упомянутого уже генетического картирования требуется идентификация в ДНК элементов (или *маркеров*), которые сильно меняются от особи к особи и широко распространены по геному. Тандемные повторы являются как раз такими маркерами. Кратность повторов подстроки в ряду у разных индивидуумов различна. Отсюда термин, используемый для этих маркеров: *переменные по числу*

*) В этой цитате отсутствуют важные сведения о том, что прямые (тандемные) повторы в изучаемых генах [48] имеют общую длину около 1500 оснований.

тандемные повторы (variable number of tandem repeats — VNTR). VNTR встречаются часто и регулярно во многих геномах, включая геном человека, и являются удобными маркерами, необходимыми для генетического картирования в больших масштабах. VNTR используются во время поиска на генетическом уровне (в отличие от физического уровня) специфических дефектных генов и для построения судебных опознаний по ДНК (так как число повторов сильно варьируется между индивидуумами, то маленький набор VNTR может однозначно охарактеризовать человека в популяции). Тандемные повторы, состоящие из очень короткой подстроки, часто всего в два символа длиной, называемые *микросателлитами*, наиболее удобны как маркеры генетического картирования.

Существование многократно повторяющихся ДНК, таких как Alu, затрудняет некоторые типы крупномасштабного секвенирования ДНК (см. пп. 16.11 и 16.16), но их существование можно использовать в некоторых методах клонирования, картирования и поиска. Например, один общий подход к физическому картированию низкого разрешения (нахождение в истинном физическом масштабе интересующих областей генома) и к нахождению генов, вызывающих болезни, предполагает вставку частей человеческой ДНК, содержащих искомые области, в геном хомяка. Эта техника называется *соматической клеточной гибридизацией*. Каждая гибридная клетка хомяка включает в себя различные части человеческой ДНК, и эти гибридные клетки можно тестировать, чтобы идентифицировать специфическую клетку, имеющую такое же интересующее нас свойство, как человеческая. В этой клетке затем распознаются те части гибридного генома, которые происходят от человека. Но в чем существенное различие между ДНК человека и хомяка?

В одном из подходов для их разделения используются последовательности Alu. Характерные для ДНК человека последовательности Alu встречаются в человеческом геноме так часто, что содержатся в большинстве фрагментов ДНК с длиной выше чем в 20 000 оснований [317]. Поэтому фрагменты человеческой ДНК в гибриде можно опознать пробой на фрагменты Alu. Та же идея используется, чтобы изолировать человеческие *онкогены* (модифицированные гены роста, которые стимулируют некоторые виды рака) из человеческих опухолей. Фрагменты человеческой ДНК из опухоли помещаются сначала в клетки мыши. Клетки, которые получили фрагмент, содержащий онкоген, начинают изменяться и воспроизводиться быстрее, чем другие. Это помогает найти искомую область человеческой ДНК, но теперь ее нужно отделить от мышиной. Для идентификации человеческой части гибридного генома и здесь учитывается близость онкогена к последовательности Alu [471]. Родственная техника, также использующая близость интересующего фрагмента к последовательностям Alu, описана в [403].

Алгоритмические задачи для повторяющихся структур

Мы рассмотрим специфические задачи относительно повторяющихся структур в строках в нескольких пунктах этой книги.*.) Правда, не каждая задача о повторяющейся строке, которую мы будем обсуждать, вполне мотивируется биологической задачей или известным сейчас феноменом. Снова можно повторить возражение,

*.) По смыслу задача о наибольшей общей подстроке и задача о k -общей подстроке (пп. 7.6 и 9.7) также относятся к повторяющимся подстрокам. Однако повторы в этих задачах появляются в различных строках, а не внутри одной и той же строки. Это различие существенно и по формулировке задач, и по технике, используемой для их решения.

что первые задачи о повторяющейся строке, которые мы рассматриваем, относятся к *точным* повторам (хотя и с допущением комплементарности и обращений), тогда как большинство случаев повторяющихся ДНК подразумевают *почти* идентичные копии. Некоторые приемы работы с неточными палиндромами (комплементарными или нет) и неточными повторами обсуждаются в пп. 9.5 и 9.6. Приемы работы с более либеральными ошибками будут рассмотрены в этой книге ниже. Другое возражение касается того, что для повторов малой длины достаточно простой техники. Например, для поиска повторяющихся ДНК длины 10 есть смысл сначала построить таблицу всех 4^{10} возможных строк и затем сканировать изучаемую ДНК с отрезками длины 10, хешируя места подстрок в рассчитанную заранее таблицу.

Несмотря на эти возражения, близость вычислительных задач, которые мы будем обсуждать, к биологическим феноменам достаточна для обоснования разработки сложной техники работы с точными и близкими к точным повторами. Эти методы проходят тест “правдоподобия” того, что они или стоящие за ними идеи могут в будущем быть использованы в вычислительной биологии. В этом свете рассмотрим задачи, относящиеся к точным повторам подстроки в отдельной строке.

7.12. APL11: Нахождение всех максимальных повторяющихся структур за линейное время

Прежде чем строить алгоритмы для нахождения повторяющихся структур, мы должны аккуратно определить эти структуры. Плохое определение может привести к лавинному выводу информации. Например, если строка состоит из n копий одного и того же символа, алгоритм, ищащий все пары идентичных подстрок (резонное первоначальное определение повторяющихся структур), выведет в результат $\Theta(n^4)$ пар, что явно нежелательно. Другие плохие определения могут не охватывать интересные структуры или затруднить рассуждения о них. Неудачные определения особенно запутывают дело при работе со множеством всех повторов конкретного типа. Поэтому ключевая задача — определить повторяющиеся структуры так, чтобы не генерировать лавинного вывода, но отследить достаточно ясным образом все осмысленные феномены. В этом пункте мы подходим к вопросу через разные понятия *максимальности*. Другие способы определения и изучения повторяющихся структур рассматриваются в упражнениях 56, 57 и 58 этой главы, в упражнениях к другим главам и в пп. 9.5, 9.6 и 9.6.1.

Определение. *Максимальная пара* (или максимальная повторяющаяся пара) в строке S — это такая пара одинаковых подстрок α и β из S , что ближайший символ слева (справа) от α отличен от ближайшего символа слева (справа) от β . Это значит, что продолжение α и β в любом направлении разрушило бы равенство этих двух строк.

Определение. Максимальная пара представляется тройкой (p_1, p_2, n') , где p_1 и p_2 задают начальные позиции двух подстрок, а n' — их длину. Для строки S определим $\mathcal{R}(S)$ как множество всех троек, описывающих максимальные пары в S .

Например, рассмотрим строку $S = xabciiizabcqabcyxar$, в которую трижды входит подстрока abc . Первое и второе вхождения образуют максимальную

пару (2, 10, 3), второе и третье — также образуют максимальную пару (10, 14, 3), тогда как первое и третье вхождения максимальной пары не образуют. Два вхождения строки $abcy$ также образуют максимальную пару (2, 14, 4). Отметим, что определение позволяет двум подстрокам максимальной пары перекрывать друг друга. Например, $cxxxaxxaxxb$ содержит максимальную пару с подстрокой $xxaxx$.

В общем, мы хотим разрешить префиксу и суффиксу S участвовать в максимальных парах. Например, два вхождения xa в $habcyiiizabcqabcyrxar$ должны бы рассматриваться в качестве максимальной пары. Чтобы смоделировать этот случай, просто добавим в начале и конце S символ, который не встречается в S . Начиная с этого места, будем считать, что это уже сделано.

Иногда представляет интерес нахождение и вывод всего множества $\mathcal{R}(S)$. Однако в некоторых ситуациях размер $\mathcal{R}(S)$ может быть слишком велик, и вполне достаточно, даже предпочтительно, более сжатое представление максимальных пар.

Определение. Определим *максимальный повтор* α как подстроку S , которая входит в какую-либо максимальную пару. Это значит, что α есть *максимальный повтор* в S , если существует такая тройка $(p_1, p_2, |\alpha|) \in \mathcal{R}(S)$, что α входит в S , начиная с позиций p_1 и p_2 . Пусть $\mathcal{R}'(S)$ обозначает множество максимальных повторов в S .

Например, в рассмотренной S обе строки, abc и $abcy$, являются максимальными повторами. Заметим, что независимо от того, сколько раз строка участвует в максимальных парах в S , в множестве $\mathcal{R}'(S)$ она представлена лишь один раз. Следовательно, размер $|\mathcal{R}'(S)|$ не превосходит размера $|\mathcal{R}(S)|$ и обычно значительно меньше. Вывод такой информации много скромнее и достаточно отражает максимальные пары.

В некоторых приложениях определение максимального повтора неадекватно моделирует желаемое понятие повторяющейся структуры. Например, в строке $S = a\alpha b x \alpha y a\alpha b$ подстрока α — максимальный повтор, но максимальным повтором является и подстрока $a\alpha b$, которая для α есть *надстрока*, хотя не каждое вхождение α в этой надстроке содержится. Может быть не всегда желательным выводить α как повторяющуюся структуру, так как большая подстрока $a\alpha b$, которая иногда содержит α , может быть более информативной.

Определение. *Супермаксимальным повтором* называется максимальный повтор, который не входит в качестве подстроки в другие максимальные повторы.

Максимальные пары, максимальные повторы и супермаксимальные повторы —ими исчерпываются возможные способы определения точных повторяющихся структур, которыми мы будем интересоваться. Другие модели точных повторов даны в упражнениях. Задачи, относящиеся к палиндромным и тандемным повторам, рассматриваются в некоторых местах книги. Неточные повторы будут обсуждаться в 9.5 и 9.6.1. Некоторые типы повторов элегантно представляются в графической форме с помощью конструкции, называемой *ландшафтом* [104]. Эффективная программа для построения ландшафта базируется в основном на суффиксных деревьях, она также описана в упомянутой работе. В следующих пунктах мы детально рассмотрим, как эффективно находить все максимальные пары, максимальные повторы и супермаксимальные повторы.

7.12.1. Линейный по времени алгоритм для нахождения всех максимальных повторов

Простейшей является задача нахождения всех максимальных повторов. Используя суффиксное дерево, их можно найти в строке длины n за время $O(n)$. Более того, существует компактное представление всех максимальных повторов, и его можно построить за время $O(n)$, несмотря на то что полная длина всех максимальных повторов может быть порядка $\Omega(n^2)$. Следующая лемма дает необходимое условие для того, чтобы подстрока была максимальным повтором.

Лемма 7.12.1. *Пусть \mathcal{T} — суффиксное дерево для строки S . Если строка α является максимальным повтором в S , то α должна быть путевой меткой какой-либо вершины v в \mathcal{T} .*

Доказательство. Если α есть максимальный повтор в S , то должны существовать по крайней мере две копии α , в которых символы справа от входления различны. Следовательно, α является путевой меткой какой-то вершины v из \mathcal{T} . \square

Ключевым моментом в лемме является то, что путь α должен кончаться в вершине \mathcal{T} . Это приводит сразу же к следующему неожиданному факту:

Теорема 7.12.1. *В любой строке длины n может быть не более n максимальных повторов.*

Доказательство. Так как \mathcal{T} имеет n листьев и каждая внутренняя вершина, отличная от корня, должна иметь не меньше двух детей, \mathcal{T} может включать в себя не более n внутренних вершин. Из леммы 7.12.1 прямо следует утверждение теоремы. \square

Теорема 7.12.1 была бы тривиальна, если бы для любой позиции i не больше одной начинающейся в ней подстроки могло входить в максимальную пару. Но это неверно. Например, в рассмотренной ранее строке $S = xabcu iiiizabcqabcyr$ обе копии подстроки $abcu$ участвуют в максимальных парах и каждая копия строки abc также участвует в максимальных парах.

Итак, теперь мы знаем, что для нахождения максимальных повторов нам нужно рассмотреть только строки, которые кончаются в вершинах суффиксного дерева \mathcal{T} . Но какие из этих вершин соответствуют максимальным повторам?

Определение. Для каждой позиции i строки S символ $S(i - 1)$ называется *левым символом i* . *Левым символом листа \mathcal{T}* называется левый символ начала суффикса, представляемого этим листом.

Определение. Вершина v дерева \mathcal{T} называется *различной влево*, если по меньшей мере два листа в поддереве v имеют различные левые символы. По определению лист не может быть различным влево.

Заметим, что свойство вершины быть различной влево распространяется вверх по дереву. Если вершина v различна влево, то все ее предшественники по дереву — тоже.

Теорема 7.12.2. *Строка α , помечающая путь к вершине v дерева \mathcal{T} , является максимальным повтором в том и только том случае, если v различна влево.*

Доказательство. Предположим сначала, что v различна влево. Это означает, что в S существуют подстроки $x\alpha$ и $y\alpha$, где x и y представляют различные символы.

Пусть за первой подстрокой следует символ p . Если за второй подстрокой следует символ, отличный от p , то α — максимальный повтор, и теорема доказана. Поэтому предположим, что мы имеем два вхождения $x\alpha p$ и $y\alpha p$. Но так как v является (ветвящейся) вершиной, то в S должна быть также подстрока αq с некоторым символом q , отличным от p . Если этому вхождению αq предшествует символ x , то оно входит в максимальную пару со строкой $y\alpha p$, а если ему предшествует y , то в максимальную пару с $x\alpha p$. В любом случае строке α не могут предшествовать одновременно x и y , так что α должно быть частью максимальной пары и, следовательно, максимальным повтором.

Обратно, если α является максимальным повтором, то она участвует в максимальной паре и должны существовать вхождения α с различными левыми символами. Следовательно, вершина v должна быть различна влево. \square

Максимальные повторы можно представить компактно

Так как свойство вершины быть различной влево распространяется вверх по \mathcal{T} , из теоремы 7.12.2 следует, что максимальные повторы в S представляются некоторой начальной частью суффиксного дерева. Более подробно, вершина называется “фронтальной” в \mathcal{T} , если она различна влево, но никакой из ее детей не различен влево. Поддерево \mathcal{T} от корня вниз к фронтальным вершинам в точности задает максимальные повторы в том смысле, что каждый путь от корня до вершины на границе или выше ее определяет максимальный повтор. Обратно, каждый максимальный повтор определяется одним таким путем. Такое поддерево, листья которого являются фронтальными вершинами \mathcal{T} , является компактным представлением^{*)} множества всех максимальных повторов в S . Заметим, что полная длина всех максимальных повторов может быть порядка $\Theta(n^2)$, но так как это представление является поддеревом \mathcal{T} , его полный размер имеет порядок $O(n)$ (включая символы для представления концевых меток). Таким образом, если вершины, различные влево, могут быть найдены за время $O(n)$, то древовидное представление множества максимальных повторов можно построить за время $O(n)$, даже несмотря на то что их полная длина может иметь порядок $\Omega(n^2)$. Опишем теперь алгоритм нахождения вершин, различных влево.

Нахождение вершин, различных влево, за линейное время

Для каждой вершины v дерева \mathcal{T} алгоритм либо записывает, что v различна влево, либо записывает символ, скажем, x , который является левым символом каждого листа в поддереве v . Алгоритм начинает с записи левого символа каждого листа суффиксного дерева \mathcal{T} . Затем он обрабатывает вершины \mathcal{T} снизу вверх. Чтобы обработать вершину v , проверяются ее дети. Если кто-либо из детей v различен влево, то записывается, что и v различна влево. Если никто из детей не различен влево, то проверяются символы, сопоставленные детям v . Если все эти символы равны, скажем, x , то x записывается в вершине v . Однако если они не все равны, то записывается, что v различна влево. Время проверки того, что все дети v имеют один и тот же записанный символ, пропорционально числу детей v . Следовательно, полное время работы алгоритма есть $O(n)$. Чтобы сформировать окончательное представление

^{*)} Этот тип дерева иногда называется *compact trie*, но мы не будем использовать эту терминологию. (По поводу слова *trie* и его перевода на русский язык см. примечание переводчика в книге: Кнут Д. *Искусство программирования*. Т. 3. М.: Вильямс, 2000. С. 527. — Прим. перев.)

множества максимальных повторов, нужно просто удалить из \mathcal{T} все вершины, которые не различны влево. Итак:

Теорема 7.12.3. Все максимальные повторы в S могут быть найдены за время $O(n)$, и древовидное представление для них может быть также построено из суффиксного дерева \mathcal{T} за время $O(n)$.

7.12.2. Нахождение супермаксимальных повторов за линейное время

Напомним, что супермаксимальный повтор — это максимальный повтор, который не является подстрокой никакого другого максимального повтора. Мы получим сейчас эффективные критерии для определения всех супермаксимальных повторов в строке S . Для этого решим более общую задачу нахождения *почти-супермаксимальных повторов*.

Определение. Подстрока α строки S является *почти-супермаксимальным повтором*, если α — максимальный повтор, который встречается *по меньшей мере один раз* в месте, где содержащего его максимального повтора нет. Такое вхождение α называется *свидетельством* почти-супермаксимальности α .

Например, в строке $aabxayaabxab$ подстрока α является максимальным повтором, но не супермаксимальным или почти-супермаксимальным, тогда как в $aabxayaab$ подстрока α также не будет супермаксимальным повтором, но будет почти-супермаксимальным. Об этом свидетельствует второе вхождение α .

В этой терминологии супермаксимальный повтор α является максимальным повтором, в котором каждое вхождение α свидетельствует о его почти-супермаксимальности. Отметим, что было бы неверно сказать, что множество почти-супермаксимальных повторов есть множество максимальных повторов, не являющихся супермаксимальными.

Используем суффиксное дерево \mathcal{T} для обнаружения почти-супермаксимальных и супермаксимальных повторов. Пусть v — вершина, соответствующая максимальному повтору α и w — один из ее детей (возможно, лист). Листья в поддереве \mathcal{T} , определяемом корнем w , идентифицируют положение некоторых (но не всех) вхождений подстроки α в S . Пусть $L(w)$ обозначает эти вхождения. Свидетельствуют ли такие вхождения α о почти-супермаксимальности α ?

Лемма 7.12.2. Если вершина w является внутренней в \mathcal{T} , то ни одно из вхождений α , определенных $L(w)$, не свидетельствует о почти-супермаксимальности α .

Доказательство. Пусть y — подстрока, помечающая дугу (v, w) . Каждый индекс в $L(w)$ задает некоторое вхождение αy . Но w — внутренняя, так что $|L(w)| > 1$, и αy является префиксом максимального повтора. Поэтому все вхождения α , задаваемые множеством $L(w)$, содержатся в максимальном повторе, который начинается с αy , и w не может быть свидетельством о почти-супермаксимальности α . \square

Таким образом, никакое вхождение α в $L(w)$ не может свидетельствовать о почти-супермаксимальности α , если только w не лист. Если w — лист, то w задает единичное вхождение подстроки $\beta = \alpha y$. Рассмотрим теперь этот случай.

Лемма 7.12.3. Предположим, что w — лист и i — (единичное) вхождение строки β , представляемое листом w . Пусть x — левый символ листа w . Тогда вхождение α в позиции i свидетельствует о почти-супермаксимальности α в том и только том случае, если x не является левым символом никакого другого листа ниже v .

Доказательство. Если имеется еще одно вхождение α с предшествующим символом x , то $x\alpha$ встречается дважды и либо является максимальным повтором, либо содержится в другом максимальном повторе. В этом случае вхождение α в позиции i содержится в максимальном повторе.

Если других вхождений α с предшествующим x нет, то $x\alpha$ встречается в S только один раз. Пусть теперь y — первый символ дуги из v в w . Так как w — лист, то αy входит в S только один раз. Поэтому вхождение α , начинающееся в i , которому предшествует x и за которым следует y , не содержится в максимальных повторах и поэтому свидетельствует о почти-супермаксимальности α . \square

Подводя итог, мы можем сформулировать следующую теорему.

Теорема 7.12.4. Внутренняя вершина v , различная влево, представляет почти-супермаксимальный повтор α в том и только том случае, если среди детей v есть лист (характеризующий, скажем, позицию i) и его левый символ $S(i - 1)$ не является левым символом никакого другого листа ниже v . Внутренняя вершина v , различная влево, представляет собой супермаксимальный повтор α в том и только том случае, если все ее дети являются листьями, и каждый лист имеет свой отдельный левый символ.

Таким образом, все супермаксимальные и почти-супермаксимальные повторы можно распознать за линейное время. Более того, мы можем определить степень почти-супермаксимальности α как долю вхождений α , которые свидетельствуют о ее почти-супермаксимальности. Эта степень каждого почти-супермаксимального повтора также может быть вычислена за линейное время.

7.12.3. Нахождение всех максимальных пар за линейное время

Обратимся теперь к вопросу о нахождении всех максимальных пар. Так как их может быть больше чем $O(n)$, время работы алгоритма будет определяться в терминах размера результата. Алгоритм обобщает метод, предложенный ранее для нахождения всех максимальных повторов.

Сначала построим суффиксное дерево для S . Для каждого листа, определяющего суффикс i , запишем его левый символ $S(i - 1)$. Теперь обойдем дерево снизу вверх, заходя в каждую вершину. Значит, в вершину v мы зайдем только после посещения всех ее детей. Во время посещения v мы создадим не более σ цепных списков, где σ — размер алфавита. Каждый список индексируется каким-то левым символом x . Список в v с индексом x содержит все начальные позиции подстрок в S , которые совпадают со строкой на пути до v и имеют левый символ x . Таким образом, это просто список номеров листьев ниже v , которые определяют такие суффиксы в S , которым непосредственно предшествует символ x .

Пусть $n = |S|$. Легко создать (но не поддерживать), двигаясь вверх по дереву, все эти списки за полное время $O(n)$. Чтобы создать список для символа x в вершине v , нужно сцепить (не копируя) списки для символа x , имеющиеся у всех детей v . Так как размер алфавита конечен, время на все эти сцепки константное. Сцепление без

копирования требуется для получения оценки времени $O(n)$. Прицепление списка, созданного в вершине v' , к другому списку разрушает список для v' . По счастью, списки, созданные для v' , не будут нужны после создания родительских списков.

Посмотрим теперь подробно, как использовать списки, имеющиеся у детей вершины v , для нахождения всех максимальных пар, содержащих α — путевую метку v . В начале посещения вершины v , до того как созданы списки для v , алгоритм может вывести все максимальные пары (p_1, p_2, α) . Для каждого символа x и каждого из детей v' вершины v алгоритм создает у каждого из детей v , отличных от v' , декартово произведение списка для x в v' с объединением каждого списка для символа, отличного от x . Каждая пара в этом списке задает начальные позиции максимальной пары для строки α . Доказательство этого по существу совпадает с доказательством теоремы 7.12.2.

Если число максимальных пар равно k , то метод требует времени $O(n + k)$. Создание суффиксного дерева, его обход снизу вверх и все сцепления списков требуют времени $O(n)$. Каждая операция, выполняемая в декартовом произведении, создает уникальную максимальную пару, так что на все эти операции затрачивается время $O(k)$. Если мы хотим только подсчитать число максимальных пар, то алгоритм можно модифицировать так, чтобы он исполнялся за время $O(n)$. Если требуются только максимальные пары определенной минимальной длины (этот случай характерен для многих приложений), то алгоритм можно модифицировать так, чтобы он исполнялся за время $O(n + k_m)$, где k_m — число максимальных пар длины не меньше требуемого минимума. Нужно просто останавливать обход снизу вверх в любой вершине, где строковая глубина опускается ниже этого минимума.

Подводя итог, мы имеем следующую теорему.

Теорема 7.12.5. *Все максимальные пары можно найти за время $O(n + k)$, где k — число максимальных пар. Если существует только k_m максимальных пар длины выше указанного порога, то все они могут быть найдены за время $O(n + k_m)$.*

7.13. APL12: Линеаризация циклической строки

Вспомним определение циклической строки S из упражнения 2 главы 1 (с. 33). Символы S первоначально нумеруются от 1 до n , начиная с какой-то точки в S .

Определение. При заданном упорядочении символов алфавита строка S_1 лексически (или лексикографически) меньше строки S_2 , если S_1 должна стоять раньше S_2 в нормальном словарном упорядочении этих строк. Это значит, что если при счете от левого конца этих двух строк i — первая позиция, где строки различны, то S_1 лексически меньше S_2 в том и только том случае, если $S_1(i)$ предшествует $S_2(i)$ в упорядочении алфавита, использованного для этих строк.

Для того чтобы справиться со случаем, когда S_1 является собственным суффиксом S_2 (и должно считаться лексически меньшим S_2), мы следуем соглашению, что пустое место считается первым символом алфавита.

Задача линеаризации циклической строки S , состоящей из n символов, заключается в следующем: выбрать место разреза S таким образом, чтобы получающаяся линейная строка была лексически наименьшей среди всех n возможных линейных строк, созданных разрезанием S .

Эта задача встречается в химических базах данных для циклических молекул. Каждая такая молекула представляется циклической строкой химических символов; для ускорения просмотра и сравнения молекул желательно хранить каждую циклическую строку в виде *канонической линейной строки*. Простая циклическая молекула сама бывает частью более сложной молекулы, так что задача может встречаться во “внутреннем цикле” более сложных задач химического поиска и сравнения.

Естественно для канонической линейной строки выбирать лексически наименьшую. С помощью суффиксных деревьев ее можно найти за время $O(n)$.^{*)}

7.13.1. Решение с помощью суффиксных деревьев

Разрежем циклическую строку S в произвольном месте, образовав линейную строку L . Затем удвоим L , образовав строку LL , и построим суффиксное дерево \mathcal{T} для LL . Как обычно, припишем к концу LL терминальный символ $\$$, который будем считать *старше*, чем любой символ алфавита, используемого для S . (Интуитивно ясно, что цель дублирования L в упрощении рассмотрения строк, составленных из суффикса L в начале и префикса L в конце.) Далее, проходим дерево \mathcal{T} таким образом, чтобы в каждой вершине двигаться по дуге с первым символом, наименьшим в смысле заданного упорядочения среди всех дуг, выходящих из вершины. Обход дерева продолжается, пока путь имеет строковую глубину n . Такая глубина будет всегда достижима (доказательство оставляется читателю). Любой лист l в поддереве той точки, где обход остановился, может быть использован для разрезания строки. Если $1 < l \leq n$, то разрез S между символами $l - 1$ и l создает лексически наименьшую линеаризацию циклической строки. Если $l = 0$ или $l = n + 1$, разрезать нужно между символами n и 1 . Каждый лист в этом поддереве дает точку разреза, приводящую к одной и той же линейной строке.

Корректность решения устанавливается легко и оставляется как упражнение.

Этот метод работает за линейное время и поэтому оптимален по времени. Другой метод с линейным временем и с меньшей константой предложен Шилочем [404].

7.14. APL13: Суффиксные массивы — большее сокращение памяти

В п. 6.5.1 мы видели, что, когда размер алфавита включается в оценки времени и памяти, построение суффиксного дерева для строки длины m требует либо памяти $\Theta(m|\Sigma|)$, либо времени $O(\min\{m \log m, m \log |\Sigma|\})$. Аналогично, поиск образца P длины n с применением суффиксного дерева может быть выполнен за время $O(n)$, только если использовать для дерева память $\Theta(m|\Sigma|)$ или предположить, что за константное время можно сделать до $|\Sigma|$ сравнений символов. В противном случае поиск требует $O(n \min\{\log m, \log |\Sigma|\})$ сравнений. По этим причинам суффиксное дерево может занимать слишком много памяти, чтобы оказаться практичным в некоторых приложениях. Следовательно, желательно иметь схему, более эффективную по памяти, но сохраняющую большую часть преимуществ поиска, присущих суффиксному дереву.

^{*)} Эта задача встречается и в одном из современных методов сжатия текстов, а именно в методе Барроуза-Уилера. — *Прим. перев.*

В контексте задачи о подстроке (см. п. 7.3), где поиск много раз ведется в фиксированной строке T , ключевыми показателями являются время, которое тратится на поиск, и память, используемая фиксированной структурой данных, представляющей T . Память, занимаемая при препроцессинге T , не имеет большого значения, хотя она все же должна быть “разумной”.

Манбер и Майерс [308] предложили новую структуру данных, названную *суффиксным массивом*, которая очень рационально использует память и может решать задачу о точном совпадении и задачу о подстроке почти так же эффективно, как суффиксное дерево. Похоже, что суффиксные массивы будут важным вкладом в решение некоторых задач вычислительной молекулярной биологии, где алфавит может быть большим (ниже мы обсудим некоторые причины появления больших алфавитов). Интересно, что, хотя более формальное определение суффиксного массива и основные алгоритмы для его построения и применения были развиты в [308], многие из этих идей были введены в биологическую литературу Мартинецом [310].

После определения суффиксных массивов мы покажем, как за линейное время конвертировать суффиксное дерево в суффиксный массив. В постановке этой задачи важна ясность. Стока T будет долгое время оставаться неизменной, а P будет изменяться. Поэтому наша цель заключается в нахождении представления для T , эффективного по памяти (суффиксный массив), которое будет сохраняться и облегчать решение задач поиска в T . Однако размер памяти, используемой при построении этого представления, не так критичен. В упражнениях мы рассмотрим способ построения самого представления, который более эффективен по памяти.

Определение. Пусть задана m -символьная строка T . *Суффиксным массивом* для T , обозначаемым Pos , называется массив целых чисел от 1 до m , определяющий лексический порядок m суффиксов строки T .

Это значит, что суффикс, начинающийся в позиции $Pos(1)$ строки T , является лексически наименьшим, и вообще суффикс $Pos(i)$ лексически меньше суффикса $Pos(i + 1)$.

Как обычно, мы припишем к концу T терминальный символ $\$$, но сейчас он будет считаться лексически меньше любого символа алфавита^{*)}, вопреки его интерпретации в предыдущем пункте. В качестве примера построим суффиксный массив для $T = mississippi$. Как видно из рис. 7.4, где суффиксы перечислены в лексическом порядке, $Pos = (11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3)$.

Отметим, что суффиксный массив состоит из целых чисел и, следовательно, не содержит информации об алфавите, используемом в строке T . Поэтому потребности в памяти для него скромны — на строку длины m требуется ровно m машинных слов в предположении, что число битов в машинном слове не меньше $\log m$.

При добавлении еще $2m$ значений (называемых *Lcp*-значениями и определяемых далее) суффиксный массив можно использовать для поиска всех вхождений в T образца P за $O(n + \log_2 m)$ операций посимвольного сравнения и подсчетов. Более того, эта оценка не зависит от размера алфавита. Так как для большинства интересующих нас задач $\log_2 m$ имеет порядок $O(n)$, задача о подстроке решается с помощью суффиксных массивов так же эффективно, как и с помощью суффиксных деревьев.

^{*)} Автор регулярно использует слово “лексический” для замены слова “лексикографический” и переносит этот термин и на порядок букв в алфавите. — Прим. перев.

```

11: i
8: ippi
5: issippi
2: ississippi
1: mississippi
10: pi
9: ppi
7: sippi
4: sisippi
6: ssippi
3: ssissippi

```

Рис. 7.4. Одиннадцать суффиксов строки *mississippi*, перечисленные в лексическом порядке. Начальные позиции этих суффиксов определяют суффиксный массив *Pos*

7.14.1. От суффиксного дерева к суффиксному массиву за линейное время

Предположим, что у нас достаточно памяти для построения суффиксного дерева для T (оно строится один раз в фазе препроцессинга), но это дерево нельзя сохранить на время (многочисленных) последующих поисков образцов. Поэтому мы конвертируем суффиксное дерево в суффиксный массив, рациональнее использующий память. Упражнения 53, 54 и 55 развивают альтернативный, более эффективный по памяти, но более медленный метод построения суффиксного массива.

Суффиксный массив для T можно получить из суффиксного дерева \mathcal{T} выполнением “лексического” обхода в глубину \mathcal{T} . После построения суффиксного массива суффиксное дерево разрушается.

Определение. Определим, что дуга (v, u) лексически меньше дуги (v, w) в том и только том случае, если первый символ дуги (v, u) лексически меньше первого символа (v, w) . (В этом приложении символ конца строки $\$$ лексически меньше любого другого символа.)

Так как метки всех дуг, выходящих из v , начинаются с разных символов, то эти дуги строго лексически упорядочены. Поэтому путь из корня \mathcal{T} , следующий по лексически наименьшей дуге каждой проходимой вершины, приводит к листу \mathcal{T} , представляющему лексически наименьший суффикс T . Более общо, обход \mathcal{T} в глубину, при котором дуги, выходящие из каждой вершины v , просматриваются в лексическом порядке, будет перебирать листья \mathcal{T} в лексическом порядке суффиксов, которые эти листья представляют. Суффиксный массив *Pos* окажется просто упорядоченным списком номеров суффиксов, перечисляющих листья \mathcal{T} при их лексическом поиске в глубину. Суффиксное дерево для T строится за линейное время, и обход также требует только линейного времени; поэтому мы получаем следующую теорему.

Теорема 7.14.1. *Суффиксный массив *Pos* для строки T длины m можно построить за время $O(m)$.*

Например, суффиксное дерево для $T = tartar$ представлено на рис. 7.5. Лексический обход суффиксного дерева в глубину посещает листья в порядке 5, 2, 6, 3, 4, 1, определяя этим значения массива *Pos*.

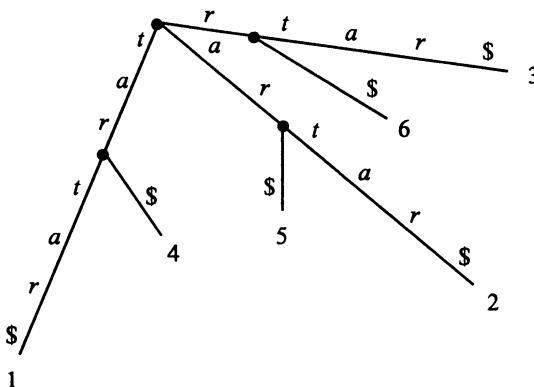


Рис. 7.5. Лексический обход суффиксного дерева в глубину посещает листья в порядке 5, 2, 6, 3, 4, 1

В качестве детали реализации отметим, что если ветви, выходящие из каждой вершины дерева, организованы в *упорядоченный* цепной список (как об этом говорилось в п. 6.5 на с. 152), то издержки на лексический поиск в глубину будут такими же, как на любой другой поиск в глубину. Каждый раз, когда этот поиск должен выбирать для перехода дугу, выходящую из v , он берет следующую дугу из цепного списка этой вершины.

7.14.2. Как искать образец, используя суффиксный массив

Суффиксный массив для строки T позволяет реализовать очень простой алгоритм поиска всех вхождений любого образца P в T . Здесь ключевым является тот факт, что если P входит в T , то все места этих вхождений будут расположены в Pos рядом. Например, $P = issi$ встречается в *mississippi*, начиная с позиций 2 и 5, которые в Pos стоят рядом (см. рис. 7.4). Таким образом, поиск вхождений P в T просто выполняет двоичный поиск в суффиксном массиве. Рассуждая подробнее, предположим, что P лексически меньше, чем суффикс в средней позиции Pos (т. е. суффикс $Pos([m/2])$). В этом случае первое место в Pos , содержащее такую позицию, где P входит в T , должно быть в первой половине Pos . Аналогично, если P лексически больше, чем суффикс $Pos([m/2])$, то места, где P входит в T , должны находиться во второй половине Pos . Следовательно, используя двоичный поиск, можно найти в Pos наименьший индекс i (если он существует), для которого P в точности совпадает с первыми n символами суффикса $Pos(i)$. Аналогично можно найти и наибольший индекс с таким свойством i' . Итак, образец P входит в T , начиная с любого места, задаваемого числами от $Pos(i)$ до $Pos(i')$.

Лексическое сравнение P с любым суффиксом занимает время, пропорциональное длине общего префикса этих двух строк. Она не превосходит n , следовательно:

Теорема 7.14.2. При использовании двоичного поиска в массиве Pos все вхождения P в T могут быть найдены за время $O(n \log m)$.

Конечно, истинное поведение алгоритма зависит от того, сколько длинных префиксов P встречается в T . Если их очень мало, то конкретное лексическое сравнение

действительно потребует времени $\Theta(n)$, и в общем граница $O(n \log m)$ будет довольно пессимистичной. В “случайных” строках (даже при больших алфавитах) этот метод работал бы за ожидаемое время $O(n + \log m)$. Если в T встречается много длинных префиксов P , метод можно улучшить с помощью двух приемов, описанных в следующих двух пунктах.

7.14.3. Простой ускоритель

В случае двоичного поиска обозначим через L и R , соответственно, левую и правую границы “текущего интервала поиска”. В момент старта имеем $L = 1$ и $R = m$. На каждой итерации двоичного поиска проверка идет в позиции $M = [(R + L)/2]$ массива Pos . Поисковый алгоритм помнит наибольшие префиксы $Pos(L)$ и $Pos(R)$, совпадающие с префиксом P . Пусть l и r обозначают, соответственно, длины этих префиксов. Положим $mlr = \min(l, r)$.

Значение mlr можно использовать, чтобы ускорить лексическое сравнение P и суффикса $Pos(M)$. Так как массив Pos обеспечивает лексический порядок суффиксов T , то для любого индекса i между L и R первые mlr символов суффикса $Pos(i)$ должны быть такими же, как у суффикса $Pos(L)$ и, следовательно, у P . Поэтому лексическое сравнение P и суффикса $Pos(M)$ можно начинать с позиции $mlr + 1$, а не с начала.

Поддержание mlr во время двоичного поиска немного усложняет алгоритм, но позволяет избежать многих ненужных сравнений. В начале поиска, когда $L = 1$ и $R = m$, P явно сравнивается с суффиксами $Pos(1)$ и $Pos(m)$, что дает значения l , r и mlr . Однако наихудшее время для этого улучшенного метода остается прежним — $O(n \log m)$. Майерс и Манбер [308] сообщают, что использование только mlr позволяет ускорить поиск в практических случаях до оценки наихудшего случая $O(n + \log m)$, которую мы вначале объявили. Так что мы представим полный метод, который гарантирует эту улучшенную оценку наихудшего случая, только из-за его элегантности.

7.14.4. Сверхускоритель

Назовем проверку символа в P *избыточной*, если этот символ был уже проверен ранее. Цель ускорения заключается в уменьшении числа избыточных проверок символов до не более одной на каждую итерацию двоичного поиска — следовательно, до не более $O(\log m)$ в общей сложности. Отсюда немедленно получится желаемая граница времени $O(n + \log m)$. Привлечение к этой цели только mlr результата не обеспечивает. Так как mlr есть минимум из l и r , то при $l \neq r$ все символы в P от $mlr + 1$ до максимума из l и r будут уже проверены, и любые сравнения этих символов будут избыточны. Теперь нужно определить, как начать сравнения с максимума из l и r .

Определение. $Lcp(i, j)$ есть длина наибольшего общего префикса (longest common prefix) суффиксов, определенных позициями i и j массива Pos , т.е. суффиксов $Pos(i)$ и $Pos(j)$.

Например, когда $T = mississippi$, суффикс $Pos(3)$ — это *issippi*, а суффикс $Pos(4)$ — это *ississippi*, и $Lcp(3, 4) = 4$ (см. рис. 7.4).

Для ускорения поиска алгоритм использует значения $Lcp(L, M)$ и $Lcp(M, R)$ для каждой тройки (L, M, R) , встречающейся в ходе двоичного поиска. Предположим, что мы можем получить эти значения за константное время, и покажем, как они могут помочь в поиске. Далее покажем, как вычислять конкретные значения Lcp , требуемые поиском, во время препроцессинга T .

Как использовать значения Lcp

Простейший случай. На любой итерации двоичного поиска если $l = r$, то сравнивать P с суффиксом $Pos(M)$, начиная с позиции $mlr + 1 = l + 1 = r + 1$, как раньше.

Общий случай. Когда $l \neq r$, не умаляя общности, предположим, что $l > r$. Тогда возможны три подслучаи:

- Если $Lcp(L, M) > l$, то общий префикс суффиксов $Pos(L)$ и $Pos(M)$ длиннее, чем общий префикс P и $Pos(L)$. Поэтому P совпадает с суффиксом $Pos(M)$ и за символом l . Другими словами, символ $l + 1$ у суффиксов $Pos(L)$ и $Pos(M)$ один и тот же, и он лексически меньше, чем символ $l + 1$ образца P (из-за того, что P лексически больше, чем суффикс $Pos(L)$). Следовательно, все начальные позиции вхождений P в T (если они существуют) должны находиться справа от позиции M в Pos . Итак, на любой итерации двоичного поиска, когда возникает такой случай, никаких проверок P не требуется, L просто заменяется на M , а l и r остаются неизменными (рис. 7.6).

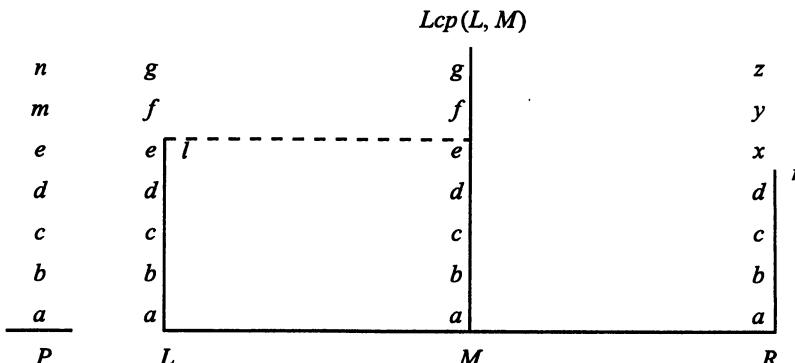


Рис. 7.6. Подслучай 1 сверхускорителя. Образец $P = abcde m n$ выписан вертикально снизу вверх. Суффиксы $Pos(L)$, $Pos(M)$ и $Pos(R)$ также выписаны вертикально. В этом случае $Lcp(L, M) > 0$ и $l > r$. Справа от M в массиве Pos должна встретиться любая стартовая позиция P в T , так как P совпадает с суффиксом $Pos(M)$ только до символа l .

- Если $Lcp(L, M) < l$, то общий префикс суффиксов $Pos(L)$ и $Pos(M)$ меньше, чем общий префикс суффикса $Pos(L)$ и P . Поэтому P совпадает с суффиксом $Pos(M)$ за символом $Lcp(L, M)$. Символ $Lcp(L, M) + 1$ у P и суффикса $Pos(L)$ один и тот же, и он лексически меньше, чем у суффикса $Pos(M)$. Следовательно, все начальные позиции вхождений P в T (если они существуют) должны находиться слева от позиции M в Pos . Итак, на любой итерации двоичного поиска, когда возникает такой случай, никаких проверок P не требуется, r заменяется на $Lcp(L, M)$, l остается неизменным, а R заменяется на M .

3. Если $Lcp(L, M) = l$, то P совпадает с суффиксом $Pos(M)$ за символом l . В этом случае алгоритм лексически сравнивает P с суффиксом $Pos(M)$, начиная с позиции $l + 1$. Таким образом, в результате этого сравнения определяется, которое из значений меняется, L или R , с соответствующим изменением l или r .

Теорема 7.14.3. *При использовании значений Lcp поисковый алгоритм делает не более $O(n + \log m)$ сравнений и работает за такое же время.*

Доказательство. Во-первых, простым анализом всех возможностей легко проверить, что во время двоичного поиска ни l , ни r убывать не могут. Кроме того, каждая итерация, которая заканчивает поиск, не проверяет ни одного символа из P или кончается после первого несовпадения, найденного на этой итерации.

В тех двух случаях ($l = r$ или $Lcp(L, M) = l > r$), когда алгоритм проверяет во время итерации символ, сравнения начинаются с символа $\max(l, r)$ образца P . Предположим, что на этой итерации проверено k символов P . Значит, обнаружено $k - 1$ совпадений, и в конце итерации $\max(l, r)$ увеличился на $k - 1$ (и это значение присвоено либо l , либо r). Следовательно, в начале любой итерации символ $\max(l, r)$ образца P может быть уже проверен, а следующий символ еще нет. Это означает, что на каждой итерации делается не более одного дублирующего сравнения. Таким образом, всего делается не более $\log_2 m$ избыточных сравнений. Число ненужных сравнений не превосходит n , что дает $n + \log m$ как оценку для общего числа. Вся остальная работа в алгоритме, очевидно, может быть выполнена за время, пропорциональное числу этих сравнений. \square

7.14.5. Как получить значения Lcp

Значения Lcp , необходимые для ускорения поиска, вычисляются заранее в препроцессной фазе при создании суффиксного массива. Сначала рассмотрим, сколько значений Lcp нам понадобится (чтобы обеспечить все возможные пути двоичного поиска). Для удобства предположим, что m является степенью 2.

Определение. Пусть B — полное двоичное дерево с m листьями, в котором каждая вершина помечена парой целых чисел (i, j) , $1 \leq i \leq j \leq m$. Корень B помечен парой $(1, m)$. Каждая нелистовая вершина (i, j) имеет двух детей, левый из них — с меткой $(i, \lfloor (i+j)/2 \rfloor)$ и правый — с меткой $(\lfloor (i+j)/2 \rfloor, j)$. Листья B имеют метки $(i, i+1)$ (плюс еще один лист с меткой $(1, 1)$) и упорядочены слева направо в порядке возрастания i (рис. 7.7).

По существу, метки вершин определяют границы (L, R) всех возможных интервалов поиска, которые могут появиться при двоичном поиске в упорядоченном списке длины m . Так как B — двоичное дерево с m листьями, оно имеет в общей сложности $2m - 1$ вершин. Таким образом, нужно предварительно вычислить только $O(m)$ значений Lcp . Предположение, что эти значения можно накопить в препроцессинге \mathcal{T} за время $O(m)$, выглядит правдоподобно. Но как именно? В следующей лемме мы покажем, что значения Lcp на листьях B легко накапливаются во время лексического обхода в глубину дерева \mathcal{T} .

Лемма 7.14.1. *При лексическом обходе в глубину дерева \mathcal{T} рассмотрим внутренние вершины, проверяемые между посещениями листа $Pos(i)$ и листа $Pos(i+1)$,*

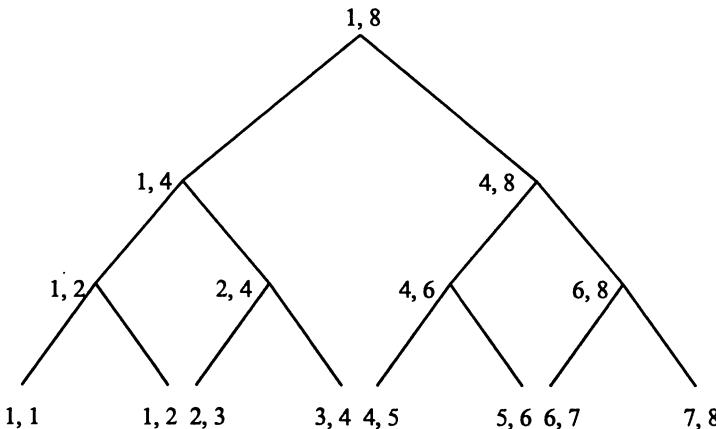


Рис. 7.7. Двоичное дерево B представляет все возможные интервалы поиска при выполнении двоичного поиска в списке длины $m = 8$

т. е. между i -м осмотренным листом и следующим. Среди этих внутренних вершин возьмем ближайшую к корню. Тогда $Lcp(i, i + 1)$ равно ее строковой глубине.

Например, рассмотрим снова суффиксное дерево, показанное на рис. 7.5 (с. 193). $Lcp(5, 6)$ равно строковой глубине родителя листьев 4 и 1, т. е. 3, так как этот родитель помечен строкой *tar*. Значения $Lcp(i, i + 1)$ для i от 1 до 5 равны, соответственно, 2, 0, 1, 0, 3.

Самое трудное в лемме 7.14.1 — прочесть ее. Когда это сделано, доказательство прямо следует из свойств суффиксных деревьев, так что оно оставляется читателю.

Если предположить, что строковые глубины вершин известны (а их можно найти за линейное время), то по лемме значения $Lcp(i, i + 1)$ для i от 1 до $m - 1$ легко вычисляются за время $O(m)$. Оставшиеся значения Lcp легко получаются согласно следующей лемме.

Лемма 7.14.2. $Lcp(i, j) = \min\{Lcp(k, k + 1) : i \leq k < j\}$.

Доказательство. Суффиксы $Pos(i)$ и $Pos(j)$ строки T имеют общий префикс длины $Lcp(i, j)$. По свойствам лексикографического порядка для каждого k между i и j суффикс $Pos(k)$ должен также иметь этот общий префикс. Поэтому $Lcp(k, k + 1) \geq Lcp(i, j)$ для каждого k между i и $j - 1$.

Теперь по транзитивности $Lcp(i, i + 2)$ должно быть не меньше, чем минимум из $Lcp(i, i + 1)$ и $Lcp(i + 1, i + 2)$. Далее, $Lcp(i, j)$ должно быть не меньше, чем наименьшее из $Lcp(k, k + 1)$, где k меняется от i до $j - 1$. Это наблюдение в сочетании с рассуждением предыдущего абзаца доказывает лемму. \square

С помощью леммы 7.14.2 оставшиеся значения Lcp для B можно найти, продвигаясь от листьев и полагая значение Lcp для любой вершины v равным минимуму из значений Lcp ее детей. Это, очевидно, требует времени только $O(m)$.

В итоге алгоритм сравнения строки и подстроки со временем $O(n + \log m)$, использующий суффиксный массив, должен заранее вычислить $2m - 1$ значений Lcp , сопоставленных вершинам двоичного дерева B . Значения, соответствующие листьям,

накапливаются за требующий линейного времени лексический просмотр дерева T в глубину, который используется при построении суффиксного массива. Остальные значения вычисляются по ним за линейное время проходом по дереву B снизу вверх, что дает следующую теорему.

Теорема 7.14.4. *Все необходимые значения Lcp можно получить за время $O(m)$, а все вхождения P в T можно найти, используя суффиксный массив, за время $O(n + \log m)$.*

7.14.6. Где встречаются задачи с большими алфавитами?

К работе с суффиксными массивами в значительной мере побуждают трудности использования суффиксных деревьев, если алфавит велик. Поэтому вполне естественно спросить, а когда же он велик?

Во-первых, существуют естественные языки, такие как китайский, с большими “алфавитами”. Однако для нас чаще всего большой алфавит появляется, когда строка содержит числа, каждое из которых трактуется как символ. Простой пример такой ситуации — строка изображения, каждый символ которой задает цвет пикселя или градацию серого цвета.

Задачи поиска совпадений строк и подстрок, когда алфавит содержит числа, а строки P и T велики, встречаются также в вычислительных задачах молекулярной биологии. Примером может служить задача *совпадения карт* (map matching). *Карта рестриктаз* для единичного фермента определяет расположения в строке ДНК копий определенной подстроки (мест рестрикций). Каждое такое место может отделяться от следующего многими тысячами оснований. Следовательно, карта рестриктаз для единичного фермента представляется строкой из чисел, задающих расстояния между последовательными местами рестрикций. При рассмотрении этой последовательности в качестве строки каждое число является символом (огромного) базового алфавита. В более общем случае карта может отображать расположения многих различных образцов (независимо от того, являются ли они местами рестриктаз), так что строка (карта) состоит из символов некоторого конечного алфавита (представляющего известные рассматриваемые образцы), чередующихся с целыми числами, задающими расстояния между такими местами. Алфавит велик, так как велика область изменения чисел, и, так как часто расстояния известны с большой точностью, эти числа не округляются. Более того, множество известных рассматриваемых образцов и само велико (см. [435]).

Часто подстрока ДНК получается и исследуется без информации о том, где она размещается в геноме и не изучалась ли ранее. Если и новая, и ранее исследованная ДНК полностью расшифрованы и помещены в базу, то вопросы о предыдущей работе или локализациях могут быть решены точным сравнением строк. Но большинство изучаемых подстрок ДНК расшифрованы не полностью — получить карты проще и дешевле, чем полные последовательности. Ввиду этого возникает следующая задача о совпадениях *карт*, которая преобразуется в задачу о совпадениях *строк* с большими алфавитами:

По имеющейся карте (рестриктаз) для большой строки ДНК и карте для меньшей строки определить, не является ли меньшая строка подстрокой большей.

Так как каждая карта представляется как строка чередующихся символов и целых чисел, то базовый алфавит огромен. Это побуждает к использованию суффиксных массивов вместо суффиксных деревьев для анализа совпадений и поиска подстрок. Конечно, задачи становятся более трудными при наличии ошибок, когда числа в строках определены неточно или места могут оказаться пропущены или неправильно добавлены. Эта задача, известная как *выравнивание карт* (map alignment), обсуждается в п. 16.10.

7.15. APL14: Суффиксные деревья в геномных проектах

Сейчас мы обсудим использование суффиксных деревьев, обобщенных суффиксных деревьев и суффиксных массивов в качестве основных структур данных в трех геномных проектах.

Arabidopsis thaliana. В рамках проекта *Arabidopsis thaliana*^{*)} в Мичиганском университете и Университете Миннесоты разрабатывают первоначально карту EST для генома *Arabidopsis* (см. обсуждение EST в п. 3.5.1 и картирования в главе 16). В этом проекте обобщенные суффиксные деревья используются несколькими способами [63–65].

Во-первых, каждый расшифрованный фрагмент проверяется на наличие загрязнений известными векторными последовательностями. Эти векторные последовательности собраны в обобщенное суффиксное дерево (см. п. 7.5).

Во-вторых, каждый вновь расшифрованный фрагмент проверяется на наличие уже расшифрованных фрагментов для нахождения дубликатов последовательностей или областей высокого сходства. С этой целью фрагменты хранятся в расширяющемся обобщенном суффиксном дереве. Так как в проекте планируется секвенировать около 36 000 фрагментов, каждый длиной около 400 оснований, то эффективность при поиске дубликатов и загрязнений здесь важна.

В-третьих, к суффиксным деревьям обращаются при поиске биологически значимых образцов в получаемых последовательностях *Arabidopsis*. Такие образцы часто представляются регулярными выражениями, и обобщенные суффиксные деревья используются для ускорения проверки совпадений образцов с регулярными выражениями, когда при совпадении разрешается небольшое число ошибок. Подход, который позволяет применять для этого суффиксные деревья, рассматривается в п. 12.4.

Дрожжи. Суффиксные деревья являются также основной структурой данных в анализе генома *Saccharomyces cerevisiae* (пивные дрожжи), выполняемом в Институте Макса Планка [320]. Суффиксные деревья “особенно удобны для поиска образцов подстрок в базах данных для последовательностей” [320]. В этом проекте высоко оптимизированные суффиксные деревья, называемые *хешированными деревьями позиций*, используются для решения задач “кластеризации данных о последовательностях в эволюционно близких семействах белков, предсказания структуры и сборки фрагментов” [320] (см. в п. 16.15 обсуждение сборки фрагментов).

Borrelia burgdorferi. *Borrelia burgdorferi* — это бактерия, вызывающая болезнь Лайма. Ее геном имеет длину около одного миллиона элементов и уже расшифрован

^{*)} *Arabidopsis thaliana* является “плодовой мушкой” для генетики растений, т. е. классическим модельным организмом при изучении молекулярной биологии растений. Размер ее генома около 100 миллионов пар оснований.

в Брукхэвенской национальной лаборатории с использованием метода прямого секвенирования для заполнения пробелов после первоначальной фазы дробового секвенирования (*shotgun sequencing*) (см. п. 16.14). Чен и Скиена [100] для решения в рамках этого проекта задачи сборки фрагментов развили методы, основанные на суффиксных деревьях и суффиксных массивах. При сборке фрагментов одним из узких мест является определение перекрытий, которое требует решения варианта задачи о суффиксно-префиксных совпадениях (с возможными ошибками) для всех пар строк из большого множества (см. п. 16.15.1). Данные по *Borrelia* [100] состояли из 4612 фрагментов (строк), насчитывавших в общей сложности 2 032 740 элементов. При использовании суффиксных деревьев и массивов необходимые перекрытия были вычислены за пятнадцать минут. Чтобы сравнить скорость и точность методов суффиксного дерева с методами чистого динамического программирования при определении перекрытий (они обсуждаются в пп. 11.6.4 и 16.15.1), Чен и Скиена тщательно проанализировали громадный объем данных. Проверка показала, что использование суффиксного дерева дает 1000-кратное ускорение по сравнению с несколько более точным динамическим программированием, находя 99 % существенных перекрытий, обнаруженных с помощью динамического программирования.

Критический вопрос — эффективность

Во всех трех проектах исключительно важна эффективность построения, поддержки и поиска суффиксных деревьев, так что детали реализации из п. 6.5 являются решающими. Однако так как суффиксные деревья очень велики (до 20 миллионов символов в проекте *Arabidopsis*), то требуются дополнительные усилия по реализации, особенно в организации хранения суффиксного дерева на диске, чтобы уменьшить число обращений к диску. Все три проекта глубоко использовали это обстоятельство и нашли несколько различные решения. См. детали в [320, 100, 63].

7.16. APL15: Подход Бойера–Мура к точному множественному совпадению

Алгоритм Бойера–Мура для поиска точных совпадений (с единичным образцом) часто сильно сдвигает образец, проверяя лишь малый процент символов текста. Напротив, алгоритм Кнута–Морриса–Пратта при поиске всех вхождений образца должен проверить все символы текста.

При поиске точных множественных совпадений алгоритм Ахо–Корасика аналогичен алгоритму Кнута–Морриса–Пратта — он проверяет все символы текста. Так как алгоритм Бойера–Мура для единичной строки на практике значительно эффективнее алгоритма Кнута–Морриса–Пратта, хочется иметь алгоритм типа Бойера–Мура и для множественной задачи точного совпадения, т. е. такой метод, который обычно проверял бы только сублинейную часть T . Ни один из известных простых алгоритмов этой цели не достигает с линейной оценкой времени для наихудшего случая. Однако синтез алгоритмов Бойера–Мура и Ахо–Корасика, выполненный Комменц–Уолтером [109], решает множественную задачу точного совпадения в духе алгоритма Бойера–Мура. Его правила сдвига позволяют обойтись без проверки многих символов T . Мы не будем описывать алгоритм Комменц–Уолтера, а вместо этого применим суффиксные деревья для получения того же результата еще проще.

Для удобства изложения сначала опишем решение, в котором участвуют два дерева — простое дерево ключей (без обратных ссылок) и суффиксное дерево. Трудная часть работы выполняется суффиксным деревом. После разъяснения идей мы реализуем метод, используя только суффиксное дерево.

Определение. Пусть P^r обозначает строку, обратную образцу P , а \mathcal{P}^r — набор строк, полученный обращением всех образцов P из исходного набора \mathcal{P} .

Как обычно, алгоритм предварительно обрабатывает набор образцов, а затем использует результат препроцессинга для ускорения поиска. Следующее изложение перемежает описания метода поиска и обработки, которая поддерживает поиск.

7.16.1. Поиск

Напомним, что в алгоритме Бойера–Мура, когда конец образца располагается у позиции i в T , сравнение отдельных символов идет *справа налево*. Однако индекс i *возрастает* на каждой итерации. Эти базовые свойства алгоритма Бойера–Мура будут сохраняться и в алгоритме для точного множественного совпадения, который мы представим.

В случае множественных образцов поиск основывается на простом дереве ключей \mathcal{K}^r (без обратных ссылок), кодирующем образцы в \mathcal{P}^r . Поиск, как и раньше, просматривает возрастающие значения индекса i и для каждого i определяет, нет ли в множестве образцов \mathcal{P} такого, который заканчивался бы в позиции i текста T . Подробности следуют ниже.

Время препроцессинга, требуемое, чтобы построить \mathcal{K}^r , имеет порядок только $O(n)$ — полной длины всех образцов в \mathcal{P} . Более того, так как обратные указатели здесь не нужны, препроцессинг особенно прост. Алгоритм построения \mathcal{K}^r последовательно вставляет каждый образец в дерево, проходя как можно дальше по совпадающему пути от корня и т. д. Напомним, что каждый лист \mathcal{K}^r определяет один из образцов в \mathcal{P}^r .

Проверка в позиции i

Построенное дерево \mathcal{K}^r можно использовать для проверки при любой конкретной позиции i из T , не заканчивается ли в этой позиции один из образцов из \mathcal{P} . Для этого нужно просто двигаться по пути от корня \mathcal{K}^r , сопоставляя символы пути с символами из T , начиная от $T(i)$ и проходя справа налево, как в алгоритме Бойера–Мура. Если лист \mathcal{K}^r будет достигнут раньше, чем левый конец T , то номер образца, приписанный листу, определяет образец, который должен входить в T , заканчиваясь в позиции i . Напротив, если совпадающий путь закончится до достижения листа и не сможет быть продолжен, то никакой образец из \mathcal{P} не входит в T , заканчиваясь в позиции i .

Первая проверка начинается с позиции i , равной длине кратчайшего образца. Алгоритм прекращает работу, когда i становится больше $|T|$.

Когда проверка для конкретной позиции i завершается, алгоритм увеличивает i и возвращается к корню \mathcal{K}^r , чтобы начать новую проверку. Приращение i аналогично сдвигу единичного образца в исходном алгоритме Бойера–Мура. Зададимся вопросом: на сколько можно изменить i ? Увеличение i на 1 аналогично наивному алгоритму для точного сопоставления. При сдвиге только на одну позицию никаких

вхождений любого образца не будет пропущено, но вычисления будут неэффективны. В худшем случае они потребуют времени $\Theta(nt)$, где n — полный размер образцов, а t — размер текста. Более эффективный алгоритм увеличит i , когда возможно, больше чем на 1, используя правила, аналогичные правилам плохого символа и хорошего суффикса из метода Бойера–Мура. Конечно, никакой сдвиг не может превышать длины кратчайшего образца P из \mathcal{P} , так как такой сдвиг мог бы пропустить вхождения P в T .

7.16.2. Правило плохого символа

Правило плохого символа из метода Бойера–Мура можно легко адаптировать к задаче множественного совпадения. Предположим, что проверка установила совпадение некоторого пути в \mathcal{K}^r с символами от i влево до $j < i$ в T , но не может продолжить этот путь до совпадения символа $T(j - 1)$. Прямое обобщение правила плохого символа увеличивает i до наименьшего индекса $i_1 > i$ (если он существует), такого что какой-либо образец \bar{P} из \mathcal{P} имеет символ $T(j - 1)$ строго в $i_1 - j + 2$ позициях от его правого конца (рис. 7.8 и 7.9). С этим правилом если i_1 существует, то при выравнивании правого конца каждого образца из \mathcal{P} с позицией i_1 строки T символ $j - 1$ из T окажется около совпадающего символа в строке \bar{P} из \mathcal{P} . (Нужно рассмотреть также особый случай, когда проверка не проходит уже на первом сравнении, т. е. в корне \mathcal{K}^r , и до применения правила сдвига положить $j = i + 1$.)

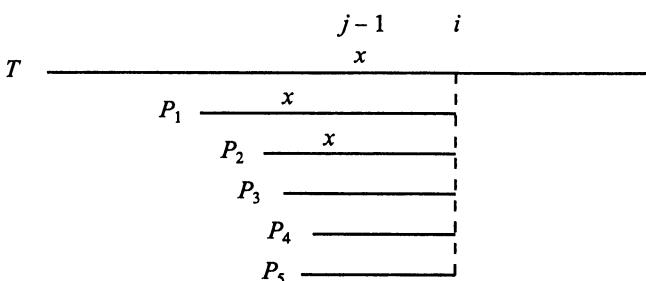


Рис. 7.8. В позиции $j - 1$ строки T дальнейших совпадений нет

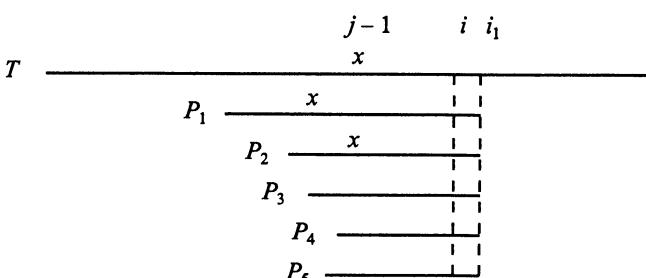


Рис. 7.9. Сдвиг при использовании правила плохого символа

Приведенное обобщение правила плохого символа не вполне корректно. Проблема в образцах из \mathcal{P} , которые меньше, чем \bar{P} . Может случиться так, что i_1 настолько велико, что если правые концы всех образцов выровнены по нему, то левый конец наименьшего образца P_{\min} из \mathcal{P} будет на уровне позиции, большей j в T . Если такое случится, то появится возможность пропустить вхождение P_{\min} (с его левым концом у позиции перед $j + 1$ в T). Следовательно, при использовании только информации о плохом символе (без суффиксных правил, о которых речь пойдет дальше) i не следует выбирать больше чем $j - 1 + |P_{\min}|$. В итоге правило плохого символа для множества образцов таково:

Если i_1 не существует, то увеличить i до $j - 1 + |P_{\min}|$; в противном случае увеличить i до минимума из i_1 и $j - 1 + |P_{\min}|$.

Препроцессинг, необходимый для реализации правила плохого символа, прост и оставляется читателю.

Обобщение правила плохого символа на множественное сопоставление легко, но, в отличие от случая единичного образца, его использование без других средств может быть не очень эффективным. Когда число образцов растет, обычный размер $i_1 - i$ имеет тенденцию к убыванию, особенно если алфавит мал. Это происходит потому, что очень правдоподобно наличие образца, имеющего символ $T(j - 1)$ близко к месту предыдущего совпадения и слева от него. Как отмечалось ранее, в некоторых приложениях из молекулярной биологии полная длина образцов в \mathcal{P} больше, чем размер T , и это делает правило плохого символа почти бесполезным. Правило плохого символа, аналогичное более простому, необобщенному правилу для единичного образца, полезно еще меньше. Поэтому в случае множественного сопоставления решающую роль в повышении эффективности подхода Бойера–Мура играет аналог правила хорошего суффикса.

7.16.3. Правило хорошего суффикса

Чтобы адаптировать (слабое) правило хорошего суффикса к задаче множественного совпадения, мы рассуждаем следующим образом. После нахождения совпадающего пути в \mathcal{K}' (независимо от того, найдено вхождение образца или нет) обозначим через j крайний левый символ из T , совпавший вдоль пути. Пусть $\alpha = T[j..i]$ — совпавшая при проходе подстрока из T (найденная в обратном порядке). Прямое обобщение (на множественное сопоставление) правила хорошего суффикса для двух строк сдвинуло бы правые концы всех образцов из \mathcal{P} до наименьшего значения $i_2 > i$ (если оно существует), такого что $T[j..i]$ совпадает с подстрокой некоторого образца \bar{P} из \mathcal{P} . Образец \bar{P} должен содержать подстроку α , начинающуюся строго в $i_2 - j + 1$ позициях от его правого конца. Этот сдвиг аналогичен правилу хорошего суффикса для случая двух образцов, но, в отличие от него, может оказаться слишком большим. Причиной опять является возможность наличия образцов меньших чем \bar{P} .

Когда существуют образцы меньшие чем \bar{P} , то при сдвиге правого конца каждого образца до i_2 может случиться, что левый конец наименьшего образца P_{\min} расположится на более чем одну позицию справа от i . Тогда вхождение P_{\min} в T будет пропущено. Даже если это и не произойдет, есть другая трудность. Предположим, что префикс β некоторого образца $P' \in \mathcal{P}$ совпадает с суффиксом α . Если P' меньше чем \bar{P} , то сдвиг правого конца P' до i_2 может переместить префикс β образца P'

за подстроку β в T . Если это случится, то вхождение P в T также может быть пропущено. Итак, пусть i_3 будет наименьшим индексом, большим чем i (если i_3 существует), таким что, когда все образцы в \mathcal{P} выровнены по позиции i_3 в T , префикс по меньшей мере одного образца выровнен с суффиксом α в T . Отметим, что, так как \mathcal{P} содержит больше одного образца, это перекрытие может не быть самым большим перекрытием префикса образца из \mathcal{P} и суффикса α . Поэтому правило хорошего суффикса таково:

Увеличить i до минимума из i_2 , i_3 и $i + |\mathcal{P}_{\min}|$. Игнорировать в этом правиле i_2 и/или i_3 , если они не существуют (одно или оба).

7.16.4. Как определить i_2 и i_3

Следующий вопрос в том, как во время поиска эффективно определять, если понадобится, i_2 и i_3 . Обсудим сначала i_2 . Напомним, что через α обозначена подстрока T , которая совпала при только что завершившемся поиске.

Чтобы найти i_2 , нам нужно отыскать в \mathcal{P} образец P , который содержит копию α , заканчивающуюся ближе всего к его правому концу, но не являющуюся его суффиксом. Если эта копия α заканчивается в r позициях от конца P , то i следует увеличить ровно на r позиций, т. е. i_2 следует положить равным $i + r$ (рис. 7.10 и 7.11).

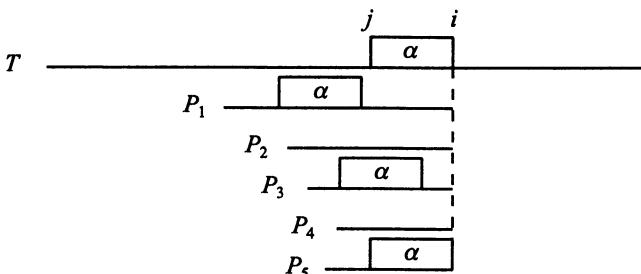


Рис. 7.10. Подстрока α в P_5 совпадает с T от позиции i влево до j ;
левее j совпадение невозможно

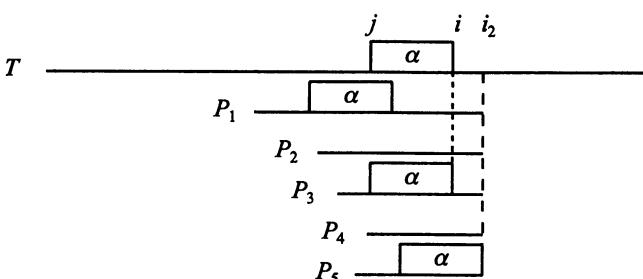


Рис. 7.11. Сдвиг при использовании слабого правила хорошего суффикса.
Здесь размер сдвига определяется образцом P_3

Задачу нахождения i_2 , если оно существует, будем решать с помощью суффиксного дерева, полученного препроцессингом множества \mathcal{P}^r . Ключевой момент — использование этого дерева для поиска в \mathcal{P}^r образца P^r , который содержит копию α^r , начинающуюся ближе всего к его левому концу, но не префикса P^r . Если это вхождение α^r начинается в позиции z образца P^r , то вхождение α заканчивается в $r = z - 1$ позициях от конца P .

Во время препроцессинга строится обобщенное суффиксное дерево \mathcal{T}^r для набора образцов \mathcal{P}^r . Напомним, что в обобщенном суффиксном дереве каждый лист ассоциируется как с образцом $P^r \in \mathcal{P}^r$, так и с номером z , определяющим стартовую позицию суффикса P^r .

Определение. Для каждой внутренней вершины v дерева \mathcal{T}^r обозначим через z_v наименьший номер z , больший чем 1 (если он существует), такой что z является номером суффиксной позиции, записанным в каком-либо листе поддерева v . Если таких листьев нет, то z_v не определено.

Определим для каждой внутренней вершины v суффиксного дерева \mathcal{T}^r число z_v . Эти два подготовительных действия легко выполняются за линейное время стандартными методами и оставляются читателю.

В качестве примера этой подготовки рассмотрим набор $\mathcal{P} = \{wxa, xaq, qxa\}$ и обобщенное суффиксное дерево для \mathcal{P}^r , показанное на рис. 7.12. Первое число в каждом листе указывает строку в \mathcal{P}^r , а второе — начальную позицию суффикса в этой строке. Число z_v равно первому (или единственному) числу, записанному в каждой внутренней вершине (второе число будет введено позже).

Теперь опишем, как используется \mathcal{T}' для определения значения i_2 , если оно существует. После того как обнаружится совпадение α' вдоль пути в \mathcal{K}' , проходим по пути с меткой α' от корня \mathcal{T}' . Такой путь существует, поскольку α является суффиксом некоторого образца в \mathcal{P} (который обнаружен при поиске в \mathcal{K}'), так что α' является префиксом некоторого образца в \mathcal{P}' . Пусть v — первая вершина на этом пути в \mathcal{T}' или ниже его конца. Если z_v определено, то из него можно получить i_2 : лист, определяющий z_u (т. е. лист, где $z = z_v$), связан со строкой $P' \in \mathcal{P}'$,

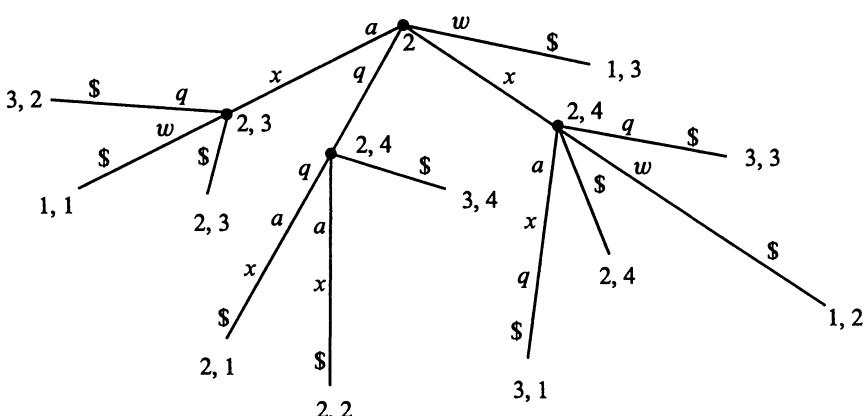


Рис. 7.12. Обобщенное суффиксное дерево \mathcal{T}' для набора $\mathcal{P} = \{wxa, xaq, qxax\}$

которая содержит копию α' , начинающуюся справа от позиции 1. Из всех таких вхождений α' в строках \mathcal{P}' , в P' эта копия α' начинается ближе всего к левому концу строки. Это означает, что P содержит копию α , которая не является суффиксом P , и из всех таких вхождений α в P копия α кончается ближе всего к правому концу строки. Таким образом, P — это та строка из \mathcal{P} , которую следует использовать для определения i_2 . Более того, α кончается в P строго в $z_v - 1$ символах от конца. Как говорилось выше, i должно быть увеличено на $z_v - 1$. В результате мы получаем следующее утверждение.

Теорема 7.16.1. *Если первая вершина v в \mathcal{T}' на пути α' или ниже его конца имеет определенное значение z_v , то $i_2 = i + z_v - 1$.*

При использовании суффиксного дерева \mathcal{T}' нахождение i_2 требует времени $O(|\alpha|)$, всего лишь удваивая время поиска α . Однако при хорошем препроцессинге поиск i_2 можно исключить. Подробности будут даны ниже в п. 7.16.5.

Теперь обратимся к вычислению i_3 . Здесь опять предполагается нужный препроцессинг \mathcal{P} . Снова мы используем обобщенное суффиксное дерево \mathcal{T}' для \mathcal{P}' . В целях описания идеи метода предположим, что $P \in \mathcal{P}$ — такой образец, у которого суффикс α является префиксом. Это означает, что префикс α' является суффиксом P' . Рассмотрим в дереве \mathcal{T}' путь с меткой α' . Так как некоторый суффикс α является префиксом P , то начальная часть пути α' в \mathcal{T}' описывает суффикс P' . Поэтому должна существовать листовая дуга (u, z) , ответвляющаяся от этого пути там, где лист z связан с образцом P' , а метка дуги (u, z) состоит просто из терминального символа $\$$. Обратно, пусть (u, z) — любая дуга, ответвляющаяся от пути α' и помеченная лишь символом $\$$. Тогда образец P , связанный с z , должен иметь префикс, совпадающий с суффиксом α . Эти наблюдения приводят к следующим методам препроцессинга и поиска.

В препроцессинге, когда строится \mathcal{T}' , надо идентифицировать каждую дугу (u, z) из \mathcal{T}' , которая помечена только терминальным символом $\$$. (Число z используется как в качестве имени листа, так и в качестве стартовой позиции связанного с ним суффикса.) Для каждой такой вершины u присвоим d_u значение z . На рис. 7.12, например, d_u — это второе число при u (для корня d_u не определено). В фазе поиска после сравнения строки α из T значение i_3 (если требуется) может быть найдено следующим образом.

Теорема 7.16.2. *Значение i_3 выбирается равным $i + d_w - 1$, где d_w есть наименьшее значение d в вершинах на пути α' в \mathcal{T}' . Если ни в одной вершине на этом пути значение d не определено, то i_3 не определено.*

Теорема 7.16.2 доказывается непосредственно и оставляется читателю. Ясно, что i_3 можно найти во время обхода пути α' в \mathcal{T}' , используемого для поиска i_2 . Если ни i_2 , ни i_3 не существуют, то i следует увеличить на длину наименьшего образца в \mathcal{P} .

7.16.5. Удаление излишеств реализации

Предложенная выше реализация строит два дерева, расходуя время и память пропорционально полному размеру образцов в \mathcal{P} . Вдобавок каждый раз, когда строка α' сравнивается в \mathcal{K}' , на поиск i_2 и i_3 расходуется только $O(|\alpha|)$ дополнительного времени. Таким образом, время на реализацию сдвигов с использованием двух деревьев

пропорционально времени, необходимому для нахождения совпадений. С асимптотической точки зрения два дерева так же малы, как одно, а два прохода так же быстры, как один. Но ясно, что в такой реализации выполняется лишняя работа — одного дерева и одного прохода на каждую поисковую фазу должно хватить. Покажем это

Препроцессинг для множественного точного совпадения Бойера–Мура

`begin`

- Построить обобщенное суффиксное дерево \mathcal{T}^r для строк \mathcal{P}^r . (Каждый лист в дереве имеет номер как конкретного образца P^r из \mathcal{P}^r , так и конкретной стартовой позиции z суффикса P^r .)
- Распознать и пометить каждую вершину в \mathcal{T}^r , включая листья, которая предшествует листу, занумерованному суффиксной позицией 1 (для какого-либо образца P^r в \mathcal{P}^r). Отметим, что вершина считается своим собственным предшественником.
- Для каждой помеченной вершины v положим z_v равным наименьшему номеру суффиксной позиции z , большему 1 (если такая существует), листа в поддереве v .
- Найти все листовые дуги (u, z) из \mathcal{T}^r , которые помечены только терминальным символом $\$$, и положить $d_u = z$.
- Для каждой вершины v из \mathcal{T}^r положить d'_v равным наименьшему значению d_u по всем предшественникам u (включая саму v).
- Удалить поддерево с корнем в любой непомеченной вершине (включая листья) \mathcal{T} . (Вершины помечались на шаге 2.)

`end.`

Перечисленные подготовительные действия легко выполняются за линейное время стандартными методами обхода дерева.

Использование \mathcal{L} в фазе поиска

Пусть \mathcal{L} обозначает дерево в конце препроцессинга. В основном оно устроено, как знакомое дерево ключей \mathcal{K}^r , но более компактно. Любой путь из вершин с единственным потомком заменен простой дугой. Следовательно, для любого i проверка того, не заканчивается ли образец из \mathcal{P} в позиции i , может быть выполнена с использованием дерева \mathcal{L} вместо \mathcal{K}^r . Более того, в отличие от \mathcal{K}^r , с каждой вершиной v в \mathcal{L} теперь связываются значения, требующиеся для вычисления i_2 и i_3 за *контантное* время. Подробнее, после того как алгоритм находит совпадение строки α из T , следя по пути α' в \mathcal{L} , он проверяет первую вершину v на пути в \mathcal{L} или ниже его конца. Если z_v определено, то i_2 существует и равно $i + z_v - 1$. Затем алгоритм проверяет первую вершину v на совпавшем пути или выше его конца. Если d'_v определено, то i_3 существует и равно $i + d'_v - 1$.

Фаза поиска не пропустит ни одного вхождения образца, если применяется какое-либо из правил хорошего суффикса или плохого символа. Однако их можно комбинировать, чтобы достичь наибольшего приращения i .

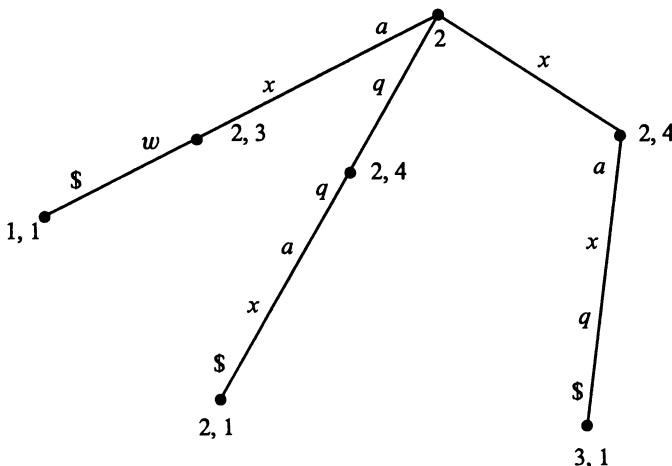


Рис. 7.13. Дерево \mathcal{L} , соответствующее дереву \mathcal{T}^r для набора $\mathcal{P} = \{wxa, xaq, qxah\}$

1234567890	1234567890	1234567890	1234567890
qxaxtqqpst	qxaxtqqpst	gxaxtqqpst	qxaxtqqpst
wxa	wxa	wxa	wxa
xaq	xaq	xaq	xaq
qxah	qxah	qxah	qxah

Рис. 7.14. Первые сравнения начинаются в позиции 3 строки T , обнаружено совпадение ax . Значение $z_v = 2$, так что происходит сдвиг на одну позицию. Совпадает строка $qxah$, z_v не определено, а d'_v определено и равно 4, а сдвиг равен 3. Стока qq совпадает, но дальше совпадения нет; z_v не определено, d'_v определено как 4, сдвиг равен 3, после чего сравнения не производятся и алгоритм завершается

Рис. 7.13 представляет дерево \mathcal{L} , соответствующее дереву \mathcal{T}^r на рис. 7.12.

Чтобы увидеть, как \mathcal{L} используется при поиске, возьмем $T = qxaxtqqpst$. Сдвиги \mathcal{P} показаны на рис. 7.14.

7.17. APL16: Сжатие данных по методу Зива—Лемпеля

Большие текстовые и графические файлы часто сжимаются для уменьшения размера памяти при хранении и времени при пересылке. Большинство операционных систем имеет утилиты сжатия, и некоторые программы пересылки файлов автоматически их запаковывают, пересылают и распаковывают обратно без вмешательства пользователя. Проблема сжатия текстов сама по себе является предметом нескольких книг (см., например, [423]) и не будет здесь исследоваться подробно. Однако популярный метод сжатия, предложенный Зивом и Лемпелем [487, 488], успешно реализуется с помощью суффиксных деревьев [382], еще раз иллюстрируя их эффективность.

Метод сжатия Зива–Лемпеля широко известен (он лежит в основе утилиты *compress* системы UNIX), хотя на самом деле есть несколько вариантов этого метода под одним и тем же именем (см. [487, 488]). В этом пункте мы предложим вариант основного метода и его эффективную реализацию с применением суффиксных деревьев.

Определение. Для любой позиции i в строке S длины m определим подстроку $Prior_i$ как самый длинный префикс $S[i..m]$, встречающийся также в $S[1..i - 1]$ в качестве подстроки.

Например, если $S = abaxcabaxabz$, то $Prior_7 = bax$.

Определение. Для любой позиции i в S определим l_i как длину $Prior_i$. Для $l_i > 0$ определим s_i как начальную позицию крайней левой копии $Prior_i$.

В предыдущем примере $l_7 = 3$ и $s_7 = 2$.

Заметим, что, когда $l_i > 0$, копия $Prior_i$, начинающаяся в s_i , полностью содержится в $S[1..i - 1]$.

Метод Зива–Лемпеля использует некоторые из значений l_i и s_i для построения сжатого представления строки S . Основная мысль в том, что если текст $S[1..i - 1]$ уже представлен (возможно, в сжатой форме) и $l_i > 0$, то следующие l_i символов S (подстрока $Prior_i$) не нужно представлять явно. Достаточно описать эту подстроку парой (s_i, l_i) , указывающей на ее более раннее вхождение. Следуя этой мысли, метод сжатия может обрабатывать ~~S~~ слева направо, выводя, когда можно, пары (s_i, l_i) вместо явных подстрок $S[i..i + l_i - 1]$ и символы $S(i)$, когда без этого не обойтись. Все подробности даются приводимым ниже алгоритмом.

1-й алгоритм сжатия

```

begin
  i := 1;
repeat
  вычислить  $l_i$  и  $s_i$ ;
  if  $l_i > 0$  then begin
    output  $(s_i, l_i)$ ;
    i := i + 1
  end
  else begin
    output  $S(i)$ ;
    i := i + 1
  end
until  $i > n$ 
end.
```

Например, строку $PS = abacabaxabz$ можно описать как $ab(1, 1)c(1, 3)x(1, 2)z$. Конечно, в этом примере число символов, использованных для представления S , не уменьшилось, а увеличилось! Это возможно для маленьких примеров. Но при росте длины строки, когда возрастает вероятность повторов подстрок, сжатие улучшается. Более того, алгоритм может предпочесть явный вывод символа $S(i)$, если l_i “мало” (реальное правило зависит от рассмотрений на битовом уровне, определяясь размером алфавита и т. п.). В качестве малого примера,

где можно наблюдать действительное сжатие, рассмотрим нарочно придуманную строку $S = abababababababababababababab$, представляющую как $ab(1, 2)(1, 4)(1, 8)(1, 16)$. Это представление использует 24 символа вместо исходных 32. Если мы расширим этот пример до k повторяющихся копий ab , то сжатое изображение будет содержать примерно $5 \log_2 k$ символов — фантастическое сокращение места.

Чтобы распаковать сжатую строку, нужно двигаться по ней слева направо, и каждая пара (s_i, l_i) будет указывать на уже полностью распакованную подстроку. Предположим для индукции, что первые j элементов сжатой строки (простые символы или пары s, l) уже обработаны и получились символы с 1 по $i - 1$ исходной строки S . Следующий элемент в сжатой строке будет либо символом $S(i + 1)$, либо парой (s_i, l_i) , указывающей на подстроку S , лежащую строго перед i . В любом случае алгоритм имеет всю информацию, нужную для распаковки j -го элемента, и так как первый элемент в сжатой строке есть первый символ S , мы устанавливаем по индукции, что алгоритм декомпрессии может воссоздать исходную строку S .

7.17.1. Реализация, использующая суффиксные деревья

Ключевой вопрос реализации — как вычислять l_i и s_i , когда они нужны алгоритму при обработке позиции i . Алгоритм сжимает S слева направо и не требует (s_i, l_i) для какой-либо позиции i в уже сжатой части S . Сжатые подстроки поэтому не перекрываются, и если каждая запрошенная пара (s_i, l_i) может быть найдена за время $O(l_i)$, то весь алгоритм будет работать за время $O(m)$. Используя суффиксное дерево для S , оценку времени $O(l_i)$ легко получить для любого запроса.

До начала сжатия алгоритм строит суффиксное дерево \mathcal{T} для S , а затем нумерует каждую вершину v номером c_v , равным наименьшей из позиций листьев в поддереве, соответствующем v . Этот номер дает самую левую начальную позицию в S копий подстроки, которая маркирует путь от r до v . Это дерево можно построить за время $O(m)$, и все номера вершин — также за время $O(m)$ любым стандартным методом обхода дерева (или распространением снизу вверх).

Когда алгоритму нужно вычислить (s_i, l_i) для какой-либо позиции i , он проходит единственным путем в \mathcal{T} , который совпадает с префиксом $S[i..m]$. Проход кончается в точке p (не обязательно в вершине), когда i равно строковой глубине точки p плюс число c_v , где v — первая вершина в p или ниже. В любом случае путь из корня в p описывает самый длинный префикс $S[i..m]$, который встречается также в $S[1..i]$. Таким образом, $s_i = c_v$ и l_i равно строковой глубине p . Из-за того что алфавит фиксирован, время на поиск (s_i, l_i) равно $O(l_i)$. Таким образом, весь алгоритм сжатия работает за время $O(m)$.

7.17.2. Однопроходный вариант

В предложенной реализации мы предполагали, что строка S известна заранее и что суффиксное дерево для S можно построить до начала сжатия. Это прекрасно работает во многих задачах, но метод можно модифицировать так, чтобы он функционировал в режиме *on-line*, как будто S вводится посимвольно. Алгоритм реализуется

таким образом, что сжатие S переплетается с построением \mathcal{T} . Самый простой способ увидеть, как это нужно делать, — сочетание с линейным по времени алгоритмом суффиксного дерева Укконена.

Алгоритм Укконена строит неявные суффиксные деревья в режиме on-line, когда символы добавляются к правому концу растущей строки. Предположим, что сжатие уже сделано для $S[1..i - 1]$ и что для этой части строки построено неявное суффиксное дерево \mathcal{T}_{i-1} . Теперь алгоритму сжатия необходимы значения (s_i, l_i) . Эту пару можно получить точно так же, как было сделано в приведенной выше реализации, если в каждой вершине v дерева \mathcal{T}_{i-1} записаны величины c_v . Однако, в отличие от упомянутой реализации, где значения c_v получались за линейное время обходом \mathcal{T} , этот алгоритм не может обходить каждое из неявных суффиксных деревьев, так как в результате линейность времени будет нарушена. Вместо этого когда в алгоритме Укконена новая внутренняя вершина v создается разделением дуги (u, w) , то c_v полагается равным c_w , а когда создается новый лист v , то c_v — это просто номер суффикса, соответствующий листу v . При таком способе действий достаточно константного времени для обновления значений c_v при добавлении в дерево новой вершины. В итоге:

Теорема 7.17.1. *I-й алгоритм сжатия можно реализовать так, чтобы он работал за линейное время как однопроходный онлайновый алгоритм при сжатии любой входной строки S .*

7.17.3. Настоящий Зив–Лемпель

Схема сжатия, предложенная в *I-м алгоритме сжатия*, хотя и не является подлинным методом Зива–Лемпеля, но вполне заменяет его и сохраняет его дух. Настоящий метод Зива–Лемпеля — это однопроходный алгоритм, результаты работы которого отличаются от результатов *I-го алгоритма сжатия* тем, что, когда он выводит пару (s_i, l_i) , он явно выписывает $S(i + l_i)$ — символ, следующий за подстрокой. Например, $S = ababababababababababababab$ скажется бы до $ab(1, 2)a(2, 4)b(1, 10)a(2, 12)b$, а не до $ab(1, 2)(1, 4)(1, 8)(1, 16)$. Однопроходный вариант *I-го алгоритма сжатия* можно trivialально преобразовать для реализации метода Зива–Лемпеля за линейное время.

Не вполне ясно, почему алгоритм Зива–Лемпеля выводит дополнительный символ. Для целей сжатия он не нужен и кажется избыточным. Можно предположить, что одной из причин такой записи является то, что $(s_i, l_i)S(i + l_i)$ определяет кратчайшую подстроку, начинающуюся в позиции i , которая не появлялась раньше в строке, тогда как (s_i, l_i) определяет самую длинную подстроку, начинающуюся в i , которая появлялась раньше. Исторически могло сложиться так, что легче рассуждать о кратчайших подстроках, не появлявшихся раньше в строке, чем о самых длинных подстроках, уже появлявшихся.*)

*) Здесь автор говорит о методе LZ77 [487]. В методе LZ78 [488] и более поздних вариантах метода Зива–Лемпеля очередная порция скимаемой строки ищет свое начало не среди всех подстрок сжатой части, а только среди уже включенных в словарь, — в разных вариантах метода это множество различно, но теперь уже не включает в себя всех мыслимых подстрок. Это уменьшает сжатие и ускоряет работу алгоритма. Ограничимся уже рассмотренным автором примером. По методу LZ78 строка $S = ababababababababababababab$ скажется так: $(0a)(0b)(1b)(3a)(2a)5b 4b 7a)(6a)(9b)(3)$, что дает 21 символ, если не считать скобок. — Прим. перев.

7.18. APL17: Код минимальной длины для ДНК

Недавно несколько исследовательских групп в области молекулярной биологии и информатики использовали метод Зива–Лемпеля для сжатия строк ДНК не в целях их эффективного хранения, а в целях вычисления меры “сложности” или “информационного содержания” строк [14, 146, 325, 326, 386]. Не определяя полностью основные технические термины “сложность”, “информация”, “энтропия” и т. п., мы отметим основную идею: подстроки наибольшей биологической значимости должны быть более сжимаемы, чем подстроки, которые в основном случайны. Можно ожидать, что случайные строки будут иметь слишком мало структур, чтобы допустить сильное сжатие, так как сильное сжатие основано на нахождении в строке повторяющихся элементов. Таким образом, поиском подстрок, которые сжимаются лучше, чем случайные строки, можно найти строки, которые несут определенные биологические функции. (С другой стороны, наиболее часто повторяющиеся ДНК встречаются вне экзонов.)

Сжатие применялось также для изучения “родственности” *) двух строк ДНК, S_1 и S_2 [14, 324]. В основном идея заключается в том, чтобы строить суффиксное дерево для S_1 и затем сжимать строку S_2 , используя только это дерево. Такое сжатие S_2 эффективно учитывает подстроки S_2 , которые появились в S_1 , но не использует выгоды повторяющихся подстрок в самой S_2 . Аналогично, S_1 можно сжать с помощью только суффиксного дерева для S_2 . Эти сжатия отражают и оценивают “родственность” строк S_1 и S_2 . Если две строки сильно похожи, то оба вычисления должны существенно их сжать.

Другое биологическое использование алгоритмов типа Зива–Лемпеля оценивало “энтропию” коротких строк для того, чтобы установить различие между экзонами и инtronами в эукариотических ДНК [146]. Согласно Фарауху и др. [146], нет существенных различий в среднем сжатии инtronов и экзонов, и следовательно, по сжатию их друг от друга не отличить. Однако эти авторы сообщают также о следующем обобщении подхода, которое делает различие экзонов от инtronов эффективным.

Определение. Для любой позиции i строки S пусть $ZL(i)$ обозначает длину самой длинной подстроки, начинающейся в i , которая появляется где-либо в строке $S[1..i - 1]$.

Определение. Пусть строка ДНК разбита на экзоны и интроны. Назовем **экзон-средним ZL значением** среднее $ZL(i)$, взятое по всем позициям i в экзонах S , а **инtron-средним** — аналогичное среднее по инtronам S .

Интуитивно ясно, что экзон- и инtron-средние значения ZL можно вычислить за время $O(n)$, используя суффиксные деревья для вычисления всех значений $ZL(i)$. Техника напоминает способ вычисления статистики совпадений, но более сложна, так как подстрока, начинающаяся в позиции i , должна появляться также и слева от i .

Главный эмпирический результат работы [146] заключается в том, что экзон-среднее значение ZL меньше, чем инtron-среднее, на статистически значимую величину. Этот результат противоречит высказанному выше ожиданию, что биологически существенные подстроки (в данном случае экзоны) должны быть более сжимаемы,

*) Другие более общие способы изучения родственности, или сходства, двух строк широко обсуждаются в части III.

чем более случайные подстроки (каковыми принято считать интроны). Следовательно, вопрос о полной биологической значимости сжимаемости строк остается открытым.

7.19. Другие приложения

Много дополнительных приложений суффиксных деревьев появится в следующих ниже упражнениях, в главе 9, в пп. 12.2.4, 12.3 и 12.4 и в упражнениях к главе 14.

7.20. Упражнения

1. При заданном наборе \mathcal{S} из k строк мы хотим найти все строки из \mathcal{S} , которые являются подстроками других строк этого набора. Предполагая, что полная длина всех строк равна n , предложите для этой задачи алгоритм со временем $O(n)$. Этот результат понадобится в алгоритмах для задачи о кратчайшей надстроке (п. 16.17).
2. Для строки S длины n покажите, как вычислить значения $N(i)$, $L(i)$, $L'(i)$ и sp_i (введенные в пп. 2.2.4 и 2.3.2) за время $O(n)$ прямо из суффиксного дерева для S .
3. Мы можем определить суффиксное дерево в терминах дерева ключей, использованного в алгоритме Ахо–Корасика (AC). Входом в алгоритм AC является набор образцов \mathcal{P} , а дерево AC — компактное представление этих образцов. Можно рассматривать n суффиксов единственной строки S как набор образцов и строить суффиксное дерево для S , конструируя сначала дерево AC, а затем сжимая в простую дугу любой максимальный путь через вершины с одним потомком. Если мы примем такой подход, каково будет соотношение между связями неудач в дереве ключей и суффиксными связями в алгоритме Укконена? Почему не следует строить суффиксные деревья таким способом?
4. Суффиксное дерево для строки S может рассматриваться как дерево ключей, составленное для суффиксов S . При таком подходе суффиксное дерево полезно при эффективном построении дерева ключей, когда строки для дерева заданы неявно. Рассмотрим теперь следующий неявно заданный набор строк. Рассмотрим две строки, S_1 и S_2 , пусть D — набор всех подстрок S_1 , которые не содержатся в S_2 . Предположив, что длина этих двух строк равна n , покажите, как построить дерево ключей для D за время $O(n)$. Далее постройте дерево ключей для D вместе с самим набором подстрок S_2 , не входящих в S_1 .
5. Предположим, что построено обобщенное суффиксное дерево для строки S вместе с его суффиксными связями (или указателями связи). Покажите, как эффективно конвертировать это дерево в дерево ключей Ахо–Корасика.
6. Обсудите относительные преимущества метода Ахо–Корасика в сравнении с использованием суффиксных деревьев для множественной задачи точного совпадения, когда текст фиксирован, а набор образцов меняется во времени. Рассмотрите препроцессинг, время поиска и использование памяти. Исследуйте как случай, когда текст больше набора образцов, так и противоположный.
7. Почему суффиксное дерево показывает структуру строки более глубоко, чем дерево ключей Ахо–Корасика или препроцессинг по методам Кнута–Морриса–Пратта и Бойера–Мура? Иначе говоря, значения sp дают некоторую информацию о строке, но суффиксное дерево дает много больше информации о структуре строки. Уточните это утверждение. Ответьте на тот же вопрос о суффиксных деревьях и значениях Z .

8. Предложите алгоритм, который по набору из k строк находит наибольшую общую подстроку каждой из C_k^2 пар строк. Предположим, что каждая строка имеет длину n . Так как наибольшая общая подстрока любой пары может быть найдена за время $O(n)$, время $O(k^2n)$ легко достижимо. Теперь предположим, что длины строк различны и в сумме равны m . Покажите, как найти все наибольшие общие подстроки за время $O(km)$. Затем попытайтесь — за $O(m + k^2)$ (как достичь этой последней границы, автор, кстати, не знает).
9. Задача нахождения подстрок, общих для набора различных строк, обсуждалась отдельно от задачи нахождения подстрок, общих для отдельной строки, и кажется, что первую задачу решить много труднее, чем вторую. Почему нельзя первую задачу свести ко второй конкатенацией строк набора для формирования одной большой строки?
10. Модифицируя алгоритм сокращения и добавляя немного информации (линейно по памяти) к результирующему ориентированному ациклическому графу (для него использовалась аббревиатура DAG), можно использовать DAG не только для определения того, встречается ли образец в тексте, но и для поиска всех его вхождений. Проиллюстрируем эту идею случаем, когда имеется только простое слияние вершин p и q . Допустим, что p имеет большую строковую глубину, чем q , и что i является родителем p до слияния. Во время слияния нужно удалить поддерево p и присвоить величину смещения -1 новой дуге из i в pq . Смещение на дуге будет учитываться при определении начала вхождения P . Теперь предположим, что мы ищем в тексте образец P и устанавливаем, что он присутствует. Пусть i — лист ниже пути, помеченного строкой P (т.е. ниже точки завершения поиска). Если поиск проходит дугу из i в pq , то P присутствует с началом в позиции $i - 1$; в противном случае он присутствует с началом в позиции i .
Обобщите эту идею и разработайте детали для любого числа сливаемых вершин.
11. В некоторых приложениях желательно знать, сколько раз входная строка P входит в большую строку S . После очевидной подготовки, линейной по времени, на запросы такого рода с помощью суффиксного дерева можно отвечать за время $O(|P|)$. Покажите, как подготовить DAG за линейное время, чтобы на эти запросы можно было отвечать за время $O(|P|)$, используя DAG.
12. Докажите корректность алгоритма сокращения суффиксных деревьев.
13. Пусть S' — строка, обратная S . Есть ли связь между числом вершин в DAG для S и для S' ? Докажите ее. Найдите соотношение DAG для S и для S' (оно немного проще, чем для суффиксных деревьев).
14. В теореме 7.7.1 мы предложили легко вычисляемое условие изоморфности двух поддеревьев суффиксного дерева для строки S . Альтернативное условие, не такое уж полезное для эффективного вычисления, заключается в следующем. Пусть α — подстрока, помечающая вершину p , а β — подстрока, помечающая вершину q , в суффиксном дереве для S . Поддеревья p и q изоморфны в том и только том случае, если набор позиций в S , где кончаются вхождения α , равен набору позиций, где кончаются вхождения β .
Докажите корректность этого альтернативного условия изоморфизма поддеревьев.
15. Выполняется ли теорема 7.7.1 для обобщенного суффиксного дерева (для более чем одной строки)? Если нет, то можно ли ее изменить так, чтобы она выполнялась?
16. Ациклический граф D для строки S можно преобразовать в конечный автомат, заменив каждую дугу с меткой, состоящей более чем из одного символа, последовательностью дуг, помеченных каждой одним символом. Этот автомат будет распознавать подстроки S , но не будет обязательно наименьшим таким автоматом. Приведите пример.

Посмотрим теперь, как построить наименьший автомат для распознавания подстрок S . Снова начнем с суффиксного дерева для S , сольем изоморфные поддеревья, а затем заменим каждую дугу, которая помечена больше чем одним символом. Однако операция слияния должна быть выполнена аккуратнее, чем раньше. Более того, представим себе, что имеется суффиксная связь из каждого листа i в каждый лист $i + 1$, где $i < n$. Тогда появится путь из суффиксных связей, соединяющий все листья, и под каждым листом будет нуль листьев. Следовательно, все листья будут слиты.

Напомним, что Q — это множество всех пар (p, q) , для которых в \mathcal{T} существует суффиксная связь из p в q , где p и q имеют одинаковые номера листьев в соответствующих им поддеревьях. Предположим, что $(p, q) \in Q$. Пусть v — родитель p , γ — метка дуги (v, p) , а δ — метка дуги, входящей в q . Объясните, почему $|\gamma| \geq |\delta|$. Так как каждая дуга ациклического графа будет безусловно разбита на столько дуг, какова длина ее метки, мы хотим сделать каждую дуговую метку по возможности малой. Ясно, что δ является суффиксом γ , и мы будем пользоваться этим фактом, чтобы сливать дуговые метки по возможности лучше. Во время слияния p с q удаляются, как раньше, все дуги, выходящие из p , но дуга из v не обязательно направляется в q . Вместо этого при $|\delta| > 1$ дуга δ делится на две дуги с помощью новой вершины u . Первая из них помечается первым символом δ , а вторая — оставшейся частью строки δ . После этого дуга из v направляется в u , а не в q . Дуга (v, u) помечается первыми $|\gamma| - |\delta| + 1$ символами γ .

Использование этого модифицированного слияния проясняет описание всего процесса сокращения и доказывает, что результатирующий DAG распознает подстроки S . Конечный автомат для S получается разбиением каждой дуги этого графа, помеченной более чем одним символом. Каждая вершина этого графа становится состоянием автомата.

17. Покажите, что автомат, построенный выше, имеет меньше состояний, чем любой другой автомат, распознающий подстроки S . Основная идея доказательства: детерминированный автомат имеет меньше состояний при условии, что никакое его состояние не эквивалентно другому. Два состояния, с которых начинается работа, эквивалентны, если принимается один и тот же набор строк (см. [228]).
18. Предположим, что вы уже имеете дерево ключей Ахо–Корасика (с обратными связями). Можете ли вы применить его для вычисления за линейное время статистики совпадений или, если нет, за некоторое “разумное” нелинейное время? Можно ли его использовать для решения за разумное время задачи о наибольшей общей подстроке? Если нет, то в чем трудность?
19. В п. 7.16 мы рассматривали использование суффиксного дерева для поиска всех вхождений набора образцов в данный текст. Если длина всех образцов равна n , а длина текста m , то метод требует места $O(n + m)$ и памяти $O(m)$. Другой взгляд: решение требует $O(m)$ времени на препроцессинг и $O(n)$ — на поиск. А метод Ахо–Корасика решает эту задачу за то же полное время, но с памятью $O(n)$. Кроме того, он расходует время $O(n)$ на препроцессинг и $O(m)$ — на поиск.
Так как нет определенного соотношения между n и m , то иногда один метод будет требовать меньше памяти или времени на препроцессинг, чем другой. Используя обобщенное суффиксное дерево для набора образцов и обратную роль суффиксных деревьев, рассмотренную в п. 7.8, можно решить задачу с суффиксным деревом, полученным с теми же границами времени и памяти, что и в методе Ахо–Корасика. Покажите подробно, как это сделать.
20. Используя обратную роль суффиксных деревьев из п. 7.8, покажите, как решить общую задачу загрязнения ДНК из п. 7.5 с помощью суффиксного дерева для S_1 , а не обобщенного суффиксного дерева для S_1 со всеми возможными его контаминантами.

21. В п. 7.8.1 мы использовали суффиксное дерево для маленькой строки P , чтобы вычислить статистику совпадений $ms(i)$ для каждой позиции i в длинной текстовой строке T . Предположим теперь, что мы также хотим вычислить статистику совпадений $ms(j)$ для каждой позиции j из P . Число $ms(j)$ определяется как длина самой длинной подстроки, начинающейся в позиции j из P , которая совпадает с некоторой подстрокой T . Можно было бы действовать, как раньше, но это потребовало бы суффиксного дерева для длинного текста T . Покажите, как найти всю статистику совпадений и для T и для P за время $O(|T|)$, с помощью только суффиксного дерева для P .

22. В нашем обсуждении статистики совпадений мы прибегали к суффиксным связям, созданным алгоритмом Укконена. Суффиксные связи можно получить также обращением указателей связи в алгоритме Вайнера, но в предположении, что дерево уже нельзя модифицировать. Можно ли вычислить статистику совпадений за линейное время, используя дерево и указатели связи, как в алгоритме Вайнера?

23. В п. 7.8 мы обсуждали обращенное использование суффиксного дерева для решения задачи точного поиска образца: найти все вхождения образца P в текст T . При этом рассчитывалась статистика совпадений $ms(i)$ для каждой позиции в тексте i . Вот модификация этого метода, которая решает задачу о точном совпадении, но не вычисляет статистику совпадений: следовать во всех деталях алгоритму статистики совпадений, но проверять новые символы в тексте, только когда проходится путь от корня до листа с меткой 1. Это значит, что на каждой итерации в суффиксном дереве не нужно идти ниже строки $\alpha\gamma$, за исключением случая, когда вы находитесь на пути до листа с меткой 1. Будучи на другом пути, алгоритм просто следует суффиксным связям и выполняет операции скачка по счетчику, пока не вернется на нужный путь.

Докажите, что эта модификация корректно решает задачу о точном совпадении за линейное время.

Какими преимуществами или недостатками обладает этот метод по сравнению с вычислением статистики совпадений?

24. Есть простое практическое улучшение предыдущего метода. Пусть v — точка пути к листу 1, в которой кончается некоторый поиск, а v' — вершина рассматриваемого пути, в которую алгоритм войдет после этого (вслед за некоторым числом итераций, на которых посещаются вершины вне пути). Создадим прямую связь “короткого замыкания” от v к v' . Дело в том, что в любой будущей итерации, заканчивающейся v , эта перемычка будет использоваться вместо длинного непрямого возврата до v' .

Докажите, что это улучшение работает (т.е. что с ним задача о точном совпадении корректно решается).

Каково взаимоотношение этих перемычек с функцией неудач, применяемой в методе Кнута–Морриса–Пратта? Когда суффиксное дерево кодирует больше одного образца, какова связь этих перемычек с обратными указателями, используемыми в методе Ахок–Корасика?

25. Мы могли бы модифицировать предыдущий метод и еще дальше. На каждой итерации нужно просто следовать суффиксной связи (до конца α) и не делать скачков по счетчику или сравнений символов, пока вы не находитесь на пути до листа 1. В этой точке нужно делать все требуемые вычисления скачков по счетчику, чтобы пропустить уже проверенные части текста.

Разработайте эту идею в деталях и проверьте, решает ли она корректно задачу о точном совпадении за линейное время.

26. Вспомним обсуждение STS в п. 7.8.3 (с. 173). Покажите подробно, как статистику совпадений можно применить для идентификации любых STS, содержащихся в строке, в предположении, что возможно “скромное” число ошибок в строках STS или в новой строке.
27. Пусть задан набор из k строк длины n каждая. Найдите наибольший общий префикс каждой пары строк. Полное время должно быть $O(kn + p)$, где p — число пар строк, имеющих непустой общий префикс. (Это можно сделать, используя алгоритм поиска наименьшего общего предшественника, обсуждаемый дальше, но возможен более простой метод.)
28. Для любой пары строк мы можем вычислить длину наибольшего общего префикса за время, линейно зависящее от их общих длин. Это простое использование суффиксного дерева. Теперь предположим, что задано k строк общей длины n и мы хотим вычислить минимальную длину двух наибольших попарно общих префиксов по всем C_k^2 парам строк, т. е. наименьшую длину попарных совпадений. Очевидный способ достичь результата — решить задачу о наибольшем общем префиксе для каждой из C_k^2 пар строк за время $O(k^2 + kn)$. Покажите, как найти решение за время $O(n)$ независимо от k . Рассмотрите также задачу вычисления максимальной длины по всем попарно общим префиксам.
29. Проверьте, что задача суффиксно-префиксного совпадения всех пар, рассмотренная в п. 7.10, может быть решена за время $O(km)$ при использовании любого метода сравнения строк, линейного по времени. То есть что оценка времени $O(km)$ не требует суффиксного дерева. Объясните, почему эта оценка не содержит слагаемого k^2 .
30. Рассмотрим снова задачу суффиксно-префиксного совпадения всех пар. Ее можно решить с той же оценкой времени без явного обхода дерева. Сначала построим обобщенное суффиксное дерево $\mathcal{T}(\mathcal{S})$ для набора k строк \mathcal{S} (как раньше) и введем вектор V длины k , который вначале заполним нулями. Сравним каждую строку набора с деревом. Сравнение, использующее строку S_j , кончается в листе, помеченном суффиксом 1 для строки S_j . Если во время прохода для S_j встретится вершина v , содержащая в своем списке $L(v)$ индекс i , запишем строковую глубину v в позицию i вектора V . Когда мы достигнем листа для суффикса 1 строки S_j , для каждого i значение $V(i)$ будет равно длине наибольшего суффикса S_i , совпадающего с префиксом S_j .
Проведите для этого метода анализ времени наихудшего случая. Сравните все его преимущества и недостатки (по практическому использованию памяти и/или времени) с методом обхода дерева из п. 7.10. Затем предложите модификацию метода обхода дерева, которая сохраняет все его преимущества и исправляет недостатки.
31. Подстрока α называется *префиксным повтором* строки S , если α является префиксом S и имеет вид $\beta\beta$ для некоторой строки β . Предложите линейный по времени алгоритм для отыскания наибольшего префиксного повтора входной строки S . Эта задача была для Вайнера одним из побуждений для создания суффиксных деревьев.

Очень часто в литературе по анализу последовательностей методы, предназначенные для отыскания интересных свойств биологической последовательности, начинают с каталогизации определенных подстрок длиной строки. Эти методы почти всегда выделяют *окно фиксированной длины* и затем находят все различные строки этой длины. Результат этого оконного, или *q-граммного*, подхода, конечно, сильно зависит от выбора длины окна. В следующих трех упражнениях мы покажем, как суффиксные деревья позволяют снять эту трудность, обеспечивая естественное и более эффективное развитие оконного подхода. (См. также упражнение 26 к главе 14.)

32. В строке T длины M есть $m(m+1)/2$ подстрок. Некоторые из них одинаковы и встречаются в строке больше одного раза. Так как число подстрок равно $\Theta(m^2)$, мы не можем подсчитать, сколько раз появляется в T каждая из них за время $O(m)$. Однако использование

суффиксного дерева позволяет получить за такое время *неявное* представление этих чисел. В частности, когда задана любая строка P длины n , это неявное представление должно позволить нам вычислить частоту P в T за время $O(n)$. Покажите, как создать неявное представление частот и как им пользоваться.

33. Покажите, как подсчитать число различных подстрок строки T за время $O(m)$, где $m = |T|$. Покажите, как перечислить по одной копии каждой отличной подстроки за время, пропорциональное длине всех таких строк.
34. Один из способов охоты за “интересными” последовательностями в базе данных последовательностей ДНК состоит в том, чтобы находить там подстроки, которые появляются значительно чаще, чем это можно было бы предсказать при чисто случайном их появлении. Это делается сейчас и станет еще привлекательное, когда будет доступно гигантское количество анонимных последовательностей ДНК.
Предполагая, что есть статистическая модель для определения того, насколько правдоподобно случайное появление любой конкретной подстроки, и порог, выше которого подстрока “интересна”, покажите, как эффективно найти все интересные подстроки в базе данных. Если база данных имеет общую длину m , то этот метод должен требовать времени $O(m)$ плюс время, пропорциональное числу интересных подстрок.
35. **Наименьший k -повтор.** По заданной строке S и числу k мы хотим найти наименьшую подстроку S , которая появляется в S ровно k раз. Покажите, как решить эту задачу за линейное время.
36. Теорема 7.12.1, которая устанавливает, что в строке длины n может быть не больше n максимальных повторов, была получена соединением максимальных повторов с суффиксными деревьями. Представляется, что эта граница должна получаться просто и непосредственно. Попробуйте найти такое обоснование. Напомним, что неправильно считать, что в любой позиции S начинается не более одного максимального повтора.
37. Пусть заданы две строки, S_1 и S_2 , и мы хотим найти все *максимальные общие пары* S_1 и S_2 . Общая подстрока C максимальна, если добавление к ней любого символа в начале или конце меняет строку так, что она не содержится в одной из строк S_1 и S_2 . Например, если $A = aayxrt$ и $B = aquxhrtw$, то строка $uxhr$ является максимальной общей подстрокой, а ux нет. *Максимальная общая пара* — это тройка (p_1, p_2, n') , где p_1 и p_2 — позиции, соответственно, в S_1 и S_2 , а n' — длина максимальной общей подстроки, начинающейся в этих позициях. Это обобщение максимальной пары для единичной строки.
Обозначив через m полную длину S_1 и S_2 , предложите решение этой задачи за время $O(m + k)$, где k — число троек в результате. Предложите метод со временем $O(m)$ только для подсчета числа максимальных общих пар и алгоритм со временем $O(n + l)$ для нахождения одной копии каждой максимальной общей подстроки, где l — полная длина этих строк. Это обобщение задачи о максимальном повторе для единичной строки.
38. Другой способ распознания супермаксимальных повторов, столь же эффективный, но менее компактный, заключается в следующем. Максимальный повтор в S , представленный различной влево вершиной v в суффиксном дереве для S , является супермаксимальным повтором в том и только том случае, если никакой собственный потомок v не является различным влево и никакая вершина в поддереве v (включая v) не достижима путем из суффиксных связей из различной влево вершины, отличной от v . Докажите.
Покажите, как воспользоваться этим для нахождения всех супермаксимальных повторов за линейное время.
39. В биологических приложениях мы часто интересуемся не только повторяющимися подстроками, но и вхождениями подстрок, где одна подстрока является инвертированной

копией другой, комплементарной копией или (почти всегда) и тем и другим. Покажите, как адаптировать все определения и технику, развитую для повторов (максимальные повторы, максимальные пары, супермаксимальные повторы, почти-супермаксимальные повторы, общие подстроки), к работе с инвертированными и комплементарными копиями с теми же оценками времени.

40. Предложите линейный по времени алгоритм, который применяется к строке S и находит самую длинную максимальную пару без перекрытия. Это значит, что если эти подстроки начинаются в позициях $p_1 < p_2$ и имеют длину n' , то $p_1 + n' < p_2$.
41. Техника для работы с повторами в ДНК нужна не только для повторяющихся структур, которые встречаются в самой ДНК, но и для повторов в данных, полученных из ДНК. Пример тому можно найти в работе Леунга и др. [298]. В ней обсуждается задача анализа последовательностей ДНК из *E. coli*, где данные берутся из более чем 1000 независимо расшифрованных фрагментов, хранящихся в базе данных *E. coli*. Так как эти последовательности были получены при независимых попытках расшифровки, некоторые фрагменты содержат другие, некоторые перекрываются, а многие интервалы генома *E. coli* остаются нерасшифрованными. Поэтому до начала своего анализа авторы хотели “очистить” данные вручную, находя повторно расшифрованные области генома *E. coli* и пакуя все доступные последовательности в несколько *контигов*^{*}, т. е. строки, которые содержат все подстроки из базы данных (эти контиги могут быть кратчайшими из возможных, но это не обязательно).

Как бы вы очищали данные и организовывали последовательности в контиги, используя рассмотренную технику нахождения повторов, суффиксно-префиксных перекрытий и т. п.? (Это приложение поучительно, так как *E. coli*, как и большинство прокариот, содержит мало повторов ДНК. Однако из этого не следует, что техника работы с повторяющимися структурами не применима к прокариотам.)

Аналогичные проблемы очистки возникали в базе данных генома дрожжей, где вдобавок к проблемам типа перечисленных выше строки из других организмов некорректно идентифицировались в дрожжи, дрожжевые строки некорректно идентифицировались в больших составных базах данных, а части клонированных векторов загрязняли вновь открытые дрожжевые строки. Еще больше задача усложняется из-за того, что более позднее высококачественное секвенирование дрожжевой ДНК дает последовательности, в которых ошибок на порядок меньше, чем в последовательностях, уже хранящихся в базах данных. Как эти новые и старые расшифровки объединять вместе — этот вопрос еще не ставился, но ясно, что любой крупномасштабный пересмотр базы данных для дрожжей потребует какого-то варианта обсуждавшихся вычислительных средств.

42. Задача о k -покрытии. Пусть заданы две строки, S_1 и S_2 , и параметр k . Набор C подстрок S_1 , каждая длиной не менее k , назовем k -покрытием, если S_2 можно представить как конкатенацию подстрок C в каком-то порядке. Заметим, что подстроки C могут перекрываться в S_1 , но не в S_2 . То есть S_2 является перестановкой подстрок S_1 , каждая длиной не меньше k . Предложите линейный по времени алгоритм для нахождения k -покрытия по строкам S_1 и S_2 или установления, что такого покрытия не существует.

Если k -покрытие отсутствует, то найти набор подстрок S_1 , каждая длиной не менее k , покрывающих большую часть S_2 . Или указать наибольшее $k' < k$, для которого существует k' -покрытие. Предложите для этих задач алгоритмы, линейные по времени.

^{*}) Используемый автором термин *contig* в словаре отсутствует, но он явным образом происходит от латинского *contiguus*, означающего “соприкасающиеся”. В русскоязычной литературе по молекулярной биологии этим термином обозначают набор клонированных перекрывающихся фрагментов ДНК, полностью насыщающих какой-то участок ДНК (например, область поиска гена). — Прим. перев.

Рассмотрите теперь задачу поиска *неперекрывающихся* подстрок S_1 , каждая длиной не меньше k , покрывающих S_2 полностью или максимально. Эта задача труднее, попробуйте свои силы на ней.

- 43. Тасование экзонов.** В эукариотах ген составляется из чередующихся **экзонов**, конкатенация которых задает определенный белок, и **инtronов**, функция которых не ясна. Сходные экзоны часто наблюдаются в разных генах. Белки нередко формируются из различных доменов (единиц, имеющих различные функции или пространственную организацию, которые относительно независимы друг от друга). Одни и те же домены наблюдаются во многих белках, правда, в разном порядке и комбинациях. Естественно спросить, не соответствуют ли экзоны отдельным белковым доменам, тем более что имеются факты, подтверждающие эту точку зрения. Следовательно, доменная конструкция белка может отражаться в ДНК составной конструкцией гена, основанной на повторном использовании и переупорядочении находящихся в избытке экзонов. Считается, что все белки, расшифрованные к настоящему времени, составлены из всего нескольких тысяч экзонов [468]. Этот феномен повторного использования экзонов называется *тасованием экзонов* (*exon shuffling*), а белки, созданные таким тасованием, называются *мозаичными*. Эти факты приводят к следующей общей задаче поиска.

Пусть заданы неизвестные, но расшифрованные строки ДНК из кодирующих белок областей, в которых экзоны и интроны неизвестны. Попробуйте опознать экзоны, находя общие области (в идеале — одинаковые подстроки) в двух и более строках ДНК. Ясно, что здесь можно применить многие из рассмотренных в этой главе средств для поиска общих и повторяющихся подстрок, хотя их нужно испытать на реальных данных, чтобы проверить их полезность и ограниченность. Никакого элегантного аналитического результата ожидать не приходится. Не кажется ли вам, что кроме методов для общих и повторяющихся подстрок в изучении тасования экзонов была бы полезна задача о k -покрытии? Этот вопрос наверняка требует эмпирического, а не теоретического ответа. Вероятно, полезно найти сначала все максимальные подстроки длиной не меньше k , несмотря на то, что это не может привести к элегантной оценке для наихудшего случая.

44. Докажите лемму 7.14.1.
45. Докажите корректность метода, предложенного в п. 7.13 для задачи линеаризации циклической строки.
46. Разберитесь в вопросе, можно ли эффективно использовать суффиксный массив при решении более сложных задач со строками, рассмотренных в этой главе. Целью является сохранение свойств эффективности по памяти, присущих суффиксному массиву, при достижении эффективности по времени, свойственной суффиксному дереву. Поэтому было бы нечестно сначала использовать суффиксный массив для строки, чтобы затем по нему строить суффиксное дерево этой строки.
47. Уточните подробности предварительной подготовки, необходимой для реализации правила плохого символа в подходе Бойера–Мура к точному множественному совпадению.
48. В п. 7.16.3 мы использовали суффиксное дерево для реализации слабого правила хорошего суффикса для алгоритма множественного точного сопоставления Бойера–Мура. В этой реализации приращение индекса i определялось за константное время после любого теста, независимо от размера алфавита. Распространите этот подход на реализацию *сильного* правила хорошего суффикса, где снова приращение i может быть найдено за константное время. Можете ли вы убрать в этом случае зависимость от алфавита?
49. Докажите теорему 7.16.2.

50. Почему в алгоритме Зива–Лемпеля при вычислении (s_i, l_i) для некоторой позиции i обход кончается в точке p , если строковая глубина p плюс c_v равно i ? Почему продолжение сравнения за символ i вызвало бы проблемы?
51. Постарайтесь объяснить, для чего алгоритм Зива–Лемпеля выводит лишний символ по сравнению с алгоритмом сжатия 1.
52. Покажите, как вычислить все n значений $ZL(i)$, определенные в 7.18, за время $O(n)$. Одно решение связано с вычислением статистики совпадений (п. 7.8.1).

53. **Методы последовательного измельчения.** Последовательное измельчение — это общая алгоритмическая техника, используемая в ряде задач для строк [114, 199, 265]. В нескольких следующих упражнениях мы вводим идеи последовательного измельчения для суффиксных деревьев и применяем его к конкретным задачам.

Пусть S — строка длины n . Отношение E_k определяется над парами суффиксов S . Установим, что $i E_k j$ в том и только том случае, если суффикс i и суффикс j строки S совпадают не меньше чем на k начальных символов. Заметим, что E_k — отношение эквивалентности, так что оно разбивает элементы на классы эквивалентности. Далее, так как S имеет n символов, то каждый класс в E_n состоит из одного элемента. Проверьте следующие два факта:

Факт 1. Если $i \neq j$, то $i E_{k+1} j$ в том и только том случае, когда $i E_k j$ и $i + 1 E_k j + 1$.

Факт 2. Любой класс E_{k+1} является подмножеством класса E_k , и таким образом, разбиение E_{k+1} является измельчением разбиения E_k .

Для представления последовательных измельчений классов E_k , где k возрастает от 0 до n , мы используем дерево с метками T , называемое *деревом измельчения*. Корень T представляет класс E_0 и содержит все n суффиксов S . Каждый потомок этого корня представляет класс из E_1 и содержит элементы этого класса. Вообще, каждая вершина уровня l представляет класс из E_l , а его дети представляют классы из E_{l+1} , на которые он измельчается. Каково соотношение T и дерева ключей (п. 3.4., составленного из множества n суффиксов S)?

Теперь модифицируем T следующим образом. Если вершина v представляет то же множество суффиксов, что и ее предшественница v' , соединим v и v' в одну вершину. В новом дереве T' каждая вершина, не являющаяся листом, будет иметь не меньше двух детей. Каково соотношение T' с суффиксным деревом для строки S ? Покажите, как преобразовать суффиксное дерево для S в дерево T' за время $O(n^2)$.

54. Несколько строковых алгоритмов используют последовательное измельчение без явного нахождения или представления всех классов в дереве измельчения. Вместо этого в них создаются только некоторые из классов или дерево вычисляется неявно. Преимущество такого подхода в экономии памяти и в том, что получающийся алгоритм лучше пригоден для параллельного вычисления [116]. Исходный метод построения суффиксного массива [308] как раз таков. В этом алгоритме суффиксный массив получается как побочный продукт вычисления последовательных измельчений, где разбиения E_k получаются только для значений k , являющихся степенями 2. Мы разовьем этот метод здесь. Нам понадобится одно обобщение факта 1.

Факт 3. Если $i \neq j$, то $i E_{2k} j$ в том и только том случае, когда $i E_k j$ и $i + k E_k j + k$.

Из факта 2 следует, что классы E_{2k} мельче классов E_k .

Алгоритм [308] начинает работу с вычисления разбиения E_1 . Каждый класс из E_1 просто перечисляет все места в S с каким-нибудь конкретным символом алфавита, и классы расположены по порядку этих символов. Например, если $S = mississippi\$$, то E_1 состоит из пяти классов: $\{12\}$, $\{2, 5, 8, 11\}$, $\{1\}$, $\{9, 10\}$ и $\{3, 4, 6, 7\}$. Класс $\{2, 5, 8, 11\}$ перечисляет

позиции всех i в S и стоит перед классом, соответствующим m , который предшествует классу для s , и т. д. Символ конца строки $\$$ считается предшествующим любому другому символу.

Практический метод получения E_1 зависит от размера алфавита и от способа его представления. Это определенно можно сделать за $O(n \log n)$ сравнений символов.

Для любого $k \geq 1$ мы можем получить разбиение E_{2k} , измельчая E_k , как это предлагается фактом 3. Однако неясно, как эффективно реализовать его прямое использование. Поэтому взамен мы создадим разбиение E_{2k} за время $O(n)$, используя *обратный* подход к измельчению. Вместо проверки класса C из E_k , чтобы найти, как C измельчится, мы используем C как измельчитель и посмотрим, как он заставляет разбиваться другие классы E_k или оставляет их нетронутыми. Для каждого номера $i > k$ из C найдем и пометим число $i - k$. Затем для каждого класса A из E_k все числа в A , помеченные C , определят полный класс в E_{2k} . Корректность следует из факта 3.

Дайте полное доказательство корректности метода обратного измельчения для создания разбиения E_{2k} по разбиению E_k .

55. Каждый класс из E_k для любого k содержит начальные позиции подстрок S длины k . Алгоритм из [308] строит суффиксный массив для S , используя подход обратного измельчения с той дополнительной подробностью, что классы из E_k берутся в лексическом порядке строк, соответствующих этим классам.

Поясним на примере. Чтобы получить разбиение E_2 для $S = mississippi\$$, нужно по порядку обработать классы из E_1 , от лексически наименьшего до лексически наибольшего. Результатом обработки первого класса, $\{12\}$, будет создание в E_2 класса $\{11\}$. Второй класс из E_1 , $\{2, 5, 8, 11\}$, помечает индексы $\{1, 4, 7\}$ и $\{10\}$ и, следовательно, создает в E_2 три класса: $\{1\}$, $\{4, 7\}$ и $\{10\}$. Класс $\{9, 10\}$ из E_1 создает два класса: $\{8\}$ и $\{9\}$. Класс $\{3, 4, 6, 7\}$ из E_1 создает классы $\{2, 5\}$ и $\{3, 6\}$. Каждый класс из E_2 содержит начальные позиции одинаковых подстрок длины 1 или 2. Эти классы, лексически упорядоченные подстроками, которые они представляют, таковы: $\{12\}$, $\{11\}$, $\{8\}$, $\{2, 5\}$, $\{1\}$, $\{10\}$, $\{9\}$, $\{4, 7\}$, $\{3, 6\}$. Классы E_4 в лексическом порядке таковы: $\{12\}$, $\{11\}$, $\{8\}$, $\{2, 5\}$, $\{1\}$, $\{10\}$, $\{9\}$, $\{7\}$, $\{4\}$, $\{6\}$, $\{3\}$. Отметим, что $\{2, 5\}$ остаются в том же классе E_4 , так как $\{4, 7\}$ находятся в одном и том же классе E_2 . Классы из E_2 $\{4, 7\}$ и $\{3, 6\}$ измельчаются в E_4 . Объясните, почему.

Хотя общая идея обратного измельчения должна уже быть ясна, ее эффективная реализация требует ряда дополнительных деталей. Приведите подробности реализации и анализа, чтобы доказать, что классы E_{2k} могут быть получены из классов E_k за время $O(n)$. Проделайте за деталями, обеспечивающими обработку классов в лексическом порядке.

Предположим, что n является степенью 2. Заметьте, что алгоритм может остановиться, когда в каждом классе останется по одному элементу, что и должно произойти за $\log_2 n$ итераций. Когда алгоритм завершается, порядок этих (одноэлементных) классов описывает перестановку чисел от 1 до n . Докажите, что перестановка является суффиксным массивом для строки S . Используя это, докажите, что метод обратного измельчения создает суффиксный массив за время $O(n \log n)$. Каков выигрыш в памяти у этого метода по сравнению с методом, описанным в п. 7.14.1, время работы которого имеет порядок $O(n^2)$?

56. **Примитивные tandemные ряды.** Напомним, что строка α называется *тандемным рядом*, если она периодическая (см. п. 3.2.1), т. е. может быть записана в виде β^l для $l \geq 2$. Когда $\beta = 2$, tandemный ряд можно также назвать *тандемным повтором*. Тандемный ряд $\alpha = \beta^l$, содержащийся в строке S , называется *максимальным*, если ни перед, ни после α дополнительных копий β нет.

Максимальные tandemные ряды были введены в упражнении 4 к главе 1 (с. 34), а важность tandemных рядов и повторов обсуждалась в п. 7.11.1. Мы интересуемся распознаванием

максимальных тандемных рядов в строке. Как уже говорилось, лучше всего рассмотреть структурированное подмножество интересующих нас строк, чтобы ограничить размер результата и опознать наиболее информативные строки. Мы обсудим здесь некоторое подмножество максимальных тандемных рядов, которое компактно в неявной форме кодирует все максимальные тандемные ряды. (В пп. 9.5, 9.6 и 9.6.1 мы покажем эффективные методы нахождения в строке *всех* тандемных повторов и позволим повторам содержать некоторое число ошибок.)

Парой (β, l) будем описывать тандемный массив β^l . Рассмотрим тандемный массив $\alpha = abababababababab$. Его можно описать парой $(abababab, 2)$, или парой $(abab, 4)$, или $(ab, 8)$. Какое описание наилучшее? Так как первые две пары могут быть выведены из последней, эту последнюю мы и выберем. Этот “выбор” сейчас будет точно определен.

Строка β называется *примитивной*, если она не периодическая. Например, строка ab примитивная, а $abab$ нет. Пара $(ab, 8)$ дает предпочтительное описание $abababababababab$, так как строка ab примитивна. Предпочтение, отдаваемое примитивным строкам, естественно распространяется на описания максимальных тандемных рядов, появляющихся в качестве подстрок в строках большего размера. При заданной строке S мы используем тройку (i, β, l) , чтобы записать, что тандемный ряд (β, l) входит в S , начиная с позиции i . Тройка (i, β, l) называется *пм-тройкой*, если β примитивна и β^l — максимальный тандемный ряд.

Например, максимальные тандемные ряды в *mississippi*, описываемые пм-тройками, это $(2, iss, 2)$, $(3, s, 2)$, $(3, ssi, 2)$, $(6, s, 2)$ и $(9, p, 2)$. Отметим, что две или более пм-тройки могут иметь один и тот же первый номер, так как два различных максимальных тандемных ряда могут начинаться с одной и той же позиции. Например, оба максимальных тандемных ряда *ss* и *ssissi* начинаются с позиции 3 строки *mississippi*.

Все тандемные ряды в строке скжато кодируются пм-тройками. Крочемор [114] (в других терминах) использовал метод последовательного измельчения для нахождения всех пм-тройек за время $O(n \log n)$. Из этого следует совсем нетривиальный факт, что в любой строке длины n может быть только $O(n \log n)$ пм-тройек. Метод в [114] находит разбиение E_k для каждого k . Центральную роль играет следующая лемма.

Лемма 7.20.1. *Тандемный повтор подстроки β длины k , начинающийся в позиции i , присутствует в строке S в том и только том случае, если числа i и $i + k$ содержатся в одном и том же классе разбиения E_k и никакие числа, лежащие между i и $i + k$, этому классу не принадлежат.*

Докажите лемму. В одном направлении доказательство ясно. В другом направлении оно труднее, и может оказаться полезным использовать лемму 3.2.1 (с. 65).

57. Лемма 7.20.1 упрощает опознание пм-тройек. Предположим, что индексы в каждом классе E_k отсортированы в возрастающем порядке. Из леммы следует, что (i, β, j) является пм-тройкой, где β — подстрока длины k , в том и только том случае, если некоторый класс E_k содержит максимальную арифметическую прогрессию $i, i - k, i + 2k, \dots, i + jk$ с шагом k . Объясните это.

Используя факт 1 вместо факта 3 и модифицируя метод обратного измельчения, развитый в упражнениях 54 и 55, покажите, как вычислить разбиения E_k для всех k (а не только для степеней 2) за время $O(n^2)$. Приведите детали реализации, обеспечивающие поддержание набора индексов в каждом классе упорядоченным по возрастанию. Далее продолжите развитие этого метода, чтобы с использованием леммы 7.20.1 получить алгоритм нахождения всех пм-тройек строки S за время $O(n^2)$.

58. Чтобы найти все пм-тройки за время $O(n \log n)$, Крочемор [114] использовал одну дополнительную идею. Чтобы описать ее, предположим, что все классы E_k , кроме одного, например C , использовались как измельчители при создании E_{k+1} из E_k . Пусть p и q — два индекса, принадлежащие одному и тому же классу в E_k . Мы утверждаем, что если p и q не должны находиться в одном и том же классе E_{k+1} , то по крайней мере один из этих индексов уже находится в нужном классе в E_{k+1} . Причина в том, что по факту $p+1$ и $q+1$ не могут быть в одном и том же классе E_k . Таким образом, к тому времени, когда C применяется как измельчитель, либо p , либо q уже маркирован и перемещен каким-то классом E_k , использованным в качестве измельчителя.

Теперь предположим, что каждый класс E_k содержится в цепном списке и что, когда измельчитель опознает номер, допустим, p , то p удаляется из текущего списка и помещается в список соответствующего класса E_{k+1} . При таком уточнении если алгоритм использовал все классы E_k , кроме C , то все классы E_{k+1} корректно представлены вновь созданными цепными списками и тем, что осталось от исходных цепных списков для E_k . Подробно объясните это. Сделайте отсюда вывод, что один из классов E_k может не использоваться как измельчитель.

Возможность пропустить один класс при измельчении E_k , конечно, желательна, но ее недостаточно для получения заявленной оценки. Для нее мы должны повторить эту идею в большем масштабе.

Теорема 7.20.1. *При измельчении E_k для создания E_{k+1} предположим, что для каждого $k > 1$ ровно один (произвольный) потомок каждого класса E_{k-1} пропущен (т.е. не использован как измельчитель). Тогда результатирующие цепные списки корректно определяют классы E_{k+1} .*

Докажите теорему 7.20.1. Отметьте, что она предоставляет полную свободу в выборе не используемого для измельчения класса E_{k-1} . Это приводит к следующей теореме.

Теорема 7.20.2. *Если для каждого $k > 1$ не используется наибольший потомок каждого класса E_{k-1} , то полный размер всех классов, используемых в качестве измельчителей, не превосходит $n \log_2 n$.*

Докажите теорему 7.20.2. Теперь обеспечьте все детали реализации для нахождения всех пм-троек в S за время $O(n \log n)$.

59. Выше мы установили границу в $O(n \log n)$ пм-троек как побочный продукт алгоритма их построения. Но возможно прямое, неалгоритмическое доказательство, которое обращается к идее последовательного измельчения и лемме 7.20.1. На самом деле на этом пути легко получить границу $3n \log_2 n$. Проделайте это.
60. Существует фольклорное мнение, что для любой позиции i в S , если есть две пм-тройки, (i, β, l) и (i, β', l') и если $|\beta'| > |\beta|$, то $|\beta'| \geq 2|\beta|$. Это ограничило бы число пм-троек с тем же первым числом до $\log_2 n$, и оценка $O(n \log n)$ была бы непосредственной.
- Покажите на примере, что фольклорное убеждение неверно.
61. **Задача выбора праймера.** Пусть \mathcal{S} — набор строк над конечным алфавитом Σ . Предложите алгоритм (использующий обобщенное суффиксное дерево) для нахождения *кратчайшей* строки S над Σ , которая не является подстрокой ни одной строки из \mathcal{S} . Алгоритм должен работать за время, пропорциональное сумме длин строк из \mathcal{S} .

Более полезный вариант этой задачи — найти кратчайшую строку S , которая имеет длину не меньше заданной и не является подстрокой никакой строки из \mathcal{S} . Часто кроме этого набора задается строка α и требуется найти кратчайшую подстроку α (если такая существует), которая не входит подстрокой ни в одну строку из \mathcal{S} . В более общих терминах, для

каждого i вычислить кратчайшую подстроку (если такая существует), которая начинается в позиции i строки α и не входит как подстрока ни в одну строку δ .

Приведенные задачи могут быть обобщены во многих направлениях и решаться в основном тем же способом. Особенno интересна задача о выборе праймера в варианте точного совпадения. (В п. 12.2.5 мы рассмотрим эту задачу с возможностью ошибок.)

Задача выбора праймера часто встречается в молекулярной биологии. Примером является “хождение по хромосоме” — техника, используемая в некоторых методах секвенирования ДНК или локализации генов. Хождение по хромосоме широко использовалось при локализации гена муковисцидоза в седьмой хромосоме человека. Мы обсудим здесь только приложение, относящееся к расшифровке ДНК.

Целью секвенирования ДНК является определение полной последовательности нуклеотидов в длинной строке ДНК. Чтобы понять это приложение, вы должны знать две вещи относительно существующей технологии секвенирования. Во-первых, с помощью имеющихся сейчас лабораторных методов можно расшифровывать только небольшие участки, от 300 до 500 элементов, от одного конца более длинной нуклеотидной строки. Во-вторых, есть возможность копировать подстроки длинной строки ДНК, начиная почти с любого места, если только вы знаете маленькое число нуклеотидов, например 9, слева от этой точки. Копирование выполняется с использованием технологии, называемой *полимеразной цепной реакцией* (polymerase chain reaction — PCR), которая стала знаменательным достижением экспериментальной молекулярной биологии. Знание всего девяти нуклеотидов позволяет синтезировать строку, комплементарную к ним. Эту комплементарную строку можно использовать в качестве “праймера”, который добирается в длинной строке до места, где находится комплементарная ему подстрока. Затем он гибридизуется с длинной строкой в этой точке, что создает условия, позволяющие воспроизвести часть исходной строки справа от местонахождения праймера. (Обычно PCR выполняется с двумя праймерами, по одному на каждый конец, но здесь используется только один “переменный” праймер. Другой праймер фиксирован, и в этом обсуждении им можно пренебречь.)

Приведенные два факта лежат в основе метода секвенирования длинной строки ДНК. Предположим, что мы знаем первые девять нуклеотидов в самом начале строки. После расшифровки первых (допустим) 300 нуклеотидов нужно синтезировать праймер, комплементарный к последним девяти расшифрованным нуклеотидам. Затем скопировать строку, содержащую следующие 300 нуклеотидов, расшифровать ее и продолжать далее. Таким образом, более длинная строка расшифровывается кусками по 300 нуклеотидов за один раз, при этом конец каждой расшифрованной подстроки используется для создания праймера, инициирующего расшифровку следующей подстроки. В сравнении с методом дробового секвенирования (который будет рассмотрен в п. 16.14), этот *направленный* метод требует гораздо меньше перекрытий расшифрованных участков, но так как метод в принципе последовательный, то длинную строку ДНК приходится расшифровывать долго. (В случае муковисцидоза другая идея, называемая *генными прыжками*, применялась для частичной параллелизации этого последовательного процесса, но в общем хождение по хромосоме утомительно последовательно.)

В методе хождения по хромосоме есть одна трудность. Что случится, если строка, состоящая из последних девяти нуклеотидов, встретится в другом месте большей строки? Тогда праймер может не гибридизоваться в правильной позиции, и последовательность, определенная этой точкой, будет неправильной. Так как мы знаем последовательность слева от текущей точки, то можем проверить, нет ли в ней строки, комплементарной нашему праймеру. Если есть, то мы захотим выбрать другую подстроку длины девять, близкую к концу последней расшифрованной последовательности и не встречавшуюся ранее. Ее и нужно будет использовать как праймер. В результате секвенирование следующей подстроки

заново расшифрует некоторые уже известные нуклеотиды, и число заново секвенированных нуклеотидов будет немного менее 300.

Формализуйте задачу выбора праймера и покажите, как эффективно ее решать, используя суффиксные деревья. В более общем виде, для каждой позиции i в строке α найти кратчайшую подстроку, которая начинается в i и не появляется больше ни в α , ни в β .

62. В задаче выбора праймера добиться неправильной гибридизации *справа* от расшифрованной части гораздо труднее, так как мы еще не знаем последовательности. Однако есть некоторые известные последовательности, которых следует избегать. Как говорилось в п. 7.11.1, эукариотическая ДНК часто содержит области повторяющихся подстрок, многие из которых известны. Вайнберг^{*)} пишет о проблеме, которую повторяющиеся строки вызывают в хромосомном блуждании:

“Они — как зыбучий песок: все, что на них наступает, всасывается в них, а затем, как Алиса в Стране чудес, протаскивается через какую-то громадную систему подземных туннелей, только чтобы вынырнуть на поверхность где-то в геноме, в милях от начального места. Геном насыщен такими строками, называемыми повторяющимися последовательностями. Они гарантированно замедляют любое хождение по хромосоме до ползания”.

Итак, более общая задача о праймере состоит в следующем. Заданы строка α , состоящая из 300 нуклеотидов (последняя расшифрованная подстрока), строка β известной расшифровки (часть длинной строки слева от α , расшифровка которой известна) и набор строк δ (общие части известных повторов ДНК). Требуется найти крайнюю правую подстроку α длины девять, не являющуюся подстрокой β или какой-нибудь из строк δ . Если такой строки не существует, то можно искать строку большей длины, не присутствующую в β или δ . Однако праймер длины много больше, чем 9 нуклеотидов, может неверно гибридизоваться по другим причинам. Поэтому нужно балансировать ограничения, держащие длину праймера в определенных пределах, обеспечение его уникальности и размещение его возможно правее.

Формализуйте этот вариант задачи о выборе праймера и покажите, как к нему применить суффиксные деревья.

Выбор проб. Вариантом задачи о выборе праймера является задача о выборе *пробы для гибридизации*. В задачах геномной дактилоскопии и картирования ДНК (обсуждаемых в главе 16) часто возникает потребность посмотреть, какие *олигомеры* (короткие отрезки ДНК) гибридизуются с некоторыми интересующими нас фрагментами ДНК. Цель такой гибридизации не в том, чтобы создать праймер для PCR, а в том, чтобы извлечь некоторую информацию относительно целевой ДНК. При геномной дактилоскопии и картографии обычно происходит заражение целевой ДНК векторной ДНК, в результате которого олиго-проба может гибридизоваться с векторной ДНК вместо целевой. Один из подходов к решению этой проблемы заключается в использовании специально созданных олигомеров, последовательности которых редко встречаются в геноме вектора, но часто находятся в клонируемой ДНК. Это в точном виде задача выбора праймера (или пробы).

В некоторых смыслах задача о выборе пробы больше подходит к технике точного совпадения, обсуждавшейся в этой главе, чем задача о праймере. Это объясняется тем, что при составлении проб для картирования желательно и допустимо разрабатывать их так, чтобы даже однократное несовпадение разрушало гибридизацию. Такие строгие пробы при некоторых условиях могут быть разработаны [134, 177].

^{*)} Weinberg R. *Racing to the beginning of the Reload: The Search for the Origin of Cancer*. Harmony Books, 1996.

Глава 8

Общий наименьший предшественник

8.1. Введение

Мы начнем обсуждение с поразительного результата, значительно увеличивающего область применения суффиксных деревьев (в добавление ко многим другим приложениям).

Определение. В дереве \mathcal{T} с корнем вершина u называется *предшественником* вершины v (или u *предшествует* v), если u лежит на единственном пути из корня в v . Согласно определению вершина предшествует самой себе. *Собственным предшественником* v называется предшественник, отличный от v .

Определение. В дереве \mathcal{T} с корнем *наименьший общий предшественник* (lowest common ancestor — *lca*) двух вершин, x и y , — это самая нижняя вершина в \mathcal{T} , предшествующая и x , и y .

Например, на рис. 8.1 вершина 5 является *lca* вершин 6 и 10, а вершина 1 — это *lca* для 6 и 3.

Потрясающий результат заключается в том, что после *линейной* предварительной подготовки дерева с корнем наименьший общий предшественник любых двух вершин может быть найден за константное время. То есть дерево с корнем, имеющее n вершин, предварительно обрабатывается за время $O(n)$, после чего любой запрос о наименьшем общем предшественнике выполняется за константное время, независимо от n . Без предварительной подготовки лучшая граница наихудшего случая для единичного запроса равна $\Theta(n)$, так что это самый удивительный и полезный

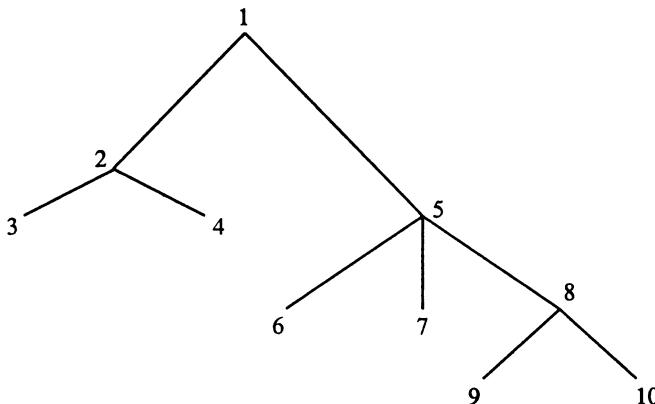


Рис. 8.1. Общее дерево \mathcal{T} с вершинами, перенумерованными обходом в глубину

результат. Он был сначала получен Харелом и Таржаном [214], а затем упрощен Шибером и Вишким [393]. Предлагаемое здесь изложение основано на втором подходе.

8.1.1. Что предшественникам делать со строками?

Поиск с константным временем особенно полезен в ситуациях, когда при фиксированном дереве нужно обрабатывать много запросов о наименьшем общем предшественнике. Такая ситуация часто встречается в строковых задачах. Чтобы дать представление об отношении между строками и общими предшественниками, отметим, что если пути от корня до двух листьев i и j суффиксного дерева совпадают до вершины v , то суффиксы i и j имеют общий префикс — строку, помечающую путь до v . Следовательно, наименьший общий предшественник листьев i и j определяет наибольший общий префикс суффиксов i и j . Нахождение наибольшего общего префикса будет важным примитивом во многих строковых задачах. Некоторые из них мы рассмотрим подробно в главе 9. В этой главе операцию нахождения lca можно считать черным ящиком, так как здесь мы хотим ознакомить читателя с побуждающими примерами до углубления в технические детали реализации.

Утверждение о возможности находить lca за константное время широко известно, а его детали — нет. Сам результат обычно принимается в литературе о строках как черный ящик, и существует общее “народное” мнение о нем, как о чисто теоретическом результате, не имеющем практического смысла. Это определенно неверно по отношению к методу Шибера–Вишкина — весьма практичному методу с низкими накладными расходами, легко программируемому и очень быстрому в работе. Его непременно следует держать в стандартном репертуаре строковых пакетов. Однако, хотя метод прост для программирования, он далеко не прост для понимания и описания, так как основан на “манипуляциях с битами”. Тем не менее результат настолько эффективен в различных строковых методах и его использование настолько для них важно, что детальное его изложение вполне обосновано. Мы надеемся, что следующее ниже обсуждение будет существенным шагом к более широкому пониманию метода.

8.2. Предполагаемая модель машины

Поскольку нам так важно константное время обработки запроса (ведь даже линейное время предварительной обработки требует особого внимания к деталям), мы должны полностью прояснить вопрос об используемой вычислительной модели. В противном случае нас могут обвинить в мошенничестве — использовании слишком мощного набора элементарных операций. Какие элементарные операции разрешено выполнять за константное время? В модели RAM единичной стоимости, когда входное дерево имеет n вершин, мы можем допустить, что любое число размера $O(\log n)$ битов записывается, читается и используется как адрес за константное время. Кроме того, за константное время два числа такого размера можно сравнивать, складывать, вычитать, перемножать и делить. Числа, представляемые более чем $\Theta(\log n)$ битами, за константное время обрабатываться не могут. Это стандартные требования единичной стоимости, запрещающие манипуляции с “большими” числами за константное время. Результат про *lca* (что на запрос об *lca* можно получить ответ за константное время зосле линейной по времени предварительной подготовки) можно доказать в этой модели RAM единичной стоимости. Однако изложение будет легче, если мы предположим, что на битовом уровне можно выполнять за константное время некоторые дополнительные операции, пока числа имеют только $O(\log n)$ битов. В частности, мы предполагаем, что за константное время можно выполнить над парами двоичных чисел (подходящего размера) операции AND, OR и XOR (исключающее или); двоичное число может быть сдвинуто (влево или вправо) на $O(\log n)$ битов; может быть создана битовая “маска” из последовательных единиц и может быть найдена позиция крайней левой и крайней правой единицы в двоичном числе. Для многих машин и нескольких языков программирования высокого уровня эти предположения вполне резонны, опять-таки, если запись операндов имеет длину не более $O(\log n)$ битов. Но для туристов мы после того, как обоснуем результат для *lca* при использовании этих более либеральных предположений, покажем, как тех же самых результатов можно достичь, используя только стандартную модель.

8.3. Полные двоичные деревья: очень простой случай

Начнем обсуждение *lca* с особо простого дерева, вершины которого перенумерованы специальным образом. Это дерево является *полным двоичным деревом*, а номера вершин будут кодами путей до них. Обозначение \mathcal{B} будет относиться к этому полному двоичному дереву, а \mathcal{T} — к произвольному дереву.

Итак, пусть \mathcal{B} — полное двоичное дерево с корнем, имеющее p листьев (всего $\tau = 2p - 1$ вершин), так что каждая внутренняя вершина имеет строго двух детей и число дуг на пути из корня до любого листа равно $d = \log_2 p$. Это значит, что дерево полное и глубина от корня у всех листьев одна и та же. Каждой вершине v в \mathcal{B} сопоставляется $(d + 1)$ -битовое число, называемое ее *путевым номером*, которое кодирует единственный путь от корня до v . Если считать от крайнего левого бита, то i -й бит путевого номера для v соответствует i -й дуге на пути от корня до v : 0 означает, что i -я дуга пути идет к левому потомку, а 1 — к правому^{*)}). Например,

^{*)} Отметим, что обычно при рассмотрении записей двоичных чисел биты нумеруются справа налево (от менее значимых к более значимым). Эта нумерация противоположна используемой для строк и путевых номеров.

путь, который идет дважды налево, затем направо, а затем опять налево, приходит в вершину, путевой номер которой начинается с 0010. Биты, которые описывают путь, называются *путевыми битами*. Каждый путевой номер дополнен до $d + 1$ бита приписыванием справа от путевых битов одной единицы и нужного числа нулей. Так, например, если $d = 6$, вершина с путевыми битами 0010 получает семибитовый номер 0010100. Корень при $d = 6$ имеет номер 1000000. Номер корня всегда так выглядит: единица в левом бите и нули на d остальных местах. (См. еще один пример на рис. 8.2.) Мы будем ссылаться на вершины \mathcal{B} , используя их путевые номера.

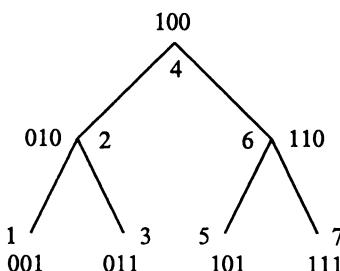


Рис. 8.2. Двоичное дерево с четырьмя листьями. Путевые номера записаны в двоичном виде, и в десятичном

Как видно по дереву на рис. 8.2, путевые номера имеют другое известное описание — это числа *фронтальной* (inorder) нумерации вершин дерева. При этом когда вершины \mathcal{B} нумеруются фронтальным обходом (рекурсивно нумеруется левая ветвь, затем корень, затем рекурсивно правая ветвь), получающиеся так номера вершин — это в точности наши путевые номера. Доказательство оставляется читателю (они мало существенно для дальнейшего). Концепция путевого номера для нас предпочтительна, так как она явно связывает номер вершины с описанием пути до нее от корня.

8.4. Как разрешать запросы об lca в \mathcal{B}

Определение. Для любых вершин i и j через $lca(i, j)$ обозначен общий наименьший предшественник i и j .

Пусть заданы две вершины, i и j , и мы хотим найти $lca(i, j)$ в \mathcal{B} (помня, что i и j — это путевые номера). Прежде всего, когда $lca(i, j)$ равно либо i , либо j (т. е. одна из этих двух вершин предшествует другой), то это может быть обнаружено описанным в упражнении 3 очень простым алгоритмом с константным временем. Итак, предположим, что $lca(i, j)$ не равно ни i , ни j . Алгоритм начинается вычислением *исключающего ИЛИ* (XOR) от двоичных представлений i и j , результат вычисления обозначим через x_{ij} . Так как длина i и j имеет порядок $O(\log n)$, операция XOR в нашей модели выполняется за константное время.

Далее в x_{ij} находится старший значимый единичный бит (самый левый). Если он находится в позиции k (считая слева), то $k - 1$ левых битов i и j совпадают, и пути до i и j должны совпадать на первых $k - 1$ дугах, а затем расходиться. Отсюда

следует, что путевое число для $lca(i, j)$ состоит из (слева направо) $k - 1$ битов i (или j), затем единичного бита, а затем $d + 1 - k$ нулей. Например, на рис. 8.2 XOR от 101 и 111 (вершины 5 и 7) равен 010, так что их пути имеют одну общую дугу — идущую вправо от корня. XOR от 010 и 101 (вершины 2 и 5) равен 111, так что пути до 2 и 5 не совпадают вообще, и следовательно, 100, корень, и является их наименьшим общим предшественником.

Поэтому, чтобы найти $lca(i, j)$, алгоритм должен вычислить XOR двух чисел, найти позицию k крайнего левого бита с единицей в результате, сдвинуть i вправо на $d + 1 - k$ позиций, положить крайний правый бит равным 1 и сдвинуть влево на те же $d + 1 - k$ позиций. По предположению, каждую из этих операций можно выполнить за константное время, и следовательно, наименьший общий предшественник i и j может быть найден в \mathcal{B} за константное время.

В итоге получается следующая теорема.

Теорема 8.4.1. *В полном двоичном дереве после подготовки, в которой за линейное время вершинам присваиваются их путевые номера, любой запрос о наименьшем общем предшественнике получает ответ за константное время.*

Этот простой случай полного двоичного дерева весьма специфичен, но было две причины показать его: чтобы развить интуицию и чтобы применить затем полные двоичные деревья в описании общего случая. Более того, используя полные двоичные деревья, можно предложить очень элегантный и относительно простой алгоритм, который может отвечать за запросы об lca за константное время, если затратить времени $\Theta(n \log n)$ на препроцессинг \mathcal{T} и память $\Theta(n \log n)$ после него. Этот метод используется в упражнении 12.

Алгоритм lca , который мы представим для общих деревьев, основан на случае полного двоичного дерева. Идея (концептуально) заключается в том, что вершины общего дерева \mathcal{T} отображаются в вершины полного двоичного дерева \mathcal{B} таким образом, что поиск lca на \mathcal{B} помогает быстро разрешать запросы lca на \mathcal{T} . Мы сначала опишем общий алгоритм lca , предполагая, что отображение \mathcal{T} в \mathcal{B} задано явно, а затем объясним, как можно обойтись без явного отображения.

8.5. Первые шаги в отображении \mathcal{T} в \mathcal{B}

Первое, что делает препроцессинг, — обход \mathcal{T} в глубину с нумерацией вершин в таком порядке, как они впервые появляются при обходе. Это *стандартная нумерация в глубину* (preorder numbering — нисходящая нумерация) (см. рис. 8.1). В ней вершины поддерева любой вершины v в \mathcal{T} получают последовательные номера в глубину, начиная с самой этой вершины (далее номера dfs). Это значит, что если в поддереве с корнем v имеется q вершин и v получает номер k , то остальные вершины в поддереве получают номера от $k + 1$ до $k + q - 1$.

Для удобства с этого момента на вершины \mathcal{T} мы будем ссылаться, используя номера dfs . Таким образом, когда мы упоминаем вершину v , то v — это одновременно и вершина, и номер. Будьте внимательны, чтобы не спутать номера общего дерева \mathcal{T} с путевыми номерами, используемыми только для двоичного дерева \mathcal{B} .

Определение. Для любого числа k пусть $h(k)$ обозначает позицию (считая справа) наименьшей значимой единицы в двоичном представлении k (ср. рис. 8.2 и 8.3).

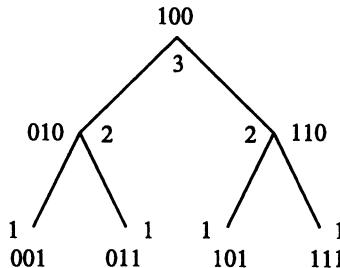


Рис. 8.3. Двоичное дерево с четырьмя листьями. Путевые номера записаны в двоичном виде. а позиции старшей значимой единицы — в десятичном

Например, $h(8) = 4$, так как $8 = 1000_2$, а $h(5) = 1$, так как $5 = 101_2$. Другой способ представить это — считать, что $h(k)$ на единицу больше числа последовательных нулей в правой части k .

Определение. В полном двоичном дереве *высота* вершины есть число вершин в пути от нее до листа. Высота листа равна 1.

Следующая лемма устанавливает решающий факт, который легко доказывается индукцией по высоте вершин.

Лемма 8.5.1. Для любой вершины k в \mathcal{B} (вершины с путевым номером k) $h(k)$, равно высоте вершины k в \mathcal{B} .

Например, вершина 8 (двоичное 1000) имеет высоту 4, и путь от нее до листа содержит четыре вершины (три дуги).

Определение. Для вершины v из \mathcal{T} обозначим через $I(v)$ вершину w из \mathcal{T} с наибольшим значением $h(w)$ среди всех вершин поддерева v (включая саму v).

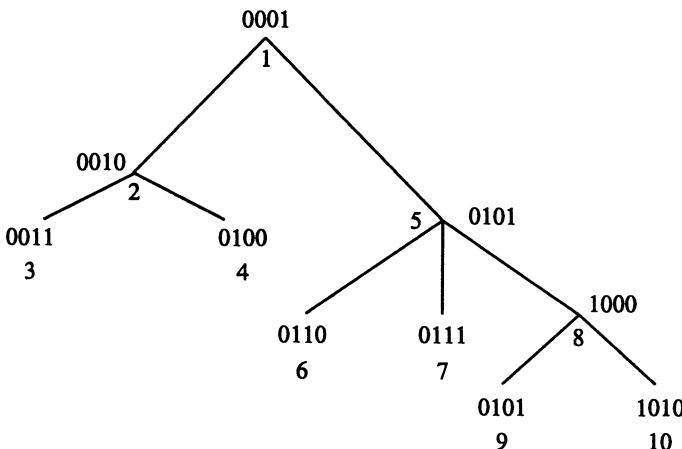


Рис. 8.4. Номера вершин в четырехбитовом представлении, иллюстрирующие определение $I(v)$

Таким образом, $I(v)$ показывает ту вершину (ее *dfs*-номер), у которой двоичное представление номера имеет наибольшее число нулей на правом конце среди всех вершин поддерева v . Рис. 8.4 показывает номера вершин с рис. 8.1 в двоичном и десятичном представлении. Мы видим, что $I(1)$, $I(5)$ и $I(8)$ равны 8, $I(2)$ и $I(4)$ — оба 4, а для всех остальных вершин рисунка $I(v) = v$.

Ясно, что если вершина v предшествует вершине w , то $h(I(v)) > h(I(w))$. По-другому можно это сказать так: значения $h(I(v))$ не убывают вдоль пути вверх в дереве \mathcal{T} . Этот факт будет учитываться в некоторых доказательствах.

В дереве на рис. 8.4 вершина $I(v)$ однозначно определена для каждой вершины v . Значит, для каждой вершины v в ее поддереве существует ровно одна w , где значение $h(w)$ максимально. Это не случайность, а факт, важный для алгоритма *lca*. Мы его сейчас докажем.

Лемма 8.5.2. Для любой вершины v дерева \mathcal{T} в поддереве v существует единственная вершина w , на которой достигается максимум $h(w)$ по этому поддереву.

	l	k	i	
u		1	10	...
w		0	10	...
N		10	...	

↓ ↓

В u , w и N биты
от l до $k + 1$ совпадают

Рис. 8.5. Номера u , w и N

Доказательство. Допустим, что это не так. Пусть u и w — две вершины поддерева v , для которых $h(u) = h(w) \geq h(q)$ для любой вершины q из поддерева. Пусть $h(u) = i$. Добавляя, если требуется, нули к левому концу, мы можем считать, что номера u и w имеют одинаковое число битов, скажем, l . Так как $u \neq w$, эти два числа должны отличаться в каком-то бите слева от i (так как по предположению и в u и в w бит i равен 1, а справа от него все биты нулевые). Пусть $u > w$ и k — самая левая позиция, где записи u и w различаются. Рассмотрим число N , составленное из левых $l - k$ битов u , за которыми следует единица в бите k , а затем $k - 1$ нулей (рис. 8.5). Тогда N строго меньше чем u и больше чем w (здесь мы пользуемся тем, что $k > i$). Следовательно, N должно быть первым *dfs*-номером, данным какой-нибудь вершине из поддерева v , так как нисходящие номера, данные вершинам ниже v , образуют сплошной интервал. Но $h(N) = k > i = h(u)$ вопреки тому, что $h(u) \geq h(q)$ для всех вершин поддерева v . Таким образом, предположение, что $h(u) = h(w)$, приводит к противоречию, и лемма доказана. \square

Итогом рассмотрения единственности $I(v)$ для всех v является следующее утверждение.

Следствие 8.5.1. *Функция $v \rightarrow I(v)$ определена.*

8.6. Отображение \mathcal{T} в \mathcal{B}

При отображении вершин \mathcal{T} в вершины двоичного дерева \mathcal{B} мы хотим в достаточной мере сохранить отношения наследования на \mathcal{T} для того, чтобы при вычислении запросов по *lca* на \mathcal{T} можно было использовать отношения *lca* на \mathcal{B} . В определении этого отображения центральная роль принадлежит функции $v \rightarrow I(v)$. Первым шагом в его понимании будет разбиение вершин \mathcal{T} на классы с одинаковым значением I .

Определение. Назовем *полосой* в \mathcal{T} максимальное подмножество вершин \mathcal{T} с одним и тем же значением I .

Итак, две вершины, u и v , принадлежат одной полосе в том и только том случае, если $I(u) = I(v)$. На рис. 8.6 показано разбиение вершин \mathcal{T} на полосы.

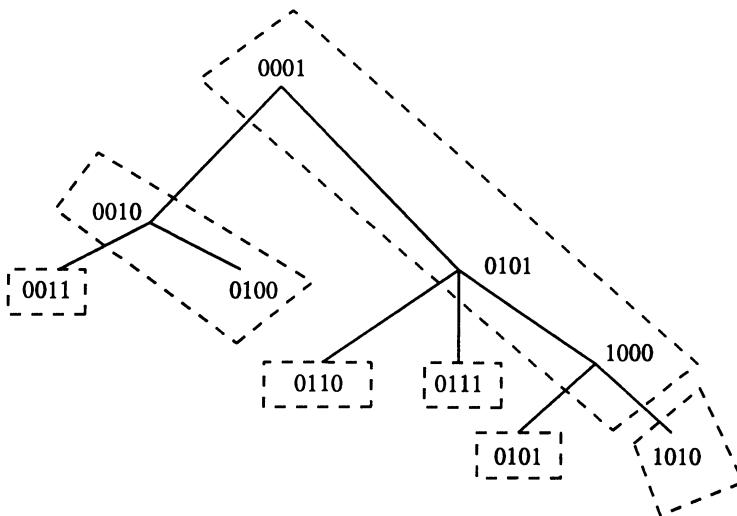


Рис. 8.6. Разбиение множества вершин на семь полос

Алгоритмически мы можем найти $I(v)$ для всех вершин за линейное время, обходя \mathcal{T} снизу вверх следующим образом. Для каждого листа u примем $I(u) = v$. Для каждой внутренней вершины v имеем $I(v) = v$, если $h(v)$ больше, чем $h(v')$ для любого потомка v' вершины v . В противном случае $I(v)$ полагается равным $I(v')$, где v' выбирается из потомков вершины v как вершина с наибольшим значением $h(I(v'))$. В результате каждая полоса образует путь вверх из вершин \mathcal{T} . И так как значения $h(I())$ не убывают вдоль пути наверх по \mathcal{T} , отсюда следует:

Лемма 8.6.1. Для любой вершины v вершина $I(v)$ — самая глубокая в полосе, содержащей v .

Эти факты иллюстрирует рис. 8.6.

Определение. Определим *голову* полосы как ее вершину, ближайшую к корню.

Например, на рис. 8.6 вершина 1 (0001) — голова полосы длины 3, вершина 2 (0010) — голова полосы длины 2, а каждая из оставшихся вершин (не принадлежащих этим двум полосам) является головой полосы, только из нее и состоящей.

Наконец, мы можем определить карту дерева.

Определение. *Карта дерева* — это отображение вершин \mathcal{T} в вершины полного двоичного дерева \mathcal{B} с глубиной $d = \lceil \log n \rceil - 1$. Именно, вершина v из \mathcal{T} отображается в вершину $I(v)$ из \mathcal{B} (напомним, что вершины \mathcal{B} именуются их путевыми номерами).

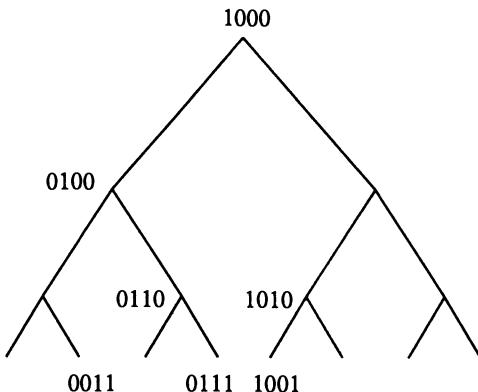


Рис. 8.7. Вершина v в \mathcal{B} нумеруется, если в \mathcal{T} есть вершина, отображаемая в v

Карта дерева определена, так как $I(v)$ является $(d - 1)$ -битовым числом и разные вершины \mathcal{B} названы разными $(d + 1)$ -битовыми числами. Все вершины из полосы в \mathcal{T} отображаются в одну и ту же вершину \mathcal{B} , но в общем случае не все вершины из \mathcal{B} имеют прообразы из \mathcal{T} . На рис. 8.7 представлено дерево \mathcal{B} для дерева \mathcal{T} с рис. 8.6. Вершина v в \mathcal{B} занумерована, если в \mathcal{T} имеется ее прообраз v .

8.7. Препроцессинг \mathcal{T} за линейное время

Теперь можно уточнить препроцессинг на дереве \mathcal{T} , выполняемый за линейное время.

Алгоритм препроцессинга для \mathcal{T}

begin

1. Найти в \mathcal{B} наименьшего общего предшественника b вершин $I(x)$ и $I(y)$. Просмотреть в нисходящем порядке вершины \mathcal{T} , назначая номера вершинам. Во время просмотра вычислить $h(v)$ для каждой вершины v . Для каждой вершины установить указатель на родительскую вершину в \mathcal{T} .

2. Используя алгоритм подъема снизу вверх, описанный ранее, вычислить $I(v)$ для каждого v . Для каждого k , такого что $I(v) = k$ для некоторой вершины v , положить $L(k)$ равным указателю на голову (*Leader*) полосы, содержащей вершину k . (Отметьте, что после этого шага голова полосы, содержащей произвольную вершину v , может быть получена за константное время: нужно вычислить $I(v)$ и затем взять $L(I(v))$.)
 {Это легко сделать при вычислении значений I . Вершина v идентифицируется как голова полосы, если значение I у ее родителя не равно $I(v)$.}
3. Пусть \mathcal{B} — полное двоичное дерево с вершинной глубиной $d = \lceil \log n \rceil - 1$. Отобразить каждую вершину v из \mathcal{T} в вершину $I(v)$ из \mathcal{B} .
 {Это отображение будет полезно потому, что оно сохраняет достаточно (хотя и не все) отношений предшествования в \mathcal{T} .}
 {Три приведенных шага составляют ядро препроцессинга, но нужен еще один шаг, более технический. Для каждой вершины v из \mathcal{T} мы хотим закодировать некоторую информацию о том, где в \mathcal{B} находятся предшественники вершин v , у которых есть прообразы. Эта информация накапливается на следующем шаге. Напомним, что $h(I(q))$ — высота в \mathcal{B} вершины $I(q)$, т. е. вершины, в которую отображается q .}
4. Для каждой вершины v из \mathcal{T} создать номер A_v (его размер в битах будет порядка $O(\log n)$). Бит $A_v(i)$ устанавливается равным 1 в том и только том случае, если вершина v имеет предшественника в \mathcal{T} , который отображается в высоту i в \mathcal{B} , т. е. в том и только том случае, если v имеет такого предшественника u , что $h(I(u)) = i$.

end.

Завершено описание препроцессинга \mathcal{T} и отображения \mathcal{T} в \mathcal{B} . Для проверки, что вы понимаете смысл A_v , убедитесь, что число единиц в двоичном разложении A_v равно числу различных полос, встретившихся на пути от корня до v . Числа A легко находятся в требующем линейного времени обходе \mathcal{T} после того, как станут известны все значения I : если v' — родитель v , то A_v получается копированием $A_{v'}$ с последующей установкой бита $A_v(i)$ в 1, если $h(I(v)) = i$ (последний шаг будет лишним, когда v и v' лежат в одной и той же полосе, но он не помешает). В качестве примера рассмотрите вершину 3 (0011) на рис. 8.6. $A_3 = 1101(13)$, так как 3 (0011) отображается в высоту 1, 2 (0010) — в высоту 3, а 1 (0001) — в высоту 4 дерева \mathcal{B} .

Что же делает это сумасшедшее отображение?

В конечном счете программистские детали этого отображения (препроцессинг) очень просты и станут еще проще (см. п. 8.9). Отображение требует лишь стандартных обходов дерева \mathcal{T} (простенькое упражнение в программировании, посильное второкурснику). Однако для большинства читателей интуитивно не ясно, что же точно делает это отображение, так как оно не взаимно однозначное. Ясно, что отношения наследования в \mathcal{T} сохраняются при отображении в \mathcal{B} не полностью (действительно, как им сохраняться, когда глубина \mathcal{T} может быть n , а глубина \mathcal{B} имеет границу $\Theta(\log n)$), но значительная часть информации о наследовании удерживается,

как показано в следующей ключевой лемме. Напомним, что по определению каждая вершина предшествует себе самой.

Лемма 8.7.1. *Если z предшествует x в \mathcal{T} , то $I(z)$ предшествует $I(x)$ в \mathcal{B} . Иначе говоря, если z предшествует x в \mathcal{T} , то либо z и x лежат в одной полосе, либо вершина $I(z)$ является собственным предшественником вершины $I(x)$ в \mathcal{B} .*

Рис. 8.6 и 8.7 иллюстрируют утверждение леммы.

Доказательство леммы 8.7.1. Доказательство тривиально, если $I(z) = I(x)$, поэтому предположим, что они не равны. Так как z предшествует x в \mathcal{T} , то $h(I(z)) > h(I(x))$ согласно определению I , но равенство возможно только при $I(z) = I(x)$. Тогда $h(I(z)) > h(I(x))$. Поскольку h задает высоту вершин в \mathcal{B} , $I(z)$ находится в \mathcal{B} на большей высоте, чем $I(x)$.

Пусть $h(I(z)) = i$. Мы утверждаем, что $I(z)$ и $I(x)$ совпадают во всех битах слева от i (напомним, что биты двоичного числа нумеруются справа). Если это не так, то пусть $k > i$ — самый левый бит, где $I(z)$ и $I(x)$ различны. Не умоляя общности, предположим, что $I(z)_k = 1$, а $I(x)_k = 0$. Так как k — это точка самого левого различия, то биты слева от k у этих чисел одинаковы, откуда $I(z) > I(x)$. Пусть теперь z предшествует x в \mathcal{T} , так что вершины $I(z)$ и $I(x)$ обе принадлежат поддереву z в \mathcal{T} . Более того, поскольку $I(z)$ и $I(x)$ — это номера вершин поддерева z дерева \mathcal{T} в нисходящей нумерации, то каждое число между $I(x)$ и $I(z)$ появляется в этом поддереве как номер одной из вершин. В частности, пусть N — число, у которого биты слева от позиции k такие же, как у $I(z)$ (или $I(x)$), дальше идет 1, а все остальные — нули. (Рис. 8.5 помогает понять эту ситуацию, только z и x играют роли, соответственно, u и w , а бит i в $I(z)$ неизвестен.) Тогда $I(x) < N < I(z)$, поэтому N также является вершиной поддерева z . Но $k > i$, так что $h(N) > h(I(z))$, что противоречит определению I . Итак, $I(z)$ и $I(x)$ должны совпадать во всех битах левее позиции i .

Теперь бит i содержит самую правую единицу в $I(z)$, так что биты слева от него описывают полный путь в \mathcal{B} до вершины $I(z)$. Эти биты слева от i -го образуют и начальную часть описания пути в \mathcal{B} до вершины $I(x)$, так как $I(x)$ имеет еще единицу справа от бита i . Совпадение показывает, что путь до вершины $I(x)$ в \mathcal{B} должен идти через вершину $I(z)$, а значит, $I(z)$ предшествует $I(x)$ в \mathcal{B} , и лемма доказана. \square

Обсудив препроцессинг \mathcal{T} и зная некоторые свойства отображения дерева, мы можем теперь описать, как выполняются ответы на запросы об *lca*.

8.8. Ответы на запросы об *lca* за константное время

Пусть x и v — две вершины из \mathcal{T} , а z — их *lca*. Предположим, что мы знаем в \mathcal{B} высоту вершины, в которую отображается z , т. е. мы знаем $h(I(z))$. Покажем, как, пользуясь только этой ограниченной информацией, найти z за константное время.

Теорема 8.8.1. *Пусть z обозначает *lca* от x и y в \mathcal{T} . Если мы знаем $h(I(z))$, то можем найти z в \mathcal{T} за константное время.*

Доказательство. Рассмотрим в \mathcal{T} полосу, содержащую z . Путь вверх по \mathcal{T} от x к z входит в эту полосу в некоторой вершине \bar{x} (возможно, в z) и затем движется вдоль полосы, пока не достигнет z . Точно так же, путь от y вверх до z входит в полосу в некоторой вершине \bar{y} и движется вдоль полосы до z . Отсюда следует,

что z совпадает либо с \bar{x} , либо с \bar{y} . Действительно, z — та из этих двух вершин, которая лежит выше, и по схеме нумерации $z = \bar{x}$ в том и только том случае, если $\bar{x} < \bar{y}$. Например, на рис. 8.6 при $x = 9$ (1001) и $y = 6$ (0110) имеем $\bar{x} = 8$ (1000) и $\bar{y} = z = 5$ (0101).

Из этого обсуждения видно, что при нахождении z нужно по $h(I(z))$ найти \bar{x} и \bar{y} , так как эти вершины определяют z . Мы объясним, как найти \bar{x} . Пусть $h(I(z)) = j$, так что высота $I(z)$ в \mathcal{B} равна j . По лемме 8.7.1 вершина x (которая лежит в поддереве z дерева \mathcal{T}) отображается в вершину $I(x)$ в поддереве вершины $I(z)$ дерева \mathcal{B} , так что если $h(I(x)) = j$, то x должна быть в той же полосе, что и z (т. е. $x = \bar{x}$), и все готово. Обратно, если $x = \bar{x}$, то $h(I(x))$ должно равняться j . Поэтому начиная с этого момента предположим, что $x \neq \bar{x}$.

Пусть w (возможно, равное x) обозначает вершину дерева \mathcal{T} на пути из z в x , лежащую как раз под полосой, содержащей z (и вне ее). Так как x не равно \bar{x} , то x не лежит в той полосе, где z , и w существует. Зная $h(I(z))$ (это предположено) и A_x (вычисленное предварительно), мы получим $h(I(w))$ и затем $I(w)$, w и \bar{x} .

Так как w принадлежит поддереву z дерева \mathcal{T} и не лежит в одной полосе с z , то w отображается в вершину \mathcal{B} с высотой, строго меньшей, чем у $I(z)$; это следует из леммы 8.7.1. Действительно, по этой лемме среди всех вершин на пути от x до z , не лежащих в одной полосе с z , w отображается в вершину с наибольшей высотой в \mathcal{B} . Таким образом, $h(I(w))$ (это и есть высота вершины, в которую отображается w) должно быть наибольшей позицией, меньшей чем j , из позиций, в которых A_x имеет бит 1. То есть мы можем получить $h(I(w))$ (даже не зная w), найдя самый значимый бит с 1 в двоичном представлении A_x в позиции, меньшей чем j . Это в предполагаемой вычислительной машине можно сделать за константное время (сначала установить все биты в 1, сдвинуть вправо на $d - j + 1$ позиций, логически умножить это число на A_x , затем найти самую левую единицу в результате.)

Пусть $h(I(w)) = k$. Найдем теперь $I(w)$. Вершина w либо равна x , либо является ее собственным предшественником в \mathcal{T} , так что либо $I(w) = I(x)$, либо вершина $I(w)$ является собственным предшественником вершины $I(x)$ в \mathcal{B} . Далее, по самой природе путевых номеров в \mathcal{B} , представляющих собой коды путей, числа $I(x)$ и $I(w)$ совпадают в битах левее k , в k -м бите $I(w)$ имеет 1 и нули правее. Таким образом, $I(w)$ можно получить из $I(x)$ (которое мы знаем) и k (которое мы получим, как выше, из $h(I(z))$ и A_x). Более того, $I(w)$ можно найти из $I(x)$ и $h(I(w))$, используя битовые операции с константным временем.

При заданном $I(w)$ мы можем найти w , потому что $w = L(I(w))$. Это значит, что w лежит сразу по выходе из полосы z , так что w должно быть головой своей полосы, и каждая вершина в \mathcal{T} указывает на голову своей полосы. Из w мы находим ее родителя \bar{x} за константное время. \square

В итоге, предположив, что знаем $h(I(z))$, мы можем найти вершину \bar{x} — ближайшего предшественника x в \mathcal{T} , который принадлежит той же полосе, что z . Аналогично мы находим \bar{y} . Тогда z равна либо \bar{x} , либо \bar{y} ; на самом деле, z равна той из этих двух вершин, у которой минимальный номер в \mathcal{T} . Конечно, мы еще должны объяснить, как найти $j = h(I(z))$.

Как найти высоту $I(z)$

Пусть b — наименьший общий предшественник $I(x)$ и $I(y)$ в \mathcal{B} . Так как \mathcal{B} — полное двоичное дерево, то b можно найти за константное время, как описано ранее. Пусть $h(b) = i$. Тогда $h(I(z))$ можно найти за константное время следующим образом.

Теорема 8.8.2. Пусть j — наименьшая позиция, не меньшая i и такая, что и A_x и A_y имеют 1 в позиции j . Тогда вершина $I(z)$ имеет высоту j в \mathcal{B} или, иными словами, $h(I(z)) = j$.

Доказательство. Пусть высота $I(z)$ в \mathcal{B} равна k . Покажем, что $k = j$. Так как z предшествует x и y , то и A_x и A_y имеют 1 в позиции k . Более того, так как $I(z)$ предшествует $I(x)$ и $I(y)$ в \mathcal{B} (по лемме 8.7.1), то $k \geq i$, и поэтому (по выбору j) $k \geq j$. Отсюда также следует, что существует позиция $j \geq i$, где и A_x и A_y имеют бит 1.

A_x содержит 1 в позиции j , и $j \geq i$, так что x имеет в \mathcal{T} предшественника x' , такого что $I(x')$ предшествует $I(x)$ в \mathcal{B} и высота $I(x')$ равна j . Отсюда вытекает, что $I(x')$ предшествует b . Аналогично, у вершины y в \mathcal{T} есть предшественник y' , такой что высота $I(y')$ равна j и $I(y')$ предшествует b в \mathcal{B} . Но если $I(x')$ и $I(y')$ имеют одинаковую высоту (j) и обе предшествуют вершине b , то $I(x') = I(y')$, что означает принадлежность x' и y' одной полосе. Значит, одна из этих вершин предшествует другой. Допустим, не умоляя общности, что x' предшествует y' в \mathcal{T} . Тогда x' является общим предшественником для x и y и предшествует z в \mathcal{T} . Следовательно, x' должно отображаться на высоту не ниже z в \mathcal{B} . Это значит, что $j \geq k$. Но неравенство $k \geq j$ уже установлено, так что, как и утверждалось, $k = j$, и теорема доказана. \square

Все этапы получения *lca* в \mathcal{T} теперь описаны, и каждый занимает только константное время. В итоге наименьший общий предшественник z любых двух вершин x и y в \mathcal{T} (предполагая, что z не равно ни x , ни y) может быть найден за константное время следующим методом.

Нахождение *lca* за константное время

begin

1. Найти в \mathcal{B} наименьшего общего предшественника b вершин $I(x)$ и $I(y)$.
 2. Найти наименьшую позицию j , не меньшую чем $h(b)$ и такую, что $A_x(j) = A_y(j) = 1$; тогда j будет $h(I(z))$.
 3. Найти вершину \bar{x} — ближайшую к x вершину из той же полосы, что и z (хотя мы и не знаем z), следующим образом:
 - 3a. Найти позицию l с крайней правой единицей в A_x .
 - 3b. Если $l = j$, то положить $\bar{x} = x$ (x и z лежат в \mathcal{T} в одной и той же полосе) и перейти к шагу 4.
 - 3c. В противном случае (когда $l < j$):
 - 3c. Найти позицию k самой левой единицы в A_x справа от позиции j . Сформировать число, состоящее из битов $I(x)$ слева от позиции k , затем 1 в позиции k , затем нули. (Это число будет $I(w)$, хотя мы еще не знаем w .) Найти вершину $L(I(w))$, которая должна быть вершиной w . Взять в качестве \bar{x} родителя вершины w в \mathcal{T} .
 4. Найти вершину \bar{y} , ближайшую к y в той же полосе, что и z , так же как в шаге 3.
 5. Если $\bar{x} < \bar{y}$, то положить z равной \bar{x} , в противном случае — равной \bar{y} .
- end.

8.9. Двоичное дерево нужно только из концептуальных соображений

Проницательный читатель заметит, что двоичное дерево \mathcal{B} можно целиком исключить из алгоритма, хотя оно существенно в изложении и доказательствах. Дерево \mathcal{B} совершенно не используется в шагах 3, 4 и 5 при получении z из $h(I(z))$. Однако оно применяется в шаге 1 для нахождения вершины b по $I(x)$ и $I(y)$. Но все, что нам реально нужно от b , — это $h(b)$ (шаг 2), а его можно получить, находя самый правый общий бит с 1 в $I(x)$ и $I(y)$. Таким образом, отображение из \mathcal{T} в \mathcal{B} важно только концептуально, в основном в целях полноты изложения.

В итоге после препроцессинга \mathcal{T} , когда заданы вершины x и y , алгоритм находит $i = h(b)$ (не находя сначала b), используя самый правый общий бит с 1 в $I(x)$ и $I(y)$. После этого он находит $j = h(I(z))$ из i и A_x и A_y , и, зная j , находит $z = lca(x, y)$. Несмотря на то что логика, стоящая за этим методом, трудна для понимания, программировать его очень просто.

8.10. Для пуристов: как избежать битовых операций

Мы предположили, что машина может выполнять некоторые побитовые операции за константное время. Многие из этих предположений вполне резонны для существующих машин, но для некоторых операций, таких как возможность найти самый правый бит с 1, предположение не кажется резонным. Можно ли избежать всех побитовых операций, выполняя алгоритм lca на стандартной модели памяти? Вспомним, что наша память может за константное время читать, писать, адресовать, складывать, вычитать, умножать и делить, причем только числа с $O(\log n)$ битами.

Идея заключается в том, чтобы во время препроцессинга \mathcal{T} создать таблицы размера $O(n)$, которые задают результаты необходимых побитовых операций. Битовые операции над *отдельным* числом с $O(\log n)$ битами проще всего. Сдвиг влево на $i = O(\log n)$ битов выполняется умножением на 2^i , что является допустимой операцией с константным временем, так как $2^i = O(n)$. Аналогично, сдвиг вправо выполняется делением. Теперь рассмотрим проблему нахождения крайней левой единицы. Построим таблицу из n элементов, храня в k -м элементе число $\lceil \log_2 n \rceil$. Первый элемент в ней равен 1, потом идут два элемента, равные 2, потом четыре элемента, равные 3, и т. д. Каждое такое число имеет $O(\log n)$ битов, так что таблица легко генерируется в стандартной модели за время $O(n)$. Создание таблицы для самой правой единицы немного сложнее, но может быть проведено аналогично. Для построения “битовых масок” нужна таблица даже меньшего размера, с $\lceil \log_2 n \rceil$ элементами. Мaska размера i состоит из i нулей в правой части из $\lceil \log_2 n \rceil$ и единиц во всех остальных позициях. i -я маска используется при нахождении самой правой единицы слева от позиции i . В качестве маски $\lceil \log_2 n \rceil$ принимается 0. Дальше каждая маска i получается из маски $i + 1$ делением ее на 2 (сдвигом маски вправо на одну позицию) и прибавлением $2^{\lceil \log_2 n \rceil - 1}$ (что добавляет единицу в самой левой позиции).

Представляется, что таблицы для *двуместных* битовых операций над $O(\log n)$ -битовыми числами, таких как XOR, AND и OR, построить труднее. Полная таблица для каждой из таких двуместных операций имеет размер n^2 , так как число возможных чисел с $\log n$ битами равно n . Таблицу размера n^2 за время $O(n)$ не создать.

Фокус в том, что каждая из необходимых операций выполняется побитово. Например, XOR для двух $\lceil \log n \rceil$ -битовых operandов можно найти, разбив каждый operand на две части, грубо говоря, по $\lceil \log n \rceil / 2$ бит каждая, дважды выполнить XOR и затем конкатенировать ответы (требуются некоторые простые подробности о том, как сделать это в стандартной модели). Так что достаточно для XOR построить таблицу для чисел только в $\lceil \lceil \log n \rceil / 2 \rceil$ бит. Но $2^{\lceil \log n \rceil / 2} = \sqrt{n}$, и таблица для них состоит всего из n элементов, а ее за время $O(n)$ построить можно. Что касается операции XOR, то она выражается через операции AND, OR и NOT, и таблицы для всех можно построить за время $O(n)$ (мы оставляем это как упражнение).

8.11. Упражнения

- Используя нисходящий обход, покажите, как построить путевые номера вершин \mathcal{B} за время, пропорциональное n — числу этих вершин. Будьте осторожны с ограничениями стандартной модели.
- Докажите, что путевые номера в \mathcal{B} равны номерам при *нисходящем обходе*.
- Был описан алгоритм *lca* для полного двоичного дерева, если *lca*(*i*, *j*) не совпадало ни с *i*, ни с *j*. В случае, когда *lca*(*i*, *j*) совпадает с *i* или с *j*, существует очень простой алгоритм с константным временем для определения *lca*(*i*, *j*). Идея в том, чтобы сначала занумеровать вершины двоичного дерева \mathcal{B} в нисходящем порядке и зафиксировать для каждой вершины *v* число вершин в ее поддереве (включая ее самое). Пусть *l*(*v*) — номер, данный *v*, а *s*(*v*) — число вершин в ее поддереве. Тогда вершина *i* предшествует вершине *j* в том и только том случае, если $l(i) \leq l(j) < l(i) + s(i)$.
Покажите, что это верно и что необходимый препроцессинг может быть сделан за время $O(n)$.
Продемонстрируйте применимость метода к любому дереву, а не только к полным двоичным деревьям.
- В случае полного двоичного дерева \mathcal{B} можно по-другому разобраться с ситуацией, когда *lca*(*i*, *j*) совпадает с *i* или *j*. Используя *h*(*i*) и *h*(*j*), мы можем определить, какая из вершин *i* и *j* выше в дереве (пусть это *i*) и сколько дуг в пути от корня до вершины *i*. Возьмем XOR двоичных представлений чисел *i* и *j* и найдем в нем, как раньше, крайнюю левую единицу, пусть она стоит в позиции *k* (считая слева). Вершина *i* предшествует *j* в том и только том случае, если *k* больше, чем число дуг в пути до вершины *i*. Обоснуйте этот метод.
- Объясните, почему в алгоритме *lca* для \mathcal{B} необходимо предполагать, что *lca*(*i*, *j*) не совпадает ни с *i*, ни с *j*. Что будет неправильно в алгоритме, если этот вопрос упустить, и какой случай будет проверяться неверно?
- Докажите, что высота любой вершины *k* в \mathcal{B} равна *h*(*k*).
- Напишите программу на Си для препроцессинга и выполнения запросов об *lca*. Испытайте программу на больших деревьях и хронометрируйте результаты.
- Предложите явный алгоритм для стандартной машины, который за время $O(n)$ строил бы таблицу, содержащую позицию самой правой единицы в каждом $\log_2 n$ -битовом числе. Помните, что запись для *i*-го числа должна быть в *i*-й позиции таблицы. Уточните детали построения таблиц операций AND, OR и NOT для чисел в $\lceil \log n \rceil / 2$ битов за время $O(n)$.

9. Может быть, резоннее предположить у стандартной машины константное время для операций сдвига влево и вправо, чем для умножения и деления. Покажите, как решить задачу *lca* за константное время с линейным временем подготовки при таких предположениях.
10. В доказательстве теоремы 8.8.1 мы показали, как найти $I(w)$ по $h(I(w))$ за константное время. Можем ли мы использовать ту же технику для нахождения $I(z)$ по $h(I(z))$? Если да, то почему бы не применить этот метод вместо привлечения вершин w , \bar{x} и \bar{y} ?
11. Алгоритм *lca* с константным временем труден для понимания, и читатель может поинтересоваться, не работают ли более простые идеи. Мы знаем, как найти *lca* за константное время в полном двоичном дереве после затрат $O(n)$ времени на препроцессинг. Допустим, мы отбросили предположение, что дерево полное. Таким образом, теперь \mathcal{T} является двоичным, но не обязательно полным. Считая, что d снова обозначает глубину \mathcal{T} , мы вычислим путевые номера длины $d + 1$, которые кодируют пути до вершин, и эти путевые номера снова позволят легко строить общего наименьшего предшественника. Может показаться, что даже в неполных двоичных деревьях этим простым способом легко найти *lca*, не прибегая к полному алгоритму *lca*. Либо обоснуйте этот алгоритм, либо покажите, почему он не позволит найти *lca* за константное время.

Если вы убедились, что предыдущий простой метод решает задачу *lca* за константное время для любого двоичного дерева, то рассмотрите возможность его применения к произвольным деревьям. Идея в том, чтобы использовать хорошо известную технику сведенияния произвольного дерева к двоичному, модифицируя каждую вершину, у которой больше двух детей, следующим образом. Пусть вершина v имеет детей v_1, v_2, \dots, v_k . Заменим детей v на двух детей v_1 и v^* и сделаем вершины v_2, \dots, v_k детьми v^* . Будем повторять это действие, пока у каждого исходного сына v_i вершины v не останется только по одному брату, и создадим указатель из каждой новой вершины v^* , созданной в этом процессе, на v . Как изменится при этом процессе число вершин? Как будут соотноситься *lca* двух вершин в этом новом двоичном дереве и исходном дереве? Теперь в предположении, что при решении задачи *lca* за константное время для любого двоичного дерева могут быть использованы путевые метки длины $d + 1$, ответьте, не приведет ли это преобразование дерева к алгоритму поиска *lca* за константное время для любого дерева?

12. Более простой (но более медленный) алгоритм *lca*. В п. 8.4.1 мы упоминали, что если разрешено препроцессное время $\Theta(n \log n)$ и память $\Theta(n \log n)$ на все время работы, то возможен (концептуально) более простой метод нахождения *lca* за константное время. Во многих приложениях граница $\Theta(n \log n)$ вполне приемлема, так как она не намного хуже, чем полученная в тексте граница $O(n)$. Здесь мы набросаем идею этого метода. Вашей задачей будет восполнить детали и доказать корректность метода.

Сначала мы сведем общую задачу *lca* к задаче выбора наименьшего числа в интервале фиксированного списка чисел.

Сведение *lca* к задаче о списке

Шаг 1. Выполнить нисходящий обход дерева \mathcal{T} , помечая вершины в нисходящем порядке и создавая мультисписок L этих вершин в порядке их посещения. (Для каждой вершины v , отличной от корня, число вхождений v в L равно степени v .) Единственное свойство нисходящей нумерации, которое нам понадобится, заключается в том, что числа, сопоставляемые вершине, меньше, чем числа, сопоставляемые ее потомкам. Начиная с этого места мы ссылаемся на вершины только по их номерам.

Например, список для дерева на рис. 8.1 (с. 228) таков:

$$\{1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 7, 5, 8, 9, 8, 10, 8, 5, 1\}.$$

Отметим, что если \mathcal{T} имеет n вершин, то L имеет $O(n)$ элементов.

Шаг 2. Получить lca для любых двух вершин x и y можно следующим образом. Найти любые вхождения x и y в L ; эти элементы ограничивают собой интервал I списка L . Найти наименьшее число в I ; оно и будет искомым $lca(x, y)$.

Например, если $x = 6$ и $y = 9$, то они определяют интервал $I = \{6, 5, 7, 5, 8, 9\}$, минимум в нем дает $lca(6, 9) = 5$.

Это конец сведений. Теперь первое упражнение.

- a. Игнорируя временну́ю сложность, докажите, что в общем случае можно получить lca двух вершин так, как это описано выше в двух шагах.

Продолжим описание метода. Дальнейшие упражнения еще последуют.

После описанного сведений выполнение запроса об lca становится задачей нахождения наименьшего числа в интервале I фиксированного списка L из $O(n)$ целых чисел. Пусть m обозначает точный размер L . Чтобы иметь возможность разрешить каждый запрос об lca за константное время, мы сначала должны выполнить за время $O(m \log m)$ препроцессинг списка L . Для удобства предположим, что m является степенью 2.

Препроцессинг L

Шаг 1. Построить полное двоичное дерево B с m листьями и перенумеровать листья слева направо (как при фронтальном обходе). Для каждого i от 1 до m вписать i -й элемент L в лист i .

Шаг 2. Для произвольной внутренней вершины v из B пусть B_v обозначает поддерево B с корнем в v и пусть $L_v = \{n_1, n_2, \dots, n_z\}$ — упорядоченный список элементов L , записанных в листьях B_v в том порядке, в котором они появляются в B . Создать два списка, $Pmin(v)$ и $Smin(v)$, для каждой внутренней вершины v . Каждый лист будет иметь размер, равный числу листьев в B_v . k -й элемент листа $Pmin(v)$ будет наименьшим числом в списке $\{n_1, n_2, \dots, n_k\}$. Таким образом, k -й элемент $Pmin(v)$ будет наименьшим номером в префиксе списка L_v , кончающемся позицией k . Аналогично, k -й элемент списка $Smin(v)$ — это наименьшее число в суффиксе L_v , начинающемся в позиции k . На этом подготовка заканчивается, далее следуют упражнения.

- b. Докажите, что полный размер всех списков $Pmin$ и $Smin$ имеет порядок $O(m \log m)$. и покажите, как их можно создать с такой оценкой времени.

После препроцессинга, занимающего время $O(m \log m)$, наименьшее число в любом интервале I может быть найдено за константное время. Итак, пусть интервал I в L имеет концы l и r ; напомним, что они соответствуют листьям B . Для организации поиска наименьшего числа в I сначала найдем $lca(l, r)$, пусть это вершина v . Пусть l и r — дети v в B , соответственно, левый и правый. Наименьшее число в I может быть найдено одним обращением к списку $Smin(v')$, одним обращением к $Pmin(v'')$ и одним дополнительным сравнением.

- c. Приведите все подробности нахождения наименьшего числа в I и объясните, почему нам достаточно константного времени.

13. При улучшении метода, развитого в упражнении 12, границу $\Theta(m \log m)$ (по времени и памяти) для препроцессинга можно свести к $\Theta(m \log \log m)$, сохраняя время обработки любого запроса об *lca* константным. (Нужно очень большое значение m , чтобы ощутить различие между $O(m)$ и $\Theta(m \log \log m)$!) Идея в том, чтобы разделить список L на $m/\log m$ блоков размером $\log m$, и затем отдельно обрабатывать каждый блок, как в упражнении 12. Дальше, вычислив минимальное значение в каждом блоке, следует собрать эти $m/\log m$ значений в упорядоченный список L_{min} и подготовить L_{min} , как в упражнении 12.

- a. Показать, что приведенная обработка требует времени и памяти $\Theta(m \log \log m)$.

Теперь набросаем схему обработки запроса в этом более быстром методе. Задав интервал I с начальной и конечной позициями l и r , ищем наименьшее число в I следующим образом. Если l и r находятся в одном и том же блоке, то действуем, как в упражнении 12. Если они — в смежных блоках, то находим минимальное число в интервале от l до конца l -го блока, минимальное число от начала r -го блока до r и берем минимум из двух чисел. Если l и r лежат в несмежных блоках, то делаем то же самое, но еще используем L_{min} , чтобы найти минимум в блоках, полностью лежащих строго между l и r . Наименьшее число в I будет минимумом из этих трех чисел.

- b. Дайте подробное описание метода обработки запроса и обоснуйте, что он требует только константного времени.

14. Можно ли описанную выше реализацию с подготовительным временем $O(m \log \log m)$ достичь до времени $O(m \log \log \log m)$? Можно ли продолжить эти улучшения до произвольного числа логарифмов?

Дополнительные приложения суффиксных деревьев

Благодаря возможности выполнять запросы о наименьшем общем предшественнике за константное время, суффиксные деревья можно применять для решения многих других задач о строках. Многие из этих задач возникают при переходе от постановок с точным совпадением к *неточному*, или приближенному, совпадению (совпадению, в котором дозволяется несколько ошибок). Эта глава иллюстрирует такой переход несколькими примерами.

9.1. Наибольшее общее продолжение: мост к неточному совпадению

Задача о наибольшем общем продолжении возникает как вспомогательная во многих классических строковых алгоритмах. Она лежит в основе почти всех приложений этой главы — всех, кроме последнего, а также *алгоритма k-разности*, рассматриваемого в п. 12.2.

Задача о наибольшем общем продолжении. Две строки, S_1 и S_2 , общей длины n задаются на фазе препроцессинга. Затем указывается *длинная* последовательность пар индексов. Для каждой такой пары (i, j) требуется найти длину наибольшей подстроки S_1 , начинающейся в позиции i и совпадающей с подстрокой S_2 , начинающейся в позиции j . Таким образом, мы должны найти длину наибольшего общего префикса i -го суффикса строки S_1 и j -го суффикса строки S_2 (рис. 9.1).

Конечно, каждый раз, когда задается пара индексов, наибольшее общее продолжение можно найти прямым поиском за время, пропорциональное длине совпадения.

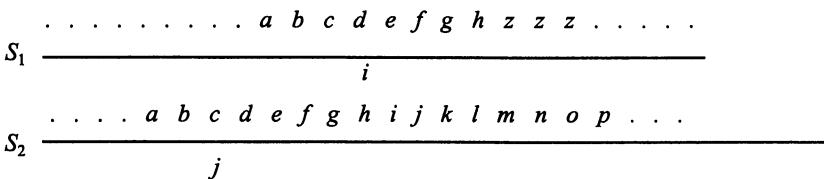


Рис. 9.1. Наибольшее общее продолжение пары (i, j) имеет длину 8.

Совпадает подстрока $abcdefgh$

Но цель в том, чтобы вычислять каждое продолжение за константное время, независимо от длины совпадения. Более того, было бы нечестно разрешить для препроцессинга S_1 и S_2 больше чем линейное время.

Чтобы оценить мощь суффиксных деревьев в сочетании с константным временем ответа на запросы об lca , читателю рекомендуется сначала попробовать решить задачу о наибольшем общем продолжении без этих двух средств.

9.1.1. Решение за линейное время

Решение задачи о наибольшем общем продолжении сначала строит обобщенное суффиксное дерево \mathcal{T} для S_1 и S_2 , а затем готовит \mathcal{T} для ответов на запросы об lca за константное время. Во время этой подготовки вычисляются также строевые глубины всех вершин дерева \mathcal{T} . Построение и подготовка \mathcal{T} требуют времени $O(n)$.

По заданной паре (i, j) алгоритм находит общего наименьшего предшественника v тех листьев \mathcal{T} , которые отвечают суффиксам i в S_1 и j в S_2 . Здесь важнее всего то, что строка, помечающая путь до v , в точности совпадает с наибольшей подстрокой S_1 , начинающейся в i , и с подстрокой S_2 , начинающейся в j . Следовательно, строковая глубина v равна длине *наибольшего общего продолжения*, так что на любой запрос о наибольшем общем продолжении можно ответить за константное время.

9.1.2. Поиск наибольшего общего продолжения, эффективный по памяти

Когда S_2 существенно меньше, чем S_1 , мы можем отказаться от построения обобщенного суффиксного дерева для S_1 и S_2 . Строя только суффиксное дерево для меньшей из двух строк, можно существенно сэкономить память. Однако будет ли задача о наибольшем общем продолжении решаться столь же эффективно при использовании только этого уменьшенного дерева? Ответ: да; решение достигается с помощью статистики совпадений.

Напомним (п. 7.8.1), что статистика совпадений $m(i)$ есть длина наибольшей подстроки S_1 , начинающейся в позиции i и совпадающей с подстрокой, начинающейся в некоторой позиции S_2 , и что $p(i)$ — это одна из таких начальных позиций в S_2 . В пп. 7.8.1 и 7.8.3 мы показали, как вычислить $m(i)$ и $p(i)$ для каждого i за полное время $O(|S_1|)$, используя только суффиксное дерево для S_2 и копию S_1 .

Запрос об общем продолжении для любой пары (i, j) разрешается нахождением сначала $v = lca(p(i), j)$ для листьев суффиксного дерева для S_2 . После этого длина наибольшего общего продолжения (i, j) находится как минимум из $t(i)$ и строковой глубины вершины v . Доказательство непосредственно и оставляется читателю. Так как любое вычисление lca требует только константного времени, мы получаем следующую теорему.

Теорема 9.1.1. *После препроцессинга, занимающего линейное время, на любой запрос о наибольшем общем продолжении можно ответить за константное время, используя лишь память, необходимую для суффиксного дерева строки S_2 (наименьшей из двух строк), и $2|S_1|$ слов для значений $t(i)$ и $p(i)$.*

Эффективное по памяти решение задачи о наибольшем общем продолжении обычно влечет за собой и эффективное по памяти решение различных ее приложений. Это не всегда явно упоминается в каждом из приводимых ниже приложений, так что подробности оставляются читателю.

9.2. Нахождение всех максимальных палиндромов за линейное время

Определение. Подстрока S' строки S , имеющая четную длину, называется *максимальным палиндромом радиуса k* , если S' читается одинаково в обоих направлениях на k символов, начиная с середины S' , а на любое $k' > k$ — уже не одинаково. Максимальный палиндром S' нечетной длины определяется аналогично после исключения центрального символа S' .

Например, если $S = aabactgaaccaaat$, то и aba , и $aaccaa$ — максимальные палиндромы радиусов 1 и 3 соответственно, а каждое вхождение aa — это максимальный палиндром радиуса 1.

Определение. Стока называется *максимальным палиндромом*, если она является максимальным палиндромом радиуса k для какого-либо k .

Например, в строке $cabaabad$ максимальные палиндромы и aba , и $abaaba$. Любой палиндром содержится в каком-либо максимальном палиндроме с той же средней точкой, так что максимальные палиндромы могут служить для компактного представления множества всех палиндромов. Более того, в большинстве приложений интересны именно максимальные палиндромы.

Задача о палиндроме. Пусть задана строка S длины n . Требуется найти все максимальные палиндромы в S .

9.2.1. Решение за линейное время

Мы объясним, как найти все максимальные палиндромы четной длины за линейное время $O(n)$, — это довольно серьезная задача. Для нахождения максимальных палиндромов нечетной длины можно будет слегка модифицировать алгоритм для четного случая.

Пусть S' — перевернутая строка S . Предположим, что в S имеется максимальный палиндром четной длины, середина которого находится сразу после символа q . Пусть k обозначает длину этого палиндрома. Тогда строка длины k , начинающаяся в позиции $q + 1$ строки S , должна совпадать со строкой, начинающейся в позиции $n - q + 1$ строки S' . Так как наш палиндром максимальен, то следующие символы (в позициях $q + k + 1$ и $n - q + k + 1$ соответственно) не совпадают. Отсюда следует, что k равно длине наибольшего общего продолжения для позиций $q + 1$ в S и $n - q + 1$ в S' . Итак, для любой фиксированной позиции q длина максимального палиндрома (если такой существует) со средней точкой в q может быть вычислена за константное время.

Это приводит к следующему простому методу с линейным временем для нахождения в S всех максимальных палиндромов четной длины:

1. За линейное время создать из S перевернутую строку S' и обработать обе строки так, чтобы любой запрос о наибольшем общем продолжении разрешался за константное время.
2. Для каждого q от 1 до $n - 1$ разрешить запрос о наибольшем общем продолжении для пары индексов $(q + 1, n - q + 1)$ в S и S' соответственно. Если это продолжение имеет непустую длину k , то имеется максимальный палиндром радиуса k с центром в q .

Получился метод со временем $O(n)$, так как именно такое время требуется на подготовку суффиксного дерева, и каждый из $O(n)$ запросов о продолжении разрешается за константное время. В итоге мы имеем:

Теорема 9.2.1. *Все максимальные палиндромы четной длины в строке можно найти за линейное время.*

9.2.2. Комплементарные и разделенные палиндромы

Палиндромы вкратце рассматривались в п. 7.11.1, во время общего обсуждения повторяющихся структур в биологических строках. Там упоминалось, что в структуре ДНК (или РНК) представляют интерес *комплементарные палиндромы*. Это означает, что две половины подстроки образуют палиндром (в нормальном смысле этого слова), только если символы одной половины переводятся в комплементарные им символы, такие как *A* и *T* (или *U* в случае РНК), а также *C* и *G*. Например, *ATTAGCTAAT* — это комплементарный палиндром.

Задачу нахождения в строке всех комплементарных палиндромов можно также решить за линейное время. Пусть $c(S')$ — комплементация строки S' (т.е. строка, в которой каждый символ заменен комплементарным к нему). Нужно действовать, как в задаче о палиндроме, только использовать $c(S')$ вместо S' .

Другой вариант задачи о палиндроме, характерный для биологических последовательностей, получается при снятии требования смежности двух половинок палиндрома (обычного или комплементарного). Такая структура называется *разделенным палиндромом*, хотя в биологической литературе различие между разделенными и неразделенными палиндромами часто расплывчато. Задача нахождения всех разделенных палиндромов совпадает с задачей нахождения всех инвертированных (обращенных) повторов (см. п. 7.12), и следовательно, она сложнее, чем поиск палиндромов. Однако если задать фиксированную границу разрешенного расстояния между

половинками, то опять-таки все разделенные палиндромы можно найти за линейное время. Здесь прямо привлекается задача о наибольшем общем продолжении, детали этого использования оставляются читателю.

Еще один вариант задачи о палиндроме называется задачей о палиндроме с точностью до k (k -mismatch palindrome problem). Мы рассмотрим его позже, после задачи о поиске совпадения с ограниченным числом несовпадений.

9.3. Точное совпадение с джокерами

Напомним рассмотренную в пп. 4.3.2 и 3.5.2 задачу о поиске всех вхождений образца P в текст T , а именно случай, когда в обеих строках допустимы джокеры. С этой задачей трудно справиться методами Кнута–Морриса–Пратта и Бойера–Мура, хотя метод быстрого преобразования Фурье, метод счета совпадений, может быть к этой задаче приспособлен. Используя приведенный метод решения задачи о наибольшем общем продолжении, в случае если в обеих строках располагается в общей сложности k джокеров, мы можем найти все вхождения P в T (разрешая джокеру совпадать с любым *отдельным* символом) за время $O(km)$, где $m = |T| \geq n = |P|$.

Если описывать алгоритм на высоком уровне, то в нем просматриваются все i от 1 до $m - n + 1$, и для каждого i проверяется, не входит ли P в T , начиная с позиции i . Джокер трактуется как дополнительный символ алфавита. Идея метода заключается в том, чтобы выровнять левый конец P по позиции i в T и затем двигаться по обеим строкам слева направо, последовательно выполняя запросы о наибольшем общем продолжении и проверяя, что каждое несовпадение происходит в позиции, содержащей джокер. После препроцессинга со временем $O(n + m)$ (для запросов о наибольшем общем продолжении) алгоритм требует только $O(k)$ времени для любого фиксированного i . Таким образом, всего требуется время $O(km)$. В приводимом ниже алгоритме уточняются подробности одного шага при фиксированной позиции i в T .

Проверка совпадения в случае метасимвола

`begin`

1. Положить $j := 1$, $i' := i$.
2. Вычислить длину l наибольшего общего продолжения, начиная с позиций j в P и i' в T .
3. Если $j + l = n + 1$, то P входит в T , начиная с i ; стоп.
4. Проверить, не стоит ли метасимвол в позиции $j + l$ строки P или позиции $i' + l$ строки T . Если да, то положить $j := j + l + 1$, $i' := i' + l + 1$, и перейти к шагу 2. В противном случае P не входит в T , начиная с i ; стоп.

`end.`

Для этого метода требуется память $O(n + m)$, поскольку он использует суффиксное дерево для двух строк. Однако, как уточняется в теореме 9.1.1, нужны только суффиксное дерево для P и статистика совпадений для T (хотя еще надо хранить исходные строки). Так как $m > n$, то получаем:

Теорема 9.3.1. *Задача о точных совпадениях с k джокерами, расположеннымными в двух строках, может быть решена за время $O(km)$ с расходом памяти $O(m)$.*

9.4. Задача о k несовпадениях

Общая задача о неточном, или приближенном, совпадении (совпадение, при котором разрешено несколько ошибок) будет подробно рассмотрена в части III этой книги (в п. 12.2), где основной темой будет техника динамического программирования. Но динамическое программирование не всегда необходимо, и сейчас мы имеем все средства для решения одной из классических “тестовых” задач приближенного совпадения: задачи о k несовпадениях.

Определение. Пусть заданы образец P , текст T и фиксированное число k , значение которого не зависит от длин P и T . Совпадением с P с точностью до k называется подстрока T длины $|P|$, которая совпадает с P по меньшей мере в $|P| - k$ символах. Значит, она совпадает с P при не более чем k несовпадениях.

Отметим, что это определение не допускает какого-либо включения или удаления символов, а лишь совпадения и несовпадения. В дальнейшем, в п. 12.2, мы обсудим проблемы с ограниченным числом исправлений, когда включение и удаление символов будут допускаться.

Задача о k несовпадениях — это задача о поиске всех совпадений с точностью до k строки P в T .

Например, если $P = bend$, $T = abentbananaend$ и $k = 2$, то T содержит три совпадения с P с точностью до k : P совпадает с подстрокой *bent* с одним нарушением, с *banana* — с двумя нарушениями и с *aend* — с одним нарушением.

Приложения этой задачи в молекулярной биологии вместе с более общими задачами, где ограничивается число отклонений, будут рассматриваться в п. 12.2.2. Задача о совпадении с точностью до k является частным случаем задачи о подсчете совпадений, рассмотренной в п. 4.3, и обсуждавшийся там подход здесь вполне применим. Но поскольку k — это фиксированное число, не зависящее от длин P и T , то удается получить более быстрое решение. В частности, Ландау и Вишкин [287] и Майерс [341] были первыми, кто продемонстрировал решение с временем $O(km)$, где P и T имели, соответственно, длины n и $m > n$. Значение k никогда не может превосходить n , но важность результата $O(km)$ объясняется приложениями, в которых k очень мало в сравнении с n .

9.4.1. Решение

Идея в основном совпадает с идеей поиска совпадений в случае метасимволов (хотя смысл k в этих задачах различен). Для любой позиции i строки T мы определяем, не начинается ли совпадение с точностью до k в позиции i . Это можно сделать за время $O(k)$, просто выполняя k запросов (константное время) о наибольшем общем продолжении. Если эти продолжения доходят до конца P , то P совпадает с подстрокой, начинающейся в i , с не более чем k несовпадениями. Если же продолжения не достигают конца P , то требуется больше чем k несовпадений. В любом случае для всякого i обрабатывается не больше k запросов о продолжении, и времени $O(k)$ оказывается достаточно для определения того, не начинается ли с i подстрока с не более чем k несовпадениями с P . А на все возможные позиции в T метод требует времени не более $O(km)$.

Проверка k несовпадений**begin**

1. Положить $j := 1$, $i' := i$ и $count := 0$.
2. Вычислить длину l наибольшего общего продолжения, начинающегося в позициях j строки P и i' строки T .
3. Если $j + l = n + 1$, то найдено совпадение с точностью до k строки P и подстроки T , начинающейся в i (на самом деле произошло только $count$ несовпадений); стоп.
4. Если $count \leq k$, то увеличить $count$ на 1, положить $j := j + l + 1$, $i' := i' + l + 1$ и перейти к шагу 2.

Если $count = k + 1$, то совпадения с точностью до k у строки P в позиции i нет; стоп.

end.

Заметьте, что для этого решения нужна память всего $O(n + m)$ и метод можно реализовать, используя суффиксное дерево только для маленькой строки P .

Нужно отметить, что существует иной практический подход к задаче совпадения с точностью до k , он основан на суффиксных деревьях и используется при поиске в биологических базах данных [320]. Его идея заключается в генерировании всех строк P' , которые могут быть получены из P изменением k символов P , и в дальнейшем поиске каждой такой строки P' в суффиксном дереве для T . При использовании суффиксного дерева поиск P' требует времени, пропорционального длине P' (и может выполняться предельно быстро), так что этот подход может победить, если k и размер алфавита относительно малы.

9.5. Приблизительные палиндромы и повторы

Мы уже говорили (п. 7.11.1) о важности палиндромов в молекулярной биологии. Из этого обсуждения в основном было видно, зачем нужно заниматься задачей о палиндроме. Но в биологических приложениях две части “палиндрома” редко оказываются одинаковы. Это побуждает к рассмотрению задачи о палиндроме с точностью до k .

Определение. *Палиндромом с точностью k называется подстрока, которая становится палиндромом после изменения не более k символов.* Например, *axabbccaa* — это палиндром с точностью 2.

Согласно этому определению обычный палиндром — это палиндром с точностью 0. Теперь легко поупражняться в подробностях метода с трудоемкостью $O(kn)$ для нахождения всех палиндромов с точностью k в строке длины n . Мы оставим это читателю и обратимся к более сложной задаче нахождения *тандемных повторов*.

Определение. *Тандемный повтор α* — это строка, которая может быть записана как $\beta\beta$, где β — подстрока.

Каждый tandemный повтор задается начальной позицией повтора и длиной подстроки β . Это определение не требует, чтобы β была максимальной длины. Например,

в строке $xababababy$ есть шесть тандемных повторов. Два из них начинаются в позиции два: $abab$ и $abababab$. В первом случае $\beta = ab$, а во втором $\beta = abab$.

Легко видеть, что с помощью запросов о наибольшем общем продолжении все тандемные повторы можно найти за время $O(n^2)$ — просто предположив, что начальная позиция тандема находится в i , а средняя позиция — в j , и выполнив запрос о наибольшем общем продолжении из i и j . Если это продолжение из i достигает j или простирается дальше, то нашелся тандемный повтор длины $2(j - i + 1)$, начинающийся в позиции i . Имеется $\Theta(n^2)$ выборов i и j , что дает оценку времени $O(n^2)$.

Определение. Тандемным повтором точности k называется подстрока, которая становится тандемным повтором после изменения не более k символов. Например, $axabaybb$ — тандемный повтор точности 2.

И здесь все тандемные повторы точности k можно найти за время $O(kn^2)$, подробности оставляются читателю. Ниже мы приведем метод решения этой задачи за время $O(kn \log(n/k))$. А то, что мы имеем сейчас, формулируется так:

Теорема 9.5.1. Все тандемные повторы в S , в которых два экземпляра отличаются не более чем в k местах, можно найти за время $O(kn^2)$. Обычно k фиксировано, и эта оценка трактуется как $O(n^2)$.

9.6. Более быстрые методы для тандемных повторов

Полное число тандемных повторов, точных и с точностью k (даже для фиксированного k), может расти как $\Theta(n^2)$ (что видно из случая, когда все n символов одинаковы). Поэтому в задаче нахождения всех тандемных повторов оценка наихудшего случая не может быть меньше, чем $O(n^2)$. Но метод со временем работы, зависящим от числа тандемных повторов в строке, вполне возможен для обоих вариантов задачи. В упражнениях 56, 57 и 58 главы 7 был предложен иной подход, в котором распознаются только максимальные примитивные тандемные ряды.

Ландау и Шмидт [288] разработали метод нахождения всех тандемных повторов с точностью k за время $O(kn \log(n/k) + z)$, где z — число таких повторов в строке S . Величина z может достигать $\Theta(n^2)$, но на практике z в сравнении с n^2 мало, так что снижение оценки до $O(kn \log(n/k) + z)$ существенно. Отметим, что мы по-прежнему ищем все тандемные повторы, но время счета зависит от числа имеющихся повторов, а не от его максимальной границы.

Мы объясним этот метод, приспособив его сначала к поиску всех точных тандемных повторов за время $O(n \log n + z)$, где z — полное число тандемных повторов в S . Эта граница для случая точных повторов была впервые получена в работе Мэйна и Лоренца [307], которые использовали похожую идею, но обошлись без суффиксных деревьев. Их подход используется в упражнении 8.

Решение Ландау–Шмидта — это рекурсивный алгоритм типа “разделяй и властвуй”, который основан на возможности вычислять запросы о наибольшем общем продолжении за константное время. Пусть $h = \lfloor n/2 \rfloor$. В укрупненном масштабе метод Ландау–Шмидта делит задачу нахождения всех тандемных повторов на четыре

подзадачи:

1. Найти все тандемные повторы, содержащиеся целиком в первой половине S (до позиции h включительно).
2. Найти все тандемные повторы, содержащиеся целиком во второй половине S (после позиции h).
3. Найти все тандемные повторы, в которых первая копия покрывает позицию h .
4. Найти все тандемные повторы, в которых вторая копия покрывает позицию h .

Ясно, что никакой тандемный повтор не может встретиться больше чем в одной из этих подзадач. Первые две подзадачи решаются рекурсивно применением того же алгоритма Ландау–Шмидта. Две последние подзадачи симметричны, поэтому мы рассмотрим только третью подзадачу, и алгоритм для нее определит весь алгоритм.

Алгоритм для задачи 3

Мы хотим найти все тандемные повторы, у которых первая копия покрывает позицию h (но не обязательно начинается в ней). Идея алгоритма заключается в следующем. Для любого фиксированного l можно за константное время проверить, существует ли такой тандемный повтор длины $2l$, у которого первая копия покрывает позицию h . Применение этого теста ко всем допустимым значениям l означает, что за время $O(n)$ мы можем найти все длины таких тандемных повторов. Более того, для каждой такой длины мы можем перечислить начальные точки этих тандемных повторов за время, пропорциональное их числу. Ниже описано, как проверить число l .

`begin`

1. Положить $q := h + l$.
2. Вычислить наибольшее общее продолжение (в прямом направлении из позиций h и q . Пусть l_1 — длина этого продолжения).
3. Вычислить наибольшее общее продолжение (в обратном направлении из позиций $h - 1$ и $q - 1$. Пусть l_2 — длина этого продолжения).
4. Тандемный повтор длины $2l$, первая копия которого покрывает позицию h , существует в том и только том случае, если $l_1 + l_2 \geq l$ и обе длины l_1 и l_2 положительны. Более того, если такой тандемный повтор длины $2l$ существует, то он может начинаться в любой позиции от $\max(h - l_2, h - l + 1)$ до $\min(h + l_1 - l, h)$ включительно. Вторая копия повтора начинается в l позициях вправо. Вывести каждую из этих начальных позиций вместе с длиной $2l$ (рис. 9.2).

`end.`

Для решения главной части подзадачи 3 (нахождение всех тандемных повторов, у которых первая копия покрывает позицию h), нужно просто выполнить приведенный алгоритм для каждого l от 1 до h .

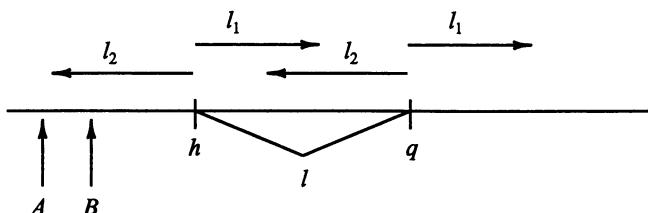


Рис. 9.2. Любая позиция между A и B включительно может быть начальной точкой тандемного повтора длины $2l$. Как указывается в шаге 4, длины l_1 и l_2 обе положительны, поэтому подинтервал этих начальных точек определяет тандемные повторы, в которых первая копия покрывает h

Лемма 9.6.1. Приведенный метод корректно решает подзадачу 3 для фиксированного h . Это значит, что он находит все тандемные повторы, у которых первая копия покрывает позицию h . Для фиксированного h его время работы равно $O(n/2) + z_h$, где z_h — число таких тандемных повторов.

Доказательство. Предположим сначала, что имеется тандемный повтор, у которого первая копия покрывает позицию h , и его длина равна, скажем, $2l$. Это означает, что позиция $q = h + l$ во второй копии соответствует позиции h в первой копии. Следовательно, для того чтобы обеспечить совпадение суффиксов в этих копиях, некоторая подстрока, начинающаяся в h , должна совпадать с подстрокой, начинающейся в q . Такая подстрока может иметь длину не более l_1 . Аналогично, должна быть подстрока, кончающаяся в $h - 1$, которая совпадает с подстрокой, кончающейся в $q - 1$, для того чтобы обеспечить совпадение префиксов. Длина этой подстроки не должна превосходить l_2 . Так как все символы между h и q содержатся в одной из двух копий, то $l_1 - l_2 \geq l$. Обратно, рассуждая в основном так же, если $l_1 + l_2 \leq l$ и l_1 и l_2 положительны, то можно найти тандемный повтор длины $2l$, у которого первая копия покрывает h . Необходимое и достаточное условие существования такого тандема теперь доказано.

Обратное доказательство, что все начальные позиции попадают в названный диапазон, аналогично и оставляется читателю.

Для подсчета времени сначала отметим, что при фиксированном h методу нужно константное время на один выбор l для того, чтобы выполнить общие запросы о продолжении, и, таким образом, время $O(n/2)$ на все такие запросы. Для любого фиксированного l методу нужно константное время на обнаруживаемый тандем, и никакой тандем не обнаруживается дважды, так как у всех обнаруживаемых тандемов длины $2l$ стартовые точки различны. Поскольку сообщение о каждом повторе состоит из начальной точки и длины, то алгоритм никогда не сообщает о тандемном повторе дважды. Следовательно, время, расходуемое на вывод информации о тандемных повторах, пропорционально z_h — числу повторов, у которых первая копия покрывает позицию h . \square

Теорема 9.6.1. Каждый тандемный повтор в S находится решением подзадач 1–4 и при этом только единожды. Время работы алгоритма равно $O(n \log n + z)$, где z — полное число тандемных повторов в S .

Доказательство. То, что алгоритм находит все тандемные повторы, прямо следует из факта, что каждый тандем имеет форму, предусмотренную одной из подзадач 1–4. Чтобы показать, что никакой тандемный повтор не выводится дважды, вспомним, что для $h = n/2$ никакой из тандемов не принимает формы, предусмотренной более чем одной из четырех подзадач. Рекурсивно это выполняется для подзадач 1 и 2. Далее, при доказательстве леммы 9.6.1 мы установили, что решение подзадачи 3 (а также 4) никогда не получает один и тот же тандем дважды. Следовательно, при полном решении четырех подзадач никакой тандемный повтор не выводится дважды. Отсюда следует, что время вывода всех тандемных повторов равно $O(z)$.

Чтобы завершить анализ, рассмотрим время, расходуемое на запросы о продолжении. Это время пропорционально числу выполняемых запросов. Пусть $T(n)$ — число запросов при длине строки n . Тогда $T(n) = 2T(n/2) + 2n$ и $T(n) = O(n \log n)$, как и утверждалось. \square

9.6.1. Ускорение для тандемных повторов точности k

Идея алгоритма трудоемкости $O(kn \log(n/k) + z)$ Ландау и Шмидта [288] является непосредственным развитием метода для нахождения точных тандемных повторов трудоемкости $O(n \log n + z)$, но реализация его немного сложнее. Этот метод также рекурсивен, и важной его частью также является подзадача 3 (или 4), находящая все тандемные повторы точности k , у которых первая копия покрывает позицию h . Решение заключается в том, чтобы пропустить k последовательных запросов о наибольшем общем продолжении от h и q в прямом направлении и k запросов от $h - 1$ и $q - 1$ в обратном направлении. Теперь сосредоточимся на интервале между h и q . Чтобы найти все тандемные повторы точности k , у которых первая копия покрывает h , нужно найти в этом интервале каждую позицию t (если такая существует) где число несовпадений от h до t (найденное при продолжении вперед) плюс число несовпадений от $t + 1$ до $q - 1$ (найденное при продолжении назад) не превосходит k . Любое такое t дает среднюю точку тандемного повтора. Мы оставляем проверку корректности этого утверждения читателю.

Чтобы достичь объявленной временной границы, мы должны найти все средние точки тандемных повторов точности k , у которых первая копия покрывает h . за время, пропорциональное их числу. Но, в отличие от точных тандемных повторов, множество правильных средних точек не должно быть сплошным. Каких находить? Мы очертим основную идею и оставим детали на упражнения. На протяжении k прямых запросов о продолжении накапливается упорядоченный список позиций в интервале $[h, q]$, где встречаются несовпадения, и то же самое делается в запросах в обратном направлении. После этого оба списка сливаются в порядке слева направо), и для каждой позиции в списке вычисляется полное число ее несовпадений до нее от h до $q - 1$. Так как оба списка упорядочены по построению, время на их слияние и полное время равны $O(k)$. Полное число несовпадений может изменяться только в позициях, которые присутствуют в общем списке; следовательно, сканирование за время $O(k)$ находит все подинтервалы, содержащие разрешенные средние точки тандемов точности k . Вдобавок каждая точка в таком подинтервале является разрешенной средней точкой. Таким образом, для фиксированного h полное время запроса для подзадачи 3 равно $O(k)$, а полное время вывода результата — kz_h .

В расчете на весь алгоритм полное время вывода равно $O(kz)$, а число запросов удовлетворяет соотношению $T(n) = 2T(n/2) + 2k$. Таким образом, выполняется не более $O(kn \log n)$ запросов. В итоге:

Теорема 9.6.2. Все tandemные повторы с точностью k в строке длины n можно найти за время $O(kn \log n + z)$.

Граница может быть уменьшена до $O(kn \log(n/k) + z)$ благодаря тому наблюдению, что не всякое $l \leq k$ должно проверяться в подзадачах 3 и 4. Детали оставляются в качестве упражнения.

На читателя оставляется также адаптация решений для палиндрома с k несовпадениями и задач о tandemных повторах к случаям комплементарных строк и разделения частей полиндрома на ограниченное расстояние.

9.7. Решение задачи о множественной общей подстроке за линейное время

Все приведенные выше приложения однотипны, они используют возможность разрешения запросов о наибольшем общем продолжении за константное время. Сейчас мы рассмотрим еще одно применение суффиксных деревьев с lca за константное время, и оно будет другого типа.

Задача о k -общей подстроке уже рассматривалась в п. 7.6 (читателю следовало бы взглянуть на это обсуждение, прежде чем идти дальше). В этом пункте обобщенное суффиксное дерево \mathcal{T} строится для K строк суммарной длины n , и затем с помощью операций над \mathcal{T} получается таблица значений всех $l(k)$. Этот метод имеет время счета $O(Kn)$. Мы уменьшим время до $O(n)$. Предлагаемое решение было получено Лукасом Хью [236]. *!

Напомним, что $C(v)$ обозначает для любой вершины v из \mathcal{T} число различных листовых идентификаторов в поддереве v и при известных значениях $C(v)$ таблица всех значений $l(k)$ может быть вычислена за время $O(n)$. Напомним также, что $S(v)$ — общее число листьев в поддереве v и $S(v)$ можно легко вычислить для всех вершин за время $O(n)$.

Ясно, что $S(v)$ не меньше, чем $C(v)$, для любой вершины v и может быть строго больше, когда в поддереве v имеются два или больше листа с одинаковым строковым идентификатором. Наш подход к вычислению $C(v)$ заключается в вычислении $S(v)$ и коррекции $U(v)$, которая подсчитывает число “дублей” суффиксов для строк, встречающихся в поддереве v . Зная их, получаем $C(v) = S(v) - U(v)$.

Определение. $n_i(v)$ — число листьев с идентификатором i в поддереве вершины v . Пусть n_i — полное число листьев с идентификатором i .

Опираясь на определение, мы непосредственно получаем следующее:

Лемма 9.7.1. $U(v) = \sum_{i:n_i(v)>0} (n_i(v) - 1)$ и $C(v) = S(v) - U(v)$.

*) Во введении к более ранней неопубликованной рукописи [376] Пратт объявил о решении этой задачи за линейное время, но в его утверждении не уточнялось, какая задача решалась — для фиксированного k или для всех значений k . Раздел работы, в котором должно было быть уточнение, недоступен и предположительно никогда не был закончен [375].

Ниже покажем, что коррекции для всех внутренних вершин можно вычислить за суммарное время $O(n)$. Это обеспечит нам такое же время и при решении задачи о k -общей подстроке.

9.7.1. Метод

Алгоритм сначала совершает нисходящий обход дерева \mathcal{T} , нумеруя листья по мере их появления. Эти номера обладают тем знакомым свойством, что для любой внутренней вершины v номера листьев в каждом поддереве последовательны (т. е. они образуют сплошной интервал).

Для удобства изложения возьмем отдельный идентификатор i и покажем, как вычислить $n_i(v) - 1$ для каждой внутренней вершины v . Пусть L_i обозначает список листьев с идентификатором i в порядке возрастания их номеров. Например, на рис. 9.3 листья с идентификатором i показаны в прямоугольниках, и соответствующий список L_i — это $1, 3, 6, 8, 10$. По свойству нумерации для любого поддерева v все $n_i(v)$ листьев с идентификатором i заполняют в списке L_i сплошной интервал. Обозначим этот интервал через $L_i(v)$. Если x и y — какие-то два листа из $L_i(v)$, то вершина $lca(x, y)$ принадлежит поддереву v . Так что если мы вычислим lca для пары последовательных листьев из $L_i(v)$, то все $n_i(v) - 1$ вычисленных lca будут находиться в поддереве v . Более того, если какая-то из вершин x и y не принадлежит этому поддереву, то и их lca не будет ему принадлежать. Это приводит нас к следующим лемме и методу.

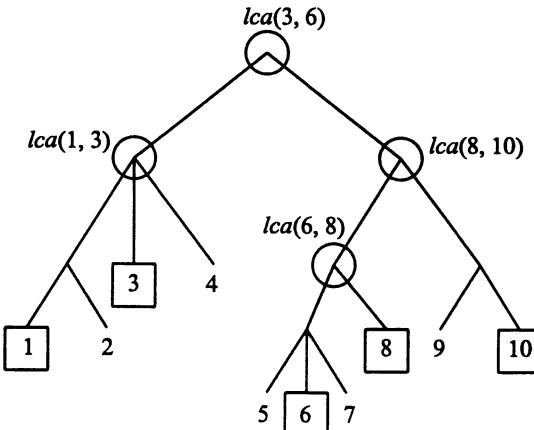


Рис. 9.3. Листья в прямоугольниках имеют идентификатор i . Внутренние вершины в кружках — это наименьшие общие предшественники четырех пар последовательных листьев из списка L_i .

Лемма 9.7.2. *Если вычислить lca для каждой пары последовательных листьев в L_i , то для любой вершины v ровно $n_i(v) - 1$ из вычисленных lca будет лежать в поддереве v .*

Лемма 9.7.2 иллюстрируется на рис. 9.3.

С помощью этой леммы мы можем вычислить $n_i(v) - 1$ для каждой вершины v следующим образом. Вычислить lca для каждой последовательной пары листьев в L_i

и подсчитать для каждой вершины w количество раз $h(w)$, когда результатом вычисления lca было w . Тогда для любой вершины v если обозначить через \mathcal{T}_v поддерево v , то $n_i(v) - 1 = \sum_{w \in \mathcal{T}_v} h(w)$. Теперь для нахождения $n_i(v) - 1$ для каждой вершины v можно использовать стандартный обход \mathcal{T} снизу вверх, требующий времени $O(n)$.

Чтобы найти $U(v)$, нам нужно получить не $n_i(v) - 1$, а $\sum_i [n_i(v) - 1]$. Однако алгоритму незачем делать отдельный обход дерева для каждого идентификатора, поскольку это приведет к оценке времени $O(Kn)$. Вместо этого алгоритм должен обходить дерево снизу вверх, пока не будет просмотрен каждый список L_i . При этом обходе по всем спискам $h(w)$ считается полное число раз, когда w названо в качестве lca . Таким образом, достаточно одного обхода \mathcal{T} . Когда он завершается, $U(v) = \sum_{i:n_i > 0} [n_i(v) - 1] = \sum_{w \in \mathcal{T}_v} h(w)$.

Мы можем теперь полностью описать весь метод трудоемкости $O(n)$ для задачи о k -общей подстроке.

Алгоритм для множественной общей подстроки

begin

1. Построить обобщенное суффиксное дерево \mathcal{T} для K строк.
2. Перенумеровать листья \mathcal{T} в порядке их появления в нисходящем обходе дерева.
3. Для каждого строкового идентификатора i извлечь упорядоченный список L_i листьев с идентификатором i . (Мелкие детали реализации, необходимые для выполнения этого за время $O(n)$, оставляются читателю.)
4. Для каждой вершины w в \mathcal{T} положить $h(w)$ равным нулю.
5. Для каждого идентификатора i вычислить lca всех пар последовательных листьев в L_i и увеличить на 1 счетчик $h(lca)$.
6. Следуя обходу \mathcal{T} снизу вверх, вычислить для каждой вершины v значения $S(v)$ и $U(v) = \sum_{i:n_i > 0} [n_i(v) - 1] = \sum_{w \in \mathcal{T}_v} h(w)$.
7. Положить $C(v) = S(v) - U(v)$ для каждой вершины v .
8. Накопить таблицу значений $l(k)$, как описано в п. 7.6.

end.

9.7.2. Анализ времени

Суффиксное дерево имеет размер $O(n)$, и его подготовка к вычислениям lca также требует времени $O(n)$. Делается $\sum_{i=1}^K |n_i| - 1 < n$ вычислений lca , каждое из них занимает константное время, и в общей сложности требуют времени $O(n)$. Следовательно, и для получения всех значений $C(v)$ необходимо только время $O(n)$. А после — только время $O(n)$ для построения таблицы вывода. Эта часть алгоритма остается такой же, как в уже обсуждавшемся в п. 7.6 алгоритме трудоемкости $O(Kn)$. Итак:

Теорема 9.7.1. *Пусть \mathcal{S} — набор из K строк суммарной длины n , а $l(k)$ — длина наибольшей подстроки, присутствующей не меньше чем в k различных строках*

из \mathcal{S} . Таблицу всех значений $l(k)$, для k от 2 до K , можно построить за время $O(n)$.

Объем информации о подстроках \mathcal{S} , который можно получить за время, пропорциональное времени прочтения этих строк, очень впечатляет. Было бы заманчиво попробовать получить этот результат без использования суффиксных деревьев (или аналогичных структур данных).

9.7.3. Родственные применения

Методология, развитая для задачи о k -общей подстроке, может быть легко обобщена для решения важных родственных задач о наборах строк.

Например, предположим, что у вас есть два набора строк \mathcal{S} и \mathcal{P} и вы хотите знать про каждую строку $A \in \mathcal{P}$, в сколько строк из \mathcal{S} она входит. Пусть n обозначает суммарный размер всех строк из \mathcal{S} , а m — всех строк из \mathcal{P} . Задача может быть решена за время $O(n + m)$, и та же времененная оценка достижима при помощи метода Ахо–Корасика.

Вот еще одна задача. Задана длина l . Найти строку длины не меньше l , которая присутствует в самом большом числе строк данного набора. То есть требуется найти самую общую подстроку длины не менее l . Эта задача применяется во многих методах множественного выравнивания. (См. упражнение 26 главы 14.)

9.8. Упражнения

1. Докажите теорему 9.1.1.
2. Восполните подробности и докажите корректность метода решения задачи о наибольшем общем продолжении с эффективным использованием памяти.
3. Приведите подробности поиска в строке всех максимальных палиндромов нечетной длины за линейное время.
4. Покажите, как решить все задачи о палиндромах за линейное время, используя только суффиксное дерево для строки S , а не для обеих строк S и S' .
5. Приведите детали организации поиска комплементарных палиндромов в линейной строке.
6. Напомним, что *плазмида* — это кольцевая молекула ДНК, типичная для бактерий. Некоторые бактериальные плазмиды содержат относительно длинные комплементарные палиндромы (функции которых не вполне известны). Предложите алгоритм с линейным временем для нахождения всех максимальных комплементарных палиндромов в циклической строке.
7. Покажите, как найти все палиндромы точности k в строке длины n за время $O(kn)$.
8. **Тандемные повторы.** В рекурсивном методе, описанном в п. 9.6 (с. 252), для нахождения точных tandemных повторов решалась задача 3 с линейным числом выполняемых за константное время запросов об общем продолжении. В выполнении этих запросов применялись суффиксные деревья и вычисления наименьшего общего предшественника. Имеется более раннее решение задачи 3, столь же эффективное и не использующее суффиксных деревьев; оно было предложено Мэйном и Лоренцем [307].

Идея в том, что задачу можно решать без суффиксных деревьев с *амортизационной линейной оценкой времени*. По сути задачи 3, h остается фиксированным, тогда как $q = h + l - 1$ изменяется, когда l пробегает все подходящие значения. Каждый запрос об общем продолжении вперед является задачей нахождения длины *наибольшей подстроки начинающейся в позиции q и совпадающей с префиксом $S[h..n]$* . Все длины должны быть найдены за линейное время. Но эта цель может быть достигнута вычислением значений Z (см. главу 1) для подходящей подстроки S . Восполните детали этого подхода и докажите линейную амортизационную оценку.

Теперь покажите, как можно решить за линейное время и задачу об общих продолжениях в обратном направлении, также используя вычисление значений Z для сконструированной нужным образом подстроки S . Эта подстрока формируется чуть сложнее, чем для прямых продолжений.

9. Восполните детали алгоритма трудоемкости $O(kn \log n + z)$ для задачи о тандемном повторе точности k . Рассмотрите и корректность и время.
10. Восполните детали построения границы $O(kn \log(n/k) + z)$ для решения задачи о тандемном повторе точности k .
11. Попробуйте модифицировать метод Мэйна–Лоренца для поиска всех точных тандемных повторов, чтобы он решал задачу о тандемных повторах точности k за время $O(kn \log n + z)$. Если не выйдет, то объясните, в чем трудности и как при решении этих задач использовать суффиксные деревья и общих предшественников.
12. Метод тандемных повторов, описанный в п. 9.6, находит все тандемные повторы, даже если они не максимальны. Например, в строке *hababababy* он находит шесть повторов, хотя самый левый повтор *abab* содержится в более длинном повторе *abababab*. В зависимости от приложения такой результат может быть нежелателен. Дайте определение *максимальности*, которое уменьшило бы размер вывода, и постарайтесь предложить эффективные алгоритмы для различных определений.
13. Рассмотрим следующую ситуацию. Задается и остается фиксированной длинная строка S . Затем предполагается последовательность более коротких строк S_1, S_2, \dots, S_k . После ввода каждой строки S_i (но до S_{i+1}) делается несколько запросов о наибольшем общем продолжении S_i и S . Пусть r — полное число запросов, а n — полная длина всех коротких строк. Требуется эффективно отвечать на все эти онлайновые запросы. Самый прямой подход — построить обобщенное суффиксное дерево для S и S_i , когда S_i станет известным, подготовить его (назначив *dfs*-номера нисходящего обхода, найдя значения $I()$ и т. д.) для алгоритма *lca* с константным временем и затем отвечать на запросы для S_i . Но это даст время $\Theta(k|S| + n + r)$. Слагаемое k получается из двух источников: время на построение k обобщенных суффиксных деревьев и время на подготовку каждого из них для запросов об *lca*.

Уменьшите оба источника этого слагаемого до $|S|$ и получите общую границу $O(|S| + n + r)$. Уменьшить время на построение всех обобщенных суффиксных деревьев легко, а на подготовку к выполнению запросов об *lca* — немного сложнее.

Найдите правдоподобное приложение этого результата.

Часть III

Неточное сопоставление,
выстраивание последовательностей
и динамическое программирование

Мы переходим из общей области *точного сопоставления* и *точного обнаружения образцов* к общей области *неточного, приближенного сопоставления* и *выравнивания последовательностей*. “Приближенное” означает, что в принимаемом совпадении возможны некоторые ошибки разных типов, уточняемых в дальнейшем. “Выравниванию” точный смысл будет придан позднее, но вообще оно означает расположение символов по строкам, допускающее совпадения и несовпадения, а также расположение символов из одной строки напротив пустых мест, оставляемых в другой строке.

Мы переходим также от задач относительно *подстрок* к задачам относительно *подпоследовательностей*. Подпоследовательность отличается от подстроки тем, что символы в подстроке должны идти подряд, тогда как в подпоследовательности, входящей в строку, это не обязательно^{*)}. Например, строка *xuz* является подпоследовательностью, но не подстрокой в строке *axayaz*. Переход от подстрок к подпоследовательностям является естественным следствием перехода от точного к неточному совпадению. Переход к неточному совпадению и сравнению подпоследовательностей сопровождается изменением в *технике*. Большая часть методов, рассматриваемых в части III, и многие из методов в части IV используют в качестве средства *динамическое программирование*, которое в частях I и II нам не требовалось.

Немного о разделах вычислительной биологии, относящихся к выравниванию последовательностей

Область приближенного совпадения и сравнения последовательностей занимает в вычислительной молекулярной биологии центральное место, как из-за наличия ошибок в молекулярных данных, так и ввиду активных мутационных процессов, которые методы сравнения (под)последовательностей призваны моделировать и обнаруживать. Эта тема разрабатывается в следующей главе и иллюстрируется дальше по всей книге. С технической стороны выравнивание последовательностей становится основным средством сравнения последовательностей в молекулярной биологии. Хайнайкофы пишут:

“Среди наиболее полезных компьютерных средств в современной биологии стоят те, которые включают в себя выравнивание последовательностей белков, так как эти выравнивания обеспечивают изучение функций гена и белка. Есть несколько разных типов выравнивания: глобальные выравнивания пар белков, связанных общим происхождением, по всей их длине; локальные выравнивания, включающие родственные сегменты белков; множественные выравнивания членов семейств белков; наконец, выравнивания, которые делаются во время поисков в базах данных для определения гомологий” [222].

Это положение обеспечивает основу для большинства материалов части III. Мы займемся детальным анализом всех четырех типов выравниваний, перечисленных

^{*)} В биологической литературе стало общей и запутывающей практикой называть подстроки подпоследовательностями. Но техника и результаты для задач о подстроках и таких же задач о подпоследовательностях отличаются очень сильно, так что важно выдерживать четкое различие. В этой книге мы никогда не используем термин “подпоследовательность”, если имеется в виду “подстрока”.

выше (и некоторыми их вариантами). Мы покажем также, как разные модели выравнивания относятся к различным проблемам биологии. В главе 10 мы начнем с более подробного описания того, почему сравнение последовательностей заняло центральное место в современной молекулярной биологии. Но нам не следует забывать о роли точного сопоставления.

Роль точного сопоставления

Центральное положение приближенного сопоставления в молекулярной биологии бесспорно. Однако из этого не следует, что методы точного сопоставления здесь мало применимы, и некоторые биологические приложения точного сопоставления были развиты в частях I и II. В качестве примера напомним п. 7.15, где отмечалась центральная роль суффиксных деревьев в нескольких работах, связанных с биологическими базами данных. Более того, ранее было показано, что некоторые способы точного сопоставления прямо развиваются до задач приближенного сопоставления или применяются к ним (задачи счета совпадений, о метасимволах, о k несовпадениях, о палиндроме точности k и о tandemном повторе точности k). В частях III и IV мы предложим дополнительные методы приближенного сопоставления, которые решительным образом опираются на эффективные методы точного сопоставления, суффиксные деревья и др. Мы увидим также задачи точного сопоставления, которые встречаются как подзадачи в множественном сравнении последовательностей, в крупномасштабном сравнении последовательностей, при поиске в базах данных и в других биологически важных приложениях.

Глава 10

Значение сравнения (под)последовательностей

Сравнение последовательностей, особенно в сочетании с систематическим сбором, поддержанием и поиском информации в базах данных, содержащих биомолекулярные последовательности, стало одним из основных методов современной молекулярной биологии. Комментируя близкие тогда к завершению (и ныне завершенные) попытки расшифровки всего генома дрожжей, Стивен Оливер сказал:

“За короткое время трудно осознать, как бы мы управились без техники работы с последовательностями. Биология уже никогда не будет такой, как прежде” [478].

Один факт объясняет значение данных о молекулярных последовательностях и сравнения последовательностей в биологии.

Первый факт анализа биологических последовательностей

В биомолекулярных последовательностях (ДНК, РНК или аминокислотных последовательностях) высокое сходство последовательностей обычно влечет существенное функциональное или структурное сходство.

Эволюция многократно использует, надстраивает, копирует и модифицирует “удачные” структуры (белки, экзоны, регулирующие последовательности ДНК, морфологические черты, ферментативные пути и т. д.). В основе феномена **жизни** лежит целый репертуар структурированных и взаимосвязанных молекулярных строительных блоков, которые воспроизводятся и распространяются. Идентичные или родственные молекулярные структуры и механизмы многократно проявляются и в геноме простейших видов, и вместе с тем в очень широком спектре различных видов. “Копирование с модификацией” [127–130] — центральная парадигма эволюции

белков, в которой новые белки и/или новые биологические функции моделируются из имевшихся ранее. Дулилл подчеркивает эту точку зрения следующим образом:

“Подавляющее большинство существующих белков появилось после непрерывного ряда генетических копирований и последовательных модификаций. В результате избыточность является изначальной характеристикой белковых последовательностей, и нас не должно удивлять, что так много новых последовательностей напоминает уже известные” [129].

Он же добавляет, что

“... все в биологии основано на чудовищной избыточности...” [130].

Следующие цитаты придают этой точке зрения дополнительный вес и указывают на полезность “чудовищной избыточности” в практике молекулярной биологии. Первая цитата принадлежит Эрику Вайсхаузу, одному из лауреатов Нобелевской премии по медицине в 1995 г. за работу по генетике развития *дрозофилы*. Цитата взята из статьи для *Associated Press* от 9 октября 1995 г. Описывая работу, сделанную много лет назад, Вайсхауз говорит:

“В то время мы обнаружили, что все в жизни настолько одинаково, что одни и те же гены работают и в мушке, и в человеке”.

И плодовые мушки не стоят особняком. Следующая цитата взята из рецензии на книгу о восстановлении ДНК:

“На протяжении всей книги мы узнаем об открытиях, найденных благодаря нашей возможности обнаруживать гомологии последовательностей сравнением ДНК различных видов. Изучение дрожжей замечательно предсказывает системы человека!” [424]

Итак, “избыточность” и “сходство” — вот центральные феномены биологии. Но сходство имеет свои пределы — человек и мушка в некоторых аспектах все же различны. Эти различия делают сохранившееся сходство еще более существенным, что в свою очередь превращает сравнение и аналогию в очень мощные инструменты биологии. Леск пишет:

“Для биологических систем характерно, что наблюдаемые нами объекты, чтобы получить определенную форму, эволюционно развивались из родственных объектов со сходными, но не идентичными формами. Поэтому они должны быть устойчивыми, чтобы выдерживать некоторое варьирование. Мы можем использовать преимущества этой робастности в нашем анализе: идентифицируя и сравнивая родственные объекты, различить меняющиеся и консервативные черты и таким образом определить, что существенно для структуры и функции” [297].

Важные “родственные объекты”, выбираемые для сравнения, включают в себя гораздо больше, чем последовательные данные, поскольку биологическая универсальность проявляется на многих уровнях детализации. Однако обычно легче получать и проверять последовательности, чем мельчайшие подробности генетики или биохимии и морфологии клеток. Например, известно несравненно больше первичных последовательностей белков (полученных на основании имеющихся последовательностей ДНК), чем трехмерных белковых структур. Эти последовательности становятся важными не просто из соображений удобства. Скорее, в биологических последовательностях закодированы и отражаются более сложные общие молекулярные структуры и механизмы, которые проявляются как свойства на клеточном и биохимическом уровнях. Более того, “нигде в биологическом мире дарвиновское понятие “наследование с модификацией” не проявляется больше, чем в последовательностях

генов и генных продуктов” [130]. Следовательно, возможным, хотя и частично эвристическим путем поиска функциональной или структуральной универсальности в биологических системах является поиск сходства и консерватизма на уровне последовательностей. Мощь этого подхода становится ясной из следующих цитат:

“Сегодня наиболее мощный способ для установления биологической функции гена (или закодированного им белка) заключается в поиске похожих последовательностей в базах данных белков и ДНК. С развитием быстрых методов сравнения последовательностей, наряду с эвристическими алгоритмами и мощными параллельными компьютерами, открытия, основанные единственно на сходстве последовательностей, превратились в рутину” [360].

“Определение функции, выполняемой последовательностью, — это дело поразительной сложности, требующее биологических экспериментов с высочайшим уровнем творческой мысли. Тем не менее с помощью лишь последовательности ДНК можно выполнить компьютерный анализ, сравнивающий эту последовательность с базой данных по генам, охарактеризованным ранее. Примерно в 50 % случаев такое механическое сравнение укажет на достаточное сходство, чтобы сделать предположение о ферментативной или структурной функции, которой может обладать неизвестный ген” [91].

Итак, крупномасштабное сравнение последовательностей, обычно организованное в виде поиска в базах данных, является очень мощным средством для получения выводов в современной молекулярной биологии. И это средство действительно почти универсально используется молекулярными биологами. Когда новый ген клонируется и расшифровывается, стало стандартной практикой переводить его последовательность ДНК в последовательность аминокислот и затем искать сходство между ней и записями баз данных для известных белков. Сейчас никому в голову не придет публиковать расшифровку вновь клонированного гена без проведения такого поиска.

Заключительная цитата отражает потенциальное полное влияние на биологию упомянутого *первого факта* и его использование в форме поиска в базе данных последовательностей. Цитата взята из статьи Уолтера Гилберта, Нобелевского лауреата, награжденного за участие в открытии практического метода расшифровки ДНК. Гилберт пишет:

“Проявляющаяся сейчас новая парадигма состоит в том, что все “гены” будут известны (в том смысле, что будут помещены в электронно доступные базы данных и что начальная точка биологического исследования будет теоретической. Ученый будет начинать свою работу с теоретического предположения, только после этого обращаясь к эксперименту, чтобы следовать своей гипотезе или проверять ее” [179].

Ежегодно появляются сотни (если не тысячи) журнальных публикаций, которые сообщают о биологических исследованиях, где сравнение последовательностей и/или поиск в базах данных составляет неотъемлемую часть работы. Примеры таких работ, поддерживающих и иллюстрирующих *первый факт*, рассыпаны по этой книге. В частности, некоторые важные примеры сосредоточены в главах 14 и 15, где обсуждается множественное сравнение строк и поиск в базах данных. Но прежде чем обсуждать эти примеры, мы должны сначала подготовить в нескольких главах аппарат, предназначенный для приближенного сопоставления и сравнения (под)последовательностей.

Caveat

Первый факт анализа биологических последовательностей — это чрезвычайно мощное средство, и его значение будет дальше иллюстрироваться на протяжении всей книги. Однако нет одно-однозначного соответствия между последовательностью и структурой или последовательностью и функцией, так как утверждение, обратное *первому факту*, неверно. Высокое сходство последовательностей обычно указывает на значительное структурное или функциональное сходство (вот он, первый факт), но структурное или функциональное сходство не влечет непременно сходства последовательностей. По поводу структуры белков Ф. Коэн пишет: “... похожие последовательности приводят к похожим структурам, но совершенно различные последовательности могут производить поразительно похожие структуры” [106]. Этот вопрос *обращения* обсуждается более глубоко в главе 14, которая посвящена множественному сравнению последовательностей.

Глава 11

Ядро методов редактирования строк и выстраивания

11.1. Введение

В этой главе мы рассматриваем как задачи неточного сопоставления и выравнивания, которые составляют ядро этой области, так и другие задачи, иллюстрирующие более общую технику. Изложение некоторых задач и методов будет в последующих главах улучшено и обобщено. Мы начнем с подробного анализа классической задачи неточного совпадения, решаемой с помощью динамического программирования — задачи о *редакционном расстоянии*. Темы из молекулярной биологии, побуждающие к рассмотрению неточного совпадения (и более общо, сравнения последовательностей), будут возникать постоянно, на протяжении всей книги. Мы обсудим много специфических примеров того, как сравнение строк и неточное сопоставление используются в современной молекулярной биологии. Однако для начала сосредоточимся на чисто формальных и технических аспектах определения и вычисления неточного сопоставления.

11.2. Редакционное расстояние между двумя строками

Часто требуется измерить различие или *расстояние между двумя строками* (например, в эволюционных, структуральных или функциональных исследованиях биологических строк, в хранении текстовых баз данных, в методах проверки правописания). Есть несколько способов формализации понятия расстояния между строками. Одна общая, и простая, формализация [389, 299] называется *редакционным расстоянием*; она основана на *преобразовании* (или *редактировании*) одной строки в другую серией операций редактирования, выполняемых над отдельными символами. Разрешенные

операции редактирования — это *вставка* (insertion) символа в первую строку, *удаление* (deletion) символа из первой строки и *подстановка* или *замена* (substitution или, лучше, replace) символа из первой строки символом из второй строки. Пусть, например, I обозначает insert, D — delete, R — replace и M — “не-операцию” над правильной буквой (от match). Тогда строка *vintner* может быть доредактирована до *writers* следующим образом:

R	I	M	D	M	D	M	M	I
v		i	n	t	n	e	r	
w	r	i		t		e	r	s

Это значит, что v заменяется на w , r вставляется, i совпадает и не изменяется, так как присутствует в обеих строках, n удаляется, t не изменяется, n удаляется, er совпадают и не изменяются и, наконец, s вставляется. Мы можем теперь формально определить редакционные предписания и строковые преобразования.

Определение. Стока над алфавитом I, D, R, M , которая описывает преобразование одной строки в другую, называется *редакционным предписанием* или, для краткости, *предписанием* этих двух строк.

В общем, если задано две входных строки, S_1 и S_2 , и редакционное предписание для них, то преобразование выполняется последовательным применением операций из предписания к очередным символам соответствующей строки (или символам строк). Пусть $next_1$ и $next_2$ — указатели места в S_1 и S_2 . Оба указателя начинают со значения 1. Редакционное предписание читается и исполняется слева направо. Когда считывается символ I , буква $next_2$ из S_2 вставляется перед буквой $next_1$ в S_1 , и указатель $next_2$ увеличивается на одну букву. Когда считывается D , буква $next_1$ удаляется из S_1 и $next_1$ увеличивается на одну букву. Когда считывается R или M , буква $next_1$ в S_1 , соответственно, заменяется или совпадает с буквой $next_2$ из S_2 и оба указателя увеличиваются на 1.*)

Определение. *Редакционное расстояние* между двумя строками определяется как минимальное число редакционных операций — вставок, удалений и подстановок, необходимое для преобразования первой строки во вторую.

Подчеркнем, что совпадения операциями не являются и не засчитываются.

Редакционное расстояние иногда называют *расстоянием Левенштейна* по статье В. Левенштейна [299], где оно рассматривалось, вероятно, впервые.

Мы будем иногда называть редакционное предписание, использующее минимальное число редакционных операций, *оптимальным предписанием*. Заметим, что оптимальных предписаний может быть больше одного. Когда мы захотим подчеркнуть это, предписания будут называться *кооптимальными*.

Задача о редакционном расстоянии — это задача о вычислении редакционного расстояния между двумя данными строками, вместе с оптимальным редакционным предписанием, описывающим преобразование, на котором этот минимум достигается.

*) Здесь нужно иметь в виду, что строка S_1 существует одновременно в двух вариантах — исходном и результирующем — и указатель $next_1$ описывает положение буквы в исходной строке. Иначе нам нужно при изменении строки менять и указатели. Поэтому операция удаления в этом контексте не удаляет букву, а просто не копирует ее. — Прим. перев.

Из определения редакционного расстояния следует, что все операции применяются только к одной строке. Но иногда его рассматривают как минимальное число операций, выполняемых над любой из двух строк для преобразования обеих в общую третью строку. Этот взгляд эквивалентен данному определению, так как включение в одну строку может рассматриваться как удаление из другой, и наоборот.

11.2.1. Выравнивание строк

Редакционное предписание — это способ *представления* конкретного преобразования одной строки в другую. Альтернативный (и часто предпочтительный) способ заключается в показе явного *выравнивания* (alignment) этих двух строк.

Определение. (Глобальное) *выравнивание* двух строк, S_1 и S_2 , получается вставкой пробелов в строки S_1 и S_2 (возможно, на их концах) и размещением двух получившихся строк друг над другом так, чтобы каждый символ или пробел одной строки оказался напротив одного символа или пробела другой строки.

Термин “глобальный” подчеркивает тот факт, что обе строки участвуют в выравнивании полностью. Иная картина складывается в случае локального выравнивания, обсуждаемого дальше. Отметим, что наше использование слова “выравнивание” значительно точнее, чем в частях I и II. Там выравнивание употреблялось в обычном смысле для указания на расположение одной строки относительно другой и вставлять пробелы в какую-либо из строк не разрешалось.

В качестве примера рассмотрим глобальное выравнивание строк $qacbdb$ и $qa\cancel{a}xb$:

$$\begin{array}{ccccccc} q & a & c & - & d & b & d \\ q & a & w & x & - & b & - \end{array}$$

Здесь символ c не совпадает с w , оба d и x стоят напротив пробелов, изображенных знаком подчеркивания, а все остальные символы совпадают со своими напарниками из другой строки.

Другой пример выравнивания был показан на с. 270, где слова *vintner* и *writers* были выстроены под их редакционным предписанием. Этот пример показывает также связь между выравниванием и редакционным предписанием, которую нам еще предстоит обсудить.

Выравнивание и редакционное предписание

С математической точки зрения выравнивание и редакционное предписание — это эквивалентные способы описания отношения между двумя строками. Выравнивание можно легко конвертировать в эквивалентное редакционное предписание и наоборот, как это видно по примеру *vintner-writers*. Именно, пара несовпадающих символов в выравнивании соответствует подстановке в редакционном предписании, пробел в первой строке выравнивания — вставке парного символа из второй строки в первую, а пробел во второй строке — удалению парного символа из первой строки. Таким образом, редакционное расстояние между двумя строками задается выравниванием, минимизирующим число пар несовпадающих символов плюс число пар, содержащих пробелы.

Хотя выравнивание и редакционное предписание математически эквивалентны, с модельной точки зрения предписание совершенно отлично от выравнивания. Предписание подчеркивает предполагаемые *мутационные события* (точковые мутации в имеющейся модели), которые переводят одну строку в другую, тогда как выравнивание просто показывает соотношение между двумя строками. Это различие *процесса и результата*. Разные эволюционные модели формализуются с помощью разных строковых операций, которые могут отображаться в то же самое выравнивание. Поэтому выравнивание само по себе затирает мутационную модель. Эта точка зрения представляется педантичной, но она доказала свою полезность в некоторых дискуссиях о моделировании эволюции.

Мы будем переключаться с языка редакционных предписаний на выравнивание и обратно, как это будет удобно. Однако язык выравниваний зачастую предпочтительнее, так как он более нейтрален и не содержит намеков на процесс. Он более нейтрален также в области множественного сравнения последовательностей.

11.3. Вычисление редакционного расстояния с помощью динамического программирования

Обратимся теперь к алгоритмическому вопросу: как вычислить с помощью динамического программирования редакционное расстояние между двумя строками вместе с редакционным предписанием или выравнивание, на котором оно достигается. Общая парадигма динамического программирования, вероятно, хорошо известна читателям этой книги. Однако, так как это средство исключительно важно и часто используется в строковых алгоритмах, стоит подробно объяснить как общий подход динамического программирования, так и его специфическое приложение к задаче о редакционном расстоянии.

Определение. Для двух строк, S_1 и S_2 , значение $D(i, j)$ определяется как редакционное расстояние между $S_1[1..i]$ и $S_2[1..j]$.

Значит, $D(i, j)$ определяет минимальное число редакционных операций, необходимых для преобразования первых i символов S_1 в первые j символов S_2 . При использовании этой нотации если $|S_1| = n$ и $|S_2| = m$, то редакционное расстояние между S_1 и S_2 равно *в точности* значению $D(n, m)$.

Чтобы найти $D(n, m)$, будем решать более общую задачу вычисления значений $D(i, j)$ для всех комбинаций i и j , когда i меняется от 0 до n , а j от 0 до m . Это стандартный подход *динамического программирования*, применяемый в большом числе вычислительных задач. Он содержит три существенные компоненты: *рекуррентное соотношение*, *табличная форма вычислений* и *обратный проход*. Теперь поясним каждую компоненту.

11.3.1. Рекуррентное соотношение

Рекуррентное соотношение задает *рекурсивное* отношение между значением $D(i, j)$ для положительных i и j и значениями D с индексными парами, меньшими i , j . Когда меньших индексов нет, значение $D(i, j)$ должно получаться явно из условий, которые будем называть *базовыми условиями* для $D(i, j)$.

Для задачи о редакционном расстоянии базовые условия таковы:

$$D(i, 0) = i$$

и

$$D(0, j) = j.$$

Базовое условие $D(i, 0) = i$ очевидно корректно (т. е. оно дает число, требуемое определением $D(i, 0)$), так как единственный способ преобразования первых i символов S_1 в нуль символов S_2 заключается в удалении всех этих i символов. Аналогично, условие $D(0, j) = j$ корректно, потому что для преобразования нуля символов S_1 в j символов S_2 требуется вставить j символов.

Рекуррентное соотношение для $D(i, j)$, когда i и j строго положительны, выглядит так:

$$D(i, j) = \min\{D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1) + t(i, j)\},$$

где значение $t(i, j)$ по определению равно 1, если $S_1(i) \neq S_2(j)$, и 0, если $S_1(i) = S_2(j)$.

Корректность общей рекуррентции

Мы установим корректность в следующих двух леммах, использующих концепцию редакционного предписания.

Лемма 11.3.1. *Значение $D(i, j)$ должно совпадать с $D(i, j - 1) + 1$, $D(i - 1, j) + 1$ или $D(i - 1, j - 1) + t(i, j)$. Других возможностей нет.*

Доказательство. Рассмотрим редакционное предписание для преобразования $S_1[1..i]$ в $S_2[1..j]$, использующее минимальное число редакционных операций, и посмотрим на последний символ этого предписания. Он может быть равен I , D , R или M . Если он равен I , то последняя редакционная операция — это вставка символа $S_2(j)$ в конец (преобразуемой) первой строки. Отсюда следует, что символы в предписании до этого I должны обеспечивать минимальное число операций, преобразующих $S_1[1..i]$ в $S_2[1..j - 1]$ (если они этого не делают, то данное преобразование $S_1[1..i]$ в $S_2[1..j]$ использовало бы число операций, большее минимального). По определению это последнее преобразование требует $D(i, j - 1)$ редакционных операций. Следовательно, если последний символ в предписании равен I , то $D(i, j) = D(i, j - 1) + 1$.

Аналогично, если последний символ в предписании равен D , то последняя редакционная операция — это удаление $S_1(i)$, и символы в предписании влево от этого D должны задавать минимальное число редакционных операций для преобразования $S_1[1..i - 1]$ в $S_2[1..j]$. По определению это последнее преобразование требует $D(i - 1, j)$ редакционных операций. Таким образом, если последний символ в предписании равен D , то $D(i, j) = D(i - 1, j) + 1$.

Если последний символ в предписании равен R , то последняя редакционная операция заменяет $S_1(i)$ на $S_2(j)$, а символы влево от R определяют минимальное число редакционных операций для преобразования $S_1[1..i - 1]$ в $S_2[1..j - 1]$. В этом случае $D(i, j) = D(i - 1, j - 1) + 1$. Наконец, по аналогичным причинам, если последний символ в предписании равен M , то $S_1(i) = S_2(j)$ и $D(i, j) = D(i - 1, j - 1)$. Используя введенную ранее переменную $t(i, j)$ (т. е. 0, если $S_1(i) = S_2(j)$, и 1 — в противном случае), мы можем соединить два последних случая в один: если последний символ предписания равен R или M , то $D(i, j) = D(i - 1, j - 1) + t(i, j)$.

Так как последний символ предписания должен быть равен I , D , R или M , мы исчерпали все случаи и получили утверждение леммы. \square

Посмотрим теперь с другой стороны.

Лемма 11.3.2. $D(i, j) \leq \min\{D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1) + t(i, j)\}$.

Доказательство. Рассуждение очень похоже на предложенное в предыдущей лемме, но его цель несколько отлична. Сейчас мы намерены конструктивно показать существование преобразований, на которых достигается каждое из трех значений, выписанных в неравенстве. После этого, так как каждое из трех значений достижимо, их минимум также будет достижим.

Прежде всего, можно преобразовать $S_1[1..i]$ в $S_2[1..j]$ ровно за $D(i, j - 1) + 1$ редакционных операций. Нужно просто преобразовать $S_1[1..i]$ в $S_2[1..j - 1]$ за минимальное число операций и использовать еще одну операцию, чтобы вставить символ $S_2(j)$ в конце. По определению число редакционных операций в этом конкретном способе преобразования S_1 в S_2 равно $D(i, j - 1) + 1$. Далее, можно преобразовать $S_1[1..i]$ в $S_2[1..j]$ ровно за $D(i - 1, j) + 1$ операций: преобразовать $S_1[1..i - 1]$ в $S_2[1..j]$ за наименьшее число операций и затем удалить $S_1(i)$, что и даст в точности $D(i - 1, j) + 1$. Наконец, видно, как сделать преобразование, используя ровно $D(i - 1, j - 1) + t(i, j)$ операций. \square

Из лемм 11.3.1 и 11.3.2 прямо следует корректность общего рекуррентного соотношения для $D(i, j)$.

Теорема 11.3.1. $D(i, j) = \min\{D(i - 1, j) + 1, D(i, j - 1) + 1, D(i - 1, j - 1) + t(i, j)\}$, когда i и j строго положительны.

Доказательство. Лемма 11.3.1 утверждает, что $D(i, j)$ должно равняться одному из трех значений: $D(i - 1, j) + 1$, $D(i, j - 1) + 1$ или $D(i - 1, j - 1) + t(i, j)$. Лемма 11.3.2 утверждает, что $D(i, j)$ не должно превосходить наименьшего из этих трех значений. Отсюда следует, что $D(i, j)$ должно равняться наименьшему из этих трех значений, и теорема доказана. \square

Здесь завершается описание рекуррентного соотношения — первой компоненты метода динамического программирования для редакционного расстояния.

11.3.2. Табличный расчет редакционного расстояния

Вторая существенная компонента любой задачи динамического программирования — это использование рекуррентных соотношений для эффективного вычисления значения $D(n, m)$. Мы могли бы запрограммировать рекуррентные соотношения и базовые условия для $D(i, j)$ в виде рекурсивной процедуры на любом языке программирования, допускающем рекурсию. Сделав это, мы могли бы вызвать процедуру с параметрами m и n и ждать ответа^{*}). Эта *нисходящая* рекурсия в вычислении $D(n, m)$ легко программируется, но крайне неэффективна для больших значений n и m .

Проблема в том, что с ростом n и m число рекурсивных вызовов растет экспоненциально (проверка этого будет несложным упражнением). Но существует только $(n + 1) \times (m + 1)$ комбинаций i и j , так что среди рекурсивных вызовов будет лишь

* И ждать, и ждать...

$(n + 1) \times (m + 1)$ различных. Следовательно, неэффективность исходящего вычисления объясняется громадным числом избыточных рекурсивных вызовов процедуры. Прелестное обсуждение этого феномена содержится в [112]. Ключом к (несравненно) более эффективному вычислению $D(n, m)$ является отказ от простоты исходящего вычисления и переход к вычислению восходящему.

Восходящий расчет

При восходящем методе мы вычисляем $D(i, j)$ для наименьших возможных значений i и j , а затем для других значений, увеличивая i и j . Обычно восходящий расчет организуется в таблице $(n + 1) \times (m + 1)$. Таблица содержит значения $D(i, j)$ для всех выборов i и j (рис. 11.1). В ней строка S_1 соответствует вертикальной оси, а S_2 — горизонтальной. Так как изменение i и j начинается от нуля, таблица^{*)} включает в себя нулевую строчку и нулевой столбец, значения в которых получаются прямо из базовых условий для $D(i, j)$. После подготовки таблицы оставшаяся подтаблица размером $n \times m$ заполняется строчкой за строчкой, в порядке возрастания i . Внутри каждой строчки клетки заполняются в порядке возрастания j .

$D(i, j)$		w	r	i	t	e	r	s
	0	1	2	3	4	5	6	7
v	0	0	1	2	3	4	5	6
i	1	1						
n	2	2						
t	3	3						
n	4	4						
e	5	5						
r	6	6						
	7	7						

Рис. 11.1. Таблица для вычисления редакционного расстояния между *vintner* и *writers*. Значения в нулевой строчке и нулевом столбце, определяемые по базовым условиям, уже включены в таблицу

Чтобы увидеть, как заполняется подтаблица, отметим, что согласно общему рекуррентному соотношению для $D(i, j)$ все значения, необходимые для вычисления $D(1, 1)$, известны, когда вычислены $D(0, 0)$, $D(1, 0)$ и $D(0, 1)$. Следовательно, когда нулевая строчка и нулевой столбец заполнены, можно вычислить $D(1, 1)$. Далее, снова согласно рекуррентным соотношениям после вычисления $D(1, 1)$, мы имеем все значения для вычисления $D(1, 2)$. Следуя по этому пути, видим, что все значения в первой строчке можно вычислить в порядке возрастания индекса j . Теперь становятся известными все значения, необходимые для вычисления второй строчки, и ее можно заполнить в порядке возрастания j . Так заполняется вся таблица, строчка за строчкой по возрастанию i , и в каждой строчке клетки заполняются по возрастанию j (рис. 11.2).

^{*)} Автор, разумеется, использует термины “string” и “row”, и одновременно переводить их правильным термином “строка” было невозможно. Чтобы не было путаницы, всюду для строк таблицы мы используем перевод “строчка”. — Прим. перев.

$D(i, j)$		w	r	i	t	e	r	s
	0	1	2	3	4	5	6	7
v	0	0	1	2	3	4	5	6
i	1	1	1	2	3	4	5	6
n	2	2	2	2	2	3	4	5
t	3	3	3	3	3	3	4	5
n	4	4	4	4	4	*		
e	5	5						
r	6	6						
	7	7						

Рис. 11.2. Таблица заполняется редакционными расстояниями по строчкам, и в каждой строчке — слева направо. Пример показывает момент, когда вычислены расстояния вплоть до столбца 3 в строчке 4. Следующим вычисляется $D(4, 4)$, где стоит звездочка. Значение для клетки $(4, 4)$ равно 3, так как $S_1(4) = S_2(4) = i$ и $D(3, 3) = 3$

Анализ времени

Насколько трудоемок этот подход? Когда вычисляется значение для конкретной клетки (i, j) , проверяются только клетки $(i - 1, j - 1)$, $(i, j - 1)$ и $(i - 1, j)$, а также два символа — $S_1(i)$ и $S_2(j)$. Следовательно, при заполнении одной клетки выполняется постоянное число проверок клеток, арифметических операций и сравнений. Всего в таблице $O(nm)$ клеток, так что мы получаем следующую теорему.

Теорема 11.3.2. *Таблица динамического программирования для вычисления редакционного расстояния между строкой длины n и строкой длины m может быть заполнена с трудоемкостью $O(nm)$. Следовательно, используя динамическое программирование, редакционное расстояние $D(n, m)$ можно вычислить за время $O(nm)$.*

От читателя ожидается, что он установит, что таблицу можно заполнять и *по столбцам* (вместо строчек), т. е. заполнить сначала первый столбец, потом второй и т. д. Существует вариант заполнения по последовательным анти-диагоналям. Подробности оставляются как упражнение.

11.3.3. Обратный ход

Каким образом после вычисления редакционного расстояния, можно найти соответствующее оптимальное предписание? Простейший путь (по идеи) заключается в том, чтобы по мере вычисления значений в клетках получать *указатели*.

Именно, когда вычисляется значение в клетке (i, j) , нужно установить в ней указатель на $(i, j - 1)$ при $D(i, j) = D(i, j - 1) + 1$, на $(i, j - 1)$ при $D(i, j) = D(i - 1, j) + 1$ и на $(i - 1, j - 1)$ при $D(i, j) = D(i - 1, j - 1) + t(i, j)$. Это правило применяется также к клеткам в нулевой строчке и нулевом столбце. Следовательно, для большинства целевых функций каждая клетка в нулевой строчке указывает на клетку слева от нее, а каждая клетка в нулевом столбце — на клетку выше ее.*)

В остальных клетках может быть (и это типично) больше одного указателя, как, например, показано на рис. 11.3.

*). Очевидное исключение составляет клетка $(0, 0)$, не имеющая ссылок. — Прим. перев.

$D(i, j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	←1	←2	←3	←4	←5	←6	←7
v	1	↑1	↖1	↖2	↖3	↖4	↖5	↖6	↖7
i	2	↑2	↖12	↖2	↖2	↖3	↖4	↖5	↖6
n	3	↑3	↖13	↖13	↖13	↖3	↖4	↖5	↖6
t	4	↑4	↖14	↖14	↖14	↖3	↖4	↖5	↖6
n	5	↑5	↖15	↖15	↖15	↑4	↖4	↖5	↖6
e	6	↑6	↖16	↖16	↖16	↑5	↖4	↖5	↖6
r	7	↑7	↖17	↖6	↖17	↑6	↑5	↖4	←5

Рис. 11.3. Полная таблица динамического программирования с добавленными указателями. Стрелка \leftarrow в клетке (i, j) указывает на клетку $(i, j - 1)$, стрелка \uparrow — на клетку $(i - 1, j)$, а стрелка \nwarrow — на клетку $(i - 1, j - 1)$

Указатели позволяют легко найти оптимальное предписание: просто следовать по любому пути из указателей от клетки (n, m) до клетки $(0, 0)$. Редакционное предписание строится прохождением по этому пути с интерпретацией каждой горизонтальной дуги из клетки (i, j) в клетку $(i, j - 1)$ как *включения* (I) символа $S_2(j)$ в S_1 , каждой вертикальной дуги из (i, j) в $(i - 1, j)$ как *удаления* (D) $S_1(i)$ из S_1 и каждой диагональной дуги из (i, j) в $(i - 1, j - 1)$ как *совпадения* (M), если $S_1(i) = S_2(j)$, или как *замены* (R), если $S_1(i) \neq S_2(j)$. Доказательство того, что такой обратный путь задает оптимальное предписание, похоже на способ, которым были получены рекуррентные соотношения для редакционных расстояний. Мы оставляем это на упражнения.

В качестве альтернативы, в терминах *выравнивания* S_1 и S_2 , каждая горизонтальная дуга задает пробел, включаемый в S_1 , каждая вертикальная дуга — пробел, включаемый в S_2 , а каждая диагональная дуга — совпадение или несовпадение, в зависимости от конкретных символов.

Так, в примере, представленном на рис 11.3, имеется три пути от клетки $(7, 7)$ до клетки $(0, 0)$. Они одинаковы от клетки $(7, 7)$ до клетки $(3, 3)$, где можно пойти или наверх, или по диагонали. Соответствующие им оптимальные выравнивания таковы:

$$\begin{array}{ccccccccc} w & r & i & t & - & e & r & s \\ v & i & n & t & n & e & r & - \end{array}$$

$$\begin{array}{ccccccccc} w & r & i & - & t & - & e & r & s \\ v & - & i & n & t & n & e & r & - \end{array}$$

и

$$\begin{array}{ccccccccc} w & r & i & - & t & - & e & r & s \\ - & v & i & n & t & n & e & r & - \end{array}$$

Если в клетке (n, m) указателей больше одного, то путь из (n, m) в $(0, 0)$ может начаться с любого. Каждый из них лежит на пути из (n, m) в $(0, 0)$. Это свойство сохраняется и у всех проходимых клеток. Следовательно, обратный путь из (n, m) в $(0, 0)$ можно начать, выбрав произвольный указатель в клетке (n, m) , и продолжать в том же духе в каждой из проходимых клеток. Поскольку любая клетка, кроме $(0, 0)$,

имеет указатель, позволяющий из нее выйти, никакой путь из (n, m) не заведет в тупик. Любой путь из указателей от (n, m) до $(0, 0)$ определяет оптимальное редакционное предписание или выравнивание, и мы получаем следующую теорему.

Теорема 11.3.3. *Когда таблица динамического программирования с указателями уже построена, оптимальное редакционное предписание можно получить за время $O(n + m)$.*

Мы полностью описали три важные компоненты общей парадигмы динамического программирования на примере задачи о редакционном расстоянии. Позднее рассмотрим, как увеличить скорость решения и уменьшить потребность в памяти.

Указатели представляют все оптимальные предписания

Указатели, которые строятся при вычислении значений в таблице, позволяют найти все оптимальные предписания.

Теорема 11.3.4. *Любой путь от (n, m) до $(0, 0)$ по указателям, полученным при вычислении $D(i, j)$, определяет редакционное предписание с минимальным числом редакционных операций. И обратно. любое оптимальное редакционное предписание определяется таким путем. Более того, так как путь описывает только одно предписание, то соответствие между путями и оптимальными предписаниями одно-однозначное.*

Теорему можно доказать в основном теми же рассуждениями, из которых получается корректность рекуррентного соотношения для $D(i, j)$, и это оставляется читателю. Альтернативный путь нахождения оптимальных редакционных предписаний без использования указателей рассмотрен в упражнении 9. Когда указатели уже получены, все кооптимальные предписания можно перечислить за время $O(n + m)$ на предписание. Это будет темой упражнения 12.

11.4. Редакционные графы

Зачастую полезно представлять решения строковых задач методами динамического программирования в терминах *взвешенного редакционного графа*.

Определение. Пусть заданы строки S_1 и S_2 длиной n и m соответственно. Взвешенный редакционный граф имеет $(n + 1) \times (m + 1)$ вершин, все они помечены различными парами (i, j) ($0 \leq i \leq n$, $0 \leq j \leq m$). Набор дуг и их весов зависит от особенностей строковой задачи.

В случае задачи о редакционном расстоянии редакционный граф содержит дугу из каждой вершины (i, j) в каждую из вершин $(i, j + 1)$, $(i + 1, j)$ и $(i + 1, j + 1)$, если эти вершины существуют. Веса первых двух дуг равны 1; вес третьей (диагональной) дуги равен $t(i + 1, j + 1)$. На рис. 11.4 показан редакционный граф для строк CAN и ANN .

Основное свойство редакционного графа заключается в том, что любой *кратчайший* путь (полный вес которого минимален) от начальной вершины $(0, 0)$ до вершины назначения (n, m) определяет редакционное предписание с минимальным числом операций. Эквивалентно, любой кратчайший путь определяет глобальное выравнивание минимального полного веса. Более того, можно сформулировать теорему и следствие.

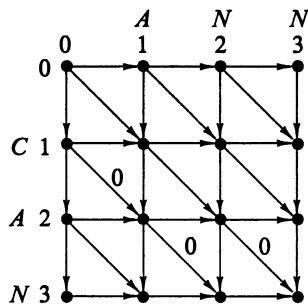


Рис. 11.4. Редакционный граф для строк *CAN* и *ANN*. Вес каждой дуги равен 1, кроме трех дуг нулевого веса, отмеченных на рисунке

Теорема 11.4.1. *Редакционное предписание для S_1 и S_2 имеет минимальное число операций в том и только том случае, если оно соответствует кратчайшему пути из $(0, 0)$ в (n, m) в редакционном графе.*

Следствие 11.4.1. *Множество всех кратчайших путей из $(0, 0)$ в (n, m) в редакционном графе в точности определяет множество всех оптимальных редакционных предписаний, преобразующих S_1 в S_2 . То есть оно определяет все оптимальные (имеющие минимальный вес) выравнивания S_1 и S_2 .*

Взгляд на динамическое программирование как на задачу о кратчайшем пути полезен ввиду того, что есть много средств исследования и компактного представления кратчайших путей в графах. Эта возможность используется в п. 13.2, где рассматриваются субоптимальные решения.

11.5. Взвешенное редакционное расстояние

11.5.1. Веса операций

Важным обобщением редакционного расстояния является допущение произвольного веса, или *стоимости*, или *балльной оценки* (*score*)^{*)}, приписываемых каждой редакционной операции, в том числе и совпадению. Таким образом, любое включение или удаление имеет вес, обозначаемый через d , замена имеет вес r , а совпадение — вес e (который обычно мал в сравнении с другими весами и часто равен нулю). Другими словами, *выравнивание с весами операций* имеет стоимость несовпадения r , стоимость совпадения e и стоимость пробела d .

Определение. При произвольных весах операций задачей об *операционно-взвешенном редакционном расстоянии* называется задача о поиске редакционного предписания, которое переводит строку S_1 в S_2 с минимальным полным весом операций.

^{*)} Термины “вес” и “стоимость” сильно загружены в литературе по информатике, а термин “оценка” используется в биологической литературе. При обсуждении алгоритмов мы будем использовать эти термины более или менее равноправно, но в специфических биологических приложениях будут использоваться “оценки”.

В этих терминах задача о редакционном расстоянии, которую мы рассматривали до сих пор, — это задача о нахождении предписания с минимальным весом операций, когда $d = 1$, $r = 1$ и $e = 0$. Но, например, если каждое несовпадение имеет вес 2, пробел — вес 4, а совпадение — вес 1, то выравнивание

$$\begin{array}{ccccccc} w & r & i & t & - & e & r & s \\ v & i & n & t & n & e & r & - \end{array}$$

имеет полный вес 17 и является оптимальным.

Ввиду того что целевая функция — минимизируемый полный вес, а замена может быть выполнена как удаление, за которым следует вставка, то, если допускаются замены, их вес должен быть меньше, чем сумма весов удаления и вставки.

Вычисление операционно-взвешенного редакционного расстояния

Задача об операционно-взвешенном редакционном расстоянии для двух строк длиной n и m может быть решена за время $O(nm)$. Нужно только немного модифицировать рекуррентные соотношения для редакционного расстояния. Пусть $D(i, j)$ обозначает минимум полного веса редакционных операций, переводящих $S_1[1..i]$ в $S_2[1..j]$. Мы снова используем обозначение $t(i, j)$, чтобы охватить им и замену и равенство, так что теперь $t(i, j)$ равно e при $S_1(i) = S_2(j)$ и r — в противном случае. Теперь базовыми условиями будут

$$D(i, 0) = i \times d$$

и

$$D(0, j) = j \times d.$$

Общее рекуррентное соотношение выглядит так:

$$D(i, j) = \min\{D(i, j - 1) + d, D(i - 1, j) + d, D(i - 1, j - 1) + t(i, j)\}.$$

Задача об операционно-взвешенном редакционном расстоянии также может быть представлена и решена как задача о кратчайшем пути на взвешенном редакционном графе, в котором веса дуг естественным образом соответствуют весам редакционных операций. Подробности очевидны и оставляются читателю.

11.5.2. Алфавитно-взвешенное редакционное расстояние

Другое важное обобщение редакционного расстояния получается, если допустить, что вес или оценка замены зависит от того, какая буква алфавита удаляется и какая добавляется. Например, может оказаться, что A дороже заменить на T , чем на G . Другими словами, мы можем захотеть, чтобы оценка удаления или вставки зависела от того, какая именно буква алфавита удаляется или вставляется. Мы назовем эту форму расстояния *алфавитно-взвешенным редакционным расстоянием*, чтобы отличить его от операционно-взвешенного расстояния.

Задача об операционно-взвешенном редакционном расстоянии является частным случаем задачи об алфавитно-взвешенном расстоянии. Модификация предыдущих рекуррентных соотношений (для операционно-взвешенного расстояния) на случай алфавитных весов тривиальна и оставляется на упражнения. Имея в виду алфавитные

веса, мы будем пользоваться простым термином *взвешенное редакционное расстояние*. Отметим, что во взвешенном редакционном расстоянии вес операции зависит от символов, участвующих в операции, но не от положения этих символов в строке.

При сравнении белков “редакционное расстояние” почти всегда означает алфавитно-взвешенное расстояние над алфавитом аминокислот. Существует общирная (и продолжающая прибывать) литература о том, какие оценки нужно выбирать для операций над символами аминокислот и как их нужно определять. Сейчас среди оценочных схем аминокислот доминируют РАМ-матрицы Дейхоф [122] и более новые оценочные BLOSUM-матрицы Хайнайкофов [222], хотя на самом деле эти матрицы определены в терминах задачи максимизации (сходства), а не редакционного расстояния.*). Недавно была разработана математическая теория [16, 262] того, как следует интерпретировать оценки и как оценочная схема может соотноситься с данными, из которых она получена, и с типами поисков, для которых она предназначена. Мы кратко обсудим этот вопрос еще раз в п. 15.11.2.

При сравнении строк ДНК чаще вычисляется редакционное расстояние невзвешенное или операционно-взвешенное. Например, популярная программа поиска в базах данных BLAST приписывает совпадениям оценку -5 , а несовпадениям -4 . Однако алфавитно-взвешенное редакционное расстояние также представляет интерес, и для ДНК предлагались алфавитные оценочные схемы (например, см. [252]).

11.6. Сходство строк

Редакционное расстояние — это один из способов формализации родственности двух строк. Альтернативным, и часто предпочитаемым, способом является измерение не их различия, а их *сходства*. Этот подход выбирается в большинстве биологических приложений по техническим причинам, которые станут ясны позднее. При ориентации на сходство язык выравниваний обычно становится удобнее, чем язык редакционных предписаний. Мы начнем с точного определения сходства.

Определение. Пусть Σ — алфавит, используемый для строк S_1 и S_2 , а Σ' — это Σ с добавленным символом пробела $_$. Тогда для любых двух символов $x, y \in \Sigma'$ через $s(x, y)$ обозначается значение (или *оценка*), полученное при выравнивании символа x напротив символа y .

Определение. Введем выравнивание \mathcal{A} строк S_1 и S_2 . Обозначим через S'_1 и S'_2 эти строки после вставки пробелов, а через l — (одинаковую) длину получившихся строк. Значение выравнивания \mathcal{A} определяется как $\sum_{i=1}^l s(S'_1(i), S'_2(i))$.

Это значит, что каждая позиция i в \mathcal{A} определяет пару противоположных друг другу символов алфавита Σ' , и значение \mathcal{A} получается суммированием значений этих пар.

*.) В чисто математических и вычислительных обсуждениях алфавитно-взвешенных расстояний мы бы предпочли для матрицы алфавитно-зависимых оценок замен общий термин “весовая матрица”. Однако для этих матриц молекулярные биологи используют термины “матрица замен аминокислот” или “матрица замен нуклеотидов”, а термин “весовая матрица” занят совсем другим объектом (см. п. 14.3.1). Поэтому, чтобы сохранить общность и оставаться в некоторой гармонии с молекулярно-биологической литературой, мы будем применять общий термин “оценочная матрица”.

Например, пусть $\Sigma = \{a, b, c, d\}$, а попарные оценки определены матрицей

s	a	b	c	d	$-$
a	1	-1	-2	0	-1
b		3	-2	-1	0
c			0	-4	-2
d				3	-1
$-$					0

Тогда выравнивание

$$\begin{array}{cccccc} c & a & c & - & d & b & d \\ & c & a & b & b & d & b & - \end{array}$$

имеет полное значение $0 + 1 - 2 + 0 + 3 + 3 - 1 = 4$.

В задачах сходства строк элементы оценочных матриц $s(x, y)$ обычно неотрицательны при совпадении символов x, y и отрицательны при несовпадении. При такой оценочной схеме ищут выравнивание с возможно *большим* значением. Такое выравнивание подчеркивает совпадения (или сходства) двух строк со штрафами за несовпадения и вставленные пробелы. Конечно, осмысленность получающегося выравнивания может сильно зависеть от используемой схемы оценки и от сопоставления оценок за совпадение с оценками за несовпадения и пробелы. Для белков и ДНК предлагались различные оценочные матрицы [81, 122, 127, 222, 252, 400], но ни одна конкретная схема не оказалась пригодной для всех приложений. Мы вернемся к этому вопросу в пп. 13.1, 15.7 и 15.10.

Определение. Пусть задана матрица оценок пар символов алфавита Σ' .

Сходство двух строк, S_1 и S_2 , определяется как максимальное полное значение выравнивания \mathcal{A} этих строк. Оно называется также *значением оптимального выравнивания* S_1 и S_2 .

Сходство строк очевидно родственно алфавитно-взвешенному редакционному расстоянию, и в зависимости от используемой конкретной оценочной матрицы можно преобразовывать одну задачу в другую. Важное различие между сходством и взвешенным редакционным расстоянием станет ясно в п. 11.7, после обсуждения локального выравнивания.

11.6.1. Вычисление сходства

Сходство двух строк, S_1 и S_2 , и соответствующее оптимальное выравнивание могут быть вычислены методом динамического программирования с помощью рекуррентных соотношений, которые теперь уже интуитивно вполне ясны.

Определение. $V(i, j)$ определяется как значение оптимального выравнивания префиксов $S_1[1..i]$ и $S_2[1..j]$.

Напомним, что черта ($_$) используется для изображения пробела, вставляемого в строку. Базовые условия таковы:

$$V(0, j) = \sum_{1 \leq k \leq j} s(_, S_2(k))$$

и

$$V(i, 0) = \sum_{1 \leq k \leq i} s(S_1(k), _)$$

Поскольку i и j строго положительны, общее рекуррентное соотношение выглядит так:

$$V(i, j) = \max \left\{ \begin{array}{l} V(i - 1, j - 1) + s(S_1(i), S_2(j)), \\ V(i - 1, j) + s(S_1(i), _), \\ V(i, j - 1) + s(_, S_2(j)) \end{array} \right\}.$$

Корректность этого рекуррентного соотношения устанавливается рассуждениями, сходными с теми, которые использовались в случае редакционного расстояния. Именно, при любом выравнивании \mathcal{A} имеется три возможности: символы $S_1(i)$ и $S_2(j)$ находятся в одной и той же позиции (напротив друг друга), $S_1(i)$ находится после $S_2(j)$ или $S_1(i)$ находится до $S_2(j)$. Проверка основана на анализе этих вариантов. Детали оставляются читателю.

Если S_1 и S_2 имеют, соответственно, длины n и m , то значение их оптимального выравнивания равно $V(n, m)$. Эта величина и вся таблица динамического программирования могут быть получены за время $O(nm)$, так как на каждую клетку требуется только по три сравнения и по две арифметические операции. Если при заполнении таблицы оставлять указатели, как это делалось при работе с редакционным расстоянием, то оптимальное выравнивание можно построить, пройдя по любому пути из указателей от клетки (n, m) до клетки $(0, 0)$. Таким образом, задача об оптимальном (глобальном) выравнивании может быть решена за время $O(nm)$, т. е. за то же время, что и редакционное расстояние.

11.6.2. Частные случаи сходства

За счет выбора подходящей оценочной схемы многие задачи можно моделировать как частные случаи оптимального выравнивания или сходства. К ним относится важная задача о наибольшей общей подпоследовательности.

Определение. В строке S подпоследовательность определяется как подмножество символов S , выстроенных в их исходном “относительном” порядке. Более формально, подпоследовательность строки S длины n определяется списком индексов $i_1 < i_2 < i_3 < \dots < i_k$ для некоторого $k \leq n$. Подпоследовательностью, заданной этим списком индексов, является строка $S(i_1)S(i_2)S(i_3)\dots S(i_k)$.

Еще раз подчеркнем, что подпоследовательность не обязана состоять из идущих подряд символов из S , тогда как символы подстроки обязаны идти подряд^{*)}. Конечно, подстрока удовлетворяет определению подпоследовательности. Например, *its* — это подпоследовательность строки *winters*, но не подстрока, тогда как *inter* — это подстрока, и подпоследовательность.

Определение. Общей подпоследовательностью двух заданных строк, S_1 и S_2 , называется подпоследовательность, которая появляется одновременно в S_1 и S_2 . Задача о наибольшей общей подпоследовательности заключается в поиске самой длинной подпоследовательности (*lcs*) S_1 и S_2 .

^{*)} Различие между подпоследовательностью и подстрокой часто теряется в биологических задачах. Но алгоритмы для подстрок часто совершенно отличны по духу и эффективности от алгоритмов для подпоследовательностей, так что различие очень важно.

Задача о *lcs* важна и сама по себе — мы обсудим некоторые ее применения и подходы к улучшению ее решения в п. 12.5. А сейчас покажем, что ее можно моделировать и решать как задачу об оптимальном выравнивании.

Теорема 11.6.1. *При оценочной схеме, которая дает 1 за каждое совпадение и 0 за несовпадение или пробел, совпадающие символы в выравнивании с максимальным значением образуют наибольшую общую подпоследовательность.*

Доказательство простое и оставляется читателю. Из теоремы следует, что наибольшая общая подпоследовательность строк длиной n и m может быть вычислена за время $O(nm)$.

Здесь мы встречаемся с одним из различий между задачами о подстроках и подпоследовательностях и видим, почему важно их четко разделять. В п. 7.4 было установлено, что наибольшую общую подстроку можно найти за время $O(n+m)$, тогда как найденная временная граница для нахождения наибольшей общей подпоследовательности равна $O(n \times m)$ (хотя она и будет несколько уменьшена). Это типично — задачи о подстроках и подпоследовательностях обычно решаются разными методами, и затраты времени и памяти на их решения различны.

11.6.3. Граф выравнивания для вычисления сходства

Как и в случае редакционного расстояния, вычисление сходства можно рассматривать как задачу о пути в ациклическом графе, называемом *графом выравнивания*. Здесь имеется ввиду тот же самый редакционный граф, рассматривавшийся ранее, но веса в нем другие: это значения выравнивания конкретных пар символов, один из которых может быть пробелом. Начальной вершиной в графе выравнивания снова будет вершина, соответствующая клетке $(0, 0)$, а вершина назначения соответствует клетке (n, m) таблицы динамического программирования, но оптимальное выравнивание идет по самому длинному пути от начала до конца, а не по кратчайшему. И вновь есть одно-однозначное соответствие между самыми длинными путями в графе выравнивания и оптимальными (по максимуму значения) выравниваниями. В общем случае самые длинные пути в графах вычислить непросто, но для ациклических графов самый длинный путь находится за время, пропорциональное числу дуг, с помощью варианта динамического программирования (что не удивительно). Следовательно, для графов выравнивания самые длинные пути можно найти за время $O(nm)$.

11.6.4. Вариант с бесплатными концевыми пробелами

Часто встречается вариант выравнивания строк, называемый *выравниванием с бесплатными концевыми пробелами*. В этом варианте любые пробелы в начале или конце выравнивания имеют нулевой вес, независимо от веса остальных пробелов. Например, в выравнивании

$$\begin{array}{ccccccccc} - & - & c & a & c & - & d & b & d \\ l & t & c & a & b & b & d & b & - \end{array}$$

бесплатны два пробела в левом конце выравнивания и один пробел в правом.

Бесплатность концевых пробелов в целевой функции побуждает одну строку выравниваться по внутренней части другой или суффикс одной строки — по префиксу

другой. Этот вариант предпочтителен, когда есть ощущение, что такие типы выравнивания отражают “подлинное” отношение этих двух строк. Без механизма, способствующего таким выравниваниям, оптимальное выравнивание могло бы иметь совершенно другую форму и не отражать желаемого отношения.

Один из примеров того, когда концевые пробелы должны быть бесплатны, — *дробовая сборка последовательности* (shotgun sequence assembly, см. пп. 16.14 и 16.15). В этой задаче имеется большое число частично перекрывающихся подстрок, полученных из многих копий одной первоначальной, но неизвестной строки; задача состоит в том, чтобы, используя сравнения пар подстрок, правильно получить исходную строку. Мало вероятно, чтобы две случайные подстроки из этого набора соседствовали в исходной строке, и это отражается в низкой цене выравнивания таких двух подстрок при бесплатных концевых пробелах. Но если две подстроки перекрываются в исходной строке, то суффикс “приличного размера” одной из них выравнивается с префиксом “приличного размера” другой с совсем небольшим числом пробелов и несовпадений (отражающим небольшой процент ошибок дешифровки). Такое перекрытие опознается высокой оценкой при взвешенном выравнивании с бесплатными концевыми пробелами. Аналогично может быть распознан и случай, когда одна подстрока содержит другую. Следовательно, процедура подбора кандидатур на соседство пар заключается в выравнивании всех пар подстрок с бесплатными концевыми пробелами; пары с высокими оценками считаются наилучшими кандидатами. Мы вернемся к дробовой сборке и продолжим это обсуждение в п. 16.14 части IV.

Чтобы реализовать бесплатность концевых пробелов при вычислении сходства, надо использовать рекуррентные соотношения для глобального выравнивания (где учитывались все пробелы), описанные на с. 282, но заменить базовые условия на $V(i, 0) = V(0, j) = 0$ для всех i и j . Эта замена обеспечит нужный эффект для пробелов левого конца. Затем таблица заполняется, как в случае глобального выравнивания. Однако, в отличие от него, значение оптимального выравнивания не обязательно находится в клетке (n, m) . Его еще нужно получить, выбрав наибольшее значение по всем клеткам в строчке n и столбце m . Клетки в строчке n соответствуют выравниванию, где значение выравнивания учитывает слагаемое от последнего символа строки S_1 , но не от символов S_2 справа от него, которые ставятся напротив бесплатных концевых пробелов. Клетки в столбце m заполняются аналогично. Ясно, что оптимальное выравнивание с бесплатными концевыми пробелами находится за время $O(nm)$ — такое же, как для глобального выравнивания.

11.6.5. Приближенные вхождения P в T

Рассмотрим теперь другой важный вариант глобального выравнивания.

Определение. Пусть задан параметр δ . Подстрока T' строки T называется *приближенным вхождением* P в том и только том случае, если оптимальное выравнивание P с T' имеет значение не больше δ .

Задача определения того, нет ли приближенного вхождения P в T , очень важна и естественно обобщает задачу о точном совпадении. Ее можно решить следующим образом: применить те же рекуррентные соотношения (см. с. 282), что и для глобального выравнивания P и T , и изменить только базовое условие для $V(0, j)$:

на $V(0, j) = 0$ для всех j . После этого заполняется таблица (с сохранением стандартных обратных ссылок). Используя этот вариант глобального выравнивания, можно доказать следующую теорему.

Теорема 11.6.2. *Приближенное вхождение P в T , кончающееся в позиции j из T , существует в том и только том случае, если $V(n, j) \geq \delta$. Более того, $T[k..j]$ является приближенным вхождением P в T в том и только том случае, если $V(n, j) \geq \delta$ и существует путь из обратных ссылок из клетки (n, j) в клетку $(0, k)$.*

Ясно, что эта таблица может быть заполнена за время $O(nt)$, но если нужно явно вывести все приближенные вхождения P в T , то времени $\Theta(nt)$ может оказаться недостаточно. Возможным компромиссом будет идентификация каждой позиции j из T , для которой $V(n, j) \geq \delta$, с последующим явным выводом в результат для такого j только кратчайшего приближенного вхождения P , заканчивающегося в позиции j . Подстрока T' получается проходом по обратным ссылкам из (n, j) до клетки в нулевой строчке, который нарушает связи предпочтением вертикального указателя диагональному, а диагонального — горизонтальному.

11.7. Локальное выравнивание: нахождение подстрок высокого сходства

Во многих приложениях две строки могут не слишком хорошо совпадать по всей своей длине, но содержать очень похожие участки. Задача состоит в нахождении и выделении пар участков, по одному из каждой строки, обладающих высоким сходством. Она называется задачей локального выравнивания или локального сходства и формально определяется ниже.

Задача локального выравнивания. Пусть заданы две строки, S_1 и S_2 . Требуется найти подстоки α и β из S_1 и S_2 соответственно, сходство которых (значение оптимального глобального выравнивания) максимально по всем парам подстрок из S_1 и S_2 ; обозначим его через v^* .

Например, рассмотрим строки $S_1 = pqraxabcstuvq$ и $S_2 = xyaxbacsl$. Если мы оценим совпадение в 2 балла, несовпадение в -2, а пробел в -1, то подстоки $\alpha = axabcs$ и $\beta = axbac$ из S_1 и S_2 соответственно будут иметь следующее оптимальное (глобальное) выравнивание:

$$\begin{array}{ccccccccc} a & x & a & b & - & c & s \\ a & x & - & b & a & c & s \end{array}$$

Значение выравнивания равно 8. Из всех выборов пар подстрок максимальным сходством (при выбранной системе оценки) обладает эта пара. Следовательно, для данной системы оценки оптимальное локальное выравнивание S_1 и S_2 имеет значение 8 и определяется подстоками $axabcs$ и $axbac$.

Должно быть ясно, почему локальное выравнивание определяется в терминах сходства, которое максимизирует целевую функцию, а не в терминах минимизирующего редакционного расстояния. Если бы искалась пара подстрок с минимальным

расстоянием, оптимальные пары при самых естественных системах оценок состояли бы из точно совпадающих подстрок. Но эти подстроки имели бы длину в один символ и не могли бы рассматриваться как участки высокого сходства. У задачи локального выравнивания, которая совпадение учитывает с плюсом, а несовпадение — с минусом, больше шансов найти нетривиальные участки высокого сходства.

Почему локальное выравнивание?

Обращение к глобальному выравниванию белковых последовательностей имеет смысл, если две строки входят в одно и то же семейство белков. Например, белок *цитохром c* имеет почти одинаковую длину во всех производящих его организмах, и можно ожидать соответствия цитохромов из двух разных видов по всей длине двух строк. То же верно для белков семейства *глобинов*, таких как *миоглобин* и *гемоглобин*. При попытках вывести историю эволюции из анализа сходств и различий белковой последовательности обычно сравнивают белковые последовательности из одного семейства белков, так что глобальное выравнивание в этих приложениях обычно оправданно и эффективно.

Однако во многих биологических задачах локальное сходство (локальное выравнивание) значительно более результативно, чем глобальное. Это, в частности, верно, когда сравниваются длинные фрагменты анонимных ДНК, так как только некоторые внутренние секции этих строк могут быть родственны. При сравнении белковых последовательностей локальное выравнивание также важно, так как белки из сильно различающихся семейств часто построены из одних и тех же структурных или функциональных субъединиц (мотивов или доменов) и локальное выравнивание хорошо работает при поиске этих (неизвестных) субъединиц. Аналогично, разные белки часто содержат родственные мотивы, которые образуют внутреннее ядро белка, но эти мотивы отделены друг от друга высокоспирализованными участками наружной поверхности, которые для разных белков могут значительно различаться.

Очень интересный пример консервативных доменов дают белки, закодированные *гомеобоксными* генами. Эти гены [319, 381] широко распространены, от дрозофилы до лягушки и человека. Они регулируют эмбриональное развитие, и одиночная мутация в них может преобразовать одну часть тела в другую (в одном из первоначальных экспериментов зародыш антенны плодовой мушки в результате мутации развился как конечность, но это, кажется, не очень беспокоило саму мушку). Аминокислотные последовательности, которые закодированы в этих генах, очень различаются у разных видов, кроме одного участка, называемого *гомеодоменом*. Гомеодомен состоит примерно из шестидесяти аминокислот, которые образуют часть регуляторного белка, который связывается с ДНК. Это странно, но гомеодомены у насекомых и млекопитающих особенно похожи, обнаруживая сходство от 50 до 95 % при выравниваниях без пробелов. Связывание белка с ДНК играет центральную роль в регуляции эмбрионального развития и клеточной дифференциации. Таким образом, последовательность аминокислот в наиболее биологически важной части этих белков высоко консервативна, тогда как сходство других их частей незначительно. В таких случаях, как эти, локальное выравнивание дает заведомо лучший способ сравнения аминокислотных последовательностей, чем глобальное.

Локальное выравнивание в белке имеет дополнительное значение потому, что конкретные изолированные символы родственных белков могут быть более высоко консервативны, чем оставшаяся часть белка (например, наиболее высоко консервативны аминокислоты в *активном центре* фермента или аминокислоты в *гидрофобном ядре* глобулярного белка). Вероятно, локальное выравнивание будет точнее определять эти консервативные символы, чем глобальное. Хороший пример дает семейство *сериновых протеаз*, которое характеризуется наличием нескольких изолированных, консервативных аминокислот. Другой пример — это мотив “спираль–поворот–спираль”, который часто встречается в белках, регулирующих транскрипцию ДНК присоединением к ДНК. Десятая позиция этого мотива очень часто представлена аминокислотой глицина, но остальная часть мотива весьма изменчива.

Следующая цитата из С. Чотиа еще сильнее подчеркивает биологическую значимость белковых доменов и, следовательно, локального сравнения строк:

“Имеющиеся белки построены из исходного набора не только при помощи точковых мутаций, вставок и удалений, но также при помощи комбинирования генов, дающего химерные белки. Это особенно верно в отношении очень крупных белков, произведенных на поздних стадиях эволюции. Многие из них построены из различных комбинаций белковых доменов, которые выбраны из относительно скромного репертуара” [101].

Дулиттл подытоживает:

“Подтекст сообщения состоит в том, что нужно беспокоиться об участках сходства, даже если они оказываются вложенными в окружение несходства” [129].

Таким образом, доминирующая сегодня точка зрения состоит в том, что локальное выравнивание является наиболее подходящим типом выравнивания для сравнения белков из различных семейств. Однако отмечалось также [359, 360], что часто находится широкое глобальное сходство пар белковых строк, у которых сначала было отмечено лишь существенное локальное сходство. Высказывались и предположения [316], что в некоторых ситуациях глобальное выравнивание более эффективно, чем локальное, в демонстрации важных биологических общностей.

11.7.1. Вычисление локального выравнивания

Почему бы не искать участки высокого сходства двух строк, выравнивая сначала эти строки глобально? На глобальное выравнивание двух длинных строк будут определенно влиять участки высокого сходства, и оптимальное глобальное выравнивание могло бы хорошо расположить соответствующие участки друг относительно друга. Но локальные участки высокого сходства теряются при оптимальном глобальном выравнивании. Поэтому оказывается эффективнее искать локальные участки высокого сходства непосредственно.

Мы покажем, что если длины строк S_1 и S_2 равны, соответственно, n и m , то задача локального выравнивания может быть решена за время $O(nm)$ — то же, что для глобального выравнивания. Такая эффективность неожиданна, так как имеется $\Theta(n^2m^2)$ пар строк, так что, даже если глобальное выравнивание могло бы быть вычислено для каждой выбранной пары за константное время, граница времени была бы $\Theta(n^2m^2)$. На самом же деле, если мы наивно примем $O(kl)$ как границу времени на выравнивание строк с длинами k и l , результирующая временная граница для задачи локального выравнивания будет $O(n^3m^3)$, вместо границы $O(nm)$, которую

мы получим. Временная граница $O(nm)$ была получена Темплом Смитом и Майклом Утерменом [411] с использованием алгоритма, который описан ниже.

В определении локального выравнивания, данном ранее, допускалась любая схема оценки глобального выравнивания двух выбранных строк. Одно небольшое ограничение поможет в вычислении локального выравнивания. Мы предположим, что глобальное выравнивание двух пустых строк имеет значение 0. Предположение используется для того, чтобы алгоритм локального выравнивания имел возможность выбрать в качестве α и β две пустые строки. Прежде чем описывать решение задачи локального выравнивания, будет полезно рассмотреть более ограниченную версию этой задачи.

Определение. Пусть задана пара индексов $i \leq n$ и $j \leq m$. Задача локального суффиксного выравнивания заключается в нахождении (возможно, пустых) суффиксов α строки $S_1[1..i]$ и β строки $S_2[1..j]$, для которых $V(\alpha, \beta)$ достигает максимума по всем парам суффиксов $S_1[1..i]$ и $S_2[1..j]$. Мы обозначим через $v(i, j)$ значение оптимального локального суффиксного выравнивания для данной индексной пары i, j .

Например, предположим, что целевая функция оценивает каждое совпадение в 2 балла, а каждое несовпадение или пробел — в -1. Если $S_1 = abcxdex$ и $S_2 = xxxcde$, то $v(3, 4) = 2$ (совпали два c), $v(4, 5) = 1$ (cx выстроилось по cd), $v(5, 5) = 3$ (x_d выстроилось по xcd) и $v(6, 6) = 5$ (x_de выстроилось по $xcde$).

Так как определение позволяет, чтобы один или оба суффикса были пустыми, $v(i, j)$ всегда неотрицательно.

Следующая теорема показывает соотношение между задачей локального выравнивания и задачей локального суффиксного выравнивания. Напомним, что v^* равно значению оптимального локального выравнивания для двух строк длин n и m .

Теорема 11.7.1. $v^* = \max\{v(i, j) : i \leq n, j \leq m\}$.

Доказательство. Очевидно, что $v^* \geq \max\{v(i, j) : i \leq n, j \leq m\}$, так как оптимальное решение задачи локального суффиксного выравнивания для любых i, j является допустимым решением задачи локального выравнивания. Обратно, пусть α, β — подстроки в оптимальном решении задачи локального выравнивания, и α кончается в позиции i^* , а β — в j^* . Тогда пара α, β определяет также локальное суффиксное выравнивание для пары индексов i^*, j^* , и поэтому $v^* \leq v(i^*, j^*) \leq \max\{v(i, j) : i \leq n, j \leq m\}$, так что оба направления соотношения теоремы получены. \square

Теорема 11.7.1 задает только значение v^* , но из ее доказательства ясно, как найти подстроки, на выравнивании которых это значение достигается. Именно:

Теорема 11.7.2. Если i', j' — пара индексов, на которой достигается максимум $v(i, j)$ по всем парам i, j , то пара подстрок, решающих задачу локального суффиксного выравнивания для i', j' , решает также задачу локального выравнивания.

Таким образом, решение задачи локального суффиксного выравнивания решает задачу локального выравнивания. Обратимся к задаче нахождения $\max\{v(i, j) : i \leq n, j \leq m\}$ и пары строк, выравнивание которых имеет максимальное значение.

11.7.2. Как решить задачу локального суффиксного выравнивания

Прежде всего, $v(i, 0) = 0$ и $v(0, j) = 0$ для всех i и j , так как мы всегда можем выбрать пустой суффикс.

Теорема 11.7.3. При $i > 0$ и $j > 0$ рекуррентное соотношение для $v(i, j)$ такое

$$v(i, j) = \max \left\{ \begin{array}{l} 0, \\ v(i - 1, j - 1) + s(S_1(i), S_2(j)), \\ v(i - 1, j) + s(S_1(i), _), \\ v(i, j - 1) + s(_, S_2(j)) \end{array} \right\}.$$

Доказательство. Аргументация похожа на обоснования предыдущих рекуррентных соотношений. Пусть α и β — подстроки S_1 и S_2 , глобальные выравнивания которых дают оптимальное локальное выравнивание. Так как строки α и β могут быть пустыми суффиксами $S_1[1..i]$ и $S_2[1..j]$, то включение 0 в качестве *кандидата* на значение $v(i, j)$ возможно. Однако если оптимальная подстрока α непуста, то символ $S_1(i)$ должен быть выровнен либо с пробелом, либо с символом $S_2(j)$. Аналогично, если непуста оптимальная β , то символ $S_2(j)$ выровнен либо с пробелом, либо с $S_1(i)$. Это обосновывает рекуррентное соотношение, которое сравнивает способы сопоставления символов $S_1(i)$ и $S_2(j)$ в оптимальном локальном суффиксном выравнивании для i, j .

Если символ $S_1(i)$ выстроен с $S_2(j)$, то вклад этих двух символов в $v(i, j)$ равен $s(S_1(i), S_2(j))$, и оставшаяся часть $v(i, j)$ определяется локальным суффиксным выравниванием для индексов $i - 1, j - 1$. Это выравнивание должно быть оптимальным и, стало быть, имеет значение $v(i - 1, j - 1)$. Поэтому если $S_1(i)$ и $S_2(j)$ выровнены друг с другом, то $v(i, j) = v(i - 1, j - 1) + s(S_1(i), S_2(j))$.

Если $S_1(i)$ выровнено с пробелом, то, рассуждая аналогично, видим, что $v(i, j) = v(i - 1, j) + s(S_1(i), _)$, а если $S_2(j)$ выровнено с пробелом, то $v(i, j) = v(i, j - 1) + s(_, S_2(j))$. Так как все случаи исчерпаны, мы доказали, что $v(i, j)$ должно быть нулем либо совпадать с одним из трех остальных членов в рекуррентном соотношении.

С другой стороны, для каждого из четырех членов в рекуррентном соотношении имеется такой способ выбора суффиксов $S_1[1..i]$ и $S_2[1..j]$, что выравнивание этих двух суффиксов имеет значение, даваемое соответствующим членом. Следовательно, значение оптимального суффиксного выравнивания не меньше, чем максимум из этих четырех членов. Доказав, что $v(i, j)$ должно совпадать с одним из них и не превосходит максимального, мы установили, что значение $v(i, j)$ должно равняться максимальному, что и доказывает теорему. \square

Рекуррентные соотношения для локального суффиксного выравнивания почти идентичны соотношениям для глобального выравнивания, разница только во включении нуля. Это интуитивно ясно. И в глобальном, и в локальном суффиксном выравнивании префиксов $S_1[1..i]$ и $S_2[1..j]$ разбираются случаи с концевыми символами, но в случае локального суффиксного выравнивания любое число начальных символов можно игнорировать. Нуль в рекуррентии реализует это, выполняя ее “перезапуск”.

С учетом теоремы 11.7.2 метод вычисления v^* заключается в вычислении таблицы динамического программирования для $v(i, j)$ и последующем нахождении наибольшего из значений в клетках таблицы, скажем, в клетке (i^*, j^*) . Как обычно, при

заполнении таблицы создаются указатели. После нахождения клетки (i^*, j^*) подстроки α и β , дающие оптимальное локальное выравнивание S_1 и S_2 , находятся обратной трассировкой по указателям от клетки (i^*, j^*) до (i', j') , в котором достигается нульевое значение. Теперь подстроками оптимального локального выравнивания будут $\alpha = S_1[i'..i^*]$ и $\beta = S_2[j'..j^*]$.

Временной анализ

Так как для вычисления $v(i, j)$ в одной клетке требуется только четыре сравнения и три арифметических операции, то трудоемкость заполнения всей таблицы оценивается временем $O(nm)$. Поиск v^* и обратный проход тоже занимают только время $O(nm)$, так что мы получили следующую теорему.

Теорема 11.7.4. Для двух строк длины n и m задача локального выравнивания может быть решена за то же время $O(nm)$, что и глобальное выравнивание.

Напомним, что указатели в таблице динамического программирования для редакционного расстояния, глобального выравнивания и сходства кодируют все оптимальные выравнивания. Аналогично, указатели в таблице для локального выравнивания кодируют оптимальные локальные выравнивания следующим образом.

Теорема 11.7.5. Все оптимальные локальные выравнивания двух строк представлены в таблице динамического программирования для $v(i, j)$ и могут быть найдены обратным трассированием по указателям от любой клетки со значением v^* .

Мы оставляем это доказательство как упражнение.

11.7.3. Три заключительных замечания о локальном выравнивании

Терминология для локального и глобального выравнивания

В биологической литературе на глобальное выравнивание (сходство) часто ссылаются как на выравнивание Нидлмана–Вунча [347], по имени авторов, которые рассматривали глобальное сходство. На локальное выравнивание часто ссылаются как на выравнивание Смита–Уотермена [411], по имени авторов, которые это локальное выравнивание ввели. Есть, однако, некоторая путаница в литературе между “Нидлманом–Вунчом” и “Смитом–Уотерменом” в связи с постановкой задач и методами их решения. Исходное решение, данное Нидлманом–Вунчом, считает за кубическое время и редко используется. Следовательно, “Нидлман–Вунч” обычно относится к задаче глобального выравнивания. Метод Смита–Уотермена считает за квадратичное время и используется повсеместно, так что на “Смита–Уотермена” часто ссылаются и в связи с их методом решения, и в связи с постановкой задачи. Но есть методы решения для задачи локального выравнивания (Смита–Уотермена), отличающиеся от метода Смита–Уотермена, но на которые иногда также ссылаются как на “Смита–Уотермена”.

Нахождение нескольких участков высокого сходства методом Смита–Уотермена

Очень часто в биологических приложениях недостаточно найти только пару подстрок исходных строк S_1 и S_2 с оптимальным локальным выравниванием. Скорее можно сказать, что требуется найти все или “многие” пары подстрок, имеющие сходство

выше некоторого порогового значения. Конкретное приложение такого рода будет рассматриваться в п. 18.2, а более общая задача будет глубоко изучаться в п. 13.2. Здесь мы просто отмечаем, что на практике таблица динамического программирования, используемая для решения задачи локального суффиксного выравнивания, часто применяется для нахождения дополнительных пар подстрок с “высоким” сходством. Главное, что для любой клетки таблицы (i, j) можно найти пару подстрок S_1 и S_2 (обратным проходом) со сходством (значением глобального выравнивания), равным $v(i, j)$. Простой путь получения набора высоко схожих подстрок заключается в том, чтобы найти в таблице множество клеток со значением выше некоторого порога. Таким способом можно получить не все схожие подстроки, но этот подход общеупотребителен.

Потребность в хороших схемах оценки

Результат оптимального локального выравнивания определяется используемой схемой оценки. Например, если совпадения оценивать единицей, а несовпадения и пробелы — нулем, то оптимальное локальное выравнивание будет определяться наибольшей общей подпоследовательностью. Обратно, если несовпадениям и пробелам сопоставить большие штрафные (отрицательные) оценки, а каждое совпадение оценивать единицей, то оптимальное локальное выравнивание будет наибольшей общей подстрокой. В большинстве случаев ни одна из этих двух схем интереса не представляет, и следует специально позаботиться о том, чтобы обеспечить схему оценки, приемлемую для конкретного приложения и дающую осмысленные с его точки зрения локальные выравнивания. Для локального выравнивания элементы матрицы оценок должны иметь отрицательное среднее. В противном случае результирующее “локальное” оптимальное выравнивание стремится стать глобальным. Недавно несколько авторов разработали элегантную теорию того, что значит схемы оценки локальных выравниваний в контексте поиска в базах данных и как схемы оценок получать. Мы вкратце обсудим эту теорию в п. 15.11.2.

11.8. Пропуски

11.8.1. Что такое пропуски

До этого момента основными конструктами, использованными при измерении значения выравнивания (и для определения сходства), были *совпадения, несовпадения и пробелы*. Сейчас мы введем еще один важный конструкт — а именно *пропуски*. Пропуски помогут создавать выравнивания, которые лучше согласуются с подразумеваемыми биологическими моделями и более полно соответствуют образцам, которые хочется найти в осмысленных выравниваниях.

Определение. Пропуском назовем любой *максимальный последовательный ряд* пробелов в *одной* строке данного выравнивания.*)

*) Иногда в биологической литературе термин “пробел” (space), как мы его используем, не употребляется, а термин “пропуск” или “гэп” (gap) применяется и для “пробела” и для “пропуска”, как мы его здесь определили. Это может привести к большой путанице, — в нашей книге термины “пропуск” и “пробел” имеют различный смысл.

Пропуск может начаться до начала S , и в этом случае он ограничивается слева первым символом S , или закончиться после конца S , и в этом случае он ограничивается справа последним символом S . В противном случае пропуск должен ограничиваться символами из S . Пропуск может состоять из единственного пробела. В качестве примера пропусков рассмотрим выравнивание на рис. 11.5, где четыре пропуска содержат в общей сложности семь пробелов. Это выравнивание имеет в своем составе пять совпадений, одно несовпадение, четыре пропуска и семь пробелов. Отметим, что за последним пробелом в первой строке идет пробел во второй строке, но эти два пробела составляют два пропуска, а не один.

c	t	t	t	a	a	c	—	—	a	—	a	c
c	—	—	—	c	a	c	c	c	a	t	—	c

Рис. 11.5. Выравнивание с семью пробелами, распределенными по четырем пропускам

Введение в целевую функцию члена, отражающего количество пропусков в выравнивании, может оказать некоторое влияние на *распределение* пробелов в выравнивании и, следовательно, на всю форму выравнивания. В простейшей целевой функции, включающей пропуски, каждый пропуск добавляет постоянный вес W_g , не зависящий от длины пропуска. Это значит, что каждый индивидуальный пробел бесплатен, так что $s(x, _) = s(_, x) = 0$ для любого символа x . Используя нотацию, введенную в п. 11.6 (с. 281), запишем значение выравнивания, содержащего k пропусков, как

$$\sum_{i=1}^l s(S'_1(i), S'_2(i)) - k W_g.$$

Изменение значения W_g относительно других весов в целевой функции может изменить расположение пробелов в оптимальном выравнивании. Большое W_g заставит выравнивание ограничиться несколькими пропусками, и выстроенные куски двух строк распадутся на несколько подстрок. Меньшее W_g разрешит более фрагментированные выравнивания. Влияние W_g на выравнивание будет глубже обсуждаться в п. 13.1.

11.8.2. Почему пропуски?

Большинство биологических обоснований важности локального выравнивания (см. п. 11.7) применимо и для обоснования пропуска как явной концепции при выравнивании строк.

Точно так же, как пробел в выравнивании соответствует вставке или удалению отдельного символа редакционным предписанием, пропуск в строке S_2 напротив подстроки α в строке S_1 соответствует либо удалению α из S_1 , либо вставке α в S_2 . Концепция пропуска в выравнивании важна поэтому во многих биологических приложениях, так как вставка или удаление целой подстроки (особенно в ДНК) часто появляется как отдельное мутационное событие. Более того, многие из этих единичных мутационных событий могут давать пропуски сильно различающегося размера с почти равным правдоподобием (в широком, но ограниченном диапазоне значений).

Многие из повторяющихся ДНК, рассмотренных в п. 7.11.1, появились вследствие единичных мутаций, в результате которых копировались и вставлялись длинные куски ДНК. Другие мутационные механизмы, вызывающие длинные вставки и удаления в ДНК, включают: *неравный кроссинговер* в мейозе (включение фрагмента в одну строку и эквивалентное удаление из другой); *проскальзывание ДНК* во время репликации (когда участок ДНК повторяется в реплицируемой копии из-за того, что система теряет текущее место репликации в образце, соскальзывает назад и повторяет секцию); включение в строку ДНК *переместимых элементов* (прыгающих генов); включение ДНК *ретровирусами* и *транслокации* ДНК между хромосомами [301, 317]. Пример пропусков в геномной последовательности см. на рис. 11.6.

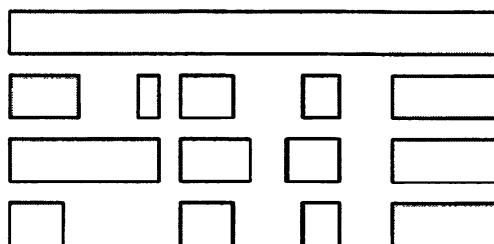


Рис. 11.6. Каждая из четырех строчек представляет часть последовательности РНК одного из штаммов вируса HIV-1 (ВИЧ). Вирус ВИЧ быстро мутирует, так что мутации можно наблюдать и прослеживать. Три нижние строчки взяты из штаммов вируса, которые мутировали из первоначального штамма (верхняя строчка). Каждая из трех последовательностей выровнена по верхней последовательности. Каждый темный прямоугольник представляет подстроку, совпадающую с соответствующей подстрокой в верхней последовательности, а каждое белое место соответствует пропуску, получающемуся из-за удаления известной последовательности. Рисунок взят с некоторыми изменениями из [123].

При вычислении выравниваний для изучения и эволюции за долгий период времени часто бывает, что именно пропуски в выравниваниях наиболее информативны. В строках ДНК замены отдельных символов благодаря точковым мутациям случаются повсеместно и обычно гораздо чаще, чем происходят нефатальные мутационные события, вызывающие пропуски. Аналогичные гены (определяющие “один и тот же” белок) в двух видах могут, таким образом, сильно различаться на уровне последовательности ДНК, затрудняя восстановление эволюционных связей на основе сходства строк (без учета пропусков). Но большие включения и удаления в молекулах, которые в выравниваниях отображаются как пропуски, встречаются гораздо реже, чем замены. Поэтому пропуски в паре выровненных строк иногда могут дать ключ к общей эволюционной истории набора строк [45, 405]. Далее, в п. 17.3.2, мы увидим, что такие пропуски можно рассматривать как эволюционные символы в некоторых подходах к построению эволюционных деревьев.

Напомним, что многие белки “построены из различных комбинаций доменов, выбранных из относительно скромного репертуара” [101]. Следовательно, две белковые строки могут быть относительно похожи на некоторых участках, но отличаться там, где одна строка содержит белковый домен, которого нет у другой. Такой интервал при выравнивании двух белков наиболее естественно изображается как пропуск.

В некоторых контекстах многие биологи рассматривают правильную классификацию главных (длинных) пропусков как *самую* существенную задачу выравнивания белков. Если длинные (главные) пропуски выбраны правильно, то остальную часть выравнивания — отражение точковых мутаций — выполнить относительно легко.

Выравнивание двух строк должно отражать стоимость (или правдоподобие) мутационных событий, нужных для преобразования одной строки в другую. Так как пропуск в более чем один пробел может возникнуть в результате единичного мутационного события, модель выравнивания должна отражать истинное распределение пробелов в пропусках, а не просто число пробелов в выравнивании. Отсюда следует, что модель должна определить, какие веса задавать пропускам, чтобы отразить их биологический смысл. В этой главе мы обсудим различные предлагаемые схемы оценки пропусков, а в последующих главах — некоторые дополнительные аспекты этой оценки. Сначала рассмотрим конкретный пример, показывающий, как применяется концепция пропуска.

11.8.3. Сравнение кДНК: конкретная иллюстрация

Пример использования пропусков в модели выравнивания возникает в связи с задачей *сравнения кДНК*. В этой задаче одна строка существенно длиннее другой и выравнивание, лучше всего отражающее их соотношение, должно состоять из нескольких участков очень высокого сходства, перемежающихся “длинными” пропусками в более короткой строке (рис. 11.7). Отметим, что участки совпадения могут иметь несовпадения и пробелы, но их процент в участке сходства должен быть мал.

Длинная строка

Набор коротких строк, перемежающихся пропусками

Рис. 11.7. При сравнении кДНК ожидается, что выравнивание меньшей строки с большей состоит из нескольких областей очень высокого сходства, перемежающихся относительно длинными пропусками

Биологическая постановка задачи

У эукариот ген, который кодирует белок, обычно построен из чередующихся экзонов (экспрессируемые последовательности — *expressed sequences*), которые, собственно, и кодируют белок, и инtronов (перемежающие последовательности — *intervening sequences*), которые в коде не участвуют. Число экзонов (а следовательно, и инtronов) обычно невелико (скажем, от четырех до двадцати), но длины инtronов могут быть громадными по сравнению с длинами экзонов.

На очень грубом уровне белок, кодируемый эукариотическим геном, синтезируется по следующей схеме. Сначала на матрице ДНК гена транскрибируется молекула РНК. При этом соблюдается условие комплементарности, т. е. каждому *A* из гена соответствует *U* (урасил) в РНК, каждому *T* — *A*, каждому *C* — *G* и каждому *G* — *C*. Образующаяся в результате транскрипции РНК покрывает весь ген, как инtronы, так и экзоны. Затем процессом, который еще не вполне понятен, в первичном транскрипте находятся границы инtronов и экзонов, и РНК, соответствующая инtronам,

вырезается (или “отшнуровывается” — *snurped* — молекулярным комплексом, именуемым *snrp* [420]), а участки РНК, соответствующие экзонам, сцепляются. Мы не будем описывать происходящие здесь некоторые дополнительные действия. Получающаяся молекула РНК называется *информационной* или *матричной РНК* (*iРНК* или *mРНК*, английский термин — *messenger* или *tRNA*); она покидает ядро клетки и используется для синтеза закодированного в ней белка.

Каждая клетка (обычно) содержит копию всех хромосом и, следовательно, всех генов целого организма, хотя в каждой специализированной клетке (клетке печени, например) экспрессируется только малая доля этих генов. Это значит, что только малая доля белков, закодированных в геноме, на самом деле производится в специализированной клетке. Стандартный метод для определения того, какие белки экспрессируются в специализированной клеточной линии, и для локализации кодирующих генов включает в себя очистку *иРНК* из цитоплазмы клетки, после того как она покидает ядро клетки. Затем эта *иРНК* используется для создания комплементарной строки ДНК. Эта строка называется *кДНК* (комплементарная ДНК). По сравнению с исходным геном строка *кДНК* состоит только из его сцепленных экзонов.

Очистка *иРНК* и составление библиотек *кДНК* (полных коллекций *иРНК* клетки) для конкретных клеточных линий стала рутинной работой. По мере составления все большего числа библиотек накапливаются транскрипты всех генов генома и систематика клеток, в которых эти гены экспрессируются. На самом деле основной компонентой проекта “Геном человека” [111, 399] и является получение *кДНК*, отражающих большинство генов в человеческом геноме. Эта деятельность проводится также несколькими частными компаниями и уже вызвала несколько интересных диспутов относительно патентования последовательностей *кДНК*.

После получения *кДНК* задачей становится обнаружение того, где находится ген, ассоциированный с этой *кДНК*. В настоящее время эта задача чаще всего решается лабораторными методами. Однако если *кДНК* дешифрована полностью или частично (а в проекте “Геном человека”, например, есть намерение дешифровать части каждой из полученных *кДНК*) и если уже расшифрована часть генома, содержащая ген, ассоциированный с данной *кДНК* (как, например, было бы после расшифровки всего генома), то задача нахождения положения гена, заданного последовательностью *кДНК*, становится строковой задачей. Она превращается в задачу выравнивания строки *кДНК* и более длинной строки расшифрованной ДНК методом, которым находят экзоны. Она превращается в задачу совпадения *кДНК*, рассмотренную выше.

Почему в целевой функции нужно учитывать пропуски

Если целевая функция включает в себя только члены, зависящие от совпадений, несовпадений и пробелов, то, по-видимому, нет способа добиться желаемой формы оптимального выравнивания. Попробуем объяснить, почему это так.

Конечно, вам не захочется устанавливать большой штраф за пробелы, так как от этого вся строка *кДНК* будет выравниваться более плотно и пропуски в выравнивании, соответствующие длинным инtronам, не будут поощряться. Вы пожелаете назначить более высокий штраф за несовпадения. Хотя в данных может быть небольшое количество ошибок расшифровки, от чего несовпадения могут возникать даже при полном соответствии *кДНК* экзонам, тем не менее большого процента несовпадений быть не должно. В итоге вы хотите малых штрафов за пробелы, относительно больших штрафов за несовпадения и положительных оценок за совпадения.

От какого типа выравнивания можно ждать результата, если целевая функция имеет низкий штраф за пробелы, высокий штраф за несовпадения, конечно же, положительный эффект от совпадения и никакого эффекта от пропусков? Помните, что длинная строка содержит больше чем один ген, что экзоны разделены длинными инtronами и что алфавит ДНК состоит всего из четырех букв, представленных в приблизительно равных количествах. При этих условиях оптимальное выравнивание было бы, вероятно, *наибольшей общей подпоследовательностью* короткой строки кДНК и длинной анонимной строки ДНК. И так как интроны длинные, а алфавит ДНК состоит всего из четырех символов, то общая подпоследовательность, скорее всего, совпадет по всем символам кДНК. Более того, ввиду малой, но вполне реальной доли ошибок расшифровки истинное выравнивание кДНК и ее экзонов не должно совпасть по всем символам. Следовательно, наибольшая общая подпоследовательность, скорее всего, будет иметь более высокую оценку, чем правильное выравнивание кДНК с экзонами. Но наибольшая общая последовательность раздробила бы строку кДНК вдоль более длинной ДНК и не дала бы выравнивания желаемой формы — оно не отметило бы ее экзонов.

Добавление в целевую функцию стагаемого для пропусков проясняет задачу. Добавляя постоянный вес W_g для каждого пропуска в выравнивании и выбирая для W_g подходящее значение (экспериментируя с различными значениями W_g), мы можем заставить оптимальное выравнивание разрезать кДНК так, чтобы она совпадала с экзонами в более длинной строке.*¹) Как и раньше, штраф за пробел берется нулевым, совпадение оценивается положительно, а штраф за несовпадение высок.

Обработанные псевдогены

Более трудный вариант задачи о совпадении кДНК встречается при поиске анонимной ДНК для *обработанных псевдогенов*. Псевдоген — это практически копия работающего гена, который мутировал достаточно далеко от оригинала, чтобы перестать выполнять свою функцию. Псевдогены очень распространены у эукариот и могут играть важную эволюционную роль, обеспечивая готовый запас разнообразных “почти генов”. Следуя тому взгляду, что новые гены созданы в процессе *дупликации с модификацией* существующих генов [127, 128, 130], можно сказать, что псевдогены представляют собой либо неудачные экспериментальные гены, либо будущие гены, которые будут функционировать после дополнительных мутаций.

Псевдоген может быть размещен очень далеко от гена, которому он соответствует, даже совсем на другой хромосоме, но он обычно содержит и интроны и экзоны, полученные от его работающего родственника. Поэтому задача нахождения псевдогенов в анонимной дешифрованной ДНК родственна задаче нахождения повторяющихся подстрок в очень длинной строке.

Более интересный тип псевдогена, *обработанный псевдоген*, содержит только экзонные подстроки из своего исходного гена. Подобно кДНК, интроны удалены, а экзоны склеены. Думается, что обработанный псевдоген возникает как *иРНК*, которая транскрибируется обратно в ДНК (посредством фермента обратной транскриптазы) и вставляется в геном в случайном месте.

Пусть теперь задана длинная строка анонимной ДНК, в которой могут содержаться и обработанный псевдоген, и его работающий предшественник. Спрашивается,

*¹) Этот прием действительно работает, и опровергнуть его практически — очень важное упражнение

как можно найти обработанные псевдогены? Задача похожа на задачу о совпадениях кДНК, но труднее, так как у нас нет на руках кДНК. Мы оставим читателю рассмотрение методов поиска повторов, локального выравнивания и выбора весов пропусков применительно к этой задаче.

Caveat

Задачи проверки совпадения кДНК и псевдогена иллюстрируют эффективность включения пропусков в выравнивание целевой функции и важность надлежащей оценки пропусков. Следует, однако, отметить, что на практике к этим задачам о совпадении можно подойти с благоразумным использованием локального выравнивания без пропусков. Идея в том, чтобы воспользоваться возможностью при вычислении локального выравнивания находить не только самые похожие, но и многие другие сильно похожие пары подстрок (см. пп. 13.2.4 и 11.7.3). В контексте проверки совпадения кДНК и псевдогена эти пары весьма правдоподобно будут экзонами, и таким образом, необходимое сопоставление кДНК с экзонами можно составить из ряда неперекрывающихся локальных выравниваний. Для практики наиболее характерен именно такой подход.

11.8.4. Выбор весов пропусков

Как показывает пример сопоставления с кДНК, правильное использование пропусков в целевой функции помогает найти выравнивания ожидаемого типа. Однако ясно, что способ оценки пропусков существенно влияет на эффективность концепции пропусков. Мы детально разберем четыре общих типа весов для пропусков: *постоянный, аффинный, вогнутый и произвольный*.

Простейшим выбором веса пропуска является *постоянный*. Этот тип был введен раньше, в нем каждый отдельный пробел бесплатен, а каждый пропуск имеет вес W_g независимо от числа пробелов в нем. Обозначая через W_m и W_{ms} веса совпадений и несовпадений соответственно, мы можем представить вариант задачи с пооперационным весом в следующем виде:

Найти выравнивание A , максимизирующее $W_m N_m - W_{ms} N_{ms} - W_g N_g$, где N_m — число совпадений в A , N_{ms} — число несовпадений, N_g — число пропусков.

В более общем виде, позволяющем учесть алфавитно-зависимые веса для совпадений и несовпадений, целевая функция при постоянном весе пропуска выглядит так:

Найти выравнивание A , максимизирующее $(\sum_{i=1}^l s(S'_1(i), S'_2(i)) - W_g N_g)$, где $s(x, _) = s(_, x) = 0$ для каждого символа x , и S'_1 и S'_2 представляют собой строки S_1 и S_2 после вставки пробелов.

Обобщение модели с постоянным весом пропуска заключается в добавлении веса W_s для каждого пробела в пропуске. В этом случае W_g называется *стартовым весом пропуска* и представляет стоимость инициализации пропуска, а W_s называется *весом приращения пропуска*, соответствуя стоимости увеличения пропуска на один пробел. Тогда вариант задачи с пооперационным весом становится таким:

Найти выравнивание, максимизирующее $W_m N_m - W_{ms} N_{ms} - W_g N_g - W_s N_s$, где N_s — число пробелов.

Эта модель называется *аффинным весом пропуска*^{*)}, так как вес одного пропуска длины q определяется аффинной функцией $W_g + qW_s$. Постоянный вес пропуска соответствует аффинной модели с $W_s = 0$.

Алфавитно-весовой вариант модели с аффинным весом пропуска опять-таки полагает $s(x, _) = s(_, x) = 0$, и в нем требуется:

$$\text{максимизировать } \left(\sum_{i=1}^l s(S'_1(i), S'_2(i)) - W_g N_g - W_s N_s \right).$$

Модель с аффинным весом пропуска является, вероятно, наиболее используемой в литературе, хотя имеются существенные разногласия по поводу выбора W_g и W_s [161] (в добавление к вопросам относительно W_m и W_{ms}). Для выравнивания аминокислотных строк широко используемая поисковая программа FASTA [359] берет по умолчанию значения $W_g = 10$ и $W_s = 2$. Мы вернемся к вопросу о выборе этих установок в п. 13.1.

Высказывалось мнение [57, 183, 466], что некоторые биологические феномены моделируются лучше весовой функцией пропуска, в которой каждый дополнительный пробел пропуска добавляет к целевой функции меньше, чем предыдущий (функция с отрицательной второй производной). Другими словами, вес пропуска является *вогнутой* функцией^{**)} от ее длины. Примером такой функции может служить $W_g + \ln q$, где q — длина пропуска. Некоторые биологи считают, что функция пропуска, которая сначала возрастает до максимального значения, а затем убывает почти до нуля, отражала бы *сочетание* различных биологических феноменов включения и удаления ДНК.

Наконец, наиболее общим видом оценочной функции для пропуска из рассматриваемых нами является *произвольный вес пропуска* в виде произвольной функции $w(q)$ от длины пропуска q . Константа, аффинная и вогнутая функции являются, конечно, ее частными случаями.

Временные оценки при различных оценках пропуска

Как можно было бы ожидать, время, необходимое для нахождения оптимального выравнивания, при произвольной оценке пропуска больше, чем в других моделях. В случае, когда $w(q)$ — полностью произвольная функция длины пропуска, оптимальное выравнивание можно найти за время $O(nm^2 + n^2m)$, где n и $m \geq n$ — длины строк. Для случая вогнутой $w(q)$ мы покажем, что время можно уменьшить до $O(nm \log m)$ (возможно и дальнейшее уменьшение, но алгоритм становится слишком сложным для наших целей). В аффинном случае (и следовательно, в случае константы) временная оценка равна $O(nm)$, как и в случае выравнивания без учета пропусков. В последующих пунктах мы сначала обсудим выравнивание для произвольной оценочной функции, а затем покажем, как уменьшить время счета при аффинной функции. Алгоритм трудоемкости $O(nm \log m)$ для вогнутых весов более сложен, чем другие, и он откладывается до главы 13.

^{*)} Модель аффинного веса иногда называется *линейной* моделью веса, и я предпочитаю этот термин. Однако термин “аффинный” становится в биологической литературе доминирующим, а “линейный” обычно относится к аффинной функции с $W_g = 0$.

^{**)} Автор повсеместно называет такую функцию выпуклой, отмечая, что “некоторые называют ее вогнутой”. Поскольку такой разброс терминологии в русскоязычной литературе уже преодолен и термин “вогнутая” стал общепринятым, мы всюду, где потребовалось, переставили термины “выпуклый” и “вогнутый”. — *Прим. перев.*

11.8.5. Произвольная функция оценки пропуска

Эта задача была сформулирована и решена в классической работе Нидлмана и Вунча [347], хотя и с некоторыми отличиями в деталях и терминологии от используемых здесь.

Для произвольной оценки пропуска мы построим рекуррентные соотношения, похожие на те, которые были введены в п. 11.6.1 для оптимального выравнивания без пропусков (но более подробные). Останется, однако, деликатный вопрос, насколько корректно такие соотношения моделируют точку зрения биолога на пропуски. Этот вопрос будет рассматриваться в упражнении 45.

Чтобы выстроить строки S_1 и S_2 , рассмотрим, как обычно, префиксы $S_1[1..i]$ строки S_1 и $S_2[1..j]$ строки S_2 . Возможна реализация одного из следующих трех типов выравнивания двух префиксов (рис. 11.8):

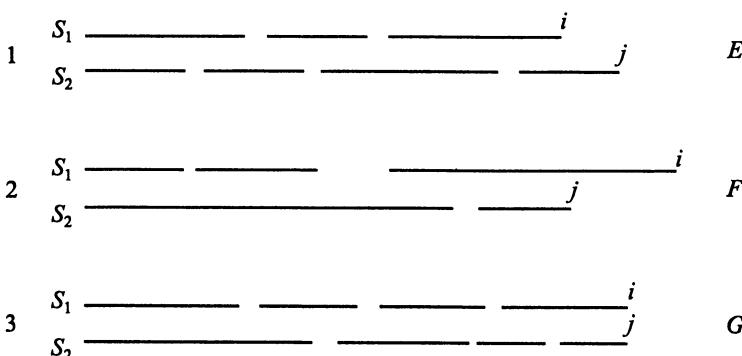


Рис. 11.8. Рекуррентные соотношения при выравнивании с пропусками делятся на три типа выравнивания: 1 — $S_1(i)$ левее $S_2(j)$; 2 — $S_1(i)$ правее $S_2(j)$; 3 — $S_1(i)$ и $S_2(i)$ стоят вровень

1. Выравнивания $S_1[1..i]$ и $S_2[1..j]$, в которых символ $S_1(i)$ выровнен с символом строго левее символа $S_2(j)$. В этом случае выравнивание кончается пропуском в S_1 .
2. Выравнивания, в которых $S_1(i)$ выровнен строго справа от $S_2(j)$. В этом случае выравнивание кончается пропуском в S_2 .
3. Выравнивания, в которых символы $S_1(i)$ и $S_2(j)$ выстроены друг против друга. Это включает в себя и случай $S_1(i) = S_2(j)$, и случай $S_1(i) \neq S_2(j)$.

Ясно, что эти три типа выравниваний исчерпывают все возможности.

Определение. Определим $E(i, j)$ как максимальное значение выравнивания типа 1, $F(i, j)$ как максимальное значение выравнивания типа 2, $G(i, j)$ как максимальное значение выравнивания типа 3 и, наконец, $V(i, j)$ как максимальное из трех значений $E(i, j)$, $F(i, j)$ и $G(i, j)$.

Рекуррентные соотношения в случае произвольной оценки пропуска

Разделяя типы выравнивания на три случая, как выше, мы можем написать следующие рекуррентные соотношения для получения $V(i, j)$:

$$\begin{aligned} V(i, j) &= \max\{E(i, j), F(i, j), G(i, j)\}, \\ G(i, j) &= V(i - 1, j - 1) + s(S_1(i), S_2(j)), \\ E(i, j) &= \max\{V(i, k) - w(j - k): 0 \leq k \leq j - 1\}, \\ F(i, j) &= \max\{V(l, j) - w(i - l): 0 \leq l \leq i - 1\}. \end{aligned}$$

Чтобы завершить формирование соотношений, нужно уточнить базовые условия и определить, откуда берется значение оптимального выравнивания. Если все пробелы включаются в целевую функцию, даже те, которые стоят в начале и конце строки, то оптимальное значение выравнивания находится в клетке (n, m) , а базовые условия таковы

$$\begin{aligned} V(i, 0) &= -w(i), \\ V(0, j) &= -w(j), \\ E(i, 0) &= -w(i), \\ F(0, j) &= -w(j), \end{aligned}$$

где $G(0, 0) = 0$, но $G(i, j)$ не определено, когда один из индексов i или j равен нулю. Отметим, что $V(0, 0) = w(0)$, и это значение естественнее всего принять равным 0.

Если концевые пробелы и, следовательно, концевые пропуски бесплатны, то значение оптимального выравнивания равно максимальному значению в строке n и столбце m , и базовые условия таковы

$$\begin{aligned} V(i, 0) &= 0, \\ V(0, j) &= 0. \end{aligned}$$

Временной анализ

Теорема 11.8.1. *Если обозначить $|S_1| = n$ и $|S_2| = m$, то рекуррентные соотношения можно вычислить за время $O(nm^2 + n^2m)$.*

Доказательство. Расчеты по рекуррентным соотношениям мы ведем как обычно, заполняя таблицу размера $(n + 1) \times (m + 1)$ по строчкам, а каждую строчку — слева направо. Для каждой клетки (i, j) алгоритм проверяет одну клетку для вычисления $G(i, j)$, j клеток строчки i для вычисления $E(i, j)$ и i клеток столбца j для вычисления $F(i, j)$. Поэтому вычисление значений E в любой фиксированной строчке требует просмотра $m(m + 1)/2 = \Theta(m^2)$ клеток, а значений F в любом фиксированном столбце — просмотра $\Theta(n^2)$ клеток. Поскольку счет идет в n строчках и m столбцах, отсюда следует утверждение теоремы. \square

Увеличение времени счета по сравнению с предыдущим случаем (временем $O(nm)$, когда пропуски в модель не включаются) вызывается необходимостью при определении $V(i, j)$ просматривать j клеток влево и i клеток наверх. До того как пропуски были включены в модель, значение $V(i, j)$ зависело только от трех клеток, смежных с (i, j) , и каждое значение $V(i, j)$ вычислялось за константное время. Посмотрим теперь, как уменьшить число просмотров клеток в случае аффинной функции оценки пропуска; далее рассмотрим более сложную редукцию для случая выпуклой функции оценки.

11.8.6. Аффинная (и постоянная) оценка пропуска

Обсудим в деталях простейшую модель аффинной оценки пропуска и покажем, что оптимальные выравнивания в этой модели можно вычислить за время $O(nm)$. Эта величина совпадает с границей для модели без учета пропусков. То есть хотя явный член, учитывающий в целевой функции пропуски, делает модель выравнивания много богаче, он не увеличивает время счета (по асимптотической оценке наихудшего случая) для нахождения оптимального выравнивания. Этот важный результат был получен несколькими авторами (например, [18, 166, 186]). Тот же результат непосредственно следует для постоянной оценки пропуска. Напомним, что задача выглядит так:

$$\text{максимизировать } (W_m N_m - W_{ms} N_{ms} - W_g N_g - W_s N_s).$$

Мы будем использовать те же переменные $V(i, j)$, $E(i, j)$, $F(i, j)$ и $G(i, j)$, что и в рекуррентных соотношениях для произвольных весов пропуска. Определение и смысл этих переменных остаются неизменными, но рекуррентные соотношения будут модифицированы.

Ключевое наблюдение, приводящее к большей эффективности в случае аффинной функции, заключается в том, что приращение полной оценки пропуска от каждого дополнительного пробела — это постоянная W_s , не зависящая от размера пропуска до этой точки. Другими словами, в аффинной модели $w(q+1) - w(q) = W_s$ при любой положительной длине пропуска q . В этом и состоит отличие от случая произвольной оценки, где никакого предсказуемого соотношения между $w(q)$ и $w(q+1)$ не существует. Ввиду того что оценка пропуска возрастает на одно и то же W_s с каждым дополнительным пробелом после первого, при расчете $E(i, j)$ или $F(i, j)$, нам не нужно точно знать, где пропуск начинается, а только — продолжается ли пропуск или создается новый (напротив символа i в S_1 или символа j в S_2). Это наблюдение, как обычно, формализуется в наборе рекуррентных соотношений.

Рекуррентные соотношения

Когда концевые пропуски включаются в оценку выравнивания, базовые условия выглядят так:

$$V(i, 0) = E(i, 0) = -W_g - iW_s,$$

$$V(0, j) = F(0, j) = -W_g - jW_s,$$

поэтому нулевую строчку и столбец в таблице для V можно легко заполнить. В случае бесплатных концевых пропусков $V(i, 0) = V(0, j) = 0$.

Рекуррентные соотношения для общего случая таковы:

$$V(i, j) = \max\{E(i, j), F(i, j), G(i, j)\},$$

$$G(i, j) = \begin{cases} V(i-1, j-1) + W_m, & \text{если } S_1(i) = S_2(j), \\ V(i-1, j-1) - W_{ms}, & \text{если } S_1(i) \neq S_2(j), \end{cases}$$

$$E(i, j) = \max\{E(i, j-1), V(i, j-1) - W_g\} - W_s,$$

$$F(i, j) = \max\{F(i-1, j), V(i-1, j) - W_g\} - W_s.$$

Чтобы лучше понять эти соотношения, рассмотрим вычисление $E(i, j)$. По определению $S_1(i)$ будет выровнено слева от $S_2(j)$. Соотношение утверждает, что либо 1) $S_1(i)$ находится строго в одной позиции слева от $S_2(j)$, в этом случае пропуск начинается в S_1 напротив символа $S_2(j)$ и тогда $E(i, j) = V(i, j - 1) - W_g - W_s$, либо 2) $S_1(i)$ находится левее $S_2(j - 1)$, в этом случае один и тот же пропуск в S_1 лежит напротив и $S_2(j - 1)$ и $S_2(j)$, а $E(i, j) = E(i, j - 1) - W_s$. Объяснение для $F(i, j)$ аналогично, а $G(i, j)$ — это простой случай выравнивания $S_1(i)$ напротив $S_2(j)$.

Как и раньше, значение оптимального выравнивания находится в клетке (n, m) , если правые концевые пробелы входят в целевую функцию. В противном случае берется максимальное из значений в n -й строчке и m -м столбце.

Читатель должен проверить корректность этих соотношений, но его может удивить, почему в соотношении для $E(i, j)$ используется $V(i, j - 1)$, а не $G(i, j - 1)$. То есть почему $E(i, j)$ не равно $\max\{E(i, j - 1), G(i, j - 1) - W_g\} - W_s$? Это было бы некорректно, так как не учитывается возможность выравнивания с двумя смежными пропусками, в S_2 и затем сразу же в S_1 . Расширенное соотношение $E(i, j) = \max\{E(i, j - 1), G(i, j - 1) - W_g, V(i, j - 1) - W_g\} - W_s$ предусматривает все типы выравниваний и корректно, но включение среднего члена $(G(i, j - 1) - W_g)$ излишне, так как последний член $(V(i, j - 1) - W_g)$ его учитывает.

Временной анализ

Теорема 11.8.2. *Оптимальное выравнивание с аффинной оценкой пропуска может быть вычислено за время $O(nm)$ — за то же время, что и оптимальное выравнивание без учета пропусков.*

Доказательство. Проверка рекуррентных соотношений показывает, что для любой пары (i, j) каждый из членов $V(i, j)$, $E(i, j)$, $F(i, j)$ и $G(i, j)$ вычисляется за конечное число просмотров ранее вычисленных значений, арифметических операций и сравнений. Следовательно, времени $O(nm)$ достаточно для заполнения всех $(n + 1) \times (m + 1)$ клеток в таблице динамического программирования. \square

11.9. Упражнения

- Выпишите редакционное предписание для примера выравнивания на с. 281.
- Данное в этой книге определение для преобразования строк и редакционного расстояния позволяет выполнять в каждой позиции строки не более одной операции. Но мотивировка изучения преобразования строк и редакционного расстояния идет, в частности, от попыток создания модели эволюции, в которой нет ограничений на число мутаций в одной позиции. Удаления, включения и замены — все это может случиться в одной и той же позиции. Однако, даже если разрешить выполнять произвольное число операций в одной позиции, они не появятся в преобразовании, использующем наименьшее число операций. Докажите это.
- При рассмотрении редакционного расстояния все преобразования выполняются только над одной строкой, и было высказано, на уровне “размахивания руками”, соображение, что разрешение проводить операции над обеими строками не увеличит общности. Подробно объясните, почему нет потери общности при ограничении преобразований одной строкой.

4. Приведите подробности заполнения таблицы динамического программирования для редакционного расстояния либо выравнивания по столбцам или последовательным антидиагоналям. Случай антидиагоналей полезен в контексте практических параллельных вычислений. Объясните это.
 5. В п. 11.3.3 мы описали, как создать редакционное предписание из обратного прохода по таблице динамического программирования для редакционного расстояния. Докажите, что сформированное так редакционное предписание будет оптимальным.
 6. В части I мы обсуждали задачу о точном совпадении, где допускались джокеры — символы, совпадающие со всеми другими. Формализуйте задачу о редакционном расстоянии при допущении джокеров в обеих строках и покажите, как с ними обращаться в методе динамического программирования.
 7. Докажите теорему 11.3.4, показав, что указатели в таблице динамического программирования полностью охватывают все оптимальные выравнивания.
 8. Покажите, как использовать значение оптимального (глобального) выравнивания для вычисления редакционного расстояния между двумя строками и наоборот. Обсудите в общем формальное отношение между редакционным расстоянием и сходством строк. При каких обстоятельствах эти концепции в основном эквивалентны и когда они различны?
 9. В этой главе рассматривался метод построения левых обратных ссылок для оптимального выравнивания при заполнении таблицы динамического программирования (ДП) с последующим использованием этих ссылок для обратного прохода от клетки (n, m) до клетки $(0, 0)$. Однако имеется альтернативный подход, который работает и при отсутствии ссылок. Если задана полная таблица ДП без ссылок, то выравнивание можно построить алгоритмом, который “прорабатывает” таблицу в один просмотр от клетки (n, m) до клетки $(0, 0)$. Уточните этот алгоритм и покажите, что он может работать так же быстро, как алгоритм, заполняющий таблицу.
 10. Для большей части выравниваний (например, для глобального выравнивания без произвольных весов пропусков) обратный ход, использующий ссылки (как описано в п. 11.3.3), работает за время $O(n + m)$, что меньше времени, необходимого для заполнения таблицы. Определите, какие типы выравниваний допускают это ускорение.
 11. Так как пути обратного хода в таблице динамического программирования находятся в одно-однозначном соответствии с оптимальными выравниваниями, число различных кооптимальных выравниваний может быть получено счетом числа различных обратных путей. Предложите алгоритм для расчета этого числа за время $O(nm)$.
- Совет.** Используйте динамическое программирование.
12. Как отмечалось в предыдущем упражнении, кооптимальные выравнивания можно найти, перечисляя все пути обратного хода в таблице динамического программирования. Предложите метод для нахождения всех путей и всех кооптимальных выравниваний за время $O(n + m)$ на путь.
 13. Должны ли элементы таблицы динамического программирования для редакционного расстояния идти вдоль строчки не убывая? А если вниз по столбцу или вниз по диагонали таблицы? Рассмотрите те же вопросы для оптимального глобального выравнивания.
 14. Обоснуйте корректность формулы в теореме 11.6.1. Затем приведите подробности нахождения наибольшей общей подпоследовательности, а не только ее длины, используя алгоритм для взвешенного редакционного расстояния.

15. Как показано в тексте, задача о наибольшей общей подпоследовательности может быть решена как задача оптимального выравнивания или сходства. Ее можно также рассмотреть как задачу о редакционном расстоянии с весами операций.

Пусть u обозначает длину наибольшей общей подпоследовательности двух строк длиной n и m . Используя веса операций $d = 1$, $r = 2$ и $e = 0$, мы утверждаем, что $D(n, m) = m + n - 2u$ или $u = (m + n - D(n, m))/2$. Таким образом, $D(n, m)$ минимизируется при максимизации u . Докажите это утверждение и объясните подробно, как найти наибольшую общую подпоследовательность, используя программу для поиска редакционного расстояния с весами операций.

16. Напишите рекуррентные соотношения для задачи о наибольшей общей подпоследовательности, не используя веса. Это значит, что задачу lcs нужно решить более непосредственно, не представляя ее как частный случай сходства или редакционного расстояния с весами операций.
17. Объясните корректность рекуррентных соотношений для сходства, данных в п. 11.6.1.
18. Объясните, как вычислить редакционное расстояние (в отличие от сходства), когда концевые пробелы бесплатны.
19. Докажите однозначность соответствия между кратчайшими путями в редакционном графе и глобальными выравниваниями минимального веса.
20. Подробно объясните, как можно корректно решить вариант задачи о сходстве с бесплатными концевыми пробелами, используя метод, предложенный в п. 11.6.4.
21. Докажите теорему 11.6.2 и подробно обоснуйте корректность метода для нахождения кратчайшего приближенного вхождения P в T , кончающегося в позиции j .
22. Объясните, как использовать таблицу динамического программирования и обратный проход для нахождения всех оптимальных решений (пар подстрок) задачи локального выравнивания двух строк, S_1 и S_2 .
23. В п. 11.7.3 мы упомянули, что таблица динамического программирования часто используется для идентификации пар подстрок высокого сходства, которые могут не быть оптимальным решением задачи локального выравнивания. При заданном пороге сходства t этот метод ищет пары подстрок со значением сходства не менее t . Постройте пример, показывающий, что возможен пропуск некоторых приемлемых пар подстрок.
24. Покажите, как решить задачу выравнивания с алфавитными весами и аффинной оценкой пропусков за время $O(nm)$.
25. Обсуждение выравнивания с весами пропусков концентрировалось на вычислении значений в таблице динамического программирования и игнорировало вопросы построения оптимального выравнивания. Покажите, как изменить алгоритм, чтобы он строил оптимальное выравнивание. Постарайтесь ограничить требуемое количество дополнительной памяти.
26. Объясните, почему рекуррентное соотношение для модели аффинной оценки пропуска $E(i, j) = \max\{E(i, j - 1), G(i, j - 1) - W_g, V(i, j - 1) - W_g\} - W_s$ корректно, но излишне и почему средний член $(G(i, j - 1) - W_g)$ может быть удален из соотношения.
27. Рекуррентные соотношения, которые мы построили для модели аффинных платежей за пропуск, следовали логике платежа $W_g + W_s$, когда пропуск “инициализировался”, и затем W_s за каждый дополнительный пробел. Альтернативной была бы логика платить $W_g + W_s$ в точке, где пропуск “завершается”. Напишите рекуррентные соотношения для

модели с этой логикой. Выравнивание должно вычисляться за время $O(nm)$. Такой тип соотношений обсуждается в [166].

28. В версии задачи выравнивания с бесплатными концевыми пропусками пробелы и пропуски на любом конце выравнивания ничего не добавляют в цену выравнивания. Покажите, как использовать предложенные в тексте рекуррентные соотношения с аффинной оценкой пропуска для решения версии также с аффинной оценкой пропуска и с бесплатными концевыми пропусками. Затем сделайте то же самое с альтернативными рекуррентными соотношениями из предыдущего упражнения. В обоих случаях время счета должно быть $O(nm)$. Имеются ли какие-нибудь преимущества в использовании того или другого из рассмотренных вариантов?
29. Покажите, как обобщить метод *agrep* из п. 4.2.3, чтобы он допускал включения и исключения символов.
30. Предложите *простой* алгоритм для решения задачи о локальном выравнивании за время $O(nm)$, если использовать пробелы не разрешается.
31. **Повторяющиеся подстроки.** Локальное выравнивание двух различных строк находит пары их подстрок, которые имеют высокое сходство. Важно также находить обладающие высоким сходством подстроки одной и той же строки. Такие подстроки представляют *неточные повторяющиеся подстроки*. Предположительно поиск неточных повторов в отдельной строке можно вести локально, выравнивая строку рядом с ней самой. Но в таком подходе таится трудность. Если локально выстроить строку с ней самой, то наилучшей подстрокой будет вся строка. Даже при использовании всех значений таблицы на наилучший путь из клетки (i, j) , где $i \neq j$, может сильно влиять главная диагональ. Против этой трудности есть простое средство. Укажите его. Может ли ваш метод давать перекрывающиеся подстроки? Желательно ли это? Позднее в упражнении 17 главы 13 мы рассмотрим задачу нахождения самых похожих *неперекрывающихся* подстрок одной строки.
32. **Тандемные повторы.** Пусть P — образец длины n , а T — текст длины m . Пусть P^m — сцепка P с собой m раз, так что длина P^m равна mn . Мы хотим вычислить локальное выравнивание P^m и T . При этом найдется интервал в T , который имеет наилучшее глобальное выравнивание (согласно стандартному критерию выравнивания) с некоторым tandemным повтором P . Эта задача отличается от задачи, рассмотренной в упражнении 4 главы 1, так как теперь разрешены ошибки (несовпадения, включения и удаления). Эта задача встречается при изучении повторной структуры белков, которые образуют так называемую *двукратную спираль* (*coiled-coil*) [158]. В этом контексте P представляет собой *мотив*, или *домен* (для наших целей — образец), который может повторяться в белке неизвестное число раз, а T представляет собой белок. Локальное выравнивание P^m и T захватывает интервал T , который “оптимально” составлен из tandemных повторов мотива (при допускаемых ошибках). Если строка P^m создана явно, то стандартное локальное выравнивание решит эту задачу за время $O(nm^2)$. Но ввиду того что P^m состоит из одинаковых копий P , возможно решение с временем $O(nm)$. Этот метод в основном имитирует работу алгоритма динамического программирования для локального выравнивания, если тот будет выполняться с P^m и T явно. Ниже мы набросаем этот метод.

Алгоритм ДП заполнит таблицу V размером $(m + 1) \times (n + 1)$, строчки которой занумерованы от 0 до n , а столбцы — от 0 до m . Нуевые строчка и столбец первоначально заполнены нулями. Далее в каждой строчке i , от 1 до m , алгоритм делает следующее. Он ведет расчет по рекуррентным соотношениям для локального выравнивания в строчке i , полагает $V(i, 0)$ равным $V(i, n)$ и затем снова считает по тем же стандартным соотношениям строчки i . После полного заполнения каждой строчки выбирается клетка с наибольшим значением V , как в стандартном решении для задачи локального выравнивания.

Ясно, что такой алгоритм требует времени только $O(nm)$. Докажите, что он корректно находит значение оптимального локального выравнивания P^m и T . Затем приведите подробности обратного хода при построении этого выравнивания. Объясните, почему P расширено (концептуально) до P^m , а не больше и не меньше.

33. a. Пусть заданы две строки, S_1 и S_2 , с длинами n и m , а также параметр δ . Покажите, как построить за время $O(nm)$ матрицу \mathcal{M} , в которой элемент $\mathcal{M}(i, j) = 1$ в том и только том случае, если существует выравнивание S_1 и S_2 , где символы $S_1(i)$ и $S_2(j)$ выровнены друг с другом и значение выравнивания отклоняется от максимального не больше чем на δ . Таким образом, если $V(S_1, S_2)$ — значение оптимального выравнивания, то наилучшее выравнивание, которое ставит $S_1(i)$ напротив $S_2(j)$, должно иметь значение не меньше $V(S_1, S_2) - \delta$. Матрица \mathcal{M} используется [490] для получения некоторой информации, такой как общие и необщие свойства, относительно множества субоптимальных выравниваний S_1 и S_2 . Поскольку биологическая значимость оптимального выравнивания иногда не ясна и оптимальность зависит от (часто подвергаемых сомнению) весов, полезно в добавление к оптимальному эффективно находить и изучать также множество субоптимальных (но близких) выравниваний. Можно ли использовать матрицу \mathcal{M} для создания и изучения этих выравниваний?
- b. Покажите, как модифицировать матрицу \mathcal{M} , чтобы $\mathcal{M}(i, j) = 1$ в том и только том случае, если $S_1(i)$ и $S_2(j)$ стоят вровень в каждом выравнивании S_1 и S_2 , значение которого не меньше $V(S_1, S_2) - \delta$. Как эффективно вычислить эту матрицу? Мотивировка для рассмотрения такой матрицы в основном та же, что и для матрицы, описанной в предыдущей задаче: она использована в [443, 445].
34. Поиск выравнивания с учетом пропусков в целевой функции реализуйте методом динамического программирования. Поэкспериментируйте с программой, чтобы найти правильные веса для решения задачи о совпадении кДНК.
35. Процесс, в результате которого в иРНК можно установить интроно-экзонные границы (называемые склейками), не вполне понятен. Простейшая надежда — на то, что склейки помечены образцами, которые встречаются только в них, и нигде больше, — неверна. Однако верно, что некоторые короткие образцы очень часто встречаются в склейках инtronов. В частности, большинство инtronов начинаются с динуклеотида *GT* и кончаются *AG*. Модифицируйте соотношения динамического программирования, используемые в задаче совпадения кДНК, чтобы учесть этот факт.
У инtronов есть и другие характерные образцы. Выполните поиск в библиотеке, чтобы найти информацию об этих свойствах, — во время поиска вы обнаружите массу интересных вещей.
36. **Переход от последовательности к структуре через выравнивание.** Важным приложением выравнивания белковых строк является построение неизвестной вторичной структуры одного белка по известной вторичной структуре другого. Исходя из этой вторичной структуры можно пытаться определить трехмерную структуру белка методами построения моделей. Прежде чем описывать эксперимент по выравниванию, нам понадобятся некоторые сведения о структуре белков.

Строка обычного глобулярного белка (например, фермента) состоит из подстрок, формирующих плотное белковое ядро. Они перемежаются с подстроками, которые образуют петли, обволакивающие белок снаружи. Различают три основных типа вторичных структур в глобулярных белках: α -спирали и β -структуры, составляющие ядро белка, и петли снаружи. Есть также витки, которые меньше, чем петли. Структура ядра белка устойчиво сохраняется во времени, так что большие включения или исключения значительно вероятнее в петлях, чем в ядре.

Теперь предположим, что известна вторичная (или трехмерная) структура белка одного вида и имеется последовательность гомологичного белка другого вида, но неизвестна его двух- или трехмерная структура. Пусть S_1 обозначает строку для первого белка, а S_2 — для второго. Определение двух- или трехмерной структуры методами кристаллографии или ЯМР очень сложно и дорого. Вместо этого хочется использовать выравнивание последовательностей S_1 и S_2 с тем, чтобы распознать α - и β -структуры в S_2 . Есть надежда, что при надлежащей модели выравнивания, матрице оценок и штрафах за пропуски подстроки α - и β -структур этих двух строк выстраиваются друг напротив друга. Так как положения участков α и β в S_1 известны, то “успешное” выравнивание идентифицирует участки α и β в S_2 . Далее, включения и исключения в участках ядра редки, поэтому выравнивание, которое успешно идентифицирует α и β в S_2 , не должно иметь больших пропусков в α - и β -участках в S_1 . Аналогично, это выравнивание не должно иметь больших пропусков в подстроках S_2 , которые выстроены с известными участками α и β из S_1 .

Обычно для оценки совпадений и несовпадений используются оценочная матрица и аффинная (или линейная) модель штрафа за пропуски. Эксперименты [51, 447] показали, что успех при таком подходе очень чувствителен к выбору оценочной матрицы и штрафов. Более того, было высказано предположение, что штраф за пропуск должен быть выше в подстроках, формирующих участки α и β , чем в оставшейся части строки (см., например, [51, 296]). Это значит, что никакой *фиксированный* выбор штрафа за пропуск и за пробел (стартовый штраф за пропуск и штраф за приращение пропуска на диалекте вычислительной биологии) не будет работать. Или, по меньшей мере, более правдоподобно, что увеличение штрафа за пропуск во вторичных участках улучшит выравнивание. Высокие штрафы за пропуск заставят избегать разрывов в участках α и β . Однако должны позволяться пропуски в выравнивании участков вне ядра, поскольку в петлях обязательно присутствуют включения и удаления.

Это приводит к следующей задаче выравнивания: как модифицировать модель выравнивания и структуру штрафа, чтобы удовлетворить изложенным выше требованиям? И как найти оптимальное выравнивание при новых ограничениях?

Сформулированная задача не очень сложна технически, но особенно важна как приложение к выявлению вторичной структуры. Данных о белковых последовательностях доступно на порядки больше, чем о белковых структурах. В силу этого получение структур из последовательностей становится центральной задачей.

Вариант множественного выравнивания для задачи предсказания структур обсуждается в первой части п. 14.10.2.

37. Пусть заданы две строки, S_1 и S_2 , и текст T и вы хотите определить, нет ли вхождения S_1 и S_2 , *переплетенных* (без пробелов) в T . Например, строки *abac* и *bbc* входят в переплетенном виде в *cabbabccdw*. Предложите для этой задачи эффективный алгоритм. (Он может иметь отношение к задаче о наибольшей общей подпоследовательности.)
38. Как уже говорилось в упражнениях главы 1, бактериальные ДНК часто представляют собой кольцевые молекулы. В связи с этим интересна следующая задача. Заданы две линейные строки с длинами n и m ; имеется n циклических сдвигов первой строки и m циклических сдвигов второй строки, так что есть nm пар циклических сдвигов. Мы хотим вычислить глобальное выравнивание для каждой из этих nm пар строк. Можно ли это сделать эффективнее, чем решая задачу выравнивания для каждой пары с пустого места? Рассмотрите и анализ наихудшего случая, и “обычное” время счета для “естественных” исходных данных.

Рассмотрите ту же задачу для локального выравнивания.

39. **Задача о заикающейся подпоследовательности** [328]. Пусть P и T — строки с длинами, соответственно, n и m . Предложите алгоритм трудоемкости $O(m)$ для определения того, не входит ли P в T как подпоследовательность.

Теперь пусть P^i обозначает строку, в которой каждый символ из P повторен i раз. Например, если $P = abc$, то $P^3 = aaabbccccc$. Ясно, что при любом фиксированном i за время $O(m)$ можно проверить, не входит ли P^i в T в качестве подпоследовательности. Предложите алгоритм, определяющий за время $O(m \log m)$ наибольшее i , для которого P^i является подпоследовательностью T . Пусть $\text{Maxi}(P, T)$ обозначает значение этого наибольшего i .

Теперь мы набросаем подход к решению задачи, который уменьшает время счета с $O(m \log m)$ до $O(m)$. Вы должны восполнить детали.

Для строки T пусть d обозначает число различных символов в T . Для строки T и символа x из T определим $\text{odd}(x)$ как множество позиций нечетных вхождений x в T , т.е. позиций первого, третьего, пятого и т.д. вхождений. Так как в T есть d различных символов, то существует d таких нечетных наборов. Например, если $T = 0120002112022220110001$, то $\text{odd}(T)$ — это 2, 9, 18. Определим теперь $\text{half}(T)$ как подпоследовательность T , которая остается после удаления всех символов, появляющихся в нечетный раз. Например, $\text{half}(T)$ для указанного T равно 0021220101. Предполагая, что число d различных символов заранее фиксировано, предложите алгоритм трудоемкости $O(m)$ для нахождения $\text{half}(T)$. Потом докажите, что длина $\text{half}(T)$ не превосходит $m/2$. Это будет использовано затем при временному анализе.

Теперь докажите, что $|\text{Maxi}(P, T) - 2\text{Maxi}(P, \text{half}(T))| \leq 1$. Этот факт имеет для метода решающее значение.

Приведенные факты позволяют найти $\text{Maxi}(P, T)$ за время $O(m)$ алгоритмом “разделяй и властвуй”. Детализируйте метод: уточните условия завершения в “разделяй и властвуй”, докажите корректность метода, постройте рекуррентное соотношение для анализа времени счета, а затем решите это соотношение, получив оценку $O(m)$.

Более сложная задача: есть ли реалистическая интерпретация задачи о заикающейся подпоследовательности?

40. Как видно из предыдущего упражнения, легко определить, не входит ли единичный образец P в текст T в качестве подпоследовательности. Это требует времени $O(m)$. Рассмотрите теперь другую задачу: не входит ли в текст хотя бы один из набора образцов. Если n — общая длина всех образцов набора, то на решение для каждого образца отдельно понадобится время $O(nm)$. Попробуйте добиться времени, значительно лучшего, чем $O(nm)$. Вспомните, что аналогичная задача для набора подстрок может быть решена за время $O(n + m)$ методами Ахо–Корасика и суффиксного дерева.
41. **Задача складывания тРНК.** Следующее упражнение — это совершенно сырой вариант задачи, которая возникает в предсказании вторичной (планарной) структуры молекул транспортной РНК. Пусть S — строка из n символов над алфавитом РНК, состоящим из a , c , u , g . Определим **паросочетание** как множество непересекающихся пар символов из S . Паросочетание называется **собственным**, если оно содержит только пары (a, u) и (c, g) . Это ограничение возникает вследствие того, что в РНК нуклеотиды a и u , а также c и g комплементарны. Если мы изобразим S как циклическую строку, то **вложенное паросочетание** определится как собственное паросочетание, в котором каждая пара соединена линией внутри круга и эти линии не пересекаются (рис. 11.9). Задача в том, чтобы найти вложенное паросочетание наибольшего размера. Часто добавляется дополнительное ограничение: символ не может быть в паре со своими двумя непосредственными соседями. Покажите, как решить задачу складывания тРНК методами динамического программирования за время $O(n^3)$.

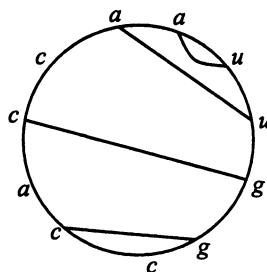


Рис. 11.9. Вложенные пары, не обязательно максимальной кардинальности

Модифицируйте эту задачу, добавив веса целевой функции так, чтобы вес пары $a-u$ отличался от веса пары $c-g$. Теперь целью будет нахождение вложенного паросочетания максимального общего веса. Предложите эффективный алгоритм для такой взвешенной задачи.

42. Транспортные молекулы РНК (тРНК) имеют своеобразную планарную вторичную структуру, называемую структурой *клеверного листа*. В ней строка состоит из четырех чередующихся черенков и петель (рис. 11.10). Каждый черенок складывается из двух параллельных подстрок, обладающих тем свойством, что любая пара стоящих напротив символов в черенке комплементарна (a с u , c с g). Каждая комплементарная пара в черенке образует химическую связь, которая усиливает общую стабильность молекулы. Связь $c-g$ сильнее, чем связь $a-u$.

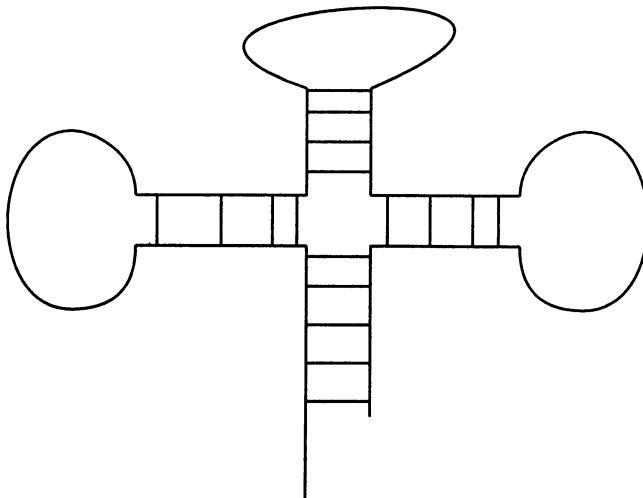


Рис. 11.10. Грубая схема “клеверного листа”. Каждая маленькая вертикальная или горизонтальная черта внутри черенка представляет пару оснований типа $a-u$ или $c-g$

Сопоставьте это (очень поверхностное) описание вторичной структуры тРНК с рассмотренной выше задачей о взвешенном вложенном паросочетании.

43. Настоящий образец со связями комплементарных оснований (в черенках) молекул тРНК в основном удовлетворяет условиям неперекрещивания из определения вложенных

паросочетаний. Однако бывают исключения, так что когда вторичная структура известных молекул тРНК изображается линиями в круге, то некоторые пересечения возможны. Эти нарушения условия неперекрещивания называются *псевдоузлами*.

Рассмотрите задачу нахождения собственного паросочетания максимального размера при допущении фиксированного числа псевдоузлов. Предложите для этой задачи эффективный алгоритм, сложность которого будет функцией от разрешенного числа перекреcиваний.

- 44. Выравнивание последовательности и структуры РНК.** Из-за того что структура РНК основана на вложенных паросочетаниях, при выравнивании строк РНК легко учитывать некоторые структурные соображения. Здесь мы и рассмотрим выравнивания такого рода.

Пусть P — образцовая строка РНК с известной структурой паросочетаний, а T — текстовая строка РНК с известной структурой паросочетаний. Чтобы представить структуру паросочетаний в P , введем $O_P(i)$ как *смещение* (offset) (положительное или отрицательное) напарника символа из позиции i , если такой существует. Например, если символ в позиции 17 имеет напарника из позиции 46, то $O_P(17) = 29$ и $O_T(46) = -29$. Если символ в позиции i не имеет пары, то $O_P(i) = 0$. Структура T аналогично представлена вектором смещений O_T . Тогда P входит в T , начиная с позиции j , в том и только том случае, если $P(i) = T(j + i - 1)$ и $O_P(i) = O_T(j + i - 1)$ для любой позиции i из P .

a. Предполагая, что длины P и T равны n и m соответственно, предложите алгоритм со временем $O(n + m)$ для нахождения всех точных вхождений P в T .

b. Теперь рассмотрите более либеральный критерий для решения, что P входит в T , начиная с позиции j . Снова потребуем, чтобы $P(i) = T(j + i - 1)$ для любой позиции i из P , но $O_P(i) = O_T(j + i - 1)$ — только при $O_P(i)$, не равном нулю.

Предложите эффективный алгоритм для нахождения всех мест, где P входит в T при более либеральном определении вхождения. Наивный алгоритм трудоемкости $O(nm)$, заключающийся в явном пристраивании P к каждой начальной позиции j и дальнейшей проверке совпадения, не эффективен. Эффективное решение может быть получено с помощью методов из частей I и II этой книги.

c. Рассмотрите вопрос о том, когда разумно использовать более либеральное определение, а когда нет.

- 45. Один вопрос о моделировании пропусков.** Рекуррентные соотношения из п. 11.8.5, выведенные для случая произвольных весов за пропуск, ставят деликатный вопрос о правильной модели пропуска, когда штраф за пропуск w произведен. С этими соотношениями любой обычный пропуск может рассматриваться как два или более пропусков, которые оказались смежными. Предположим, например, что $w(q) = 1$ для $q \leq 5$ и $w(q) = 10^6$ для $q > 5$. Тогда пропуск длины 10 будет иметь вес 10^6 , если его рассматривать как один пропуск, и вес 2, если его представить как два смежных пропуска по 5. Соотношения из п. 11.8.5 будут рассматривать эти десять пробелов как два смежных пропуска с суммарным весом 2. Хороша ли такая модель пропусков?

На этот вопрос можно смотреть с двух точек зрения. С одной стороны, целью является моделирование наиболее правдоподобного набора мутационных событий, трансформирующих одну строку в другую, и выравнивание — только средство для *представления* этого преобразования. Разрешенные примитивные мутационные события — это преобразование отдельных символов (несовпадения в выравнивании) и включение и исключение блоков символов произвольной длины (каждый из которых приводит к пропуску в выравнивании). С этой точки зрения совершенно правильно иметь два смежных пропуска в одной строке. Они просто показывают, что две операции блоковых вставок или удалений произошли рядом друг с другом. Если веса пропусков корректно моделируют стоимости таких

блоковых операций и стоимость является возрастающей выпуклой функцией длины, как в предыдущем примере, то наиболее правдоподобно, что длинный пропуск был следствием нескольких событий удаления и вставки. С этой точки зрения оправдано, чтобы задача динамического программирования допускала смежные пропуски, когда они выгодны.

С другой стороны, может быть, интерес заключается только в том, насколько “похожи” две строки, и никакой явной мутационной модели не предполагается. Тогда данное выравнивание двух строк — просто один из способов демонстрации их сходства. В таком случае пропуск представляет собой максимальное множество смежных пробелов и потому не должен разбиваться на меньшие пропуски.

С произвольными весами пропусков соотношения динамического программирования представляют корректную модель для первой точки зрения, но не для второй. Далее, в случае выпуклых весов (и следовательно, аффинных или постоянных) эти соотношения корректно моделируют обе точки зрения, так как не возникает побуждений разбить пропуск на меньшие. Однако если веса пропусков с выпуклыми возрастающими участками считаются обоснованными, то для корректного моделирования второй точки зрения требуются другие рекуррентные соотношения. Вот они:

$$\begin{aligned} V(i, j) &= \max\{E(i, j), F(i, j), G(i, j)\}, \\ G(i, j) &= V(i - 1, j - 1) + s(S_1(i), S_2(j)), \\ E(i, j) &= \max\{G(i, k) - w(j - k) : 0 \leq k \leq j - 1\}, \\ F(i, j) &= \max\{G(l, k) - w(j - l) : 0 \leq l \leq j - 1\}. \end{aligned}$$

Эти уравнения отличаются от соотношений п. 11.8.5 заменой $V(i, k)$ и $V(l, j)$ на $G(i, k)$ и $G(l, j)$ в уравнениях для $E(i, j)$ и $F(i, j)$ соответственно. В результате каждому пропуску, кроме самого левого, будут предшествовать два выровненных символа; следовательно, двух смежных пропусков в одной и той же строке не будет. Однако под запретом окажутся и два смежных пропуска в разных строках. Например, с помощью этих модифицированных соотношений выравнивание

x	x	a	b	c	$-$	$-$	$-$	y	y
x	x	$-$	$-$	$-$	i	d	e	y	y

никогда не будет найдено.

Кажется, что нет модельных оснований для запрещения смежных пропусков в разных строках. В действительности некоторые мутации, такие как *инвертирование подстроки* (что типично для ДНК), лучше всего представляются в выравнивании как смежные пропуски такого типа, если только модель выравнивания явно не включает в себя понятия инверсии (как, например, модель в главе 19). Другой пример, где смежные пробелы были бы естественны, возникает при сравнении двух строк иРНК, которые появляются при альтернативной склейке инtronов. У эукариот гены часто состоят из чередующихся участков экзонов и инtronов. При нормальной форме транскрипции все интраны в конечном счете вырезаются, так что молекула иРНК представляет собой сцепку экзонов. Но может произойти так называемая *альтернативная склейка*, когда экзоны вырезаются так же, как интраны. Рассмотрим ситуацию, когда вырезаны все интраны и еще экзон i , и ситуацию, когда вырезаны все интраны и еще экзон $i + 1$. Если сравниваются эти две строки иРНК, то наилучшее выравнивание может расположить экзон i напротив пропуска во второй строке, а экзон $i + 1$ напротив пропуска в первой строке. Другими словами, информативным было бы выравнивание с двумя смежными пропусками в альтернативных строках. В этом случае приведенные соотношения не отразят корректно вторую точку зрения.

Напишите соотношения для произвольных весов пропуска, которые допускали бы смежные пропуски в различных строках и запрещали бы их в одной строке.

Глава 12

Улучшение процедур выстраивания

Для некоторых основных задач редактирования и выстраивания (выравнивания) строк мы рассмотрим ряд усовершенствований методов их решения. В результате ускоряются процессы динамического программирования, уменьшается размер необходимой памяти или повышается эффективность метода.

12.1. Вычисление выравниваний в линейной памяти

Один из дефектов динамического программирования во всех обсуждавшихся задачах заключается в том, что таблицы ДП используют память размера $\Theta(nm)$ для строк длиной n и m . (Когда мы говорим о памяти, необходимой алгоритму, то имеем в виду максимальную память, занимаемую единовременно. Повторное использование памяти не увеличивает затрат.) Принято считать, что ограничивающим ресурсом в задачах выравнивания строк является именно память, а не время. Ограничение памяти затрудняет обработку больших строк, независимо от того, как долго мы готовы ждать завершения расчетов. Поэтому очень важно иметь методы, уменьшающие затраты памяти без критического увеличения времени счета.

Хиршберг [224] предложил элегантный и практичный метод сокращения памяти, который работает для многих задач динамического программирования. Для некоторых задач выравнивания строк этот метод уменьшает потребности в памяти с $\Theta(nm)$ до $O(n)$ (для $n < m$), всего лишь удваивая время счета в наихудшем случае. Миллер и Майерс развили эту идею и привлекли к ней внимание вычислительных биологов [344]. С тех пор метод был расширен и применен ко многим другим задачам [97]. Мы иллюстрируем его на решении динамическим программированием задачи вычисления оптимального взвешенного глобального выравнивания двух строк.

12.1.1. Уменьшение памяти при вычислении сходства

Напомним, что сходство двух строк — это число и что за целевой функцией для сходства стоит оптимальное выравнивание, значение которого равно этому числу. Если нам требуется только сходство $V(n, m)$, а не само выравнивание с этим значением, то необходимая память (в добавление к месту для строк текста) может быть уменьшена до $2m$. Идея заключается в том, что в вычислении значений V для строчки таблицы i участвуют только значения из предыдущей строчки $i - 1$, все остальные строчки до $i - 1$ не нужны. Это ясно видно из самих рекуррентных соотношений для сходства. Таким образом, мы можем реализовать решение по методу динамического программирования с помощью только двух строчек таблицы: одну — *текущую* строчку (*current*) — назовем C , а другую — *предыдущую* (*previous*) — P . На каждой итерации строчка C вычисляется с использованием строчки P , рекуррентных соотношений и двух текстовых строк. Когда строчка C полностью готова, значения P больше не нужны, и C копируется в P для подготовки к следующей итерации. После n итераций строчка C содержит значения для строчки n полной таблицы, и следовательно, $V(n, m)$ расположено в последней клетке C . Таким способом $V(n, m)$ может быть вычислено в памяти $O(m)$ и за время $O(nm)$. На самом деле любая отдельная строчка полной таблицы может быть вычислена и сохранена за это время и в такой же памяти. Указанная возможность и определяет метод, который будет излагаться.

Дальнейшее развитие этой идеи позволяет уменьшить необходимую память до одной строчки плюс одна дополнительная клетка (кроме места на исходные строки текста). Таким образом, требуется $m + 1$ клеток. И если $n < m$, то память можно уменьшить до $n + 1$. Мы оставляем эти детали как упражнение.

12.1.2. Как найти оптимальное выравнивание в линейной памяти

Предлагаемая идея хороша, если мы ищем сходство $V(n, m)$ или одну заранее выбранную строчку таблицы динамического программирования. Но что же делать, если нам нужно *выравнивание*, на котором достигается значение $V(n, m)$? В большинстве случаев ищется как раз такое выравнивание, а не только его значение. В базисном алгоритме выравнивание находится обратным проходом по указателям, найденным при вычислении всей таблицы ДП для сходства. Однако описанный метод с линейной памятью не хранит всей таблицы, и линейной памяти не достаточно для хранения указателей.

Схема Хиршберга для нахождения оптимального выравнивания в линейной памяти, если описывать ее на принципиальном уровне, выполняет несколько меньших вычислений выравнивания, каждое из которых использует только линейную память и определяет небольшую часть настоящего оптимального выравнивания. Чистым результатом всех этих вычислений становится полное описание оптимального выравнивания. Посмотрим сначала, как с использованием только линейной памяти найти начальный фрагмент полного выравнивания.

Определение. Для любой строки α пусть α^r обозначает обратную к ней строку.

Определение. Пусть заданы строки S_1 и S_2 . Определим $V^r(i, j)$ как сходство строк, состоящих из первых i символов S_1^r и первых j символов S_2^r . Эквивалентно можно определить $V^r(i, j)$ как сходство последних i символов S_1 и последних j символов S_2 (рис. 12.1).

Ясно, что таблица значений $V^r(i, j)$ может быть получена за время $O(nm)$ и любая заранее заданная строчка этой таблицы может быть сосчитана и сохранена за время $O(nm)$ в памяти $O(n)$.

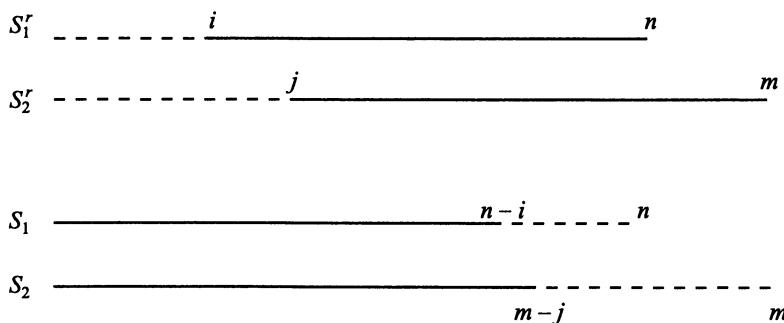


Рис. 12.1. Сходство первых i символов S_1^r и первых j символов S_2^r равносильно сходству последних i символов S_1 и последних j символов S_2 . Пунктирные линии показывают сопоставляемые подстроки

Начальный участок полного выравнивания вычисляется в линейной памяти нахождением $V(n, m)$ в двух частях. Первая часть использует исходные строки, а вторая — обращенные. Подробности двухчастного способа даются следующей леммой.

Лемма 12.1.1. $V(n, m) = \max_{0 \leq k \leq m} \{V(n/2, k) + V(n/2, m - k)\}$.

Доказательство. Этот результат почти очевиден, но тем не менее требует доказательства. Напомним, что $S_1[1..i]$ является префиксом строки S_1 , состоящим из первых i символов, и что $S'[1..i]$ — это строка, обратная к суффиксу, состоящему из последних i символов S_1 . Аналогичные определения используются для S_2 и S'_2 .

Для любой фиксированной позиции k' из S_2 имеются выравнивания S_1 и S_2 , состоящие из выравнивания $S_1[1..n/2]$ и $S_2[1..k']$, за которыми следуют непересекающиеся выравнивания $S_1[n/2 + 1..n]$ и $S_2[k' + 1..m]$. По определению V и V' наилучшее выравнивание в первом случае имеет значение $V(n/2, k')$, а наилучшее выравнивание во втором случае имеет значение — $V^r(n/2, m - k')$, так что комбинированное выравнивание имеет значение

$$V(n/2, k') + V^r(n/2, m - k') < \max_k \{V(n/2, k) + V^r(n/2, m - k)\} < V(n, m).$$

Обратно, рассмотрим оптимальное выравнивание S_1 и S_2 . Пусть k' — самая правая позиция в S_2 , которая выровнена с символом в позиции $n/2$ из S_1 или левее ее. Тогда оптимальное выравнивание S_1 и S_2 состоит из выравнивания $S_1[1..n/2]$ и $S_2[1..k']$, за которым следует выравнивание $S_1[n/2 + 1..n]$ и $S_2[k' + 1..m]$. Пусть значение первого выравнивания обозначено через p , а значение второго выравнивания — через q . Тогда p должно равняться $V(n/2, k')$, так как если бы выполнялось

неравенство $p < V(n/2, k')$, то мы могли бы заменить выравнивание $S_1[1..n/2]$ и $S_2[1..k']$ на такое, где достигается значение $V(n/2, k')$. Это позволило бы построить выравнивание S_1 и S_2 со значением, превосходящим оптимальное. Следовательно, $p = V(n/2, k')$. По аналогичным причинам $q = V^r(n/2, m - k')$. Таким образом,

$$V(n, m) = V(n/2, k') + V^r(n/2, m - k') < \max_k \{V(n/2, k) + V^r(n/2, m - k)\}.$$

Доказав неравенство в обе стороны, мы заключаем, что

$$V(n, m) = \max_k \{V(n/2, k) + V^r(n/2, m - k)\}. \quad \square$$

Определение. Пусть k^* обозначает позицию k , в которой достигается максимум $V(n/2, k) + V^r(n/2, m - k)$.

По лемме 12.1.1 существует оптимальное выравнивание, у которого обратный проход в полной таблице динамического программирования (если она заполнена) посещает клетку $(n/2, k^*)$. Другими словами, существует оптимальный (самый длинный) путь L из вершины $(0, 0)$ в вершину (n, m) в графе выравнивания, который проходит через вершину $(n/2, k^*)$. Это ключевое свойство k^* .

Определение. Пусть $L_{n/2}$ — подпуть L , который начинается в последней вершине L в строчке $n/2 - 1$ и кончается в первой вершине L в строчке $n/2 + 1$.

Лемма 12.1.2. Позиция k^* в строчке $n/2$ может быть найдена за время $O(nm)$ и с памятью $O(n)$. Более того, подпуть $L_{n/2}$ может быть найден и сохранен с теми же затратами времени и памяти.

Доказательство. Сначала выполним динамическое программирование, чтобы вычислить оптимальное выравнивание S_1 и S_2 , но остановимся после итерации $n/2$ (т. е. после вычисления значений в строчке $n/2$). Более того, при заполнении строчки $n/2$ найдем и сохраним нормальные обратные ссылки для клеток этой строчки. В этот момент $V(n/2, k)$ известно для всех $0 \leq k \leq m$. Из вышесказанного ясно, что для получения значений и ссылок в строчке $n/2$ достаточно памяти $O(m)$. Затем начнем вычислять оптимальное выравнивание S'_1 и S'_2 , но остановимся после итерации $n/2$. Сохраним значения для клеток в строчке $n/2$ вместе с обратными ссылками для этих клеток. И снова нам будет достаточно памяти $O(m)$, и значение $V^r(n/2, m - k)$ известно для каждого k . Теперь для каждого k сложим $V(n/2, k)$ и $V^r(n/2, m - k)$ и обозначим через k^* индекс k , дающий наибольшую сумму. Эти сложения и сравнения занимают время $O(m)$.

Используя первый набор сохраненных ссылок, пройдем по любому пути обратного хода из клетки $(n/2, k^*)$ в клетку k_1 строчки $n/2 - 1$. Тем самым получим подпуть, который входит в оптимальный путь из клетки $(0, 0)$ в клетку $(n/2, k^*)$. Аналогично, используя второй набор обратных ссылок, пройдем по любому пути из клетки $(n/2, k^*)$ в клетку k_2 в строчке $n/2 + 1$. Этот путь определяет подпуть оптимального пути из $(n/2, k^*)$ в (n, m) . Два подпути, составленные вместе, формируют подпуть $L_{n/2}$, являющийся частью оптимального пути L из $(0, 0)$ в (n, m) . Более того, этот оптимальный путь проходит через клетку $(n/2, k^*)$. В общей сложности для нахождения k^*, k_1, k_2 и $L_{n/2}$ использованы время $O(nm)$ и память $O(m)$. \square

Для предстоящего анализа всего метода представим время, необходимое для заполнения таблицы ДП размера $p \times q$, как $c p q$, с некоторой неуточняемой константой c , вместо обычной формы $O(pq)$. В таком представлении вычисление строчек $n/2$ в первой задаче ДП произойдет за время $c n m / 2$ и за столько же — вычисление во второй задаче. Таким образом, для получения и сохранения обеих строчек потребуется время $c n m$.

Ключевым моментом, который теперь следует отметить, является то, что при вычислениях, расходуя время $c n m$ и память $O(m)$, алгоритм нашел k^* , k_1 , k_2 и $L_{n/2}$. Это определило часть оптимального выравнивания S_1 и S_2 , а не только значение $V(n, m)$. По лемме 12.1.1 найдется оптимальное выравнивание S_1 и S_2 , состоящее из оптимального выравнивания первых $n/2$ символов S_1 с первыми k^* символами S_2 , за которыми следует оптимальное выравнивание последних $n/2$ символов S_1 с последними $m - k^*$ символами S_2 . В действительности, так как алгоритм получил также подпуть (подвыравнивание) $L_{n/2}$, задача выравнивания S_1 и S_2 сводится к двум меньшим задачам: одна — для строк $S_1[1..n/2 - 1]$ и $S_2[1..k_1]$, а другая — для строк $S_1[n/2 + 1..n]$ и $S_2[k_2..m]$. Назовем первую из них *верхней* задачей, а вторую *нижней*. Отметим, что верхняя задача выравнивает строки с длинами не больше $n/2$ и k^* , а нижняя — с длинами не больше $n/2$ и $m - k^*$.

В терминах таблицы ДП верхняя задача решается в прямоугольнике A исходной таблицы $n \times m$, показанной на рис. 12.2, а нижняя — в прямоугольнике B . Оставшуюся часть таблицы можно игнорировать. Теперь снова можем определить значения в средней строчке A (или B) за время, пропорциональное полному размеру A (или B). Следовательно, среднюю строчку верхней задачи можно найти за время, не превосходящее $c k^* n / 2$, а среднюю строчку нижней задачи — за время, не превосходящее $c(m - k^*) n / 2$. Эти два времени добавляются к $c n m / 2$. Рассмотрим полную идею вычисления оптимального выравнивания S_1 и S_2 .

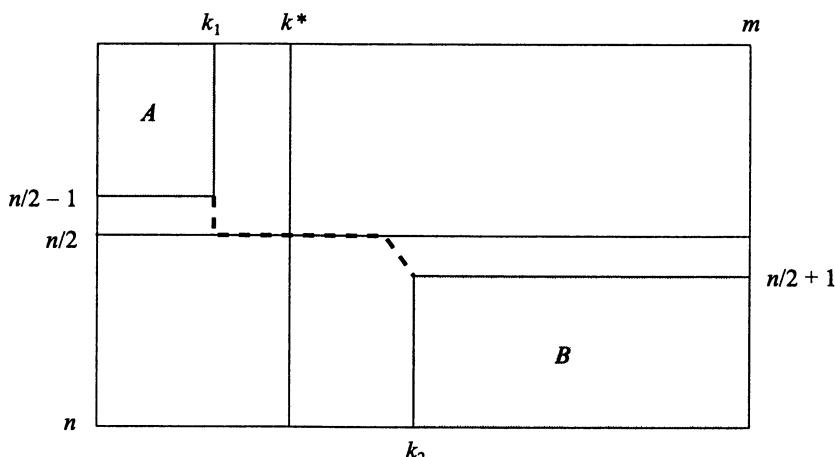


Рис. 12.2. После нахождения k^* задача выравнивания сводится к нахождению оптимального выравнивания в прямоугольнике A и еще одного оптимального выравнивания в прямоугольнике B исходной таблицы. Общая площадь подтаблиц A и B не превосходит $c n m / 2$. Подпуть $L_{n/2}$ через клетку $(n/2, k^*)$ показан пунктирной линией

12.1.3. Полная идея: использовать рекурсию

Сведя исходную задачу выравнивания n на m (для S_1 и S_2) к двум меньшим задачам выравнивания (верхней и нижней) и использовав при этом время $O(nm)$ и память $O(m)$, решаем теперь верхнюю и нижнюю задачи, рекурсивно используя то же самое сведение. (До настоящего момента мы игнорируем память, необходимую для сохранения подпутей L .) Применяя в точности ту же идею, что и при нахождении k^* в задаче $n \times m$, алгоритм использует память $O(m)$ для нахождения наилучшего столбца в строчке $n/4$ и разбивает верхнюю задачу выравнивания размера $n/2 \times k_1$. Затем он снова использует ту же память $O(m)$ для нахождения наилучшего столбца и разбиения нижней задачи выравнивания размера $n/2 \times (m - k_2)$. Иначе говоря, у нас есть две задачи выравнивания: одна — на таблице размера не более $n/2 \times k^*$, а другая — размера не более $n/2 \times (m - k^*)$. Поэтому можно найти наилучшие столбцы в средних строчках этих подзадач за время, не превышающее $c nk^*/2 + cn(m - k^*)/2 = c nm/2$, и рекурсивно перейти к решению четырех подзадач.

Продолжая действовать рекурсивным способом, мы можем найти оптимальное выравнивание двух исходных строк, проходя $\log_2 n$ уровней рекурсии, и ни на одном шаге нам не потребуется больше чем $O(m)$ памяти. Для удобства предположим, что n является степенью 2, так что любое деление пополам дает целое число. При каждом рекурсивном вызове мы находим и сохраняем подпуть оптимального пути L , но эти подпути дизъюнктны по дугам, и их полная длина равна $O(n + m)$. В итоге рекурсивный алгоритм, который нам нужен, таков *):

Алгоритм Хиршберга для оптимального выравнивания в линейной памяти

```
procedure OPTA( $l, l', r, r'$ );
```

```
begin
```

```
     $h := (l' - l)/2;$ 
```

В памяти $O(l' - l) = O(m)$ найти такой индекс k^* , $l \leq k^* \leq l'$,

что имеется оптимальное выравнивание $S_1[l..l']$ и $S_2[r..r']$,

состоящее из оптимального выравнивания $S_1[l..h]$ и $S_2[r..k^*]$,

за которым следует оптимальное выравнивание $S_1[h + 1..l']$ и $S_2[k^* + 1..r']$.

Найти также и сохранить подпуть L_h , который является частью

оптимального (самого длинного) пути L' из клетки (l, r) в клетку (l', r')

и начинается из последней клетки k_1 на L' на строчке $h - 1$

и кончается в первой клетке k_2 на L' в строчке $h + 1$.

{Это делается так, как описано выше.}

Вызов *OPTA*($l, h - 1, r, k + 1$); {новая верхняя задача}

Вывод подпути L_h ;

Вызов *OPTA*($h + 1, l', k_2, r'$); {новая нижняя задача}

```
end.
```

Вычисления начинаются с вызова *OPTA*($1, n, 1, m$). Отметим, что подпуть L_h выводится между двумя вызовами *OPTA* и что верхняя задача вызывается до нижней.

^{*)} Читателю полезно подумать об условиях завершения этого алгоритма. — *Прим. перев.*

Ввиду этого подпути выводятся в порядке возрастания значений h , так что их сцепка описывает оптимальный путь L из $(0, 0)$ в (n, m) и, следовательно, оптимальное выравнивание S_1 и S_2 .

12.1.4. Временной анализ

Мы видели, что первый уровень рекурсии потребовал времени $c n m$, а второй уровень — не более $c n m / 2$. На i -м уровне рекурсии получается 2^{i-1} подзадач, каждая из которых имеет $n/2^{i-1}$ строчек, но переменное число столбцов. Однако столбцы в этих подзадачах различны, так что полный размер всех подзадач не превосходит общего числа столбцов, $m \times n/2^{i-1}$. Следовательно, полное время на i -м уровне рекурсии не превосходит $c n m / 2^{i-1}$. Окончательный прогон динамического программирования для описания оптимального выравнивания требует времени $c n m$. Поэтому мы получаем следующую теорему.

Теорема 12.1.1. *При использовании процедуры Хиршберга OPTA оптимальное выравнивание двух строк длиной n и m может быть найдено с затратами времени $\sum_{i=1}^{\log n} c n m / 2^{i-1} \leq 2c n m$ и памяти $O(m)$.*

Для сравнения напомним, что исходным методом, в котором заполняется таблица динамического программирования размера $n \times m$, затрачивается время $c n m$. Метод Хиршберга уменьшает затраты памяти с $\Theta(nm)$ до $\Theta(m)$, лишь удваивая оценку наихудшего случая для необходимого времени вычислений.

12.1.5. Обобщение на локальные выравнивания

Метод Хиршберга с линейной памятью для задачи глобального выравнивания легко приспособить к решению задачи локального выравнивания строк S_1 и S_2 . Напомним, что оптимальное локальное выравнивание S_1 и S_2 находит подстроки α и β , глобальное выравнивание которых имеет значение, максимальное среди всех пар подстрок. Следовательно, если подстроки α и β можно найти, ограничившись линейной памятью, то их выравнивание можно построить в линейной памяти, используя метод Хиршберга для глобального выравнивания.

По теореме 11.7.1 значение оптимального локального выравнивания помещается в клетке (i^*, j^*) , содержащей максимальное значение v . Индексы i^* и j^* определяют концы строк α и β , глобальное выравнивание которых имеет максимальное значение сходства. Значения v можно вычислять построчечно, и алгоритм должен сохранять значения только двух строчек одновременно. Следовательно, конечные позиции i^* и j^* можно сосчитать в линейной памяти. Чтобы найти начальные позиции этих двух строк, алгоритм может выполнить инвертированную задачу ДП, используя линейную память (детали оставляются читателю). Можно рассмотреть вариант алгоритма ДП для v , в котором для каждой клетки (i, j) ссылка $h(i, j)$ назначалась бы следующим образом: если $v(i, j) = 0$, то $h(i, j)$ указывает на (i, j) , а если $v(i, j) > 0$ и нормальная обратная ссылка указывает на (p, q) , то $h(i, j) = h(p, q)$. При таком способе $h(i^*, j^*)$ определяет начальные позиции подстрок α и β соответственно. Так как α и β могут быть получены в линейной памяти, задача локального выравнивания может быть решена за время $O(nm)$ в памяти $O(n)$. Более глубоко этот вопрос рассмотрен в [232, 97].

12.2. Ускоренные алгоритмы для случая ограниченного числа различий

В пп. 9.4 и 9.5 рассмотрено несколько задач выравнивания и совпадения, где число разрешаемых несовпадений ограничивалось параметром k . Были получены алгоритмы, которые считают быстрее, чем без заданной границы. В частности, была задача о k несовпадениях, в которой разыскиваются все места в тексте T , куда образец P входит с не более чем k несовпадениями. Прямое решение этой задачи методами ДП требует времени $O(nt)$ для образца длины n и текста длины t . Но в п. 9.4 мы предложили решение трудоемкости $O(km)$, основанное на использовании суффиксного дерева и не требующее динамического программирования.

Результат с трудоемкостью $O(km)$ для задачи о k несовпадениях полезен потому, что многие приложения ищут только точные или почти точные вхождения P в T . Вдохновляясь теми же типами приложений (и дополнительными приложениями, рассмотренными в п. 12.2.1), мы перенесем этот результат на случай, когда разрешаются и несовпадения, и пробелы (включения и удаления с точки зрения редакционного расстояния). Мы используем термин “различия”, чтобы объединить несовпадения и пробелы.

Две конкретные задачи с ограничениями на различия

Рассмотрим две конкретные задачи: задачу глобального выравнивания с k различиями и более сложную задачу неточного совпадения с k различиями. Этот материал появился впервые в статьях Укконена [439], Фиккета [155], Майерса [341], а также Ландау и Вишкина [289]. Последняя статья была продолжена и иллюстрирована биологическими приложениями в статье Ландау, Вишкина и Нуссинова [290]. Дальнейшие алгоритмические исследования, использующие предположение о малости числа различий, см. в [341, 345, 342, 337, 483, 94, 93, 95, 373, 440, 482, 413, 414, 415]. Близкий вопрос об алгоритмах с малым ожидаемым временем работы будет изучен в п. 12.3.

Определение. Пусть заданы строки S_1 и S_2 и фиксированное число k . Задачей глобального выравнивания с k различиями называется задача о поиске наилучшего глобального выравнивания S_1 и S_2 , содержащего не более k несовпадений и пробелов (если такое существует).

Эта задача является частным случаем задачи о редакционном расстоянии. Для обращения к ней нужна уверенность, что S_1 и S_2 достаточно похожи. Она встречается как подзадача в более сложных задачах обработки строк, таких как задача о приближенном праймере PCR, которая будет рассмотрена в п. 12.2.5. Решение ее будет использовано для ускорения поиска глобального выравнивания, когда граница k не задана.

Определение. Пусть заданы строки P и T . Задачей неточного совпадения с k различиями называется задача нахождения всех (если такие есть) совпадений P с T , использующих не более k замен, вставок и удалений символов. Это значит, что требуется найти все вхождения P в T с не более чем k несовпадениями и пробелами. (Концевые пробелы в T бесплатны, а в P — нет.)

Допущение, вдобавок к несовпадениям, пробелов делает возможной более устойчивую версию задачи о k несовпадениях, рассматриваемой в п. 9.4, но оно усложняет задачу. В отличие от нашего решения задачи о k несовпадениях задача о k различиях требует, как кажется, использования динамического программирования. Подход, который мы применяем, заключается в ускорении основного решения ДП со временем $O(nm)$ за счет предположения, что интересны исключительно выравнивания с не более чем k различиями.

12.2.1. Где встречаются задачи с ограниченными различиями?

Существует обширная (и все растущая) литература по информатике, посвященная алгоритмам, эффективность которых основана на предположении об ограниченности числа различий (см. в [93] обзор и сравнение некоторых из них, включая один дополнительный метод). Поэтому, прежде чем углубляться в конкретные алгоритмические результаты, уместно спросить, настолько ли часто встречаются задачи с ограниченными различиями, чтобы оправдать такие широкие исследования.

Задачи с ограниченными различиями естественно возникают в ситуациях, когда текст многократно модифицируется (редактируется). Выравнивание текста до и после модификации может выделить измененные места. Близкое приложение [345] связано с обновлением графического экрана после внесения правок в изображаемый на нем текст. Предположение о добавляемых изменениях на экране заключается в том, что исправления текста невелики и поступают в достаточно медленном темпе, так что пользователь может их наблюдать. Выравнивание старого и нового текста определяет те незначительные изменения, которые нужно сделать, чтобы отображался новый текст. Графические дисплеи с произвольным доступом могут использовать эту информацию для очень быстрого обновления экрана. Такой подход применяется рядом текстовых редакторов. Здесь эффекты ускорения хорошо видны и зачастую впечатляющи.

12.2.2. Иллюстрации из молекулярной биологии

В биологических приложениях выравнивания менее видна обоснованность введения числа разрешенных (или ожидаемых) различий между строками. Некоторые специалисты по информатике открыто утверждали, что методы выравнивания с ограничениями различий не имеют отношения к биологии. Конечно, главные задачи о выравнивании и сравнении биологических последовательностей идут от строк (обычно это белок) с очень *малым* сходством. Тут возражений нет. Однако в молекулярной биологии есть много задач о последовательностях (в особенности задач, касающихся геномных и управляющих ДНК, а не белков), где следует ограничивать числа разрешенных (или ожидаемых) различий. Несколько часов листания биологических журналов дадут множество таких примеров.* Мы уже обсуждали одно приложение, связанное с поиском STS и EST во вновь расшифрованных ДНК (см. п. 7.8.3). Упоминалась также задача приближенного построения праймера для PCR, которая

* Я недавно присутствовал на собрании, связанном с проектом “Геном человека”, где в выступлениях приводились многочисленные примеры. После десятого я перестал записывать.

будет подробно рассмотрена в п. 12.2.5. Упомянем несколько дополнительных примеров биологических задач выравнивания, где целесообразно введение ограничения на число различий.

Чанг и Лаулер [94] отмечают, что существующие методы сборки последовательностей ДНК (см. пп. 16.14 и 16.15.1) решают огромное число раз задачу приближенного суффиксно-префиксного совпадения. Эти методы вычисляют для каждой пары строк S_1, S_2 из большого набора строк наилучшее совпадение суффикса S_1 с префиксом S_2 , где совпадение может содержать “скромную” долю различий. В стандартных методах ДП суффиксно-префиксные вычисления занимали до 90 % вычислительного времени, используемого в проектах прежних лет для сборки последовательностей [363]. Но в указанном приложении интересны только те суффиксно-префиксные совпадения, в которых число различий достаточно скромно. Согласно этому хорошо бы использовать более быстрый алгоритм, явно учитывающий это предположение. Родственная задача встречается в методе расшифровки “ВАС–РАС”, работающем с сотнями тысяч выравниваемых последовательностей (см. п. 16.13.1).

Другой пример касается локализации генов, мутации которых вызывают определенные генетические болезни или способствуют их возникновению. Основная идея заключается в том, что прежде всего надо идентифицировать (с помощью анализа генетического сцепления, анализа функций или другими средствами) ген или содержащий его участок, который вызывает рассматриваемую болезнь или способствует ее развитию. Затем копии этого гена или содержащего его участка получают (и дешифруют) от людей, пораженных этой болезнью, и от здоровых (обычно от родственников). Расшифрованные ДНК от пораженных и непораженных индивидуумов сравнивают, чтобы найти значимые различия. Так как многие генетические болезни вызываются очень малыми изменениями в гене (возможно, изменением, удалением или инверсией одного основания), задача подразумевает сравнение строк, имеющих очень маленькое число различий. Систематическое исследование генных полиморфизмов (различий) — быстро развивающаяся область исследований. Уже есть базы данных, содержащие все различные последовательности, найденные для некоторых конкретных генов. Эти последовательности в целом очень похожи друг на друга, так что средства для выравнивания и действий со строками, предполагающие ограниченное число различий, будут для таких последовательностей полезны.

Сходная ситуация встречается в развивающейся области “молекулярной эпидемиологии”, где пытаются прослеживать историю передачи патогена (обычно вируса), геном которого быстро мутирует. Для детального анализа изменяющейся вирусной ДНК или РНК требуется сравнение очень похожих строк. Выравнивание пар таких строк для обнаружения их сходств и различий — первый шаг к восстановлению их истории и вероятной последовательности их мутирования. После этого история мутаций представляется в виде дерева эволюции (см. главу 17).

Этим способом изучались коллекции возбудителей СПИДа — вирусов иммунодефицита человека (ВИЧ). Другой пример молекулярной эпидемиологии [348] — изучение истории инфекции *Hantavirus* на юго-западе США, появившейся в начале 1990-х годов.

Два заключительных примера взяты из фундаментальной статьи [162], сообщившей о первой полной расшифровке ДНК свободно живущего организма, бактерии *Haemophilus influenzae Rd*. Геном этой бактерии состоит из 1 830 137 пар оснований,

и его полная последовательность была получена дробовым секвенированием без первоначального картирования (см. п. 16.14). Перед выполнением этого крупномасштабного проекта расшифровки различными группами были секвенированы многие небольшие отдельные части генома этой бактерии, и полученные последовательности введены в базы данных по ДНК. Одним из способов проверки качества крупномасштабной расшифровки, стало сравнение, когда это было возможно, вновь полученной последовательности с расшифрованными ранее. При невозможности сопоставить новые последовательности с уже имеющимися, получив при этом мало различий, принимались дополнительные меры, чтобы достичь уверенности в корректности новых последовательностей. Цитируем: “Результаты такого сравнения показывают, что наша последовательность на 99.67 % совпадает с последовательностями из GenBank, аннотированными как *H. influenzae Rd*” [162].

С точки зрения выравнивания в рассмотренной выше задаче нужно определить, совпадают или нет новые последовательности со старыми с точностью до нескольких различий. Это приложение иллюстрирует оба введенных ранее типа задач выравнивания с ограниченными различиями. Когда известно положение в геноме последовательности из базы данных, из полной последовательности может быть выделена для сравнения соответствующая строка. Задача сравнения становится конкретной задачей глобального выравнивания с k различиями, рассматриваемой в п. 12.2.3. Когда же положение в геноме последовательности P не известно (обычная ситуация), задача сравнения заключается в отыскании всех мест в полной последовательности, где встречается P с очень малым числом допустимых различий. Это уже конкретная задача о неточном совпадении с k различиями, которая будет рассмотрена в п. 12.2.4.

Изложенная история про расшифровку *H. influenzae* будет часто повторяться, когда начнутся систематические крупномасштабные расшифровки ДНК различных организмов. Каждая полная последовательность будет сверяться с более короткими последовательностями для конкретного организма, уже записанными в базы данных. Это необходимо не только для контроля качества крупномасштабного секвенирования, но также и для проверки записей в базах данных, так как есть уверенность, что крупномасштабная расшифровка дает более корректные результаты.

Второе приложение из [162] относится к построению *неизбыточных* баз данных бактериальных белков (*nonredundant bacterial proteins* — NRBP). По ряду причин например, для ускорения поиска или для улучшения оценки статистической значимости найденных совпадений) полезно уменьшить число записей в базе данных для последовательностей (в данном случае последовательностей бактериальных белков) за счет отбраковки или какого-либо комбинирования очень похожих “избыточных” последовательностей. Это было сделано в работе, представленной в [162], и “неизбыточная” версия GenBank регулярно обновляется Национальным центром биотехнологической информации США. Согласно Флейшману и др.:

“Избыточность была удалена из NRBP в два этапа. Из Genbank были выбраны все кодовые последовательности ДНК... и последовательности из одного и того же вида сравнивались друг с другом. Последовательности, имеющие совпадение более 97 % на участках длиннее 100 нуклеотидов, комбинировались. Кроме того, эти последовательности транслировались и использовались для сравнения со всеми белковыми последовательностями в SwissProt... Последовательности, принадлежащие одинаково видам и имеющие больше 98 % сходства на участках длиннее 33 аминокислот, комбинировались” [162].

Похожий пример рассматривался в [399], где около 170 000 последовательностей ДНК “подверглись процедуре оптимального выравнивания для обнаружения пар последовательностей с совпадением не менее 97 %”. В этих задачах выравнивания можно задавать границу для числа разрешенных различий. Выравнивания, которые превышают эту границу, интереса не представляют — вычисления нужны только для того, чтобы определить, будут ли две последовательности “достаточно похожи”. Более того, так как эти приложения подразумевают большое число выравниваний (каждой записи из базы данных со всеми остальными), эффективность метода очень важна.

Общепризнано, что не каждая биологическая задача выравнивания с ограничением различий требует изощренного алгоритма. Но приложения столь массовы, размеры некоторых из них столь велики и столь велико ускорение, что кажется непродуктивным полностью упускать пользу методов с ограничением различий или несовпадений для молекулярной биологии. Имея такую мотивировку, мы обсудим специфическую технику эффективного решения задач выравнивания с ограничением числа различий.

12.2.3. Глобальное выравнивание с k различиями

Задача заключается в том, чтобы найти наилучшее глобальное выравнивание при дополнительном условии, что оно содержит не более k несовпадений и пробелов, где значение k задано. Цель в том, чтобы уменьшить границу используемого при решении времени с $O(nm)$ (требуемого при стандартном динамическом программировании) до $O(km)$. Основа подхода — вычисление *редакционного расстояния* между S_1 и S_2 с использованием ДП, но с заполнением только части таблицы размером $O(km)$.

Главное наблюдение гласит: если мы определим *главную диагональ* таблицы ДП как совокупность клеток (i, i) для $i \leq n \leq m$, то любой путь в таблице ДП, который определяет глобальное выравнивание с k различиями, не должен содержать никакой клетки вида $(i, i + l)$ или $(i, i - l)$, где $l > k$ (рис. 12.3). Чтобы понять это, отметим, что любой путь, определяющий глобальное выравнивание, начинается на главной диагонали (в клетке $(0, 0)$) и заканчивается на главной диагонали или справа от нее

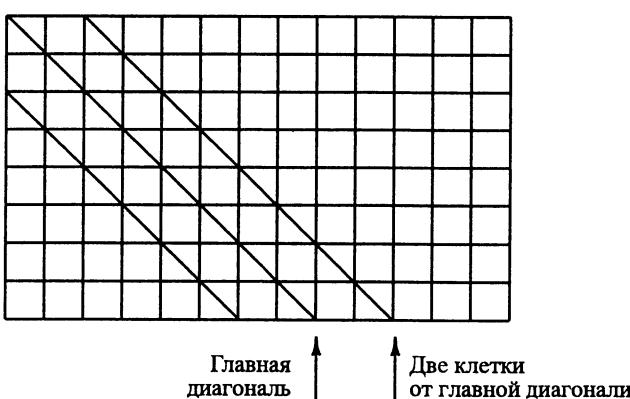


Рис. 12.3. Главная диагональ и полоса при $k = 2$

(в клетке (n, m)). Поэтому путь должен включать в выравнивание по одному пробелу на каждый горизонтальный шаг пути в сторону от главной диагонали. Таким образом, только те пути, которые нигде не удаляются по горизонтали от главной диагонали более чем на k клеток, претендуют быть глобальным выравниванием с k различиями. (Отсюда следует, что неравенство $m - n \leq k$ является необходимым условием существования решения.) Поэтому, чтобы найти какое-либо глобальное выравнивание с k различиями, достаточно заполнить в таблице ДП полосу, состоящую из $2k + 1$ клеток в каждой строчке и центрованную по главной диагонали. При вычислении значений в клетках этой полосы алгоритм использует те же рекуррентные соотношения для редакционного расстояния всюду, кроме клеток на верхней и нижней границе полосы. Любая клетка на верхней границе игнорирует в соотношениях член для клетки выше ее (так как эта клетка вышла из полосы); аналогично, любая клетка на нижней границе игнорирует в соотношениях член влево от нее. Если $m = n$, размер полосы можно наполовину уменьшить (упражнение 4).

Если глобального выравнивания S_1 и S_2 с k или менее различиями не существует, то значение, полученное для клетки (n, m) , будет больше k . В этом случае оно не обязательно равно корректному редакционному расстоянию между S_1 и S_2 , но оно покажет, что корректное значение для (n, m) больше k . Обратно, если имеется глобальное выравнивание с $d \leq k$ различиями, то соответствующий путь содержится внутри полосы, и таким образом, значение в клетке (n, m) будет корректно установлено равным d . Полная площадь полосы равна $O(kn)$, что равно $O(km)$, так как n и m могут отличаться не более чем на k . В итоге:

Теорема 12.2.1. *Глобальное выравнивание S_1 и S_2 с не более чем k различиями существует в том и только том случае, если приведенный алгоритм записывает в клетку (n, m) значение, не превосходящее k . Следовательно, задача глобального выравнивания с k различиями может быть решена с затратами времени $O(km)$ и памяти $O(km)$.*

А если k не задано?

Представленное выше решение может быть использовано в несколько ином контексте. Предположим, что редакционное расстояние между S_1 и S_2 равно k^* , но мы не знаем этого значения k^* и не можем его заранее ограничить. Непосредственное решение с помощью динамического программирования для вычисления редакционного расстояния, k^* , требует $\Theta(nm)$ времени и памяти. Мы уменьшим эти границы до $\Theta(k^*m)$. Таким образом, когда редакционное расстояние мало, метод считает быстро и использует мало памяти, а когда велико, он использует $O(nm)$ времени и памяти, как при стандартном использовании ДП.

Идея такова: изменения k , мы предполагаем, что $k^* \leq k$, и проверяем с помощью теоремы 12.2.1, что граница k достаточно велика. Метод начинает с $k = 1$ и проверяет, не найдется ли глобальное выравнивание с не более чем одним различием. Если найдется, то лучшее глобальное выравнивание (с нулем или одним различием) уже найдено. Если нет, то метод удваивает k и снова смотрит, нет ли глобального выравнивания с k различиями. На каждой последовательной итерации метод удваивает k и проверяет, не будет ли текущее k достаточным. Процесс продолжается, пока не найдется глобальное выравнивание с не более чем k различиями для текущего значения k . Когда алгоритм останавливается, наилучшее выравнивание в полосе

(ширины k по каждую сторону от главной диагонали) должно иметь значение k^* . Причина в том, что пути выравнивания делятся на два типа: содержащиеся в текущей полосе и выходящие из нее. Полученное выравнивание является наилучшим из выравниваний первого типа, а любой путь, который выходит из полосы, соответствует выравниванию с более чем k пробелами. Отсюда следует, что текущее значение в клетке (n, m) должно равняться k^* .

Теорема 12.2.2. *Последовательным удваиванием k до нахождения глобального выравнивания с k различиями редакционное расстояние k^* и соответствующее ему выравнивание вычисляются с затратами времени и памяти $O(k^*m)$.*

Доказательство. Пусть k' — наибольшее значение k , использованное методом. Ясно, что $k' \leq 2k^*$. Тогда затраты в методе равны

$$O(k'm + k'm/2 + k'm/4 + \dots + m) = O(k'm) = O(k^*m). \quad \square$$

12.2.4. Возвращение суффиксного дерева: неточное совпадение с k различиями

Рассмотрим теперь задачу неточного совпадения образца P с текстом T с учетом требования, чтобы число различий не превосходило k . Это обобщение задачи о k несовпадениях, но более трудное, поскольку кроме несовпадений разрешаются пробелы. Задача о k несовпадениях была решена с использованием только суффиксных деревьев, но суффиксные деревья — недостаточно хорошая структура, чтобы справиться с ошибками в виде вставок и удалений. Задача неточного совпадения с k различиями最难 and задачи о глобальном выравнивании с k различиями, так как мы ищем выравнивание P и T , где концевые пробелы, появляющиеся в T , не учитываются. Поэтому размеры P и T могут сильно отличаться, и при расчете нельзя ограничиться путями, которые отстоят от главной диагонали не больше чем на k клеток.

Тем не менее мы снова получим метод с временем и памятью $O(km)$, комбинируя динамическое программирование с возможностью отвечать на запросы о наибольшем общем продолжении за константное время (см. п. 9.1). Этот результат будет первым из нескольких примеров гибридного динамического программирования, в которых суффиксные деревья используются для решения подзадач в рамках вычислительной схемы ДП. Результат со временем $O(km)$ был впервые получен Ландау и Вишким [287], а также Майерсом [341] и обобщен в ряде работ. Хорошие обзоры различных методов для этой задачи представлены в [93, 421].

Определение. Как и раньше, главная диагональ таблицы ДП размера $n \times m$ состоит из клеток (i, i) , где $0 \leq i \leq n \leq m$. Диагонали выше главной нумеруются от 1 до m ; диагональ, начинающаяся в клетке $(0, i)$, — это диагональ i . Диагонали ниже главной нумеруются от -1 до $-n$; диагональ, начинающаяся в клетке $(i, 0)$ — это диагональ $-i$ (рис. 12.4).

Так как концевые пробелы в тексте T бесплатны, строчка 0 в таблице ДП заполняется нулями. Это позволяет левому концу T располагаться напротив пропуска без какого-либо штрафа.

Определение. *d-путь* в таблице ДП — это путь, начинающийся в нулевой строчке и определяющий ровно d несовпадений и пробелов.

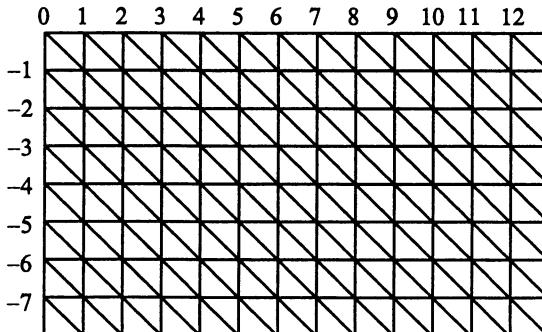


Рис. 12.4. Нумерованные диагонали в таблице динамического программирования

Определение. d -путь называется *самым продвинутым по диагонали i* , если это d -путь, который кончается на диагонали i , и индекс столбца c , где стоит его конец (на диагонали i), не меньше, чем у любого другого d -пути, кончающегося на диагонали i .

Графически d -путь является самым продвинутым по диагонали i , если никакой другой d -путь не доходит до более далекой клетки по диагонали i .

Гибридное динамическое программирование: общая идея

В целом, это метод трудоемкости $O(km)$, выполняющий k итераций, каждая из которых занимает время $O(m)$. На итерации $d \leq k$ метод находит концы самых продвинутых d -путей на всех диагоналях i , где i меняется от $-n$ до m . Самый продвинутый d -путь на диагонали i находится из самых продвинутых $(d-1)$ -путей на диагоналях $i-1$, i и $i+1$. Это будет подробно объяснено ниже. Любой самый продвинутый d -путь, достигающий строчки n , определяет положение конца (в T) вхождения P с d различиями. Мы реализуем каждую итерацию за время $O(n+m)$, что даст желаемую границу времени $O(km)$. Аналогично будет оценена требуемая память.

Подробности

Начнем с того, что при $d = 0$ самый продвинутый 0-путь, кончающийся на диагонали i , соответствует наибольшему общему продолжению $T[i..m]$ и $P[1..n]$, так как 0-путь не допускает несовпадений или пробелов. Поэтому самый продвинутый 0-путь, заканчивающийся на диагонали i , можно найти за константное время, как описано в п. 9.1.

Для $d > 0$ самый продвинутый d -путь на диагонали i можно найти, рассматривая следующие три пути, кончающиеся на диагонали i .

- Путь R_1 состоит из самого продвинутого $(d-1)$ -пути на диагонали $i+1$, затем из вертикальной дуги (пробел в тексте T), затем из максимального продолжения вдоль диагонали i , которое соответствует совпадающей подстроке P и T (рис. 12.5). Так как R_1 начинается с $(d-1)$ -пути и добавляет один пробел на вертикальной дуге, то R_1 является d -путем.

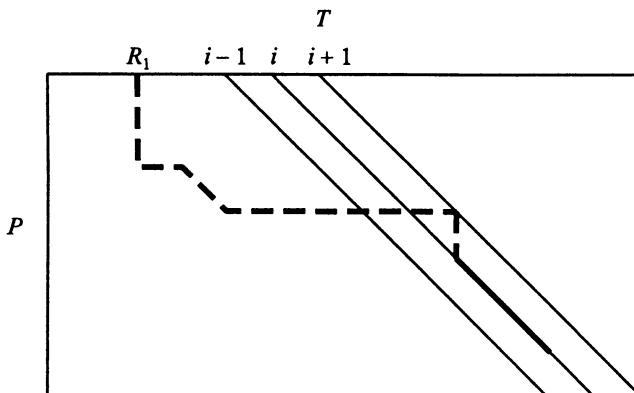


Рис. 12.5. Путь R_1 состоит из самого продвинутого $(d - 1)$ -пути на диагонали $i + 1$ (штриховая линия), затем из вертикальной дуги (точки), которая добавляет к выравниванию разность d , затем из максимального пути (сплошная линия) по диагонали i , который соответствует (максимальной) совпадающей подстроке P и T

- Путь R_2 состоит из самого продвинутого $(d - 1)$ -пути на диагонали $i - 1$, затем из горизонтальной дуги (пробела в образце P) до диагонали i , затем из максимального продолжения вдоль диагонали i , которое соответствует совпадающей подстроке P и T . Путь R_2 является d -путем.
- Путь R_3 состоит из самого продвинутого $(d - 1)$ -пути по диагонали i , затем из диагональной дуги, соответствующей несовпадению символа из P и символа из T , затем из максимального продолжения вдоль диагонали i , которое соответствует совпадающей подстроке P и T . Путь R_3 является d -путем (рис. 12.6).

Каждый из путей R_1 , R_2 и R_3 оканчивается максимальным продолжением, соответствующим совпадающей подстроке P и T . В случае R_1 (или R_2) начальные

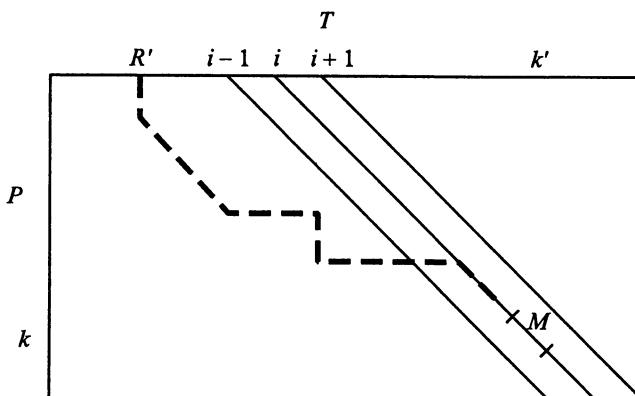


Рис. 12.6. Штриховая линия показывает путь R' — самый продвинутый $(d - 1)$ -путь, оканчивающийся на диагонали i . Дуга M на диагонали i сразу после окончания R' должна соответствовать несовпадению P и T (участвуют в несовпадении символы $P(k)$ и $T(k')$)

позиции этих двух подстрок задаются последней точкой выхода R_1 (или R_2) на диагональ i . В случае R_3 начальной позицией будет последнее несовпадение на R_3 .

Теорема 12.2.3. *Все три пути R_1 , R_2 и R_3 являются d -путями, заканчивающимися на диагонали i . Самый продвинутый d -путь на диагонали i — это тот из путей R_1 , R_2 или R_3 , который дальше всего продвинут по диагонали i .*

Доказательство. Каждый из этих трех путей является продолжением $(d - 1)$ -пути, и каждое продолжение добавляет либо один пробел, либо одно несовпадение. Следовательно, каждый из них — это d -путь, кончающийся, по определению, на диагонали i . Таким образом, самый продвинутый d -путь на диагонали i должен быть либо самым продвинутым из R_1 , R_2 и R_3 , либо должен продвигаться по диагонали i дальше, чем любой из них.

Пусть R' — самый продвинутый $(d - 1)$ -путь на диагонали i . Дуга графа выравнивания вдоль диагонали i , которая следует немедленно после R' , должна соответствовать несовпадению, в противном случае R' не был бы самым продвинутым $(d - 1)$ -путем на i . Пусть M обозначает эту дугу (см. рис. 12.6).

Пусть R^* обозначает самый продвинутый d -путь на диагонали i . Так как R^* кончается на диагонали i , то существует точка, в которой R^* ходит на диагональ i и затем никогда ее не покидает. Если эта точка находится выше дуги M , то R^* должна проходить дугу M , в противном случае R^* не может продвинуться так далеко, как R_3 . Когда путь R^* достигает точки M (обозначающей конец R'), он должен также иметь $(d - 1)$ различий; если эта часть пути R^* имеет меньше чем $(d - 1)$ различий, то она может пройти через M , создав $(d - 1)$ -путь на диагонали i , и этот путь продвигается дальше по диагонали i , чем R' , вопреки определению R' . Отсюда следует, что если путь R^* выходит на диагональ i выше M , то он будет иметь d различий после прохода M и, значит, будет кончаться точно там же, где кончается R_3 . Итак, если R^* не совпадает с R_3 , то R^* должен выходить на диагональ i ниже дуги M .

Предположим, что R^* выходит на диагональ i в последний раз ниже дуги M . Тогда R^* должен в этой точке иметь d различий; если он имеет меньше различий, то R' не смог бы быть самым продвинутым $(d - 1)$ -путем на диагонали i . Далее, выход R^* на диагональ i происходит либо с диагонали $i - 1$, либо с диагонали $i + 1$. Пусть будет $i + 1$ (случай $i - 1$ симметричен). Итак, R^* переходит вертикальной дугой с диагонали $i + 1$ на диагональ i , добавляя к R^* пробел. Это означает, что точка, в которой R^* уходит с диагонали $i + 1$, определяет $(d - 1)$ -путь на диагонали $i + 1$. Следовательно, R^* покидает диагональ $i + 1$ в той же точке, что R_1 или выше. Тогда R_1 и R^* имеют каждый по d пробелов или несовпадений в точках, где они в последний раз выходят на диагональ i , и каждый из них движется вдоль диагонали i до достижения дуги с несовпадением. Отсюда следует, что R^* не может продвинуться дальше, чем R_1 , по диагонали i . Таким образом, R^* кончается точно там же, где R_1 .

Случай, когда R^* выходит на диагональ i в последний раз с диагонали $i - 1$, симметричен. При этом R^* кончается в точности там же, где R_2 . Итак, мы показали, что в любом случае R^* , предполагаемый самым продвинутым d -путь на диагонали i , заканчивается в конечной точке одного из путей R_1 , R_2 или R_3 . Следовательно, самый продвинутый d -путь на диагонали i — это самый продвинутый из R_1 , R_2 и R_3 . \square

Теорема 12.2.3 лежит в основе метода со временем $O(km)$.

Гибридное динамическое программирование: алгоритм k различий

begin

$d := 0$

for $i := 0$ **to** m **do**

 Найти наибольшее общее продолжение $P[1..n]$ и $T[i..m]$.

 Оно определяет конечный столбец самого продвинутого

0 -пути на диагонали i .

for $d := 0$ **to** k **do begin**

for $i := -n$ **to** m **do begin**

 Используя самые продвинутые $(d - 1)$ -пути на диагоналях $i, i - 1$

 и $i + 1$, найти на диагонали i концы путей R_1, R_2 и R_3 . Самый

 продвинутый из этих трех путей является самым продвинутым
 d -путем на диагонали i ;

end;

end;

 Любой путь, достигающий строчки n , скажем, в столбце c

 определяет неточное вхождение P в T , заканчивающееся

 в символе c строки T и содержащее не более k различий.

end.

Реализация и временной анализ

Для каждого значения d и каждой диагонали i мы запоминаем тот столбец в диагонали i , где кончается самый продвинутый d -путь. Так как d изменяется от 0 до k и имеется только $O(n + m)$ диагоналей, все эти значения можно сохранить в памяти $O(km)$. На итерации d алгоритму нужны только значения, вычисленные на итерации $(d - 1)$. Полный набор значений может быть использован для построения какого-либо выравнивания P в T с не более чем k различиями. Мы оставляем подробности этого построения как упражнение.

Теперь перейдем к временному анализу. Для каждого d и i должны выдаватьсь концы трех $(d - 1)$ -путей. При любых d и i это требует константного времени, так что получение концов путей занимает время $O(km)$ на весь алгоритм. Должно быть вычислено также $O(km)$ продолжений пути вдоль диагонали. Но каждое такое продолжение соответствует максимальной совпадающей подстроке P и T , начинающейся с конкретных заданных позиций P и T . Следовательно, каждое продолжение пути требует поиска наибольшей подстроки, начинающейся в данном месте строки T и совпадающей с подстрокой, начинающейся в данном месте P . Другими словами, каждое продолжение пути требует вычисления наибольшего общего продолжения. В п. 9.1 на с. 245 мы показали, что наибольшие общие продолжения можно вычислять за константное время после линейного препроцессинга строк. Следовательно, $O(km)$ продолжений могут быть вычислены за общее время $O(n + m + km) = O(km)$. Более того, как показано в п. 9.1.2, такие продолжения могут быть найдены при использовании только этих двух строк и суффиксного дерева для наименьшей из строк. В итоге:

Теорема 12.2.4. *Все места в строке T , куда образец P входит с не более чем k различиями, можно найти, расходуя время $O(km)$ и память $O(km)$. Более того, за время $O(km)$ можно построить сами выравнивания P и T для этих мест.*

Иногда результат о k различиях излагается в более простом, но менее полезном виде, в котором расходуется меньше памяти. Если интересоваться только концами мест в T , где P и T совпадают с точностью до k различий, то границу памяти $O(km)$ можно заменить на $O(n + m)$. Идея в том, что концы самых продвинутых $(d - 1)$ -путей на каждой диагонали уже не понадобятся после итерации d и могут быть удалены. И тогда для этой более простой задачи потребуется память лишь $O(n + m)$.

Теорема 12.2.5. При затратах времени $O(km)$ и памяти $O(n + m)$ алгоритм может найти концы всех мест в T , где P совпадает с T с точностью до k различий.

12.2.5. Новый взгляд на задачу выбора праймера (и пробы): приложение совпадений с ограниченным различием

В упражнении 61 главы 7 был предложен вариант точного совпадения для задачи о выборе праймера (и пробы). Простейший случай начинает с двух строк α и β . Вариант точного совпадения таков:

Задача о праймере (пробе) с точным совпадением. Для каждого индекса j после некоторой начальной точки найти кратчайшую подстроку γ строки α (если такая есть), которая начинается в позиции j и не является подстрокой β .

Эту задачу можно решить за время, пропорциональное сумме длин строк α и β .

Вариант задачи в выборе праймера с точным совпадением может не полностью моделировать реальную задачу о выборе праймера (хотя, как уже отмечалось, вариант с точным совпадением может быть реален для выбора пробы). Напомним, что праймеры — это короткие подстроки ДНК, которые гибридизуются в нужной части строки α и в идеале не должны гибридизоваться в каких-либо частях другой строки β . Точное совпадение недекватно моделирует практическую гибридизацию потому, что подстрока ДНК может гибридизоваться, при подходящих условиях, с другой строкой ДНК даже без точного совпадения; для гибридизации может быть достаточно неточного совпадения подходящего типа. Более реалистичный вариант задачи о выборе праймера идет от точного совпадения к неточному следующим образом:

Задача о праймере с неточным совпадением. Пусть задан параметр p . Найти для каждого индекса j (после некоторой начальной точки) кратчайшую подстроку γ строки α (если такая существует), которая начинается с позиции j и имеет редакционное расстояние не меньше $|\gamma|/p$ от любой подстроки в β .

Мы решим эту задачу эффективно, рассматривая следующую задачу о k различиях:

Задача о праймере с k различиями. Пусть задан параметр k . Найти для каждого индекса j (после некоторой начальной точки) кратчайшую подстроку γ строки α , которая начинается с позиции j и имеет редакционное расстояние не меньше k от любой подстроки в β (если такая γ существует).

Замена $|\gamma|/p$ на k при переходе от одной постановки задачи к другой упрощает решение, не снижая его полезности. Причина в том, что длина практического праймера должна лежать в фиксированном и очень узком диапазоне, так что для

фиксированного p и $|y|/p$ лежит в узком диапазоне. Следовательно, для заданного p можно решить задачу о праймере с k различиями при нескольких значениях k и иметь возможность подобрать хороших кандидатов.

Как решить задачу о праймере с k различиями

Мы следуем подходу, предложенному в [243]. Метод проверяет каждую позицию j в α отдельно. Для любой позиции j задача превращается в такую:

Найти кратчайший префикс строки $\alpha[j..n]$ (если он существует), редакционное расстояние которого от каждой подстроки в β не меньше k .

Задача для фиксированного j по существу является “обратной” к задаче неточного сопоставления с точностью до k . В этой последней мы хотели найти подстроку T , с которой P совпадает при *не более чем* k различиях. А теперь мы хотим *браковать* все префиксы $\alpha[j..n]$, которые совпадают с подстрокой β с менее чем k различиями. “Картина перевернулась”, но можно пользоваться теми же средствами.

Решение заключается в применении алгоритма k различий со строкой $\alpha[j..n]$, играющей роль P , и β , играющей роль T . Алгоритм вычисляет самые продвинутые d -пути для $d = k$ на каждой диагонали. Если каким-нибудь из d -путей достигается строчка n при $d \leq k - 1$, то вся строка $\alpha[j..n]$ совпадает с подстрокой β с менее чем k различиями, так что приемлемого праймера, начинающегося с j , нет. Но если ни один из самых продвинутых ($k - 1$)-путей не достигает строчки n , то имеется приемлемый праймер, начинающийся с позиции j . Подробнее: если ни один самый продвинутый d -путь для $d = k - 1$ не достигает строчки $r < n$, то подстрока $\gamma = \alpha[j..r]$ отстоит на редакционное расстояние не меньшее k от любой подстроки в β . Более того, если r — наименьший номер строчки с этим свойством, то $\alpha[j..r]$ — кратчайшая подстрока, начинающаяся в j , которая имеет редакционное расстояние не меньше k от любой подстроки в β .

Приведенный алгоритм применяется к каждой потенциально возможной начальной позиции j из α , что дает следующую теорему.

Теорема 12.2.6. *Если α имеет длину n , а β — длину m , то задача о выборе праймера с k различиями может быть решена за полное время $O(knm)$.*

12.3. Методы исключения:

наименьшее ожидаемое время счета

Методы для k несовпадений и k различий, которые мы демонстрировали до сих пор, имели время счета в наихудшем случае порядка $\Theta(km)$. Для $k \ll n$ такое ускорение было значительным выигрышем в сравнении с оценкой $\Theta(nm)$ для обычного динамического программирования. Однако хотелось бы добиться еще большей эффективности, если $m = |T|$ велико. Типична ситуация, когда T представляет собой большую базу данных с последовательностями, а задача состоит в поиске приближенного вхождения образца P в T . Наша цель — получить методы, которые существенно быстрее, чем $\Theta(km)$, не в наихудшем случае, а для ожидаемого времени счета. Это реминисценция о том способе, с помощью которого метод Бойера–Мура, обычно

пропускающий большую часть текста, получает ожидаемое время счета, сублинейно зависящее от размера текста.

Для задач приближенного сопоставления было изобретено несколько методов с ожидаемым временем счета меньше чем $\Theta(km)$. Некоторые из них имеют ожидаемое время счета, которое *сублинейно* по m для разумного диапазона k . В них *точное* сопоставление искусно комбинируется с динамическим программированием и явно используют многие из идей, изложенных в первых двух частях книги. Хотя детали существенно различаются, но все обсуждаемые ниже методы имеют в общем родственный оттенок. Мы сосредоточимся на методах, предложенных Беза-Йетсом и Перлебергом [36], Чангом и Лаулером [94] и Майерсом [342], хотя только первый из них будет представлен и исследован во всех подробностях. Два других метода (By-Манбер [482] и Певзнера-Утермана [373]) будут только упомянуты. Эти методы не дают строгих *доказуемых* линейных и сублинейных оценок времени счета для всех практических диапазонов ошибок (и это остается прекрасной открытой задачей), но они достигают этой цели при “скромной” доле ошибок k/n .

Пусть σ — размер алфавита, используемого в P и T . Как обычно, $n = |P|$, $m = |T|$. Вхождение P в T с не более чем k ошибками (несовпадениями или различиями, в зависимости от конкретной задачи) будет называться *приближенным вхождением* P . В общих чертах большинство из этих методов можно описать так:

Приближенное сопоставление на основе разбиений

- Разбиение T или P на последовательные участки данной длины r (которая уточняется далее).
- Фаза поиска. Используя различные методы точного сопоставления, искать в T интервалы длины r (или участки, если T разбито), которые могут содержать приближенные вхождения P . Они называются *сохраняемыми* интервалами. Несохраняемые интервалы заведомо не содержат приближенных вхождений P , и цель этой фазы — исключить из рассмотрения по возможности больше интервалов.
- Фаза проверки. Для каждого сохраняемого интервала R использовать какой-либо метод приближенного сопоставления для явной проверки наличия приближенно-го вхождения P в больший интервал вокруг R .

Методы различаются, прежде всего, выбором r , разбиваемой строки и методов точного сопоставления, используемых в фазе поиска. Они отличаются также определением участка, но не зависят от конкретного выбора проверяющего алгоритма. Главная идея подхода с разбиениями — исключить большую часть T , используя только (суб)линейное ожидаемое время в фазе поиска, так что только (суб)линейное ожидаемое время потребуется для проверки немногих сохраняемых интервалов. Нужно балансировать между поиском и проверкой, так как уменьшение времени на одной фазе вызывает увеличение в другой фазе.

12.3.1. Метод BYP

Первый конкретный метод, который мы рассмотрим, принадлежит Р. Беза-Йетсу и С. Перлебергу [36] (BYP — это их инициалы). Его ожидаемое время счета оценивается как $O(m)$ для умеренной доли ошибок (уточняемой ниже).

Пусть $r = [n/(k+1)]$. Разобьем P на последовательные участки длины r (длина последнего участка может быть меньше r). В соответствии с выбором r получится $k+1$ участок полной длины r . Польза от такого разбиения устанавливается следующей леммой.

Лемма 12.3.1. *Пусть P совпадает с подстрокой T' строки T с не более чем k различиями. Тогда T' должна содержать по меньшей мере один интервал длины r , который точно совпадает с одним из участков длины r разбиения P .*

Доказательство. В выравнивании P и T' каждый участок P пристраивается к некоторой части T' (рис. 12.7), что определяет $k+1$ подвыравниваний. Если бы каждое из них содержало не меньше одной ошибки (несовпадения или пробела), то в сумме получилось бы больше k различий и возникло бы противоречие. Поэтому один из первых $k+1$ участков P должен совпасть с интервалом T' без ошибок. \square

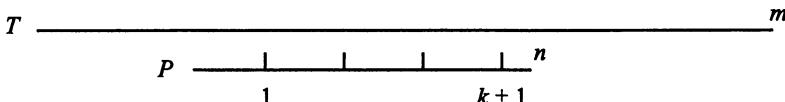


Рис. 12.7. Каждый из первых $k+1$ участков P имеет длину $r = [n/(k+1)]$

Отметим, что лемма верна и для задачи о k несовпадениях (т. е. когда вставки пробелов не допускаются). Лемма 12.3.1 приводит к следующему алгоритму приближенного сопоставления.

Алгоритм BYP

- Пусть \mathcal{P} — множество $k+1$ подстрок P , составленное из первых $k+1$ участков разбиения P .
- Построить дерево ключей (п. 3.4) для набора “образцов” \mathcal{P} .
- Используя алгоритм Ахо–Корасика (п. 3.4), найти \mathcal{I} — набор всех начальных позиций в T , куда какой-либо образец из \mathcal{P} входит точно.
- Для каждого индекса $i \in \mathcal{I}$ использовать алгоритм приближенного сопоставления (обычно основанный на динамическом программировании) в целях поиска конечных точек всех приближенных вхождений P в подстроку $T[i-n-k..i+n+k]$ (т. е. в интервал нужного размера вокруг i).

По лемме 12.3.1 легко установить, что этот алгоритм корректно находит все приближенные вхождения P в T . Главное здесь — “достаточная величина” интервала вокруг каждого i , чтобы его можно было выровнять с любым приближенным вхождением P , которое окружает i , и отсутствие приближенных вхождений P вне таких интервалов. Формальное доказательство оставляется в качестве упражнения. Теперь мы сосредоточимся на особенностях реализации и временному анализе.

Построение дерева ключей требует времени $O(n)$, а алгоритм Ахо–Корасика — в худшем случае $O(m)$ (п. 3.4). Таким образом, шаги б и с занимают время $O(n+m)$. Существует ряд альтернативных реализаций этих шагов. Например, построение суффиксного дерева для T и последующее его использование для нахождения всех

вхождений в T образцов из \mathcal{P} (см. п. 7.1). Однако такой подход требует много памяти. Эффективная по памяти реализация — построение обобщенного суффиксного дерева только для \mathcal{P} и последующее сопоставление T с ним (см. способ вычисления статистики совпадений в п. 7.8.1). Оба подхода занимают в худшем случае время $\Theta(n + m)$, но они не быстрее по ожидаемому времени, так как проверяют каждый символ из T . Практическое ускорение будет достигнуто, если использовать метод **множественного сопоставления** Бойера–Мура, основанный на суффиксных деревьях, который был развит в п. 7.16. Этот алгоритм пропускает части T и, следовательно, прорывает узкое место $\Theta(m)$. Другой вариант был предложен Ву и Манбер [482], которые реализовали шаги b и c , используя метод *Shift-And* (п. 4.2) для набора образцов. Еще один подход, найденный в работе Певзнера и Уотермена [373] и в других местах, использует для опознания длинных точно совпадающих подстрок P и T **хеширование**. Конечно, для нахождения длинных общих подстрок можно применять суффиксные деревья или развивать методы типа метода Карпа–Рабина. Хеширование и подходы, основанные на суффиксных деревьях, которые прямо ищут длинные общие подстроки в P и T , кажутся более робастными, чем BYP, так как они не используют разбиений строк. Но единственые установленные в [373] границы времени оказались такими же, как для BYP.

В фазе проверки, на шаге d , для каждого индекса из \mathcal{I} выполняется какой-либо алгоритм приближенного сопоставления P и интервала строки T длины $O(n)$. Действуя наивно, каждую из этих проверок можно выполнить за время $O(n^2)$ методом динамического программирования (глобального выравнивания). Даже в этом случае временная оценка будет адекватна получению ожидаемого полного времени счета $O(m)$ для диапазона изменения доли ошибок, который будет уточнен ниже. В качестве альтернативы можно использовать метод Ландау–Вишкина (п. 12.2), основанный на суффиксных деревьях, и тогда каждая проверка будет в худшем случае требовать времени только $O(kn)$. Если не разрешать в выравнивания P и T' пробелы (а допускать только совпадения и несовпадения), то может быть использован более простой подход, основанный на задаче о наибольшем общем продолжении (п. 9.1), требующий времени $O(kn)$. Если же внимание обращается на то, где точно в T находится совпадение, тогда на каждую проверку достаточно времени $O(n)$.

12.3.2. Анализ ожидаемого времени в алгоритме BYP

Так как шаги b и c считают в худшем случае за время $O(m)$, нам будет достаточно анализа шага d . Ключевой момент — оценка ожидаемого размера множества \mathcal{I} .

В последующем анализе мы предполагаем, что каждый символ из T выбирается равновероятно из алфавита размера σ . Однако строка P может быть произвольной. Рассмотрим любой образец $p \in \mathcal{P}$. Так как $|p| = r$ и T содержит по грубой оценке m подстрок длины r , ожидаемое количество точных вхождений p в T равно m/σ^r . Поэтому ожидаемое полное число вхождений в T образцов из \mathcal{P} (т. е. ожидаемая мощность \mathcal{I}) равняется $m(k+1)/\sigma^r$.

Для каждого $i \in \mathcal{I}$ алгоритм тратит время $O(n^2)$ (или меньше, если используются более быстрые методы) в фазе проверки. Поэтому ожидаемое время равно $m n^2 (k+1)/\sigma^r$. Наша цель — сделать ожидаемое время проверки линейным по m для достаточно скромного k , поэтому мы должны определить, при каком значении k

верна оценка

$$\frac{mn^2(k+1)}{\sigma^r} < cm$$

для некоторой константы c .

Чтобы упростить анализ, заменим k на $n - 1$ и разрешим относительно r уравнение

$$\frac{mn^3}{\sigma^r} = cm.$$

Имеем $\sigma^r = n^3/c$, так что $r = \log_\sigma n^3 - \log_\sigma c$. Но $r = [n/(k+1)]$, и в результате:

Теорема 12.3.1. Алгоритм BYP считает за время $O(m)$ при $k = O(n/\log n)$.

Иначе говоря, пока доля ошибок остается меньше чем одна на $\log_\sigma n$ символов, время счета алгоритма BYP будет линейно зависеть от m .

Узким местом в методе BYP является время $\Theta(m)$, требуемое на алгоритм Ахо–Корасика. Использование метода множественного сопоставления Бойера–Мура должно уменьшить это время в практическом использовании, но мы не можем представить расчет времени для этого случая. Однако метод Чанга–Лаулера имеет оценку ожидаемого времени, которая, вероятно, сублинейна для $k = O(n/\log n)$.

12.3.3. Метод Чанга–Лаулера

Для упрощения изложения мы ограничимся методом Чанга–Лаулера (CL) [94] для задачи о k несовпадениях, оставив обобщение на k различий как упражнение.

В CL разбивается строка T , а не P , на фиксированные последовательные участки длины $r = n/2$. Эти участки велики по сравнению с участками в BYP. Цель выбора длины $n/2$ — обеспечение того, что вне зависимости от расположения P и T (без включаемых пробелов) по крайней мере один из участков разбиения T будет полностью лежать в интервале, перекрытом P (рис. 12.8). Поэтому если P входит в T с не более чем k несовпадениями, то в T должен быть участок, перекрытый этим вхождением P и, конечно, этот участок совпадает с прилежащей частью P с не более чем k несовпадениями. Основываясь на этом наблюдении, фаза поиска метода CL проверяет каждый участок разбиения T , чтобы найти участки, которые не могут совпасть ни с какой подстрокой P с не более чем k несовпадениями. Такие участки исключаются, и затем интервал вокруг каждого из сохраненных участков проверяется методом приближенного совпадения, как в BYP. Фаза поиска в CL опирается на статистику совпадений, рассмотренную в п. 7.8.1.

Напомним, что значение статистики совпадений $ms(i)$ равно длине наибольшей подстроки, начинающейся в позиции j строки T , которая где-то (в неуточняемой

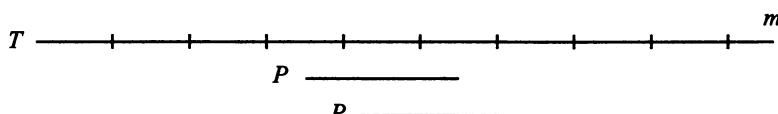


Рис. 12.8. Каждый полный участок T имеет длину $r = n/2$. Этим обеспечивается перекрытие образцом P одного полного участка независимо от положения P относительно T

позиции) совпадает с подстрокой P . Заметим также, что для любой строки S вся статистика несовпадений для позиций S может быть вычислена за полное время $O(|S|)$. Это верно, даже когда S является подстрокой большей строки T .

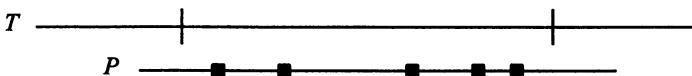


Рис. 12.9. Увеличенное изображение одного участка T , выровненного с одной копией P . Каждый черный квадрат показывает несовпадение символа P и его пары из T

Теперь пусть T' — подстрока одного из участков разбиения T , совпадающая с подстрокой P' из P с не более чем k несовпадениями (рис. 12.9). Выравнивание P' и T' можно разделить на не более чем $k + 1$ интервалов без несовпадений, чередующихся с интервалами, содержащими все несовпадения. Пусть i — начальная позиция любого из этих интервалов совпадения, а l — его длина. Ясно, что тогда $ms(i) \geq l$. Фаза поиска CL использует это замечание. Она выполняет следующий алгоритм для каждого участка R в разбиении T :

Поиск в участке R методом CL

Положить j равным начальной позиции j^* участка R в T .

$cn := 0$;

repeat

 вычислить $ms(j)$;

$j := j + ms(j) + 1$;

$cn := cn + 1$;

until $cn = k$ or $j - j^* > n/2$;

Если $j - j^* > n/2$, то сохранить участок R , в противном случае — исключить.

Если участок R сохраняется, то в фазе проверки CL выполняется алгоритм приближенного сопоставления для P и той окрестности T , которая начинается в $n/2$ позициях левее R и кончается в $n/2$ позициях правее. Эта окрестность размера $3n/2$, и поэтому каждая проверка может выполняться за время $O(kn)$.

Корректность метода CL видна из следующей леммы и того факта, что окрестности “достаточно велики”.

Лемма 12.3.2. *Когда поиск методом CL объявляет участок R исключенным, не существует таких вхождений P в T с не более чем k несовпадениями, которые бы полностью содержались в R .*

Доказательство простое и оставляется читателю, так же как и формальное доказательство корректности CL. Теперь рассмотрим анализ времени.

Поиск методом CL выполняется над $2m/n$ участками T . Для каждого участка R пусть j' — последнее значение j (т. е. значение j , когда cn достигает k или когда $j - j^*$ превышает $n/2$). Таким образом, в R статистики совпадений вычисляются для интервала длины $j' - j^* \leq n/2$. В алгоритме из п. 7.8.1 время, используемое для расчета статистик совпадений, равно $O(j' - j^*)$. Далее, ожидаемое значение $j' - j^*$ не превосходит умноженного на k ожидаемого значения $ms(i)$ для любого i . Пусть $E(M)$ обозначает ожидаемое значение статистики совпадений,

$a e$ — ожидаемое число участков, сохраняемых в фазе поиска. Тогда ожидаемое время фазы поиска равно $O(2mkE(M)/n)$, а ожидаемое время фазы проверки равно $O(kne)$.

В следующем анализе мы предполагаем, что P — случайная строка, в которой символы выбираются равномерно из алфавита размером σ .

Лемма 12.3.3. *$E(M)$, ожидаемое значение статистики совпадений, равно $O(\log_\sigma n)$.*

Доказательство. Для фиксированной длины d в P имеется, грубо говоря, n подстрок длины d , а всего можно составить σ^d различных подстрок длины d . Таким образом, для любой конкретной строки α длины d вероятность того, что α найдется где-нибудь в P , не превосходит n/σ^d . Это верно для любого d , но неинформативно при $\sigma^d = n$ (т. е. когда $d = \log_\sigma n$).

Пусть X — случайная величина, принимающая значение $\log_\sigma n$ при $ms(i) \leq \log_\sigma n$, а в противном случае — значение $ms(i)$. Тогда

$$E(M) < E(X) < \log_\sigma n + \sum_{l=\log_\sigma n}^{\infty} \frac{l}{\sigma^l} = \log_\sigma n + 2. \quad \square$$

Следствие 12.3.1. *Ожидаемое время, расходуемое методом CL в фазе поиска, равно $O(2mk \log_\sigma n/n)$, эта величина сублинейна по m для $k < n/\log_\sigma n$.*

Анализ для e , ожидаемого числа сохраняемых участков, слишком труден, чтобы излагать его здесь. В [94] показано, что, когда $k = O(n/\log_\sigma n)$, мы имеем $e = m/n^4$, так что ожидаемое время, расходуемое методом CL на фазу проверки, равно $O(km/n^3) = o(m)$. Фаза поиска этого метода настолько эффективна в исключении участков T , что ожидаемое время счета фазы проверки очень мало.

12.3.4. Множественная фильтрация для k несовпадений

Оба метода, и BYP и CL, для исключения интервалов T в фазе поиска используют совсем простые комбинаторные критерии. Можно придумать более точные условия, необходимые для того, чтобы интервал из T содержался в приближенном вхождении в P . В контексте задачи о k несовпадениях условия этого типа (называемые условиями фильтрации) развивались и изучались Певзнером и Уотерменом [373]. Эти условия используются вместе с хешированием подстрок в целях получения другого метода с линейным ожидаемым временем для задачи о k несовпадениях. Эмпирические результаты в [373] показывают меньшее время счета, чем у других методов.

12.3.5. Метод Майерса с сублинейным временем

Юджин Майерс [342, 337] развел метод исключений, более хитроумный, чем те, которые мы рассматривали до сих пор. Время работы этого метода сублинейно для широкого диапазона доли ошибок. Он справляется с приближенным совпадением и для случая включений и удалений, и для случая несовпадений. Полный алгоритм и его анализ слишком сложны для подробного рассмотрения в этой книге, но мы можем продемонстрировать некоторые его идеи, чтобы стали яснее недостатки других методов исключения.

У методов BYP и CL (и других методов исключения, которые мы упоминали) есть две основные проблемы. Во-первых, используемые ими критерии допускают большое ожидаемое число сохраняемых участков в сравнении с ожидаемым числом истинных приближенных совпадений. Поэтому не все исходные сохраняемые участки в действительности содержат приближенные совпадения, и отношение ожидаемого числа участков к ожидаемым совпадениям довольно велико (для случайных образцов и текста). При этом чем выше разрешенная доля ошибок, тем серьезнее становится эта проблема. Во-вторых, когда сохраняемый участок впервые обнаруживается, методы переходят прямо к полным вычислениям ДП (или другим относительно дорогостоящим операциям), чтобы проверить большой интервал вокруг сохраняемого участка на приближенные совпадения. Следовательно, эти методы требуют выполнения большого объема вычислений для большого числа интервалов, не содержащих приближенных совпадений.

По сравнению с другими методами исключения метод Майерса содержит две новые идеи, делающие его более селективным (отбирающим меньше сохраняемых участков) и менее дорогим в проверке оставшихся. Алгоритм Майерса начинает свою работу в манере, похожей на другие методы исключения. Он разбивает P на короткие подстроки (их длина будет уточняться ниже) и затем находит в T все места, куда эти подстроки входят с малым числом различий. Детали поиска совершенно отличны от других методов, но намерения (исключить большую часть T из дальнейшего рассмотрения) те же самые. Каждое из этих выравниваний подстроки P , которое находится (приближенно) в T , называется *сохраняемым совпадением*. Оно, грубо говоря, играет роль *сохраняемого участка* в других методах исключения, но задает пару подстрок (одну из P и одну из T), а не одну, как в сохраняемом участке. Другой способ представить себе сохраняемое совпадение — считать его, грубо говоря, диагональным путем в графике выравнивания для P и T .

Отыскав начальные сохраняемые совпадения (или сохраняемые участки), все другие упоминавшиеся методы исключения должны дальше проверять полные интервалы длины грубым счетом в $2n$ вокруг каждого сохраняемого участка в T , чтобы посмотреть, не содержат ли они приближенных совпадений с P . Напротив, метод Майерса будет *последовательно увеличивать* и проверять растущий интервал вокруг каждого исходного сохраненного совпадения, чтобы создать более длинное сохраненное совпадение или исключить его из дальнейшего рассмотрения. Это делается примерно за $O(\log n)$ итераций. (Напомним, что $|P| = n$ и $|T| = m$.)

Определение. Для данной доли ошибок ε строка S называется ε -совпадающей с подстрокой T , если S совпадает с подстрокой, используя не более $\varepsilon|S|$ вставок, удалений и несовпадений.

Например, пусть $S = aba$ и $\varepsilon = 2/3$. Тогда ac ε -совпадает с S , используя одно несовпадение и одну операцию удаления.

На первой итерации образец P делится на последовательные, неперекрывающиеся подобразцы длины $\log_\sigma m$ (мы предполагаем это число целым), и алгоритм находит все подстроки T , которые ε -совпадают с одним из этих коротких подобразцов (что подробнее обсуждается ниже). Длина этих подобразцов достаточно коротка, так что все ε -совпадения могут быть найдены за сублинейное ожидаемое время для

широкого диапазона значений ε . Эти ε -совпадения образуют исходные сохраняемые совпадения.

Далее алгоритм пытается расширить каждое исходное сохраняемое совпадение, чтобы превратить его в ε -совпадение между подстроками (в P и T), которые примерно вдвое длиннее, чем в текущем сохраняемом совпадении. Это делается с помощью динамического программирования в подходящем интервале вокруг сохраняемого совпадения. В каждой последующей итерации метод применяет более селективный и дорогой фильтр, пытаясь удвоить длину ε -совпадения вокруг каждого сохраняемого совпадения. Так как представляющие интерес интервалы удваивают свою длину, то используемое время, в расчете на интервал, растет в каждой последовательной итерации четырехкратно. Однако ожидается, что в каждой последовательной итерации число сохраняемых совпадений гиперэкспоненциально убывает, компенсируя увеличение времени вычислений в каждом интервале.

С этим итеративным расширением усилия, затрачиваемые на проверку любого начального сохраняемого совпадения, скучо расходуются в течение $O(\log(n/\log m))$ итераций и не продолжаются для сохраняемого совпадения после итерации, где оно было исключено. Опишем подробнее, как находятся исходные сохраняемые совпадения и как они последовательно расширяются от одной итерации к другой.

Первая итерация

Определение. Для строки S и значения ε пусть $d = \varepsilon|S|$. Назовем d -окрестностью S множество всех строк, ε -совпадающих с S .

Например, над двухбуквенным алфавитом $\{a, b\}$ если $S = aba$ и $d = 1$, то 1 -окрестность S — это $\{bba, aaa, abb, aab, abaa, baba, abba, abab, ba, aa, ab\}$. Она создана из S операциями несовпадения, вставки и удаления. Сжатая d -окрестность S получается из d -окрестности S удалением любой подстроки, являющейся префиксом другой строки из d -окрестности. Сжатая 1 -окрестность S равна $\{bba, aaa, aab, abaa, baba, abba, abab\}$.

Напомним, что образец P исходно разбит на подобразцы длины $\log_\sigma m$ (предполагается целым). Пусть \mathcal{P} — множество этих подобразцов. На первой итерации алгоритм (концептуально) строит сжатую d -окрестность для каждого подобразца в \mathcal{P} , а затем находит все положения подстрок в тексте T , которые точно совпадают с одной из подстрок в одной из сжатых d -окрестностей. Этим способом метод находит все подстроки T , которые ε -совпадают с одним из подобразцов в \mathcal{P} . Такие ε -совпадения образуют начальные сохраняемые совпадения.

На самом деле генерирование подстрок в сжатых d -окрестностях и поиск их точных вхождений в T переплетены и требуют препроцессинга текста T и создания некоей индексной структуры, которая может быть суффиксным деревом, суффиксным массивом или хеш-таблицей, содержащей короткие подстроки T . Детали можно найти в [342].

Майерс [342] показал, что когда длина подобразцов равна $O(\log_\sigma m)$, то первую итерацию можно реализовать так, чтобы она выполнялась с ожидаемым временем $O(km^{\rho(\varepsilon)} \log m)$. Функция $\rho(\varepsilon)$ сложна, но она вогнута и возрастает, причем медленнее, чем растет размер алфавита. Она имеет значение меньшее 1 для ДНК при $\varepsilon \leq 1/3$, а для белков при $\varepsilon \leq 0.56$.

Последующие итерации

Чтобы объяснить основную идею, положим $\alpha = \alpha_0 \alpha_1$, где $|\alpha_0| = |\alpha_1|$.

Лемма 12.3.4. *Предположим, что α ε -совпадает с β . Тогда β можно так разделить на две подстроки β_0 и β_1 , что $\beta = \beta_0 \beta_1$, и либо α_0 ε -совпадает с β_0 , либо α_1 ε -совпадает с β_1 .*

Эта лемма (применяемая в обращенном виде) является ключевой для определения способа расширения интервалов вокруг сохраняемых совпадений на каждой итерации. Для простоты предположим, что n и $\log_\sigma m$ являются степенями числа 2. Пусть B — двоичное дерево, представляющее последовательные деления P на две части равного размера, пока каждая часть не станет длиной $\log_\sigma m$ (рис. 12.10). Подстроки, записанные в листьях дерева, являются подобразцами, используемыми на первой итерации алгоритма Майерса. Итерация i проверяет подстроки P , которые помечают (некоторые) вершины B на уровне i над листьями (считая, что уровень листьев — 1).

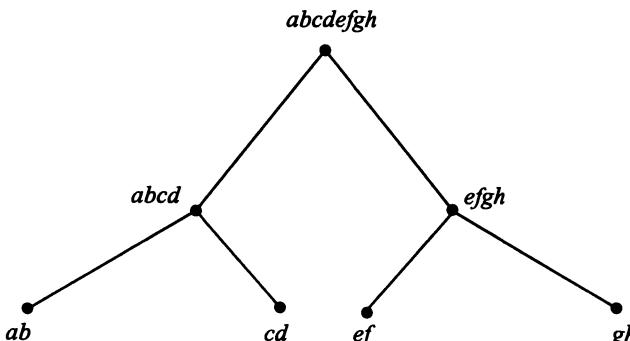


Рис. 12.10. Двоичное дерево B , определяющее последовательные деления P и его разбиение на участки длины $\log_\sigma m$ (эта длина на рисунке равна 2)

Предположим, что на итерации $i - 1$ подстроки P' и T' в запросе и тексте, соответственно, образуют сохраняемое совпадение (т. е. обнаружено, что они выровнены в ε -совпадение). Пусть P'' — предок P' в дереве B . Если P' — левый ребенок P'' , то на итерации i алгоритм пытается ε -совместить P'' с подстрокой T в интервале, который расширяет T' вправо. Напротив, если P' является правым ребенком P'' , то алгоритм пытается ε -совместить P'' с подстрокой в интервале, расширяющем T' влево. По лемме 12.3.4 если ε -совмещение P' с T' является частью ε -совмещения P с подстрокой T , то P'' будет ε -совмещено с соответствующей подстрокой T . Более того, интервал T , который должен сравниваться с P'' , всего вдвое больше интервала для T' . В результате, как это показано в [342], все проверки и, следовательно, весь алгоритм расходуют ожидаемое время $O(km^{\rho(\varepsilon)} \log m)$.

Заключительные замечания к методу Майерса

Хочется отметить сразу несколько моментов. Во-первых, наше изложение — это только набросок метода Майерса, без какого-либо анализа. Все подробности алгоритма и его анализа можно найти в [342]; в [337] есть только обзор этого метода

в сравнении с другими методами исключения. Во-вторых, в отличие от методов BYP и CL, доля ошибок, которая обеспечивает сублинейное (или линейное) время счета, не зависит от длины P . В методах BYP и CL разрешаемая доля ошибок убывает с ростом длины P . В методе Майерса эта доля зависит только от размера алфавита. В-третьих, хотя ожидаемое время счета и для CL, и для метода Майерса сублинейно (для соответствующего диапазона доли ошибок), имеется существенное отличие в природе этих сублинейностей. В методе CL сублинейность определяется множителем, который меньше единицы. А в методе Майерса сублинейность определяется экспонентой, которая меньше единицы. Так что граница в CL как функция от t растет линейно (хотя для любого фиксированного значения t ожидаемое время счета меньше чем t), а граница для метода Майерса растет сублинейно по t . Это отличие очень важно, так как многие базы данных быстро увеличиваются в размере.

Однако метод Майерса предполагает, что текст T уже предварительно организован в некоторую индексированную структуру и время на препроцессинг (хотя и линейное по t) во временну́ю оценку не включалось. Напротив, время счета методов BYP и CL включает всю работу, необходимую для этих методов. Наконец, Майерс показал, что в экспериментах над данными осмысленного размера из молекулярной биологии (образцы длины 80 при текстах длины 3 миллиона), алгоритмы для задачи о k различиях из пп. 12.2.4 и 12.2.3 считали раз в 100–500 медленнее, чем его метод, сублинейный по ожидаемому времени.

12.3.6. Заключительное замечание о методах исключения

Методы исключения, быстрые по ожидаемому времени, были развиты с ориентацией на поиск приближенных вхождений запрашиваемых строк в больших базах данных по ДНК и белкам. Но доказанные результаты немножко слабее в случае поиска в белковых базах данных, так как при сравнении белковых последовательностей большой интерес представляют доли ошибок, доходящие до 85% (так называемая сумеречная зона) [127, 360]. В сумеречной зоне еще сохраняются свидетельства в пользу существования общего предка, но требуется некоторое искусство для оценки осмысленности каждого конкретного совпадения. Другая проблема, связанная с представленными здесь методами исключения, заключается в том, что не все из них хорошо переносятся на случаи взвешенного и локального выравнивания.

Тем не менее это многообещающие результаты, и всячески приветствуется прогресс в решении проблемы построения алгоритмов с сублинейным ожидаемым временем для случая высокой доли ошибок. Более того, мы увидим в главе 15, посвященной поиску в базах данных, что эффективные практические методы поиска в базах, применяемые в настоящее время (BLAST, FASTA и их варианты), могут по большей части рассматриваться как методы исключения и основаны на идеях, родственных некоторым из представленных здесь более формальных методов.

12.4. Еще о суффиксных деревьях и гибридном динамическом программировании

Хотя суффиксное дерево было первоначально разработано и использовано для работы со сложными задачами точного сопоставления, его можно с большой пользой применять в различных задачах неточного сопоставления. Это уже демонстрировалось в пп. 9.4 и 12.2, где обсуждались задачи о k несовпадениях и k различиях. Суффиксное дерево во втором приложении использовалось в сочетании с динамическим программированием, именно в методе *гибридного динамического программирования*, который оказался быстрее обычного ДП. Недостаток этого подхода в том, что у него нет удачных обобщений на задачи *взвешенного выравнивания*. Мы предложим другой способ комбинирования суффиксных деревьев с ДП для задач взвешенного выравнивания. Эти замыслы декларировались как очень эффективные на практике, особенно для больших вычислительных проектов. Однако они не всегда пригодны для сильно улучшенных доказуемых временных границ наихудшего случая. Обсуждаемые представления следуют, причем не буквально, опубликованной работе Укконена [437] и неопубликованной заметке Гоннета и Беза-Йетса [34]. Диссертация Биганского [63] предлагает родственную идею использования суффиксных деревьев при сопоставлении (с ошибками) с образцами, заданными регулярными выражениями, и ее крупномасштабное применение в управлении геномными базами данных. Метод Гоннета и Беза-Йетса был реализован и широко используется в крупномасштабных сравнениях белков [57, 183].

Две задачи

Предположим, что существует матрица оценок для вычисления значений выравниваний, и следовательно, под “редакционным расстоянием” здесь понимается взвешенное редакционное расстояние. Мы обсудим две задачи в основном тексте и еще две родственные задачи введем в упражнениях.

1. **Задача P -против-всех.** Пусть заданы строки P и T . Вычислить редакционные расстояния между P и каждой подстрокой T' строки T .
2. **Пороговая задача все-против-всех.** Заданы строки P и T и порог d . Найти все пары подстрок P' из P и T' из T , для которых редакционное расстояние между P' и T' меньше d .

Пороговая задача все-против-всех похожа на упомянутые в п. 12.2.1 задачи о построении неизбыточной базы данных для последовательностей. Однако пороговая задача все-против-всех труднее, так как в ней нужны выравнивания всех пар *подстрок*, а не только всех пар строк. Это критическое различие было источником некоторой путаницы в литературе [50, 56].

12.4.1. Задача P -против-всех

Задача P -против-всех является примером *крупномасштабной задачи выравнивания*, в которой нужно получить большое количество информации относительно выравниваний. Если не работать аккуратно, ее решение вызовет большое количество избыточных вычислений.

Пусть P имеет длину n , а T — длину $m > n$. Самое наивное решение задачи P -против-всех — это перенумеровать все C_m^2 подстрок T и затем отдельно вычислить редакционное расстояние между P и каждой подстрокой T . Это занимает полное время $\Theta(nm^3)$. Секундное раздумье уже дает улучшение. Вместо выбора всех подстрок T нам достаточно взять каждый *суффикс* S строки T и вычислить с помощью ДП таблицу редакционных расстояний для строк P и S . Если S начинается в позиции i строки T , то последняя строчка этой таблицы дает редакционные расстояния между P и всеми подстроками T , начинающимися в позиции i . Значит, редакционное расстояние между P и $T[i..j]$ находится в клетке $(n, j - i + 1)$. Такой подход требует времени $\Theta(nm^2)$.

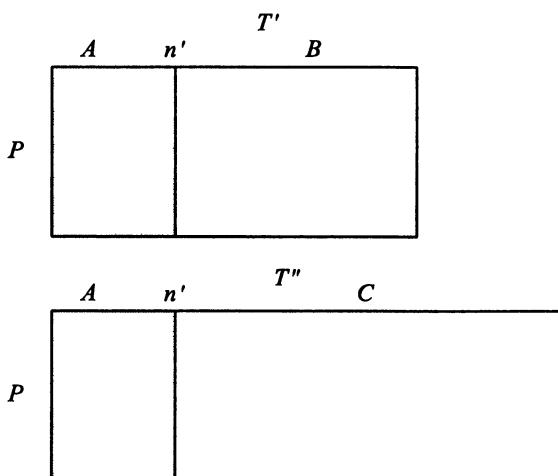


Рис. 12.11. Изображение таблиц ДП для вычисления редакционного расстояния между P и подстрокой T' (вверху) и между P и подстрокой T'' (внизу). Эти две таблицы имеют общую подтаблицу для P и подстроки A (серый прямоугольник). Общую подтаблицу достаточно вычислить один раз

Нам интересна задача P -против-всех, когда строка T очень длинна. В этом случае введение суффиксного дерева может сильно ускорить вычисления динамического программирования — в зависимости от повторяемости в строке T^*) (см. также п. 7.11.1). Чтобы уяснить основную идею метода, рассмотрим две подстроки T' и T'' строки T , у которых первые n' символов совпадают. В изложенном подходе динамического программирования, редакционное расстояние между P и T' и между P и T'' вычисляется отдельно. Но если мы вычисляем редакционное расстояние по столбцам (вместо обычного вычисления по строкам), то можем комбинировать два вычисления редакционного расстояния для первых n' столбцов, так как первые n' символов T' и T'' одинаковы (рис. 12.11). Вычисление подтаблицы $n \times n'$ отдельно для этих двух расстояний избыточно. Эту идею использования общей части T' и T'' можно formalизовать и реализовать при помощи суффиксного дерева для строки T .

* Недавние оценки повторов в человеческих ДНК дают от 50 до 60 %. Значит, от 50 до 60 % всех человеческих ДНК содержатся в структурированных подстроках *нетривиальной длины*, которые, повторяясь, проявляются во всем геноме. Сходные уровни избыточности проявляются во многих других организмах.

Рассмотрим суффиксное дерево \mathcal{T} для строки T и вспомним, что любой путь из корня дерева определяет некоторую подстроку S строки T . Пойдем по пути из корня \mathcal{T} . Пусть S обозначает растущую подстроку, соответствующую этому пути. Проходя по пути, мы можем строить (по столбцам) таблицу динамического программирования для редакционного расстояния между P и этой растущей подстрокой S . Полностью идея выглядит так: обходить дерево \mathcal{T} в глубину, вычисляя соответствующий столбец таблицы ДП (по его левому соседу) для каждой подстроки S , определенной текущим путем. Когда обход достигает вершины v , в ней сохраняется последний вычисленный столбец и последняя подстрочка текущей подтаблицы (последняя подстрочка будет всегда строчкой n). Таким образом, если S — подстрока, определенная путем до вершины v , то в v будет сохранена последняя строчка и столбец таблицы ДП для редакционного расстояния между P и S . Когда обход в глубину посещает потомка v' вершины v , к этой таблице добавляются столбцы (по одному на каждый символ на дуге (v, v')), согласно расширению подстроки S . Когда обход достигает листа, соответствующего суффиксу, который начинается (допустим) в позиции i строки T , можно вывести значения последней строчки текущей таблицы. Эти значения определяют редакционные расстояния между P и всеми подстроками, начинаящимися в позиции i строки T . Когда обход возвращается обратно в вершину v и у нее есть еще непосещенный потомок v' , строчка и столбец, сохраненные в v , прочитываются и расширяются в соответствии с продвижением по новой дуге (v, v') (рис. 12.12).

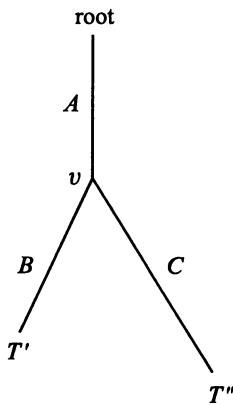


Рис. 12.12. Фрагмент суффиксного дерева для T . Проход от корня до вершины v сопровождается вычислением подтаблицы A (см. рис. 12.11). В этот момент последние строчка и столбец подтаблицы A сохраняются в вершине v . Вычисление подтаблицы B соответствует переходу из v к листу, представляющему подстроку T' . После того как обход достигает листа для T' , он возвращается в вершину v , восстанавливает записанные в ней строчку и столбец и использует их для вычисления подтаблицы C , необходимой для вычисления редакционного расстояния между P и T''

Понятно, что основанный на суффиксном дереве подход, действительно корректно вычисляет редакционное расстояние между P и каждой подстрокой T и использует повторы подстрок (больших и малых), которые могут встретиться в T . Но насколько он эффективен по сравнению с обычным подходом ДП, требующим времени $\Theta(nm^2)$?

Определение. Строковой длиной дуги в суффиксном дереве назовем длину строки, помечающей эту дугу (даже если метка компактно представлена постоянным числом символов). Длиной суффиксного дерева λ назовем сумму строковых длин всех дуг дерева.

Длина суффиксного дерева \mathcal{T} для строки T длины m может лежать между $\Theta(m)$ и $\Theta(m^2)$ в зависимости от количества повторений в T . В вычислительных экспериментах с длинными подстроками ДНК млекопитающих (длиной около миллиона) строковые длины получавшихся суффиксных деревьев были около $m^2/10$. Теперь же количество столбцов в таблице динамического программирования, которые генерируются во время обхода \mathcal{T} , равно в точности длине \mathcal{T} . Вычисление каждого столбца требует времени $\Theta(n)$, и мы получаем следующее утверждение.

Лемма 12.4.1. Время, необходимое для вычисления всех столбцов при обходе дерева, равно $\Theta(n \times \lambda(\mathcal{T}))$.

Мы должны подсчитать еще время и память, используемые для записи строчек и столбцов, сохраняемых в каждой вершине \mathcal{T} . В суффиксном дереве с m листьями есть $\Theta(m)$ внутренних вершин, а одна строчка и один столбец требуют для записи не более $O(m+n)$ времени и места. Поэтому время и место, необходимые для записи строчек и столбцов, равны $\Theta(m^2 + nm) = \Theta(m^2)$. Следовательно:

Теорема 12.4.1. Полное время подхода, основанного на суффиксных деревьях, равно $\Theta(n \times \lambda(\mathcal{T}) + m^2)$, а размер максимальной используемой памяти равен $\Theta(m^2)$.

Сокращение памяти

Размер требуемого вывода равен $\Theta(m^2)$, так как задача находит редакционное расстояние между P и каждой из $\Theta(m^2)$ подстрок T , так что член $\Theta(m^2)$ в оценке времени приемлем. С другой стороны, расход памяти кажется чрезмерным, поскольку память, запрашиваемая динамическим программированием без использования суффиксного дерева, составляет лишь $\Theta(pm)$ и может быть уменьшена до $O(m)$. Сейчас мы модифицируем подход с суффиксными деревьями, чтобы ограничиться памятью $O(n+m)$ при тех же временных границах, что и раньше.

Прежде всего, нет необходимости хранить текущий столбец в каждой вершине v . При возврате назад от потомка v' к v мы можем использовать текущий столбец в v' и строку, помечающую дугу (v, v') , чтобы вычислить заново столбец для вершины v . Однако это удвоит общее время на вычисление столбцов. Нет необходимости также хранить в каждой вершине v текущую n -ю строчку. Вместо этого на всю работу с этой строчкой требуется только память $O(m)$. Основная идея в том, что текущая таблица расширяется по столбцам, так что если строковая глубина v равна j , а строковая глубина v' равна $j+d$, то n -е строчки, хранящиеся в v и в v' , будут совпадать в первых j элементах. Оставим разработку подробностей на упражнения. В результате:

Теорема 12.4.2. Гибридный подход суффиксного дерева/динамического программирования к задаче P -против-всех может быть реализован так, чтобы он использовал время $\Theta(n\lambda(\mathcal{T}) + m^2)$ и память $O(n+m)$.

Приводимые оценки времени и памяти следует сравнить с оценками времени $\Theta(pm^2)$ и памяти $O(n+m)$ для непосредственного использования динамического программирования. Практическая эффективность этого метода зависит от длины \mathcal{T}

для реальных строк. Известно, что для случайных строк $\lambda(\mathcal{T})$ имеет порядок $\Theta(m^2)$, что делает метод непривлекательным. (Для случайных строк суффиксное дерево кустисто для строковых глубин $\log_\sigma m$ или меньше, где σ — размер алфавита. Но ниже этой глубины суффиксное дерево становится редким, так как вероятность повторного появления в строке подстроки длиннее $\log_\sigma m$ очень низка.) Однако строки с более структурными повторами (как в ДНК) должны приводить к суффиксным деревьям с длинами, достаточно малыми, чтобы сделать такой метод полезным. Мы проверили этот вопрос эмпирически для строк ДНК до миллиона символов, и длины получившихся суффиксных деревьев были порядка $m^2/10$.

12.4.2. Задача все-против-всех (с порогом)

Рассмотрим теперь более амбициозную задачу. Пусть заданы строки P и T . Найти все пары подстрок, у которых редакционное расстояние ниже фиксированного порога d . Вычисления этого типа проводились для P и T , состоящих из комбинаций белковых строк в базе данных Swiss-Prot [183]. Важность вычислений такого рода в больших объемах и применение результатов обсуждаются в [57], а способ, с помощью которого для ускорения вычислений можно использовать суффиксные деревья, — в [34].

Так как P и T имеют, соответственно, длины n и m , полная задача все-против-всех (с порогом ∞) требует вычисления n^2m^2 результатов. Следовательно, никакой метод ее решения не может считать быстрее, чем за время $\Theta(n^2m^2)$. Более того, эта временная граница легко достигается. Выберем пару начальных позиций в P и T (это nm возможных способов) и для каждого выбора i, j заполним таблицу динамического программирования для редакционного расстояния $P[i..n]$ и $T[j..m]$ (за время $O(nm)$). Для любого выбора i и j элементы соответствующей таблицы дадут редакционные расстояния для каждой пары подстрок, начинающихся с позиции i в P и с позиции j в T . Таким образом, достижение границы $O(n^2m^2)$ для полной задачи все-против-всех суффиксных деревьев не требует.

Но полная задача все-против-всех предполагает результат такого объема, который оказывается чрезмерен, и уменьшить его можно назначением осмысленного порога. В качестве варианта можно сделать критерий отбора подходящих пар функцией от длины и редакционного расстояния. Но каков бы ни был критерий отбора, если при нем не требуется выводить редакционное расстояние для каждой пары, то нет и убежденности в том, что потребуется время $\Theta(n^2m^2)$. Здесь мы предложим метод, у которого время счета в наихудшем случае выражается как $O(C + R)$, где C — время вычислений, которое может быть меньше, чем $O(n^2m^2)$, а R — размер результата (т.е. число выводимых пар подстрок). При таком подходе использование суффиксных деревьев может быть очень существенным. Эффект зависит от размера результата и количества повторов в обеих строках.

Метод со временем $O(C + R)$

Этот метод использует суффиксные деревья \mathcal{T}_P для строки P и \mathcal{T}_T для строки T . Оценка времени наихудшего случая будет записываться в виде $O(C + R)$, где $C = \lambda(\mathcal{T}_P) \times \lambda(\mathcal{T}_T)$ независимо от критерия отбора результата, и R — размер результата (определение длины суффиксного дерева λ дано в п. 12.4.1). Значит, метод будет вычислять значения некоторой таблицы динамического программирования, которые не зависят от критерия отбора, и затем, если значение в клетке таблицы

удовлетворяет данному критерию, алгоритм будет собирать приемлемые подстроки, связанные с этой клеткой. Следовательно, наше описание метода верно и для полной задачи все-против-всех, и для пороговой версии, и для любой другой версии с другим критерием отбора.

Для начала напомним, что каждая вершина в \mathcal{T}_P представляет подстроку P и что каждая под строка P является префиксом под строки, представленной вершиной \mathcal{T}_P . В частности, каждый суффикс P представлен листом \mathcal{T}_P . То же верно и для T и \mathcal{T}_T .

Определение. Таблица динамического программирования для пары вершин (u, v) из \mathcal{T}_P и \mathcal{T}_T соответственно, определяется как таблица ДП для редакционных расстояний между строками, представляющими вершины u и v .

Пороговую задачу все-против-всех можно было бы решить (если игнорировать затраты времени), вычисляя таблицу ДП для каждой пары листьев, по одной из каждого дерева, и затем проверяя каждую клетку этих таблиц. Следовательно, задача определенно может быть решена вычислением таблицы ДП для каждой пары вершин и последующей проверкой каждой клетки этих таблиц. В основном это мы и будем делать, но выберем способ, при котором можно избежать избыточных вычислений и проверок. Следующая лемма дает основную идею.

Лемма 12.4.2. *Пусть u' — предок вершины u в \mathcal{T}_P и α — строка, помечающая дугу между ними. Аналогично, пусть v' — предок v в \mathcal{T}_T и β — строка, помечающая дугу между ними. Тогда все элементы таблицы ДП для пары (u, v) , кроме нижней правой части, состоящей из $|\alpha| \times |\beta|$ клеток, имеются в одной из таблиц для (u', v') , (u', v) или (u, v') . Более того, эта нижняя правая часть таблицы для (u, v) может быть получена из других трех таблиц за время $O(|\alpha| \times |\beta|)$ (рис. 12.13).*

Доказательство этой леммы прямо следует из определений и рекуррентных соотношений для редакционного расстояния.

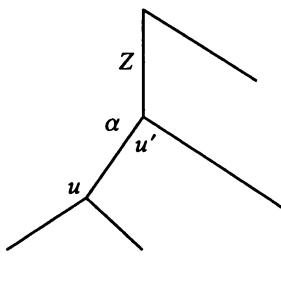
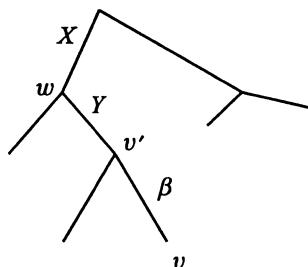
Вычисление новой части таблицы (u, v) создает прямоугольную подтаблицу $|\alpha| \times |\beta|$, которая формирует нижнюю правую секцию таблицы (u, v) . В алгоритме, который будет предложен ниже, мы храним и сопоставляем каждой паре вершин (u, v) последний столбец и последнюю строку этой таблицы $|\alpha| \times |\beta|$. Теперь можно описать алгоритм полностью.

Детали алгоритма

Прежде всего, перенумеруем некорневые вершины \mathcal{T}_P по возрастанию строковой глубины.* Отдельно перенумеруем по тому же принципу вершины \mathcal{T}_T . Затем сформируем список L пар номеров вершин, по одной из каждого дерева, в лексикографическом порядке. Следовательно, пара (u, v) появится в нем до пары (p, q) в том и только том случае, если $u < p$ или $u = p$ и $v < q$ (рис. 12.14). Отсюда следует, что если u' является родителем u в \mathcal{T}_P и v' — родителем v в \mathcal{T}_T , то (u', v') появится в списке до (u, v) .

Далее, обработаем все пары (u, v) в порядке их расположения в списке L . Снова предположим, что u' — родитель u , v' — родитель v и метки соответствующих дуг — α и β . Для обработки пары вершин (u, v) возьмем значение в клетке из

* На самом деле годится любая топологическая нумерация, но строковая глубина даст нам некоторые дополнительные выгоды после того, как будут добавлены эвристические приемы ускорения.

Суффиксное дерево для P Суффиксное дерево для T

X	w	Y	v'	β	v
Z					
u'					
α					
u					

Новая часть таблицы (u, v)

Рис. 12.13. Таблица ДП для (u, v) изображена под суффиксными деревьями для P и T . Строки на пути к вершине u — это $Z\alpha$, а к вершине v — $XY\beta$. Каждая клетка в таблице (u, v) , кроме нижнего правого прямоугольника, содержитя также в таблицах для (u, v') , (u', v) или (u', v') . Новая часть таблицы (u, v) может быть вычислена по заштрихованным клеткам и подстрокам α и β . Заштрихованные клетки содержат строго одну клетку из таблицы (u', v') , $|\alpha|$ клеток из последнего столбца таблицы (u, v') и $|\beta|$ клеток из последней строчки таблицы (u', v)

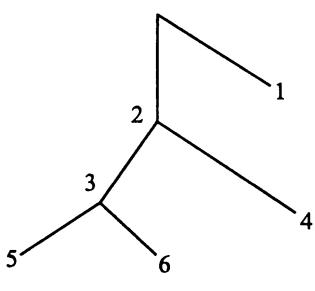
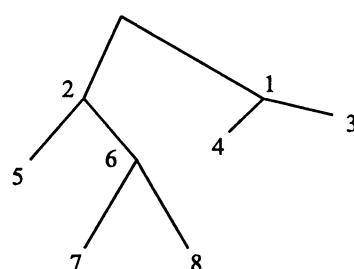
Суффиксное дерево для P Суффиксное дерево для T

Рис. 12.14. Суффиксные деревья для P и T с вершинами, занумерованными по строковой глубине. Отметим, что эти числа не совпадают со стандартными номерами суффиксных позиций, используемыми для нумерации листьев. Упорядоченный список пар вершин начинается с $(1, 1)$, $(1, 2)$, $(1, 3)\dots$ и кончается парой $(6, 8)$

нижнего правого угла сохраняемой части таблицы (u', v') , столбец, сохраняемый с парой (u, v') , и строчку, сохраняемую с парой (u', v) . Три указанные пары уже обработаны благодаря лексикографическому упорядочению списка. По этим значениям и подстрокам α и β вычисляется новая подтаблица размера $|\alpha| \times |\beta|$, завершающая формирование таблицы (u, v) . Последние строка и столбец вновь вычисленной подтаблицы сохраняются с парой (u, v) .

Теперь предположим, что клетка (i, j) находится в новой подтаблице $|\alpha| \times |\beta|$ и ее значение удовлетворяет критерию отбора. Алгоритм должен найти и вывести все расположения двух подстрок, указанных парой (i, j) . Как обычно, обход в глубину к листьям ниже u и v отыщет все начальные позиции этих строк. Их длина определяется по i и j . Следовательно, когда потребуется вывести пары подстрок, которые удовлетворяют критерию отбора, время отбора пар будет просто пропорционально их числу.

Корректность и временной анализ

Корректность метода следует из того факта, что на самом общем уровне описания метод вычисляет редакционные расстояния для всех пар подстрок, по одной из каждой строки. При этом генерируется и проверяется каждая клетка таблицы динамического программирования для всех пар подстрок (хотя избегает излишних вычислений). Единственный деликатный момент заключается в том, что метод генерирует и проверяет клетки в каждой таблице последовательным наращиванием, чтобы воспользоваться совпадениями подстрок, и следовательно, он избегает повторного счета клеток, которые входят больше чем в одну таблицу. Далее, когда метод обнаруживает клетку, удовлетворяющую критерию отбора (это функция значения и длины), он может найти все пары подстрок, определяемые этой клеткой, используя переход к подмножеству листьев в двух суффиксных деревьях. Формальное доказательство корректности оставляется читателю в качестве упражнения.

Для проведения временного анализа напомним, что длина \mathcal{T}_P равна сумме длин всех дуговых меток в \mathcal{T}_P . Если P имеет длину n , то $\lambda(\mathcal{T}_P)$ лежит между n и $n^2/2$, в зависимости от того, насколько велики повторы в P . Аналогично, $\lambda(\mathcal{T}_T)$ лежит между m и $m^2/2$, где $m = |T|$.

Лемма 12.4.3. Время, используемое алгоритмом для всех необходимых вычислений ДП и проверки клеток, пропорционально $\lambda(\mathcal{T}_P)\lambda(\mathcal{T}_T)$. Следовательно, это время, обозначенное через C , лежит между pm и n^2m^2 .

Доказательство. В алгоритме каждая пара вершин обрабатывается только один раз. На вычисление подтаблицы и проверку ее в момент обработки пары (u, v) алгоритм тратит время $O(|\alpha| \times |\beta|)$, где α и β — метки дуг, входящих, соответственно, в u и v . Поэтому каждая дуговая метка в \mathcal{T}_P образует ровно одну таблицу ДП с каждой дуговой меткой в \mathcal{T}_T . Время на построение таблиц равно $|\alpha|\lambda(\mathcal{T}_T)$. Суммирование по всем дугам в \mathcal{T}_P дает названную границу времени. \square

Приведенная лемма подсчитывает все время, используемое в алгоритме, кроме времени, необходимого для отбора и вывода пар подстрок (по их начальным позициям, длине и редакционному расстоянию). Но так как алгоритм отбирает подстроки, когда он видит в клетке значение, которое удовлетворяет критерию отбора, то время, затрачиваемое на вывод, равно времени, необходимому на обход дерева для

просмотра выводимых пар. Мы уже видели, что это время пропорционально числу выводимых пар R . Следовательно:

Теорема 12.4.3. Полное время в алгоритме равно $O(C + R)$.

Насколько эффективен подход с суффиксным деревом?

Как и в задаче *P*-против-всех, практическая эффективность этого метода зависит от длин \mathcal{T}_P и \mathcal{T}_T . Ясно, что произведение этих длин C падает по мере роста повторяемости в P и T . Мы построили суффиксное дерево для строк ДНК общей длины порядка миллиона баз и обнаружили, что длина дерева составляет около одной десятой от максимально возможной. В этом случае C имеет значение около $n^2m^2/100$, так что при прочих равных условиях (что нереально) стандартное динамическое программирование для задачи все-против-всех считало бы в сто раз медленнее, чем метод гибридного ДП.

О значительном расчете аминокислотных строк “все-против-всех” сообщалось в [183]. Хотя описание очень неопределенно, авторы в основном используют обсуждаемый здесь подход с суффиксными деревьями, вычисляя сходство вместо редакционного расстояния. Но, в отличие от стократного ускорения, они объявили, что достигли ускорения в миллион раз по сравнению со стандартным ДП.*¹) Этот уровень ускорения не подкрепляется теоретическими рассмотрениями (напомним, что для случайной строки S длины m вероятность того, что подстрока длины большей, чем $\log_\sigma m$, встретится в S более одного раза, очень мала). Не поддерживается это заявление и проделанным нами экспериментом. Объяснение может быть связано с введением правила ранней остановки, описанным в [183] только неясным утверждением: “Время экономилось благодаря тому, что сравнение с поддеревьями *patricia***²) прекращалось, когда оценки падали ниже свободно выбранной границы сходства”. Это правило оказалось очень эффективно в сокращении времени счета, но без четкого описания мы не можем определить, какая задача все-против-всех решалась.

12.5. Быстрый (комбинаторный) алгоритм для наибольшей общей подпоследовательности

Задача о наибольшей общей подпоследовательности (*lcs*) является частным случаем общей задачи взвешенного выравнивания или редакционного расстояния. Она может быть решена за время $\Theta(nt)$ либо с применением этих общих методов, либо более непосредственными рекуррентными соотношениями (упражнение 16 главы 11). Однако задача *lcs* играет особую роль в строковых алгоритмах и заслуживает дополнительного обсуждения. Эта особая роль объясняется частично историческими соображениями (многие идеи относительно строк и выравниваний впервые разрабатывались для специального случая *lcs*) и частично надеждой, что *lcs* может найти желаемое соотношение между интересующими нас строками.

Мы представим альтернативный (комбинаторный) метод для *lcs*, который не основан на динамическом программировании. Для двух строк длиной n и $m > n$

*¹) Они выполнили вычисления за 405 дней работы ЦПУ, а этот расчет, по их заявлению, без суффиксных деревьев занял бы миллион лет работы ЦПУ.

**) Дерево *patricia* представляет собой вариант суффиксного дерева. (А само слово “*patricia*” составлено по Practical Algorithm To Retrieve Information Coded In Alphanumeric. — Прим. перев.)

этот метод считает в худшем случае за время $O(r \log n)$, где r — параметр, обычно достаточно малый, чтобы такая граница оказалась привлекательной по сравнению с $\Theta(nm)$. Основная идея заключается в сведении задачи *lcs* к более простой задаче о наибольшей возрастающей подпоследовательности (*lis*). Этот подход можно приспособить для вычисления длины *lcs* за время $O(r \log n)$, используя только линейную память и не прибегая к методу Хиршберга, что будет рассмотрено в упражнении 23.

12.5.1. Наибольшая возрастающая подпоследовательность

Определение. Пусть Π — список из n целых чисел, не обязательно различных. *Возрастающая подпоследовательность* Π — это подпоследовательность Π , значения которой возрастают слева направо.

Например, если $\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$, то $\{3, 4, 6, 8, 10\}$ и $\{5, 9, 10\}$ — две возрастающие подпоследовательности Π . (Вспомним различие между подпоследовательностями и подстроками.) Мы хотим вычислить наибольшую возрастающую подпоследовательность Π . Предлагаемый здесь метод будет использован в дальнейшем для нахождения наибольшей общей подпоследовательности двух (или более) строк.

Определение. *Невозрастающая подпоследовательность* Π — это подпоследовательность Π , в которой числа не возрастают слева направо. *)

Например, $\{8, 5, 5, 3, 1, 1\}$ является невозрастающей подпоследовательностью последовательности $\{4, 8, 3, 9, 5, 2, 5, 3, 10, 1, 9, 1, 6\}$.

Определение. *Покрытие* Π — это набор невозрастающих подпоследовательностей Π , которые содержат все элементы Π .

Например, набор $\{5, 3, 2, 1\}; \{4\}; \{9, 6\}; \{8, 7\}; \{10\}$ является покрытием $\Pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$. Оно включает в себя пять невозрастающих подпоследовательностей, две из которых состоят из отдельных чисел.

Определение. *Размер покрытия* — это число невозрастающих подпоследовательностей в нем, а *наименьшее покрытие* — покрытие минимального размера.

Мы предложим метод трудоемкости $O(n \log n)$, который одновременно строит наибольшую возрастающую подпоследовательность (*lis*) и наименьшее покрытие Π . Все основывается на следующей лемме.

Лемма 12.5.1. *Если I — возрастающая подпоследовательность Π , длина которой равна длине некоторого покрытия C последовательности Π , то I является наибольшей возрастающей подпоследовательностью Π , а C — наименьшим покрытием Π .*

Доказательство. Никакая возрастающая подпоследовательность Π не может содержать больше одного числа из любой невозрастающей подпоследовательности Π .

*) Автор называет такие последовательности убывающими, отмечая возникающую асимметрию обозначений, но делая это ради более короткого термина. Учитывая терпимость русского языка к более громоздким конструкциям, мы сочли возможным в переводе придерживаться точных терминов. — Прим. перев.

Следовательно, длина никакой возрастающей подпоследовательности Π не может превышать размера любого покрытия Π .

Пусть теперь длина I равна размеру C . Тогда I является наибольшей возрастающей подпоследовательностью Π , так как никакая другая возрастающая подпоследовательность длиннее, чем размер C , быть не может. И обратно, C должно быть наименьшим покрытием Π , поскольку если бы было покрытие C' меньшего размера, то длина Π оказалась бы больше размера C' , что невозможно. \square

На лемме 12.5.1 основывается метод нахождения наибольшей возрастающей подпоследовательности и наименьшего покрытия Π . Идея заключается в разложении Π в такое покрытие C , чтобы нашлась возрастающая подпоследовательность I , содержащая ровно по одному числу из каждой невозрастающей подпоследовательности из C . Если не интересоваться эффективностью, то покрытие Π можно построить простым способом.

Наивный алгоритм построения покрытия

Начиная с левого края Π , брать последовательно каждое число из Π и ставить его в конец первой (самой левой) невозрастающей подпоследовательности, которую оно может продолжить. Если ни одну подпоследовательность это число продолжить не может, начать новую (невозрастающую) подпоследовательность справа от уже существующих подпоследовательностей.

Более подробно, если обозначить через x текущее число из Π , то x продолжает подпоследовательность i , если x не превосходит текущего числа в конце подпоследовательности i и x строго больше последнего числа в любой подпоследовательности слева от i .

Например, если взять последовательность Π из предыдущего примера, то первые два числа запишутся в невозрастающую подпоследовательность $\{5, 3\}$. Затем появится число 4, третье по порядку в Π . Число 4 нельзя ставить в конец первой подпоследовательности, так как 4 больше, чем 3. Поэтому 4 начинает новую подпоследовательность справа от первой. Далее рассматривается число 9, и так как его нельзя добавить ни к $\{5, 3\}$, ни к 4, оно начинает третью подпоследовательность. Далее рассматривается 6, его можно добавить после 9, а после двух предшествующих подпоследовательностей — нельзя. Окончательное покрытие всего Π показано на рис. 12.15, где каждая подпоследовательность записана вертикально.

5	4	9	8	10
3		6	7	
2				
1				

Рис. 12.15. Невозрастающее покрытие $\{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$

Ясно, что этот алгоритм формирует покрытие Π , которое мы, как и сам алгоритм, назовем *жадным* (*greedy cover*). Чтобы увидеть, можно ли добавить число x в конкретной невозрастающей подпоследовательности, нам нужно просто сравнить x с числом, скажем, y , находящимся в ее конце, — x можно добавить в том и только том случае, если $x \leq y$. Следовательно, если в момент, когда рассматривается x ,

имеется k подпоследовательностей, то время на добавление x к одной из них составляет $O(k)$. Так как $k \leq n$, мы получаем:

Лемма 12.5.2. *Жадное покрытие Π можно построить за время $O(n^2)$.*

Мы скоро увидим, как уменьшить время, необходимое для построения жадного покрытия, до $O(n \log n)$, но сначала покажем, что оно является наименьшим покрытием Π и что из него легко получить наибольшую возрастающую подпоследовательность.

Лемма 12.5.3. *Существует возрастающая подпоследовательность I из Π , содержащая ровно по одному числу из каждой подпоследовательности жадного покрытия C . Следовательно, I является наибольшей возможной, а C — наименьшим возможным.*

Доказательство. Пусть x — любое число, помещенное в невозрастающую подпоследовательность $i > 1$ (считая слева) жадным алгоритмом. В момент, когда алгоритм рассматривал x , последнее число у подпоследовательности $i - 1$ было меньше, чем x . Кроме того, так как у размещен до x , то он и в Π находится раньше x . Таким образом, $\{y, x\}$ образуют возрастающую подпоследовательность в Π . Так как x любое, то же рассуждение применимо и к y , и если $i - 1 > 1$, то в подпоследовательности $i - 2$ существует число z , такое что $z < y$ и z стоит перед y в Π . Повторяя это рассуждение до достижения первой подпоследовательности, мы получаем возрастающую подпоследовательность в Π , содержащую по одному числу из каждой из первых i подпоследовательностей в жадном покрытии и кончающуюся числом x . Выбор в качестве x любого числа из последней невозрастающей подпоследовательности доказывает лемму. \square

Алгоритмически мы можем найти наибольшую возрастающую подпоследовательность, задаваемую жадным покрытием, следующим образом.

Построение наибольшей возрастающей подпоследовательности

begin

0. Положить i равным числу подпоследовательностей в жадном покрытии.

Положить I равным пустому списку; взять любое число x из подпоследовательности i и поместить в начало списка I .

1. while $i > 1$ do begin

2. Просматривая подпоследовательность $i - 1$ сверху, найти первое число y , которое меньше, чем x .

3. Положить x равным y и i равным $i - 1$.

4. Поместить x в начало списка I .

end;

end.

Так как ни одно число в этом алгоритме не просматривается дважды, наибольшую возрастающую подпоследовательность, заданную жадным покрытием, можно построить за время $O(n)$.

Другой подход использует указатели. При построении жадного покрытия, когда число x добавляется к подпоследовательности i , нужно связать его указателем с числом, которое в этот момент находится в конце подпоследовательности $i - 1$. Когда жадный алгоритм кончает работу, нужно взять любое число в последней невозрастающей подпоследовательности и пройти по единственному пути из указателей от этого числа до первой подпоследовательности.

Быстрое построение жадного покрытия

Уменьшим теперь время на построение жадного покрытия до $O(n \log n)$, что уменьшит до той же границы и общее время счета для нахождения наибольшей возрастающей подпоследовательности.

Пусть L обозначает список, содержащий последние числа невозрастающих подпоследовательностей в ходе алгоритма построения жадного покрытия и упорядоченный по номерам этих подпоследовательностей, начиная от первой.

Лемма 12.5.4. В любой момент работы алгоритма значения в списке L упорядочены по возрастанию.

Доказательство. Будем доказывать лемму по индукции. Начало очевидно. Предположим, что утверждение выполнялось вплоть до итерации $k - 1$. При работе с k -м числом, назовем его x , допустим, что x добавляется в конец подпоследовательности i . Пусть w — текущее значение в конце подпоследовательности $i - 1$, y — текущее число в конце подпоследовательности i (если она существует), а z — число в конце подпоследовательности $i + 1$ (также если она существует). Тогда $w < x \leq y$ по правилам работы алгоритма, и так как $y < z$ по индуктивному предположению, то $x < z$. В итоге $w < x < z$, так что новая подпоследовательность L остается упорядоченной. \square

Отметим, что сам список L не должен быть (и обычно не будет) возрастающей подпоследовательностью Π . Хотя $x < z$, но x появляется справа от z в Π . На каждой итерации k нужно взять k -е число x из Π и найти в текущем списке L самое левое число, большее чем x . Так как L отсортирован, это можно сделать *двоичным поиском* за время $O(\log n)$. Список Π содержит n чисел, так что мы получаем:

Теорема 12.5.1. Жадное покрытие можно построить за время $O(n \log n)$. Поэтому наибольшая возрастающая подпоследовательность и наименьшее покрытие Π могут быть получены за время $O(n \log n)$.

На самом деле, если p — длина *lis*, то ее можно найти за время $O(n \log p)$.

12.5.2. Сведение наибольшей общей подпоследовательности к наибольшей возрастающей последовательности

Будем решать задачу о наибольшей общей подпоследовательности для пары строк, используя метод нахождения наибольшей возрастающей подпоследовательности в списке целых чисел.

Определение. Пусть заданы строки S_1 и S_2 (длиной m и n соответственно) над алфавитом Σ . Обозначим через $r(i)$ количество вхождений i -го символа строки S_1 в строку S_2 .

Определение. Пусть r обозначает сумму $\sum_{i=1}^m r(i)$.

Например, если мы используем нормальный латинский алфавит, то при $S_1 = abacx$ и $S_2 = baabca$ имеем $r(1) = 3, r(2) = 2, r(3) = 3, r(4) = 1$ и $r(5) = 0$, так что $r = 9$. Ясно, что для двух произвольных строк число r будет меняться в диапазоне от 0 до nm . Мы решим задачу lcs за время $O(r \log n)$ (где $n < m$), что сопоставимо с $O(nm)$, когда r велико. Однако, будучи зависимым от алфавита Σ , r зачастую оказывается существенно меньше, чем nm . Ниже обсудим это более подробно.

Сведение

Для каждого символа x , встречающегося в S_1 хотя бы один раз, создадим список тех позиций, где x входит в строку S_2 , и запишем список в убывающем порядке. Списки для различных символов окажутся дизъюнктны. В приведенном выше примере ($S_1 = abacx$ и $S_2 = baabca$) для a получим 6, 3, 2, а для b — 4, 1.

Теперь построим список длиной r , который обозначим через $\Pi(S_1, S_2)$. В нем каждое *вхождение* символа в S_1 мы заменим списком, изготовленным для этого символа. То есть для каждой позиции i в S_1 берется список, сопоставленный символу $S_1(i)$. Например, списком $\Pi(S_1, S_2)$ для того же примера будет 6, 3, 2, 4, 1, 6, 3, 2, 5.

Чтобы понять важность $\Pi(S_1, S_2)$, посмотрим, каков смысл возрастающей подпоследовательности в этом списке в терминах исходных строк.

Теорема 12.5.2. Каждая возрастающая подпоследовательность I в $\Pi(S_1, S_2)$ определяет общую подпоследовательность S_1 и S_2 той же длины, и наоборот. Таким образом, наибольшая общая подпоследовательность S_1 и S_2 соответствует наибольшей возрастающей подпоследовательности в списке $\Pi(S_1, S_2)$.

Доказательство. Прежде всего, по заданной возрастающей подпоследовательности I списка $\Pi(S_1, S_2)$ построим строку S и покажем, что S является подпоследовательностью и S_1 и S_2 . Стока S благополучно строится просмотром I слева направо. Во время этого просмотра построим также два списка индексов, определяющих подпоследовательность S_1 и подпоследовательность S_2 . Более подробно: если число j найдено во время просмотра в I и оно содержится в подсписке, представленном i -м символом S_1 , то нужно приписать символ $S_1(i)$ в правом конце S , i — в правом конце первого индексного списка и j — в правом конце второго индексного списка.*)

Например, возьмем в рассматриваемом примере список $I = 3, 4, 5$. Число 3 появилось из подсписка для символа 1, число 4 — из подсписка для символа 2, а 5 — из подсписка для символа 4. Значит, $S = abc$. Эта строка является подпоследовательностью S_1 , соответствующей позициям 1, 2, 4, и подпоследовательностью S_2 , соответствующей позициям 3, 4, 5.

Список $\Pi(S_1, S_2)$ содержит один подсписок для каждой позиции в S_1 , и каждый такой подсписок в $\Pi(S_1, S_2)$ представлен в убывающем порядке. Поэтому из каждого подсписка в I включается не более одного числа, и любая позиция из S_1 добавляется в S не более одного символа. Далее, m списков выстроены слева направо в порядке, соответствующем порядку символов в S_1 , так что S является последовательностью S_1 .

*¹) Эта конструкция неточна. Выбор пар i и j не может быть произвольным, так как должно соблюдаться условие возрастания списков S_1 и S_2 . Исправление конструкции можно рассматривать как хорошее упражнение для читателя. — Прим. перев.

Числа в I строго возрастают и соответствуют позициям в S_2 , так что S является также подпоследовательностью S_2 .

В итоге мы доказали, что каждую возрастающую подпоследовательность в $\Pi(S_1, S_2)$ можно использовать для построения общей подпоследовательности в S_1 и S_2 той же длины. Обратное утверждение, что по общей подпоследовательности строится возрастающая подпоследовательность, очень похоже и оставляется как упражнение. \square

$\Pi(S_1, S_2)$ — это список из r целых чисел, а задачу о наибольшей возрастающей подпоследовательности можно решить за время $O(r \log l)$, когда длина наибольшей возрастающей подпоследовательности равна l . Если $n \leq m$, то $l \leq n$, откуда получается следующая теорема.

Теорема 12.5.3. Задача о наибольшей общей подпоследовательности может быть решена за время $O(r \log n)$.

Метод трудоемкости $O(r \log n)$ для задачи *lcs* был впервые получен Хантом и Шиманьски [238]. Их алгоритм на первый взгляд очень отличается от приведенного, но ретроспективный анализ обнаруживает, что в него заложены близкие идеи. Связь между задачами *lcs* и *lis* была частично обнаружена Апостолико и Гуэрра [25, 27] и явно использована Джекобсоном и Во [244], а также независимо Певзнером и Утерменом [370].

Метод решения задачи *lcs*, основанный на *lis*, представляет пример *разреженного динамического программирования*, когда исходные данные составляют относительно редкое множество пар, которые разрешено выравнивать. Этот подход, да и техника решения, рассмотренные здесь, широко обобщались и детально описаны в [137, 138].

12.5.3. Насколько хорош этот метод

Насколько хорош метод решения задачи *lcs*, основанный на *lis*, по сравнению с исходным подходом ДП, работающим за время $\Theta(nm)$? Это зависит от величины r . Пусть σ обозначает размер алфавита Σ . Очень наивный анализ сказал бы, что r можно ожидать равным примерно nm/σ . При этом предполагается, что каждый символ в Σ появляется с равной вероятностью и, следовательно, ожидается, что он появится в короткой строке n/σ раз. Это означает, что $r_i = n/\sigma$ для любого i . Длинная строка имеет длину m , так что r ожидается равным nm/σ . Но, конечно, равновероятное распределение символов нетипично, и значение r сильно зависит от конкретных строк.

Для латинского алфавита со строчными и заглавными буквами, цифрами и знаками препинания размер σ около 100, но предположение о равновероятности этих знаков явно нарушается. Кроме того, может появиться вопрос, выглядит ли $(nm/100) \log n$ привлекательнее, чем nm . В случае таких алфавитов указанное ускорение не прельщает, хотя метод сохраняет свою простоту и эффективность по памяти. Так, для обычного английского текста подход, основанный на *lis*, может быть предпочтительнее, чем метод ДП. Однако во многих приложениях размер “алфавита” очень велик и растет с размером текста.*¹) Это верно, например, для утилиты *diff* системы Unix, в которой каждая строка текста рассматривается как символ в “алфавите”,

*¹) Это одно из немногих мест в книге, где мы отходим от стандартного предположения о фиксированности алфавита.

используемом в вычислении *lcs*. В некоторых приложениях из молекулярной биологии алфавит состоит из образцов или подстрок, а не из четырех символов алфавита ДНК и не из двадцати символов алфавита белков. Эти подстроки могут быть генами, экзонами или распознающими последовательностями рестриктаз. В таких случаях размер алфавита велик по сравнению с размером строки, так что r мало и $r \log n$ выглядит заманчивее, чем nm .

Задача *lcs* с ограничениями

Метод решения *lcs*, основанный на использовании *lis*, имеет еще одно преимущество перед стандартным ДП. В некоторых приложениях есть дополнительные ограничения, связанные с тем, какие пары позиций можно сочетать в *lcs*. Значит, в добавление к ограничению, что позицию i в S_1 можно сочетать с позицией j в S_2 , только если $S_1(i) = S_2(j)$, появляется еще какие-то ограничения. Сведение задачи *lcs* к *lis* можно легко модифицировать, учитывая эти дополнительные ограничения, и мы оставляем это читателю. Эффект заключается в уменьшении размера r и, следовательно, в ускорении решения *lcs*. Это еще один пример и вариант разреженного динамического программирования.

12.5.4. Задача *lcs* для более чем двух строк

Одним из приятных свойств метода решения *lcs*, основанного на *lis*, является то, что его легко можно перенести на задачу *lcs* с более чем двумя строками. Эта задача — частный случай *множественной задачи выравнивания последовательностей*, ключевой задачи вычислительной молекулярной биологии, которую мы более полно рассмотрим в главе 14. Переход с двух строк ко многим будет иллюстрироваться на примере трех строк: S_1 , S_2 и S_3 .

Идея снова состоит в сведении задачи *lcs* к задаче *lis*. Как и раньше, мы начнем с построения списка для каждого символа x из S_1 . То есть список для x будет содержать пары чисел, в каждой из которых будут позиции x в S_2 и S_3 . Далее, список для x будет упорядочен по *лексикографическому убыванию*. Это значит, что если пара (i, j) появляется в этом списке до (i', j') , то либо $i > i'$, либо $i = i'$ и $j > j'$. Например, если $S_1 = abacx$, $S_2 = baabca$ (как и раньше) и $S_3 = babbac$, то список для символа a равен $(6, 5), (6, 2), (3, 5), (3, 2), (2, 5), (2, 2)$.

Списки для всех символов снова сцепляются в том порядке, в котором символы появляются в строке S_1 , образуя последовательность пар $\Pi(S_1, S_2, S_3)$. Мы определим возрастающую подпоследовательность в $\Pi(S_1, S_2, S_3)$ как подпоследовательность пар, такую что первые числа выбранных пар образуют возрастающую подпоследовательность и вторые числа тоже образуют возрастающую подпоследовательность. Можно легко модифицировать алгоритм жадного покрытия, чтобы находить наибольшую возрастающую подпоследовательность пар в смысле этого определения. Возрастающая подпоследовательность используется следующим образом.

Теорема 12.5.4. *Любая возрастающая подпоследовательность в $\Pi(S_1, S_2, S_3)$ определяет общую подпоследовательность S_1, S_2, S_3 той же длины, и наоборот. Поэтому наибольшая общая подпоследовательность S_1, S_2, S_3 соответствует наибольшей возрастающей подпоследовательности в $\Pi(S_1, S_2, S_3)$.*

Доказательство этой теоремы похоже на случай двух строк и оставляется как упражнение. Приспособление алгоритма жадного покрытия и его временной анализ

для двух строк также оставляются читателю. Обобщение на случай большего числа строк делается непосредственно. Комбинаторный подход к вычислению *lcs* имеет также симпатичную особенность, экономящую память; мы рассмотрим этот вопрос в упражнениях.

12.6. Вогнутый вес пропусков

Сегодня молекулярные биологи используют аффинную модель штрафов за пропуски в подавляющем большинстве случаев и особенно для выравнивания последовательностей аминокислот. Однако в 1984 г. Уотерменом [466] была предложена и изучена более богатая возможностями модель пропусков — модель *вогнутых весов*. Обсуждая общее использование аффинного штрафа, Беннер, Коэн и Гоннет утверждают, что “для такой трактовки нет никакого обоснования — ни теоретического, ни эмпирического” [183], и настойчиво убеждают, что “нелинейный штраф за пропуски — единственный, который основан на эмпирических данных” [57]. Они предлагают [57], чтобы при выравнивании двух белковых последовательностей, которые различаются в d единицах РАМ (см. п. 15.7.2), пропуск длины q оценивался весом:

$$35.03 - 6.88 \log_{10} d + 17.02 \log_{10} q.$$

При такой модели начальный вес пропуска не превосходит 35.03 и падает при увеличении эволюционного (РАМ) расстояния между двумя последовательностями. Кроме этого функция добавляет $17.02 \log_{10} q$ за действительную длину пропуска q .

Трудно поверить, что функция, заданная с такой точностью, может быть правильной, но главное в том, что для фиксированного РАМ-расстояния предлагаемая весовая функция является *вогнутой* функцией длины пропуска.*)

Задача выравнивания с вогнутым весом пропуска труднее для решения, чем задача с аффинными весами, но не так сложна, как задача с произвольными весами. В этом пункте мы рассмотрим практический алгоритм оптимального выравнивания двух строк длины n и $m > n$, когда вес задан вогнутой функцией длины пропуска. Алгоритм работает за время $O(nm \log m)$ по сравнению с временем $O(nm)$ для аффинной функции и $O(nm^2)$ для произвольной. Ускорение для вогнутого случая было получено Миллером и Майерсом [322] и независимо Галилом и Джанкарло [170]. Однако во второй статье решениедается в терминах редакционного расстояния, а не сходства. Сходство зачастую полезнее, чем редакционное расстояние, поскольку его можно использовать при работе с исключительно важным случаем локального сравнения. Поэтому мы будем рассматривать задачу с вогнутыми весами в терминах сходства (максимального взвешенного выравнивания) и оставим читателю вывод аналогичных алгоритмов для вычисления редакционного расстояния при вогнутых весах пропусков. Результаты дальнейших исследований о выравнивании с выпуклыми и вогнутыми весами описаны в [136, 138, 276].

*) Автор использует термин “выпуклая” и пишет: «К сожалению, нет стандартного соглашения о терминологии, и некоторые работы называют эту модель моделью “с выпуклым” весом, а другие “с вогнутым”. Повторю, что мы были вынуждены сменить авторскую терминологию и перейти на термин “вогнутая”, так как в русскоязычной научной литературе функция с отрицательной второй производной называется зогнутой уже без сомнений. Впрочем, этот вес входит в штраф со знаком минус и получающаяся функция штрафа является уже выпуклой. — Прим. перев.

Вспомним из обсуждения произвольных весовых функций, что $w(q)$ — вес пропуска длины q . Этот пропуск добавляет штраф $-w(q)$ к полному весу выравнивания

Определение. Предположим, что $w(q)$ — неотрицательная функция q . Тогда $w(q)$ *вогнутая* в том и только том случае, если для любого q

$$w(q+1) - w(q) \leq w(q) - w(q-1).$$

Значит, при увеличении длины пропуска дополнительный штраф за пропуск уменьшается с каждой новой единицей пропуска. Отсюда следует, что $w(q+d) - w(q) \geq w(q'+d) - w(q')$ для $q < q'$ и любого фиксированного d (рис. 12.16). Функция w может иметь участки с положительным и отрицательным наклоном, хотя любой участок с положительным наклоном должен быть слева от участка с отрицательным наклоном. Заметим, что определение позволяет $w(q)$ становиться отрицательной величиной для достаточно больших длин строк. При этом $-w(q)$ становится положительным, что, вероятно, нежелательно. Следовательно, весовую функцию с отрицательным наклоном нужно использовать с осторожностью.

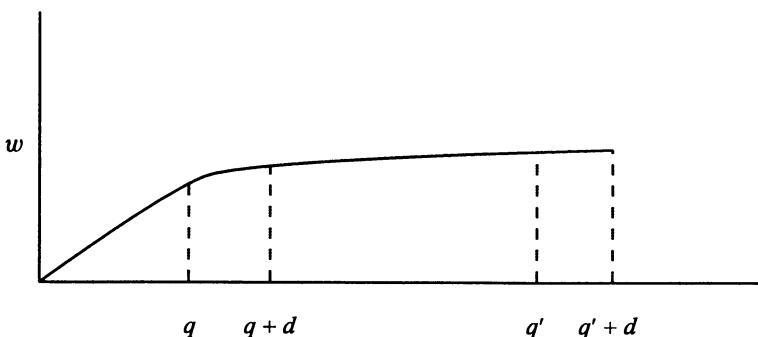


Рис. 12.16. Вогнутая функция w

Вогнутый вес пропуска был введен в [466] с предположением, что мутационные события, которые вставляют или удаляют из ДНК блоки переменной длины, могут более содержательно моделироваться с помощью вогнутых весов, чем с помощью аффинных или постоянных. Вогнутый штраф дает постановщику задачи больше возможностей отражения стоимости или вероятности различных длин пропусков и сохраняет достаточную эффективность работы по сравнению с произвольными весами. Конкретная вогнутая функция, к которой прибегают в этом аспекте, — это *логарифм*, хотя неясно, какое основание логарифма здесь было бы осмысленно.

Дискуссия вокруг вогнутых весов еще не утихла, и модели с аффинными весами продолжают доминировать на практике. Но если даже вогнутые веса не станут популярны в молекулярной биологии, они могут найти применение в других приложениях. Более того, алгоритмы выравнивания с вогнутыми весами представляют интерес сами по себе как представители класса алгоритмов из общей области “разреженного динамического программирования”.

Ускорение счета по общим рекуррентным соотношениям

Чтобы решить задачу для вогнутых весов, используем те же рекуррентные соотношения ДП, что и для произвольных весов (с. 301), просто уменьшим время, необходимое для вычислений. Для удобства перепишем эти соотношения по-другому:

$$\begin{aligned} V(i, j) &= \max\{E(i, j), F(i, j), G(i, j)\}, \\ G(i, j) &= V(i - 1, j - 1) + s(S_1(i), S_2(j)), \\ E(i, j) &= \max\{V(i, k) - w(j - k) : 0 \leq k \leq j - 1\}, \\ F(i, j) &= \max\{V(l, j) - w(i - l) : 0 \leq l \leq i - 1\}, \\ V(i, 0) &= -w(i), \quad V(0, j) = -w(j), \\ E(i, 0) &= -w(i), \quad F(0, j) = -w(j); \end{aligned}$$

$G(i, j)$ не определено, когда i или j равно 0.

Даже при произвольных весах работа в первом и втором соотношениях требует $O(m)$ на строчку, что допустимо при желаемой нами временной границе. Трудность таится в соотношениях для $E(i, j)$ и $F(i, j)$, первое из которых требует времени $\Theta(m^2)$ на строчку, а второе — $\Theta(n^2)$ на столбец, если функция w произвольна. Следовательно, в случае, когда w вогнута, нужно улучшать оценку именно E и F для данной строчки или столбца. Сосредоточимся на вычислении E для одной строчки. Вычисление F и анализ времени на расчет одного столбца вполне симметричны, при одном предупреждении, которое будет обсуждаться дальше.

Упрощение обозначений

Значение $E(i, j)$ зависит от i только через значения $V(i, k)$ для $k < i$. Следовательно, в любой фиксированной строчке можно опустить зависимость от индекса строчки i , упрощая соотношение для E . Значит, в любой фиксированной строчке мы определяем

$$E(j) = \max\{V(k) - w(j - k) : 0 \leq k \leq j - 1\}.$$

Далее для упрощения соотношений введем следующее обозначение:

$$Cand(k, j) = V(k) - w(j - k);$$

поэтому

$$E(j) = \max\{Cand(k, j) : 0 \leq k \leq j - 1\}.$$

Обозначение *Cand* происходит от *candidate*, а смысл его станет ясен позднее.

12.6.1. Впередсмотрящее динамическое программирование

Для дальнейшего изложения будет полезно изменить способ реализации процесса динамического программирования. Как правило, при вычислении значения $E(j)$ мы заглядываем в строчку *назад*, чтобы, сравнивая все значения $Cand(k, j)$ для $k < j$, выбрать наибольшее из них в качестве $E(j)$. Но возможна и другая, *впередсмотрящая* (или *прямая*) реализация, которая в этом изложении будет полезнее.*)

*) Джин Лаулер отмечал, что в некоторых кругах прямую и обратную реализацию называют *Тянитолкаем* (push you—pull me) динамического программирования. Читатель может сам определить, какой термин относится к “вперед”, а какой к “назад”.

В прямой реализации мы начинаем с инициализации переменной $\bar{E}(j')$ значением $Cand(0, j')$ для каждой клетки $j' > 0$ в строчке. Строчка E дополняется слева направо, как в обратном динамическом программировании. Однако чтобы определить $E(j)$ (для любого $j > 0$) алгоритм просто полагает его равным текущему значению $\bar{E}(j)$, так как каждая клетка слева от j уже представила свое “кандидатское” значение клетке j . Затем, перед определением $E(j+1)$, алгоритм проходит по строчке *вперед*, чтобы $\bar{E}(j')$ (для каждого $j' > j$) было максимумом из текущего значения $\bar{E}(j')$ и $Cand(j, j')$. В итоге прямая реализация для фиксированной строчки такова:

Прямое динамическое программирование для фиксированной строчки

```

for  $j := 1$  to  $m$  do begin
     $\bar{E}(j) := Cand(0, j);$ 
     $b(j) := 0$ 
end;
for  $j := 1$  to  $m$  do begin
     $E(j) := \bar{E}(j);$ 
     $V(j) := \max\{G(j), E(j), F(j)\};$ 
    {Мы предполагаем, что значения  $F(j)$  и  $G(j)$  уже вычислены
    для клетки  $j$  этой строчки, хотя и не доказываем этого.}
    for  $j' := j + 1$  to  $m$  do [Цикл 1]
        if  $\bar{E}(j) < Cand(j, j')$  then begin
             $\bar{E}(j') := Cand(j, j');$ 
             $b(j') := j;$  {Эта установка указателя из  $j'$  на  $j$  объясняется ниже.}
        end
    end;

```

Можно по-другому представить себе прямое ДП, рассмотрев для задачи выравнивания взвешенный редакционный граф (см. п. 11.4). В этом (ациклическом) графе оптимальный путь (кратчайший или самый длинный, в зависимости от типа вычисляемого выравнивания) из клетки $(0, 0)$ в клетку (n, m) определяет оптимальное выравнивание. Следовательно, алгоритмы, которые вычисляют оптимальные расстояния в (ациклических) графах, можно использовать для получения оптимальных выравниваний, а сами алгоритмы (например, алгоритм Дейкстры для кратчайших расстояний) описать как впередсмотрящие. Когда правильное расстояние $d(v)$ до вершины v уже вычислено и есть дуга из v в вершину w , правильное расстояние до которой еще не известно, алгоритм добавляет $d(v)$ к длине дуги (v, w) , чтобы получить еще одно возможное значение правильного расстояния до w . Когда вычислены правильные расстояния до всех вершин, из которых есть дуги в v , и каждая представила своего кандидата на значение для v , правильное расстояние до v равно наилучшему из этих кандидатов.

Должно быть ясно, что в прямом и обратном ДП выполняются одни и те же арифметические операции и сравнения, единственное различие — в порядке выполнения этих операций. Отсюда следует, что прямой алгоритм корректно находит все значения E в фиксированной строчке и требует по-прежнему времени $\Theta(m^2)$ на строчку. Таким образом, прямое ДП не быстрее обратного, но способствует объяснению предлагаемого ниже ускорения.

12.6.2. Основа ускорения

В момент вычисления $E(j)$ назовем клетку j *текущей клеткой*. Мы интерпретируем $Cand(j, j')$ как “кандидата на значение” для $E(j')$, которого клетка j “посыпает вперед” клетке j' . Когда клетка j является текущей, она “засыпает вперед” $m - j$ кандидатов, по одному каждой клетке $j' > j$. Каждое такое значение $Cand(j, j')$ сравнивается с текущим $\bar{E}(j')$; оно либо *выигрывает* это сравнение (когда $Cand(j, j') > \bar{E}(j')$), либо *проигрывает*. Ускорение получается за счет опознания и исключения большого числа кандидатов, которые не имеют шансов выиграть сравнения. Так алгоритм избегает большого числа бесполезных сравнений. Этот подход иногда называется методом *списка кандидатов*. Вот основное наблюдение, используемое для опознания “проигрывающих” кандидатов:

Основное наблюдение. Пусть клетка j — текущая. Если $Cand(j, j') \leq \bar{E}(j')$ для какого-либо $j' > j$, то $Cand(j, j'') \leq \bar{E}(j'')$ для всех $j'' > j'$. Значит, “пропустил удар — уходи совсем”.

Следовательно, текущая клетка j не должна засыпать кандидатов на значения справа от первой клетки $j' > j$, где $Cand(j, j') > \bar{E}(j')$. Мы получаем очевидное практическое ускорение досрочным прекращением цикла, помеченного {Цикл 1} в прямом алгоритме динамического программирования, при первом поражении кандидата от j . Однако это не приводит прямо к улучшению временной оценки наихудшего случая. Нам потребуется еще один прием, но сначала мы докажем основное наблюдение с помощью следующей более точной леммы.

Лемма 12.6.1. Пусть $k < j < j' < j''$ — любые четыре клетки в одной и той же строке. Если $Cand(j, j') \leq Cand(k, j')$, то $Cand(j, j'') \leq Cand(k, j'')$ (рис. 12.17).

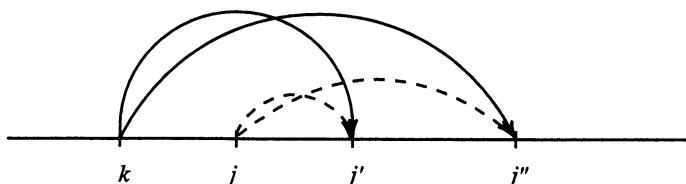


Рис. 12.17. Иллюстрация к основному наблюдению. Выигрывающие кандидаты показаны сплошной линией, а побежденные — штриховой. Если кандидат из j проиграл кандидату из k в клетке j' , то кандидат из j проиграет кандидату из k в любой клетке j'' справа от j' .

Доказательство. Из $Cand(k, j') \geq Cand(j, j')$ следует, что $V(k) - w(j' - k) \geq V(j) - w(j' - j)$, так что $V(k) - V(j) \geq w(j' - k) - w(j' - j)$.

Тривиально, $(j' - k) = (j' - j) + (j - k)$. Аналогично, $(j'' - k) = (j'' - j) + (j - k)$. Для дальнейшего использования отметим, что $(j' - k) < (j'' - k)$.

Теперь пусть $q = j' - j$, $q' = j'' - j$ и $d = j - k$. Так как $j' < j''$, то $q < q'$. Ввиду вогнутости, $w(q + d) - w(q) \geq w(q' + d) - w(q')$ (см. рис. 12.16). Подстановка дает $w(j' - k) - w(j' - j) \geq w(j'' - k) - w(j'' - j)$. Комбинируя это с результатом из первого абзаца, имеем $V(k) - V(j) \geq w(j'' - k) - w(j'' - j)$, или $V(k) - w(j'' - k) \geq V(j) - w(j'' - j)$, т. е. $Cand(k, j'') \geq Cand(j, j'')$, как и утверждалось. \square

Из леммы 12.6.1 немедленно получается основное наблюдение.

12.6.3. Указатели на клетки и разбиение строчек

Напомним, что при детальном описании прямого алгоритма ДП для каждой клетки j' была введена переменная $b(j')$. В ней хранится указатель на самую левую клетку $k < j'$, создавшую наилучшего кандидата из тех, с которыми клетка j' уже встретилась. Указатель $b(j')$ обновляется при каждом изменении значения $E(j')$. Использование этих указателей вместе с доказанной леммой и даст желаемое ускорение.

Лемма 12.6.2. *Рассмотрим момент, когда клетка j является текущей, но до того, как она заслала вперед кандидатов на значения. В этом случае $b(j') > b(j' + 1)$ для любой клетки j' от $j + 1$ до $m - 1$.*

Доказательство. Для простоты обозначений пусть $b(j') = k$ и $b(j' + 1) = k'$. Тогда в соответствии с выбором k , $Cand(k, j') > Cand(k', j')$. Теперь предположим, что $k < k'$. По лемме 12.6.1 получим $Cand(k, j' + 1) \geq Cand(k', j' + 1)$, и в этом случае $b(j' + 1)$ должно равняться k , а не k' . Следовательно, $k > k'$ и лемма доказана. \square

Теперь изменим формулировку леммы 12.6.2 на более удобную.

Следствие 12.6.1. *В момент, когда клетка j является текущей, но до того, как она заслала вперед кандидатов на значения, указатели b образуют последовательность, невозрастающую слева направо. Поэтому клетки $j, j + 1, j + 2, \dots, m$ разбиты на максимальные блоки последовательных клеток, в которых указатели b имеют одно и то же значение, убывающее от блока к блоку.*

Определение. Разбиение клеток от j до m , о котором говорится в следствии 12.6.1, называется текущим блочным разбиением (рис. 12.18).

99999	7777	666666	33	111
j				

Рис. 12.18. Разбиение клеток от $j + 1$ до m на максимальные последовательные блоки, такие что все клетки в одном блоке имеют одно и то же значение b . Это значение в каждом блоке меньше, чем в предыдущем

Благодаря следствию 12.6.1 алгоритм не должен держать указатели b для каждой клетки, достаточно записывать общий указатель b для блока. Этот факт будет использован при получении требуемого ускорения.

Подготовка к ускорению

Наша цель — уменьшить время, расходуемое на поиск значений E в пересчете на одну строчку с $\Theta(m^2)$ до $O(m \log m)$. Основная выполняемая в строчке работа — это обновление значений \bar{E} и обновление текущего блочного разбиения и относящихся к нему указателей. Мы начнем с обновления блочного разбиения и указателей b ; после этого обработка значений \bar{E} будет несложной. Итак, допустим, что вся работа со значениями \bar{E} идет бесплатно.

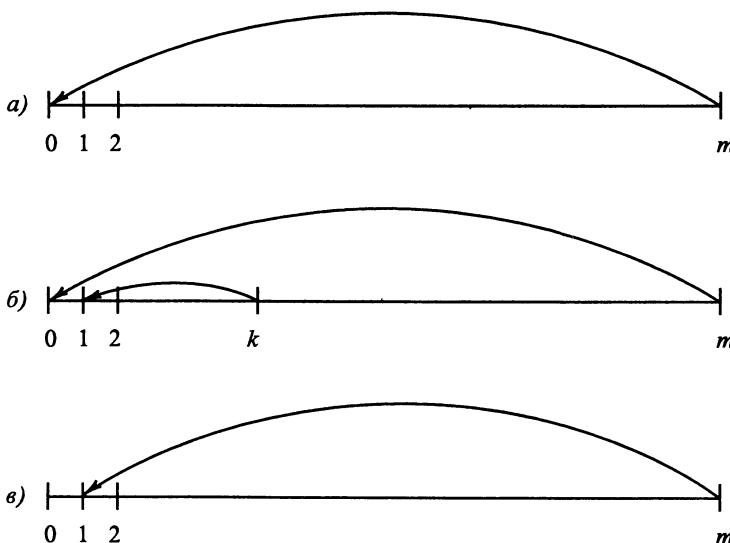


Рис. 12.19. Три возможных способа изменения блочного разбиения после вычисления $E(1)$. Стрелки изображают общие указатели блока, дуга со стрелкой выходит из последней клетки блока

Рассмотрим момент, когда клетка j является текущей, но еще не заслала своих кандидатов на значения. После вычисления $E(j)$ (и $F(j)$, а затем $V(j)$) алгоритм должен обновить блочное разбиение и требуемые указатели b . Чтобы увидеть новую идею, рассмотрим случай $j = 1$. В этот момент есть только один блок (содержащий клетки от 1 до m), с общим указателем b на клетку 0 (т.е. $b(j') = 0$ для всех клеток j' в этом блоке). После того как $E(1)$ получит значение $\bar{E}(1) = \text{Cand}(0, 1)$, любое измененное значение $\bar{E}(j')$ вызовет и изменение блочного разбиения. В частности, если изменяется $\bar{E}(j')$, то $b(j')$ изменяется с 0 на 1. Но так как значения b в новом блочном разбиении должны не возрастать слева направо, то для нового блочного разбиения есть только три возможности*) (рис. 12.19):

- Клетки от 2 до m могут остаться в одном блоке с общим указателем $b = 0$. По лемме 12.6.1 это произойдет в том и только том случае, если $\text{Cand}(1, 2) \leqslant \bar{E}(2)$.
- Клетки от 2 до m могут разделиться на два блока, в которых общий указатель на первый блок будет $b = 1$, а общий указатель на второй блок — $b = 0$. Это произойдет (снова по лемме 12.6.1) в том и только том случае, если для некоторого $k < m$ выполняются неравенства $\text{Cand}(1, j') > \bar{E}(j')$ для j' от 2 до k и $\text{Cand}(1, j') \leqslant \bar{E}(j')$ для j' от $k + 1$ до m .
- Клетки от 2 до m могут остаться в одном блоке, но общий указатель b получит значение 1. Это произойдет в том и только том случае, если $\text{Cand}(1, j') > \bar{E}(j')$ для всех j' от 2 до n .

*) Мы используем при анализе этих вариантов старые значения \bar{E} .

Поэтому, прежде чем менять значения \bar{E} , можно эффективно вычислить новое разбиение клеток от 2 до m следующим образом. Сначала сравниваются $\bar{E}(2)$ и $Cand(1, 2)$. Если $\bar{E}(2) \geq Cand(1, 2)$, то все клетки справа от 2 остаются в одном блоке с общим $b = 0$. Если же $\bar{E}(2) < Cand(1, 2)$, то алгоритм ищет крайнюю левую клетку $j' > 2$, для которой $\bar{E}(j') > Cand(1, j')$. Если такое j' найдется, то клетки от 2 до $j' - 1$ образуют новый блок с общим указателем на клетку 1, а оставшиеся клетки — другой блок с указателем на клетку 0. Если j' не нашлось, то все клетки от 2 до m остаются в одном блоке, но с $b = 1$.

И вот в чем соль: по следствию 12.6.1 индекс j' можно искать двоичным поиском. Таким образом, на него будет затрачено только $O(\log m)$ сравнений. И поскольку мы записываем один указатель на блок, потребуется обновить не более одного указателя.

Рассмотрим теперь общий случай $j > 1$. Предположим, что $E(j)$ уже найдено и что клетки $j + 1, \dots, m$ разбиты на r блоков, кончающихся в клетках $p_1 < p_2 < \dots < p_r = m$. Блок, кончающийся в p_i , будет называться i -м блоком. Обозначим через b_i общий указатель клеток в блоке i . Предположим, что есть список концов блоков $p_1 < p_2 < \dots < p_r$ и параллельный список указателей $b_1 > b_2 > \dots > b_r$.

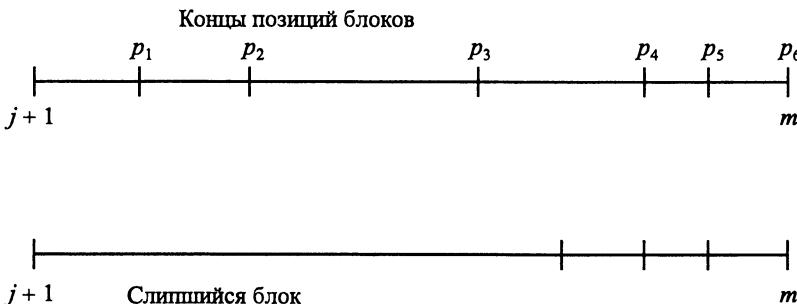


Рис. 12.20. При обновлении блочного разбиения алгоритм последовательно просматривает клетки p_i в поиске первого индекса s , для которого $\bar{E}(p_s) \geq Cand(j, p_s)$. На рисунке $s = 4$. Блоки с первого по $s - 1 = 3$ слипаются в единый блок с некоторой начальной частью блока $s = 4$. Блоки справа от s не изменяются

После вычисления $E(j)$ новое разбиение клеток от $j + 1$ до m получим следующим образом. Сначала если $\bar{E}(j + 1) \geq Cand(j, j + 1)$, то по лемме 12.6.1 $\bar{E}(j') \geq Cand(j, j')$ для всех $j' > j$, так что разбиение клеток, больших j , остается неизменным. В противном случае (если $\bar{E}(j + 1) < Cand(j, j + 1)$) алгоритм последовательно сравнивает $\bar{E}(p_i)$ с $Cand(j, p_i)$ для i от 1 до r , пока либо не исчерпается список концов блоков, либо не найдется первый индекс s , для которого $\bar{E}(p_s) \geq Cand(j, p_s)$. В первом случае клетки $j + 1, \dots, m$ попадают в общий блок с общим указателем на клетку j . Во втором случае блоки от $s + 1$ до r остаются неизмененными, а все блоки от 1 до $s - 1$ слипаются с некоторой начальной частью блока s (возможно, со всем блоком), образуя один блок с общим указателем на клетку j (рис. 12.20). Отметим, что каждое сравнение, кроме последнего, вызывает слияние двух соседних блоков в один.

Определив блок s , алгоритм находит правильное место деления блока s , выполняя двоичный поиск по клеткам блока. Это делается так же, как в рассмотренном случае $j = 1$.

12.6.4. Последние детали реализации и временной анализ

Мы описали, как обновлять блочное разбиение и общие указатели b , но в этом описании использовались значения \bar{E} в предположении, что они достаются нам бесплатно. Займемся теперь этой проблемой.

Основное наблюдение заключается в том, что алгоритм использует $\bar{E}(j)$, только когда j становится текущей клеткой, а $\bar{E}(j')$ — только когда проверяет клетку j' в процессе обновления блочного разбиения. Но текущая клетка j находится всегда в первом блоке текущего блочного разбиения (конечная клетка которого обозначена через p_1), так что $b(j) = b_1$ и $\bar{E}(j) = Cand(b_1, j)$, а это значение при необходимости может быть вычислено за константное время. Вдобавок при проверке клетки j' в процессе обновления блочного разбиения алгоритм знает блок, в который попадает j' , скажем, блок i , и следовательно, он знает b_i . Поэтому можно вычислить $\bar{E}(j)$ за константное время, найдя $Cand(b_i, j')$. В результате нет нужды хранить значения \bar{E} ; они легко вычисляются по мере надобности. В каком-то смысле это требуется только для изложения. Более того, число значений \bar{E} , которые вычисляются “на ходу”, пропорционально числу сравнений, которые алгоритм выполняет для поддержания блочного разбиения. Эти наблюдения суммируются в следующем алгоритме:

Модифицированный прямой алгоритм ДП для фиксированной строчки

Создать список концов блоков с единственным числом m .

Создать список указателей с единственным числом 0.

for $j := 1$ to m do begin

 Положить k равным первому указателю в списке указателей b .

$E(j) := Cand(k, j);$

$V(j) := \max\{G(y), E(j), F(j)\};$

 {Как раньше, мы предполагаем, что нужные значения F и G вычислены.}

 {Смотрим, как кандидаты для j меняют блочное разбиение.}

 Положить j' равным первому элементу в списке концов блоков.

 {Ищем первый индекс s в списке концов блоков, где j проигрывает.}

 if $Cand(b(j'), j + 1) < Cand(j, j + 1)$ then begin

 {кандидат для j выигрывает 1}

 while

 Список концов блоков непуст, и $Cand(b(j'), j') < Cand(j, j')$

 do begin

 Удалить первую запись из списка концов блоков

 и соответствующий указатель b .

 if список концов блоков непуст then

 Положить j' равным новому началу списка концов блоков.

 end {while};

 if список концов блоков пуст then

 Поместить m в голову этого списка.

 else begin {список концов блоков непуст}

 Пусть p_s обозначает первый элемент списка концов блоков.

 Используя двоичный поиск в клетках блока s ,

 найти самый правый элемент этого блока p , такой что

$Cand(j, p) > Cand(b_s, p)$.

```

    Добавить  $p$  в голову списка концов блоков.
end;
Добавить  $j$  в голову списка указателей  $b$ .
end {if};
end;
```

Временной анализ

Значение \bar{E} вычисляется для текущей клетки или когда алгоритм проводит одно из сравнений для поддержания текущего блочного разбиения. Следовательно, полное время на алгоритм пропорционально числу этих сравнений. На итерации j , когда клетка j является текущей, сравнения делятся на используемые для нахождения блока s и используемые в двоичном поиске для расщепления блока s . Если алгоритм делает $l > 2$ сравнений для нахождения s в итерации j , то не менее $l - 1$ полных блоков слипаются в единый блок. Затем двоичный поиск расщепляет не более одного блока на два. Таким образом, если на итерации j алгоритм делает $l > 2$ сравнений для нахождения s , то полное число блоков убывает не меньше чем на $l - 2$. Когда он делает одно или два сравнения, то полное число блоков увеличивается не больше чем на 1. Так как алгоритм начинает с одного блока и делает m итераций, то за всю работу алгоритма может быть сделано не больше $O(m)$ сравнений для нахождения s , не считая сравнений во время двоичного поиска. Ясно, что полное число сравнений, используемых в m двоичных поисках, равно $O(m \log m)$. Следовательно:

Теорема 12.6.1. Для любой фиксированной строчки все значения $E(j)$ можно вычислить за полное время $O(m \log m)$.

Случай значений F в основном симметричен

Сходный алгоритм и анализ используются для вычисления значений F , за исключением того, что для $F(i, j)$ списки разбивают столбец j из клеток от i до n . Есть, однако, один момент, который может вызвать путаницу: хотя анализ для F рассматривает один столбец и симметричен анализу для E в отдельной строчке, вычисления E и F на самом деле перемежаются, так как в рекуррентных соотношениях каждое значение $V(i, j)$ зависит и от $E(i, j)$, и от $F(i, j)$. Даром что и E и F вычисляются по строчкам (так как по строчкам вычисляется V), одна строчка после другой, $E(i, j)$ вычисляется сразу после $E(i, j + 1)$, тогда как между вычислениями $F(i, j)$ и $F(i + 1, j)$ нужно вычислить $m - 1$ других значений F ($m - j$ в строчке i и $j - 1$ в строчке $i + 1$). Итак, хотя в анализе считается, что работа в столбце проходит в сплошном интервале времени, на самом деле работу над каждым столбцом алгоритм выполняет с перерывами.*)

Для вычисления значений G и всех $V(i, j)$ нужно только время $O(nm)$, после того как известны $E(i, j)$ и $F(i, j)$. В итоге:

Теорема 12.6.2. Когда вес пропуска w является вогнутой функцией его длины, оптимальное выравнивание можно вычислить за время $O(nm \log m)$, где n и $m > n$ — длины этих двух строк.

*) Эти перерывы, конечно, сказываются на затратах памяти — алгоритм должен хранить списки по всем рассматриваемым столбцам. — Прим. перев.

12.7. Метод “четырех русских”

В этом пункте мы рассмотрим подход, который приводит и к теоретическому и к практическому ускорению во многих алгоритмах ДП. Замысел появился из статьи четырех авторов — В.Л. Арлазарова, Е.А. Диница, М.А. Кронрада и И.А. Фараджева [28] — относительно умножения булевых матриц. Эта общая идея известна на Западе как техника “четырех русских”, несмотря на то что русский из этих авторов только один.*¹) Приложения в области строк сильно отличаются от матричного умножения, но общая идея из [28] применима. Мы продемонстрируем ее на конкретной задаче вычисления (невзвешенного) редакционного расстояния. Это приложение, разработанное Мазеком и Патерсоном [313], было первым и рассматривалось теми же авторами еще в [312]; с тех пор найдено много приложений идеи “четырех русских” (например, [340]).

12.7.1. t -блоки

Определение. Назовем t -блоком квадрат $t \times t$ в таблице динамического программирования.

Грубо говоря, идея метода заключается в разбиении таблицы ДП на t -блоки и вычислении существенных значений в таблице одного t -блока за раз, а не одной клетки за раз. Цель в том, чтобы тратить на расчет одного блока только время $O(t)$ (вместо $\Theta(t^2)$), добиваясь ускорения в t раз по сравнению со стандартным ДП. В нашем изложении мы не достигнем в точности нужного разложения, так как соседние t -блоки будут перекрываться. Кроме того, приведенная идея не содержит главной мысли и преимущества излагаемого ниже метода. Он будет вычислять редакционное расстояние для двух строк длины n за время $O(n^2/\log n)$ (снова в предположении фиксированного алфавита).

Рассмотрим стандартный подход ДП к вычислению редакционного расстояния двух строк S_1 и S_2 . Значение $D(i, j)$ в клетке (i, j) , когда i и j больше 0, определяется значениями трех соседних клеток: $(i - 1, j - 1)$, $(i - 1, j)$ и $(i, j - 1)$ — и символами в позициях i и j этих строк. Продолжая, можно сказать, что значения в клетках всего t -блока с левым верхним углом в позиции, скажем, (i, j) определяются значениями первой строчки и первого столбца этого t -блока и подстроками $S_1[i..i + t - 1]$ и $S_2[j..j + t - 1]$ (рис. 12.21). Это наблюдение можно сформулировать иначе:

Лемма 12.7.1. *Значения расстояний в t -блоке, начинающемся в позиции (i, j) , являются функцией от значений в его первой строчке и первом столбце и от подстрок $S_1[i..i + t - 1]$ и $S_2[j..j + t - 1]$.*

Определение. Учитывая лемму 12.7.1 и используя обозначения рис. 12.21, определим блоковую функцию как функцию от пяти аргументов (A, B, C, D, E) с результатом F .

Из определения следует, что значения в последних строчке и столбце t -блока также являются функцией аргументов (A, B, C, D, E) . Мы назовем эту функцию ограниченной блоковой функцией.

*¹) Это отражает наш общий уровень невежества по части этнических вопросов в тогдашнем Советском Союзе. (Я не могу сказать, кто именно из этих москвичей этнический русский. — Прим. перев.)

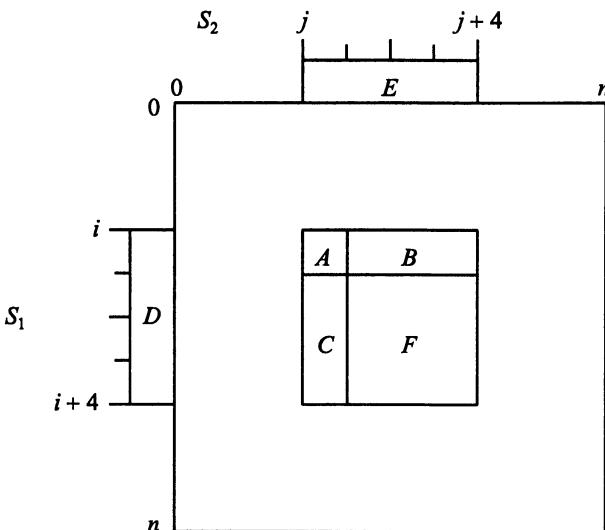


Рис. 12.21. Один блок с $t = 4$ изображен внутри полной таблицы ДП. Значения расстояния в части блока F определяются значениями в частях A , B и C , а также подстроками S_1 и S_2 в D и E . Отметим, что A является пересечением первой строчки и первого столбца блока

Отметим, что полный размер исходных данных и результата ограниченной блоковой функции имеют порядок $O(t)$.

Вычисление редакционного расстояния с помощью ограниченной блоковой функции
По лемме 12.7.1 редакционное расстояние между S_1 и S_2 можно вычислить, используя ограниченную блоковую функцию. Для простоты допустим, что обе строки S_1 и S_2 имеют длину $n = k(t - 1)$ с каким-то целым k .

Блочный алгоритм вычисления редакционного расстояния

begin

1. Покрыть таблицу ДП размера $(n + 1) \times (n + 1)$ t -блоками так, чтобы последний столбец каждого блока был первым столбцом его правого соседа (если он существует), а последняя строчка была первой строчкой блока внизу (с той же оговоркой) (рис. 12.22). При таком заполнении ввиду $n = k(t - 1)$ таблица будет состоять из k строчек и k столбцов из частично находящих друг на друга t -блоков.
2. Инициализировать значения начальной строчки и начального столбца полной таблицы согласно базовым условиям рекуррентных соотношений.
3. Проходя таблицу по строчкам и используя ограниченную блоковую функцию, определить последовательно значения в последних строчке и столбце каждого блока. Благодаря перекрытиям этих блоков значения в последнем столбце (или строчке) блока будут значениями первого столбца (строчки) блока справа (или вниз) от него.

4. Значение в клетке (n, n) дает редакционное расстояние между S_1 и S_2 .
end.

Конечно, самое важное в алгоритме — это шаг 3, где должны выполняться конкретные вызовы ограниченной блоковой функции. Любой вызов можно выполнить за время $O(t^2)$, но выигрыша мы не получим. Как выполнить этот вызов получше?

Рис. 12.22. Таблица редакционного расстояния для $n = 9$. При $t = 4$ таблица покрывается девятью налагающими друг на друга блоками. Центральный блок выделен жирными линиями. В общем, если $n = k(t - 1)$, то таблица $(n + 1) \times (n + 1)$ будет покрыта k^2 перекрывающимися t -блоками

12.7.2. Идея “четырех русских” для ограниченной блоковой функции

Общее наблюдение “четырех русских” состоит в том, что возможно получить ускорение за счет *предварительного вычисления* и сохранения информации о всех вариантах вызова подзадачи, которые могут появиться при решении задачи. Тогда при решении полной задачи и появлении конкретных подзадач вычисление ускоряется благодаря тому, что можно прямо посмотреть в ответ. Если подзадачи выбраны правильно, то полное время, расходуемое этим методом (включая время на предварительные вычисления), будет меньше, чем при стандартном способе.

В случае редакционного расстояния предварительное вычисление, предлагаемое идеей “четырех русских”, — это перечисление всех возможных значений аргументов для вызовов ограниченной блоковой функции (подходящий размер блока будет определен позднее), вычисление результата (строчка и столбец длины t) для каждого набора параметров и сохранение наборов “аргументы–результат”. Каждый раз, когда нужно будет в шаге 3 считать ограниченную блоковую функцию для конкретных значений, результат будет извлекаться из вычисленных заранее. Ясно, что этот подход работает при вычислении редакционного расстояния $D(n, n)$, но будет ли он быстрее, чем исходный метод со временем $O(n^2)$? Проницательный читатель должен сохранять свой скепсис и продолжать быть бдительным.

Счет затрат

Сначала предположим, что предварительные вычисления уже сделаны. Какое время нужно для выполнения блочного алгоритма для редакционного расстояния? Напомним, что размеры и входных и выходных данных у ограниченной блоковой функции имеют порядок $O(t)$. Нетрудно организовать данные о вычисленных заранее значениях этой функции таким образом, чтобы извлечение результата по входным параметрам занимало время $O(t)$. Подробности оставляются читателю. У нас есть $\Theta(n^2/t^2)$ блоков, следовательно, полное время, используемое на эти вызовы алгоритмом, равно $O(n^2/t)$. Считая t равным $\Theta(\log n)$, мы получим время $O(n^2/\log n)$. Однако в модели вычислений с единичной ценой каждое значение результата может быть достигнуто за константное время, так как $t = O(\log n)$. В этом случае время работы уменьшается до $O(n^2/(\log n)^2)$.

А как обстоит дело с временем предварительной подготовки? Здесь главный вопрос — количество вариантов исходных данных для ограниченной блоковой функции. По определению каждая клетка содержит целое число от 0 до n , так что любая строкка или столбец длины t может принимать $(n+1)^t$ значений. Если алфавит имеет размер σ , то существует σ^t возможных подстрок длины t . Следовательно, число различных комбинаций исходных данных равно $(n+1)^{2t}\sigma^{2t}$. Для каждого набора аргументов требуется время $\Theta(t^2)$ на вычисление последних строчки и столбца t -блока (с помощью стандартного ДП). Таким образом, общее время для предварительного вычисления результатов при всех сочетаниях параметров равно $\Theta((n+1)^{2t}\sigma^{2t}t^2)$. Но t не меньше единицы, так что $\Omega(n^2)$ в этой оценке уже есть. Не получилось! Идея хороша, но нужно еще что-то придумать.

12.7.3. Новый прием: кодирование смещений

Главный член в оценке времени предварительной подготовки — это $(n+1)^{2t}$, так как размер σ предполагается фиксированным. Этот член появляется от числа различных наборов значений в подстрочках и подстолбцах длины t . Но оценка $(n+1)^t$ явно завышена, реальное число наборов значений в векторе длины t существенно меньше, так как значение в каждой клетке сильно зависит от соседних. Уточним это обстоятельство.

Лемма 12.7.2. *В любой строчке, столбце или диагонали таблицы ДП для редакционного расстояния значения в двух смежных клетках отличаются не больше чем на 1.*

Доказательство. Ясно, что $D(i, j) \leq D(i, j-1) + 1$. Обратно, если при оптимальном выравнивании $S_1[1..i]$ и $S_2[1..j]$ происходит совпадение $S_2(j)$ с некоторым символом S_1 , то, убрав $S_2(j)$, выровняв его напарника напротив пробела и увидим, что расстояние увеличилось не больше чем на 1. Если символ $S_2(j)$ ни с чем не совпал, то, убрав его, уменьшаем расстояние на 1. Следовательно, $D(i, j-1) \leq D(i, j) + 1$, и лемма доказана для смежных клеток в строчке. Аналогичные рассуждения применимы и к столбцам.

Для смежных клеток по диагонали легко видеть, что $D(i, j) \leq D(i-1, j-1) + 1$. Обратно, если при оптимальном выравнивании $S_1[1..i]$ и $S_2[1..j]$ индекс i встал напротив j , то $D(i-1, j-1) \leq D(i, j) + 1$. Если же в оптимальном выравнивании они не встали напротив друг друга, то по меньшей мере один из символов, $S_1(i)$ или $S_2(j)$, должен встать напротив пробела и $D(i-1, j-1) \leq D(i, j)$. \square

Имея лемму 12.7.2, мы можем кодировать значения в строчке t -блока вектором длины t , указывающим значение первого элемента строчки, и затем разностями каждого значения с предыдущим (смещениями): 0 означает равенство значений, 1 указывает на увеличение, а -1 на уменьшение. Например, строчка расстояний $5, 4, 4, 5$ может быть закодирована строкой смещений $5, -1, 0, +1$. Точно так же мы можем кодировать и столбцы. Так как теперь осталось только $(n+1)3^{t-1}$ различных векторов, новая кодировка — это шаг в правильном направлении. Однако мы еще уменьшим число возможных векторов.

Определение. Вектором смещений назовем вектор длины t со значениями $\{-1, 0, 1\}$ и нулевым первым элементом.

Основой повышения эффективности метода “четырех русских” будет вычисление редакционного расстояния для векторов смещений, а не для действительных значений расстояния. Так как число возможных векторов смещений много меньше, чем число возможных векторов расстояний, потребуется значительно меньше предварительных вычислений. Покажем вычисление редакционного расстояния по векторам смещений.

Теорема 12.7.1. Рассмотрим t -блок с верхним левым углом в позиции (i, j) . Два вектора смещений для последней строчки и последнего столбца блока можно определить по двум векторам смещений для первых строчки и столбца этого блока и из подстрок $S_1[i..i+t-1]$ и $S_2[j..j+t-1]$. Значит, ни одного значения D для определения этих векторов смещений не потребуется.

Доказательство. Доказательство в основном состоит из внимательного изучения соотношений ДП для редакционного расстояния. Обозначим неизвестное значение $D(i, j)$ через C . Тогда для столбца блока q значение $D(i, q)$ равно C плюс сумма смещений в строчке i от столбца $j+1$ до столбца q . Следовательно, даже если алгоритм не знает значения C , он может выразить $D(i, q)$ как C плюс целое, которое легко определить. Так же можно выразить и любое $D(q, j)$. Пусть $D(i, j+1) = C + J$ и $D(i+1, j) = C + I$, где I и J — слагаемые, которые алгоритм может найти. Рассмотрим клетку $(i+1, j+1)$. $D(i+1, j+1)$ равно $D(i, j) = C$, если символ $S_1(i)$ совпадает с $S_2(j)$. В противном случае $D(i+1, j+1)$ равно минимуму из $D(i, j+1) + 1$, $D(i+1, j) + 1$ и $D(i, j) + 1$, т. е. минимуму из $C + I + 1$, $C + J + 1$ и $C + 1$. Алгоритм может выполнить это сравнение, сопоставляя I и J (которые известны) с числом 0. Итак, алгоритм может правильно выразить $D(i+1, j+1)$ как C , $C + I + 1$, $C + J + 1$ или $C + 1$. Продолжая в том же роде, алгоритм может правильно выразить каждое значение D в блоке как неизвестное C плюс некоторая целочисленная поправка, которую он может определить. Так как каждый член в соотношении имеет такой же вид, алгоритм правильно определит и искомые векторы смещений. \square

Определение. Функция, которая определяет два вектора смещений для последних строчки и столбца по векторам смещений для первой строчки и первого столбца блока и по подстрокам $S_1[i..i+t-1]$ и $S_2[j..j+t-1]$, называется функцией смещения.

Мы теперь имеем все, что нужно для сборки алгоритма в стиле “четырех русских” для вычисления редакционного расстояния. Снова предполагаем для простоты, что каждая строка имеет длину $n = k(t-1)$ для некоторого k .

Алгоритм редакционного расстояния в стиле “четырех русских”

- Покрыть t -блоками таблицу ДП размером $n \times n$ таким образом, чтобы последний столбец каждого t -блока был первым столбцом t -блока справа от него, а последняя строчка — первой в t -блоке снизу (если такие существуют).
- Вычислить значения в первой строчке и первом столбце полной таблицы по базовым условиям рекуррентного соотношения. Вычислить значения смещений в первой строчке и первом столбце.
- Используя блочную функцию *смещения*, определить по строкам векторы смещений последней строчки и последнего столбца каждого блока. Ввиду перекрытия блоков вектор смещений в последнем столбце (или строчке) блока задаст вектор смещений в первом столбце (строчке) следующего блока. Нужно заменить первый элемент получающегося вектора нулем.
- Пусть Q — сумма смещений, вычисленных в строчке n . Тогда $D(n, n) = D(n, 0) + Q = n + Q$.

Временной анализ

Как при анализе блокового нахождения *редакционных расстояний*, выполнение *алгоритма в стиле “четырех русских”* требует времени $O(n^2/\log n)$ (или $O(n^2/\log^2 n)$ в модели с единичной ценой), если принять t равным $\Theta(\log n)$. Итак, снова главный вопрос — это время, необходимое для предварительного вычисления блоковой функции смещения. Напомним, что первый элемент вектора смещений должен быть нулем, так что есть $3^{2(t-1)}$ возможных векторов смещений. Существует σ^t способов задания подстроки над алфавитом размера σ , так что всего получается $3^{2(t-1)}\sigma^{2t}$ вариантов исходных данных для функции смещения. Для любого конкретного их набора результат достигается за время $O(t^2)$ (средствами динамического программирования), следовательно, все предварительное вычисление занимает время $O(3^{2t}\sigma^{2t}t^2)$. Полагая $t = (\log_3 n)/2$, получаем время вычислений всего $O(n \log^2 n)$. В результате:

Теорема 12.7.2. *Редакционное расстояние между двумя строками длины n можно вычислить за время $O(n^2/\log n)$ или $O(n^2/\log^2 n)$ в случае модели вычислений с единичной ценой.*

Обобщение на строки неравной длины делается просто и оставляется как упражнение.

12.7.4. Практический подход

Теоретический результат о возможности вычисления редакционного расстояния за время $O(n^2/\log n)$ был обобщен и применен к ряду других задач выравнивания. Однако для действительно больших строк теоретические результаты навряд ли применимы. И метод “четырех русских” является большей частью теоретическим достижением и не применяется непосредственно. Используется основная идея предварительного вычисления либо ограниченной блочной функции, либо функции смещения, но только для блоков *фиксированного* размера. Обычно t имеет фиксированное значение, не зависящее от n , и часто вместо квадратного блока вводится прямоугольный — $2 \times t$. Нужно только выбрать такое t , чтобы ограниченная блочная функция или функция смещения могла быть определена за константное время

на реальных машинах. Например, t можно выбрать так, чтобы вектор смещений укладывался в одно машинное слово. Или, в зависимости от алфавита и доступного размера памяти, можно хешировать наборы входных параметров для быстрого поиска значений функции. Этот подход может привести к вычислительному времени $O(n^2/t)$, хотя на данном уровне детализации становятся важными детали практического программирования. Подробный экспериментальный анализ этих идей [339] показал, что такой подход является одним из наиболее эффективных, обеспечивая ускорение вычислений в t раз по сравнению со стандартным ДП.

12.8. Упражнения

1. Покажите, как вычислить значение оптимального выравнивания $V(n, m)$, используя только память $\min\{n, m\} + 1$, не считая той, которая расходуется на хранение исходных строк.
2. Модифицируйте метод Хиршберга для поиска выравнивания со штрафом за пропуски (аффинным и общим) в целевой функции. Может быть полезно использовать и рекуррентные соотношения для аффинных пропусков, предложенные в тексте, и альтернативные соотношения, в которых за пропуск платят, когда он заканчивается. Эти последние соотношения были предложены в упражнении 27 главы 11.
3. Метод Хиршберга вычисляет одно оптимальное выравнивание. Попробуйте найти, как его модифицировать, чтобы создать возможно больше (а может быть, и все) оптимальных выравниваний по-прежнему со значительной экономией памяти и сохранением хорошей оценки времени по сравнению с обычным методом, где и время и память имеют оценку $O(nm)$. По видимому, это открытая тема.
4. Покажите, как уменьшить требуемый в методе из п. 12.2.3 размер полосы, когда $|m - n| < k$.
5. Восполните детали поиска выравниваний P с T , имеющих не более k различий. Метод использует $O(km)$ значений, сохраняемых при алгоритме для k различий. Решение получается немного проще, если алгоритм k различий сохраняет также разреженное множество указателей, записывающих, как каждый самый продвинутый d -путь продолжает самый продвинутый $(d - 1)$ -путь. Указатели занимают только $O(km)$ памяти и являются разреженным вариантом указателей для стандартного ДП. Восполните детали и для этого подхода.
6. Задача о k различиях является задачей невзвешенного (или взвешенного с единичными весами) выравнивания, определенной в терминах числа несовпадений и пробелов. Можно ли результат $O(km)$ распространить на операторно- или алфавитно-взвешенные варианты выравнивания? Ответ: не полностью. Объясните, почему. Затем найдите частные случаи взвешенного выравнивания и правдоподобные использования этих случаев, когда результат распространяется на указанные выравнивания.
7. Докажите лемму 12.3.2 со с. 337.
8. Докажите лемму 12.3.4 со с. 341.
9. Докажите теорему 12.4.2 относительно использования памяти в задаче P -против-всех.
10. Пороговая задача P -против-всех. Задача P -против-всех была рассмотрена сначала потому, что она самым непосредственным образом иллюстрировала общий подход к использованию суффиксных деревьев для ускорения вычислений в ДП. Было высказано предположение, что такое массовое изучение соотношения P с подстроками T может быть важно для некоторых задач [183]. Тем не менее для большинства приложений результат, получаемый

в задаче P -против-всех, чрезвычайно велик, и желательно вычисление более узкой направленности. Такой характер носит *пороговая задача P -против-всех*. Пусть заданы строки P и T и порог d . Найти все подстроки T' строки T , редакционное расстояние которых от P меньше d . Конечно, было бы нечестно решить сначала задачу P -против-всех, а затем отфильтровать подстроки T , у которых редакционное расстояние до P не меньше d . Мы хотим получить метод, скорость которого связана с d . Вычисления должны с уменьшением d проходить быстрее.

Идея заключается в следовании решению задачи P -против-всех обходом в глубину суффиксного дерева T , но с распознанием поддеревьев, которые обходить не надо. Основу дает следующая лемма.

Лемма 12.8.1. В задаче P -против-всех пусть текущий путь в суффиксном дереве определяет подстроку S строки T и текущий столбец таблицы ДП (включая строку 0) не содержит значений меньше d . Тогда и столбец, представляющий расширение S , не будет содержать значений, меньших d . Следовательно, ни один столбец не должен вычисляться для продолжений S .

Докажите лемму и затем покажите, как использовать ее при решении пороговой задачи P -против-всех. Постарайтесь оценить, насколько она эффективна на практике. Не забудьте рассмотреть, насколько эффективно собираются результаты, когда процесс ДП прерывается высоко на дереве, до достижения листа.

11. Дайте полное доказательство корректности алгоритма суффиксного дерева для задачи все-против-всех.
12. Другая ускоряющая решение альтернатива задаче P -против-всех получается, если изменить постановку задачи следующим образом: для каждой позиции i из T , такой что из нее начинается подстрока с редакционным расстоянием от P , меньшим, чем d , находить только *наименьшую* из подстрок, начинающихся в этой позиции. Это будет задача о (P -против-всех) *начальных позициях*, ее можно решить, модифицируя подход к пороговой задаче P -против-всех. Задача о начальных позициях (в действительности, эквивалентная задача о конечных позициях) была темой статьи Укконена [437]. В этой статье он предложил три метода гибридного динамического программирования в том же духе, как представленные в этой главе, но с дополнительными техническими улучшениями. Этот основной результат работы был в дальнейшем усилен Коббом [105].
Детализируйте решение задачи о начальных позициях с помощью гибридного ДП.
13. Покажите, что методы суффиксного дерева и временные оценки для задач P -против-всех и все-против-всех переносятся на задачу вычисления сходства вместо редакционного расстояния.
14. Пусть R — регулярное выражение. Покажите, как модифицировать метод решения задачи P -против-всех, чтобы он решал задачу R -против-всех. То есть покажите, как использовать суффиксное дерево для эффективного поиска в большом тексте T подстроки, совпадающей с регулярным выражением R (этот задача взята из [63]).
Теперь усовершенствуйте этот метод, чтобы он допускал ограниченное число ошибок сравнения.
15. Закончите доказательство теоремы 12.5.2.
16. Покажите, что в любой перестановке из n чисел от 1 до n существует либо возрастающая подпоследовательность длины не меньше \sqrt{n} , либо убывающая подпоследовательность длины не меньше \sqrt{n} . Продемонстрируйте, что при усреднении по всем $n!$ перестановкам средняя длина наибольшей возрастающей подпоследовательности будет не меньше, чем $\sqrt{n}/2$. Убедитесь, что нижняя граница $\sqrt{n}/2$ не может достигаться.

17. Какие из результатов предыдущей задачи применимы к задаче *lcs*?
 18. Если S является подпоследовательностью другой строки S' , то S' называется *надпоследовательностью* S . Если две строки S_1 и S_2 являются подпоследовательностями S' , то S' является их *общей надпоследовательностью*. Это определение порождает естественный вопрос: какова *кратчайшая* общая надпоследовательность двух заданных строк S_1 и S_2 ? Эта задача, очевидно, связана с задачей о наибольшей общей подпоследовательности. Покажите явно связь между этими двумя задачами и между длинами их решений. Предложите эффективные методы нахождения кратчайшей общей надпоследовательности двух строк. Дополнительно о подпоследовательностях и надпоследовательностях см. [240, 241].
 19. Можно ли обобщить результаты предыдущей задачи на случай более чем двух строк? Например, есть ли естественное соотношение между наибольшей общей подпоследовательностью и кратчайшей общей надпоследовательностью трех строк?
 20. Пусть T — строка, символы которой берутся из алфавита Σ размера σ . Подпоследовательность S строки T неубывающая, если каждый следующий символ в ней лексикографически не меньше предыдущего. Например, в латинском алфавите если $T = \text{characterstring}$, то $S = aacrst$ является неубывающей подпоследовательностью T . Предложите алгоритм, который находит наибольшую неубывающую подпоследовательность строки T за время $O(n\sigma)$, где n — длина T . Как эта граница сопоставляется с границей $O(n \log n)$, полученной для задачи о наибольшей возрастающей подпоследовательности чисел?
 21. Вспомним определение r , данное для двух строк в п. 12.5.2 на с. 355. Распространите его на задачу о наибольшей общей подпоследовательности для более чем двух строк и используйте r для выражения времени нахождения *lcs* этих строк.
 22. Покажите, как использовать задачу о кратчайшем пути в ациклическом графе для моделирования и решения задачи *lis*. Есть ли какие-нибудь преимущества в таком описании этой задачи?
 23. Предположим, что мы хотим узнать только длину *lcs* двух строк, S_1 и S_2 . Это можно сделать, как раньше, за время $O(r \log n)$, но теперь уже ограничиваясь линейной памятью. Идея в том, чтобы сохранять в каждом списке покрытия (при вычислении *lis*) только последний элемент и не генерировать все $\Pi(S_1, S_2)$ сразу, а только (в линейной памяти) нужные в каждый момент части $\Pi(S_1, S_2)$. Восполните подробности этого подхода и покажите, что длину *lcs* можно вычислить так же быстро, как раньше, ограничиваясь линейной памятью.
- Открытая задача.** Разработайте предложенные комбинаторные идеи и покажите, как вычислить *lcs* двух строк, используя только линейную память без увеличения требуемого времени. Затем обобщите этот метод на случай более двух строк.
24. Эта задача требует знания систолических массивов.*^{*)} Покажите, как реализовать алгоритм поиска наибольшей возрастающей подпоследовательности, чтобы искать ее за время $O(n)$ в систолическом массиве с $O(n)$ элементами (помните, что каждый элемент занимает константную память). Чтобы сделать задачу проще, подумайте сначала, как вычислить длину *lis*, а уже затем — как вычислить саму возрастающую подпоследовательность.
 25. Разработайте метод вычисления *lcs* за время $O(n)$ в систолическом массиве размером $O(n)$.
 26. Мы свели задачу *lcs* к задаче *lis*. Покажите, как сделать обратное сведение.

^{)} Систолические массивы — это понятие из области параллельных вычислений. — Прим. перев.

27. Предположим, что каждому символу в S_1 и S_2 сопоставлен индивидуальный вес. Предложите алгоритм отыскания возрастающей подпоследовательности максимального общего веса.
28. Разработайте метод вычисления редакционного расстояния для модели вогнутых штрафов за пропуски со временем $O(nm \log m)$.
29. Идея прямого динамического программирования может использоваться для (практического) ускорения нахождения (глобального) выравнивания двух строк, даже когда пропуски не учитываются целевой функцией. Мы объясним это в терминах вычисления невзвешенного редакционного расстояния между строками S_1 и S_2 (с длинами n и m соответственно), но основная идея работоспособна и в случае вычисления сходства. Пусть во время вычисления редакционного расстояния (прямым) ДП достигнута клетка (i, j) и значение в ней равно $D(i, j)$. Предположим также, что существует быстрый способ вычисления нижней границы, $L(i, j)$, для расстояния между подстроками $S_1[i+1, \dots, n]$ и $S_2[j+1, \dots, m]$. Если $D(i, j) + L(i, j)$ не меньше известного расстояния между S_1 и S_2 , полученного при каком-либо конкретном выравнивании, то нет необходимости засыпать кандидатов на значения из клетки (i, j) . Вопрос в том, чтобы найти эффективный метод вычисления “эффективных” значений $L(i, j)$. Один простой способ — это величина $|n - m + j - i|$. Объясните это. Постарайтесь практически оценить его эффективность. Попробуйте другие простые границы, которые были бы более эффективны.

Совет. Используйте счетчик появлений каждого символа в каждой строке.

30. Как рассказывалось в тексте, метод “четырех русских” предварительно вычисляет функцию смещения для $3^{2(t-1)}\sigma^{2t}$ вариантов набора аргументов. Однако постановка задачи и временная оценка позволяют делать предварительные вычисления после того, как строки S_1 и S_2 станут известны. Можно ли использовать это замечание для сокращения времени счета?
- Альтернативная кодировка строк позволяет заменить множитель σ^{2t} на $(t+2)^t$ даже в постановке задачи, когда строки S_1 и S_2 при выполнении предварительных вычислений еще не известны. Придумайте и объясните способ кодировки и то, как вычислять редакционное расстояние, используя эту кодировку.
31. Рассмотрим ситуацию, когда редакционное расстояние нужно вычислять для каждой пары из большого набора строк. В этом случае предварительные вычисления, требуемые для метода “четырех русских”, кажутся более обоснованными. На самом деле: почему бы не выбрать “разумное” значение для t , не вычислить для него функцию смещения и не включить затем эту функцию смещения в алгоритм нахождения редакционного расстояния для всех будущих вычислений? Обсудите достоинства и недостатки этого подхода.
32. Метод “четырех русских”, представленный в тексте, вычисляет только редакционное расстояние. Как его модифицировать, чтобы вычислять и редакционное предписание?
33. Покажите, как применить алгоритм “четырех русских” к строкам неравной длины.
34. Какие трудности возникают при попытках перенести метод “четырех русских” и улучшенную границу времени на задачу *взвешенного* редакционного расстояния? Есть ли ограничения на веса (отличные от равенства), которые упрощают перенос?
35. Следуя курсу предыдущего вопроса, покажите в деталях, как применить подход “четырех русских” к решению задачи о наибольшей общей подпоследовательности двух строк длины n за время $O(n^2 / \log n)$.

Глава 13

Развитие основных задач

В этой главе подробно рассмотрим задачи выравнивания в более сложных ситуациях, обычных для строковых задач, которые постоянно встречаются в вычислительной молекулярной биологии. Такие задачи требуют техники, которая скорее расширяет, чем уточняет базовые методы выравнивания.

13.1. Параметрическое выравнивание последовательностей

13.1.1. Введение

При использовании методов выравнивания для изучения последовательностей ДНК или аминокислот часто обнаруживается решительное несогласие относительно выбора весов для совпадений, несовпадений, вставок и удалений (инделов — indel) и пропусков. Самые популярные пакеты, вычисляющие выравнивания, требуют от пользователя задания фиксированных значений этих параметров, и повсеместно обнаруживается, что биологическая значимость получающихся выравниваний в значительной мере определяется выбором установок параметров. Следующая цитата относится к выравниваниям белков из семейства глобинов; она очень характерна для комментариев, часто встречающихся в биологической литературе:

“...нужно иметь возможность менять штрафы за пропуск и размер пропуска независимо, и в каждом запросе отдельно, чтобы поиск был максимально чувствителен” [81].

Похожий комментарий:

“Выравнивание последовательностей чувствительно к выбору штрафов за пропуск и к форме зависимости. Зачастую желательно менять и то и другое...” [432].

На конец:

“Одна из наиболее значительных проблем — выбор значений параметров, особенно штрафов за пропуски. Когда сравниваются очень похожие последовательности, этот выбор не важен; но когда сохраняемость мала, получающиеся выравнивания сильно зависят от параметров” [446].

Параметрическое выравнивание последовательностей — средство, которое эффективно использует варьирование штрафов. Оно обходит проблему выбора фиксированных значений параметров, вычисляя оптимальное выравнивание как функцию от переменных параметров для весов и штрафов. Целью является разбиение пространства параметров на такие участки (как мы покажем, обязательно выпуклые), что на каждом из них какое-то одно выравнивание всюду оптимально, и обладающий этим свойством участок максимален. Таким образом, параметрическое выравнивание позволяет увидеть явно и полностью влияние выбора параметров на оптимальное выравнивание. Параметрическое выравнивание последовательностей впервые было применено в работе Фитча и Смита [161], чтобы показать, что целевая функция для выравнивания, не включающая явного члена для пропусков, не дает биологически корректных выравниваний (для последовательностей из цепей гемоглобина) при любой комбинации устанавливаемых параметров.

13.1.2. Определения и первые результаты

Задачи параметрического выравнивания встречаются и при использовании алфавитно-зависимых оценочных матриц (матрицы замены аминокислот, такие как PAM или BLOSUM), и без их использования. Поскольку изучение этих случаев несколько отличается, будем рассматривать их отдельно. Начнем с ситуации, когда оценочная матрица не используется.

Определение. Для любого выравнивания \mathcal{A} двух строк пусть $mt_{\mathcal{A}}$, $ms_{\mathcal{A}}$, $id_{\mathcal{A}}$ и $gp_{\mathcal{A}}$ обозначают, соответственно, число совпадений, несовпадений, инделов и пропусков, содержащихся в \mathcal{A} .

Если не использовать оценочных матриц, зависящих от символов, то значение выравнивания \mathcal{A} равно

$$v_{\mathcal{A}}(\alpha, \beta, \gamma) \equiv \alpha \times mt_{\mathcal{A}} - \beta \times ms_{\mathcal{A}} - \gamma \times id_{\mathcal{A}} - \delta \times gp_{\mathcal{A}},$$

где α , β , γ и δ — параметры, которые можно менять, чтобы по-разному отражать относительные вклады в целевую функцию совпадений, несовпадений, инделов и пропусков. Отметим, что значение выравнивания является линейной функцией этих четырех параметров. Когда параметры имеют фиксированные значения α_0 , β_0 , γ_0 и δ_0 , получается стандартная задача с *фиксированными параметрами*, в которой ищется выравнивание \mathcal{A} , максимизирующее целевую функцию: $\alpha_0 \times mt_{\mathcal{A}} - \beta_0 \times ms_{\mathcal{A}} - \gamma_0 \times id_{\mathcal{A}} - \delta_0 \times gp_{\mathcal{A}}$.

Параметрическое выравнивание изучает изменение оптимального выравнивания как функции от переменных параметров α , β , γ и δ . Нетрудно видеть, что на самом деле независимы только три параметра, а не четыре. Однако, поскольку трудно рисовать в трехмерном пространстве, мы сосредоточимся на параметрических задачах,

в которых изменяются только два параметра из четырех. Значения оставшихся параметров будут фиксированы. Обобщение результатов на случай трех переменных параметров оставляется читателю.

Пусть для иллюстрации параметры γ и δ изменяются, а α и β зафиксированы на значении 1. Такой выбор обычен для изучения выравнивания белков (хотя здесь принято использовать и оценочную матрицу). Если сопоставить γ и δ — две координатные оси, а по третьей (перпендикулярной) оси откладывать значение выравнивания, то \mathcal{A} как функция от γ и δ опишет *плоскость* в этом трехмерном пространстве. Две различные плоскости, если они не параллельны, пересекаются по прямой, отсюда следует:

Лемма 13.1.1. *Если плоскости для выравниваний \mathcal{A} и \mathcal{A}' пересекаются и различны, то в пространстве γ , δ существует прямая L , на которой значения \mathcal{A} и \mathcal{A}' одинаковы; значение \mathcal{A} больше, чем значение \mathcal{A}' , в одной из полуплоскостей, определенных L , и меньше на другой полуплоскости. Если плоскости для \mathcal{A} и \mathcal{A}' не пересекаются, то в каждой точке γ , δ значение одного из выравниваний больше, чем другого.*

Рассмотрим подмножество пространства γ , δ , в котором фиксированное выравнивание \mathcal{A} оптимально. Если \mathcal{A} оптимально в точке p , то, как отмечалось выше, p должна находиться в нужной полуплоскости относительно любого выравнивания \mathcal{A}' . Значит, \mathcal{A} оптимально в подмножестве этого пространства, определенном пересечением полуплоскостей. Следовательно, мы имеем

Следствие 13.1.1. *Если \mathcal{A} оптимально по крайней мере в одной точке p пространства γ , δ , то оно оптимально только в точке p , или только на сегменте прямой, содержащем p , или только в выпуклом многоугольнике, содержащем p .*

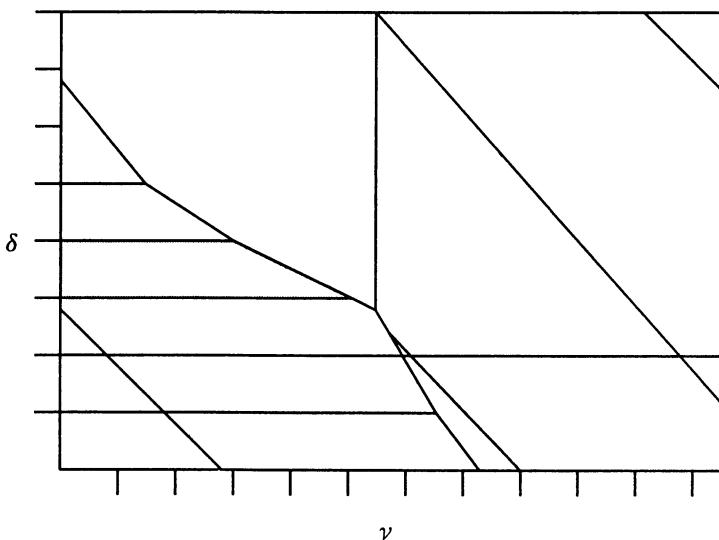


Рис. 13.1. Разбиение пространства γ , δ для двух не показанных строк на многоугольники. В программе XPARAL пользователь выбирает многоугольник (заштрихован), и программа показывает те выравнивания, которые оптимальны во всех его точках

Отсюда следует:

Теорема 13.1.1. Пусть заданы две строки, S_1 и S_2 . Пространство параметров γ, δ разбивается на выпуклые многоугольники так, что любое выравнивание, оптимальное в какой-либо внутренней точке γ, δ многоугольника \mathcal{P} , оптимально во всех точках \mathcal{P} и нигде более.

На рис. 13.1 показано такое многоугольное разбиение пространства параметров γ, δ . Программы параметрического выравнивания, такие как XPARAL [203, 205], или программа, разработанная Уотерменом и др. [447, 463], или программа, о которой идет речь в [486], позволяют задать две строки и выбрать два переменных параметра, а затем вычисляют и показывают получающееся многоугольное разбиение. После этого возможен подробный анализ выравниваний и соответствующих им многоугольников в интерактивном режиме.

13.1.3. Параметрические выравнивания с использованием оценочных матриц

Когда используется оценочная матрица, учитывающая символы (такая, как матрицы PAM и BLOSUM) и варьирующая оценки конкретных совпадений и несовпадений, параметрическая целевая функция несколько изменяется.

Определение. Для любого выравнивания \mathcal{A} двух строк обозначим, соответственно, через $smt_{\mathcal{A}}$ и $sms_{\mathcal{A}}$ полные оценки совпадений \mathcal{A} и несовпадений в \mathcal{A} (полученных по оценочной матрице). Как и раньше, $id_{\mathcal{A}}$ и $gp_{\mathcal{A}}$ обозначают числа инделов и пропусков в \mathcal{A} .

При использовании оценочных матриц параметрическое значение выравнивания \mathcal{A} равно $\alpha \times smt_{\mathcal{A}} + \beta \times sms_{\mathcal{A}} - \gamma \times id_{\mathcal{A}} - \delta \times gp_{\mathcal{A}}$. Слагаемое $\beta \times sms_{\mathcal{A}}$ добавляется к выражению для того, чтобы иметь возможность устанавливать подходящий знак.

Все результаты, полученные в п. 13.1.2, верны и при использовании оценочных матриц. Проверка утверждения оставляется читателю.

13.1.4. Эффективные алгоритмы вычисления многоугольной декомпозиции

Мы опишем сейчас эффективный алгоритм вычисления многоугольной декомпозиции, порождаемый двумя заданными строками, целевой функцией и выбором переменных параметров. Алгоритм представляется в общих терминах и не зависит от того, используется ли оценочная матрица. Для иллюстрации снова предположим, что изменяются параметры γ и δ , хотя тот же метод применим при любом выборе параметров. Следующая задача, называемая *задачей поиска по лучу*, решается во внутреннем цикле алгоритма декомпозиции.

Задача поиска по лучу. Пусть заданы выравнивание \mathcal{A} , точка p , где оно оптимально, и луч h в пространстве γ, δ , выходящий из p . Найти на луче точку, самую дальнюю от p (обозначим ее через r^*), где \mathcal{A} остается оптимальным. Если \mathcal{A} остается оптимальным вплоть до границы пространства параметров, то в качестве r^* принимается точка h на границе. Не исключается возможность $r^* = p$ (рис. 13.2).

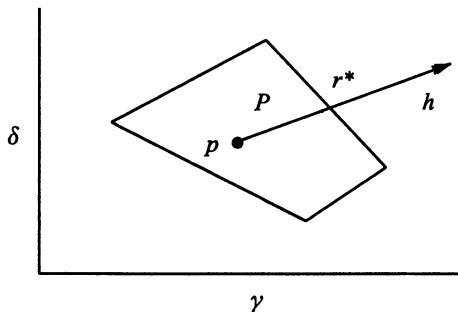


Рис. 13.2. Выравнивание \mathcal{A} оптимально в многоугольнике P . Начиная с точки p , \mathcal{A} остается оптимальным вдоль луча h до точки r^*

Эта задача решается следующим образом:

Ньютоновский алгоритм поиска по лучу

Положить r равным точке пространства (γ, δ) , где h пересекает границу пространства параметров.

while выравнивание \mathcal{A} не оптимально в точке r do begin
найти оптимальное выравнивание \mathcal{A}^* в точке r .

Положить r равным той единственной точке на h , где значение \mathcal{A} равно значению \mathcal{A}^* .

end;

Положить r^* равным r .

Алгоритм является классическим методом Ньютона для нахождения нуля в частном случае кусочно-линейной функции. В анализе потребуются следующие три факта (законы Ньютона). Их легко доказать, и это предоставляется читателю.

Лемма 13.1.2. 1) *Ньютоновский алгоритм поиска по лучу находит r^* точно.*
 2) *Если только \mathcal{A} не будет оптимально при начальном значении r , последнее вычисленное выравнивание \mathcal{A}^* будет кооптимально с \mathcal{A} в r^* и оптимально на h на некотором ненулевом расстоянии от r^* .* 3) *Когда алгоритм вычисляет выравнивание в точке r на h , ни одно из выравниваний, вычисленных ранее (в этом расчете), не будет оптимально в r .*

Отсюда следует, что если $r^* = p$, то метод обнаружит это и возвратит выравнивание \mathcal{A}^* , которое оптимально в p и на некотором ненулевом расстоянии вдоль h . Кроме того, видно, что для любого многоугольника P , пересеченного h , однократный поиск по лучу вычисляет выравнивания в не более чем двух точках P .

Нахождение многоугольника из декомпозиции

Пусть \mathcal{A} — выравнивание, которое оптимально в некоторой (известной) внутренней точке p (неизвестного) многоугольника $\mathcal{P}(\mathcal{A})$. Объясним, как найти многоугольник $\mathcal{P}(\mathcal{A})$.

Сначала выберем какой-либо луч h , выходящий из p , и решим задачу поиска по этому лучу. Могут встретиться два вырожденных случая: один — когда r^* лежит

на границе пространства параметров, а другой — когда r^* является вершиной декомпозиции. Эти случаи мы рассмотрим позднее, а сейчас предположим, что реализуется общая ситуация. Поэтому поиск на луче h даст точку r^* , которая лежит на стороне e многоугольника $\mathcal{P}(\mathcal{A})$. По второму закону Ньютона поиск на луче даст и выравнивание \mathcal{A}^* , которое оптимально внутри граничной стороны e . Пересечение двух плоскостей, соответствующих \mathcal{A} и \mathcal{A}^* , определяет линию l^* , которая содержит*) сторону e ; теперь полная протяженность стороны e может быть установлена решением еще двух задач поиска по лучу, использующих \mathcal{A} . Роль луча h в этих задачах играют две полупрямые, на которые точка r^* разбивает прямую l^* . Две задачи находят граничные точки стороны e . Полностью описав сторону e , мы ищем другую сторону $\mathcal{P}(\mathcal{A})$, выбирая другой луч h , выходящий из p и не пересекающий ребра e . Соединяя одинаковые концы полученных таким образом сторон, легко продолжить выбор лучей, начинающихся в p , которые не пересекают найденных ранее сторон или вершин $\mathcal{P}(\mathcal{A})$. Метод продолжается до тех пор, пока все найденные стороны не соединятся в замкнутый цикл, который и определит стороны и вершины $\mathcal{P}(\mathcal{A})$.

Теперь рассмотрим два упомянутых вырожденных случая. В случае если r^* находится на границе пространства параметров, то одна из сторон $\mathcal{P}(\mathcal{A})$ идет вдоль этой границы и может быть найдена с использованием этой граничной линии в качестве l^* . Во втором случае, когда r^* — вершина многоугольника, алгоритм распознает это, когда выполняет поиск вдоль луча l^* — выравнивание \mathcal{A} не будет оптимально после r^* по крайней мере на одном из лучей, идущих вдоль l^* из r^* . Если встретится такая ситуация, алгоритм останавливает текущий поиск по стороне и начинает поиск по новому лучу, выходящему из p , но обходящему r^* и все прочие найденные ранее вершины и стороны.

Заполнение пространства параметров

Чтобы вычислить многоугольную декомпозицию полностью, нужно сначала найти выравнивание, которое гарантированно оптимально внутри некоторого (неизвестного) многоугольника. Это легко сделать за константное число поисков по лучам. Например, мы можем выбрать точку p , найти оптимальное выравнивание \mathcal{A} в p и затем выполнить поиск по лучу h , выходящему из p . Если получившаяся точка r^* не совпадает с p , то \mathcal{A} оптимально для некоторого ненулевого расстояния вдоль h . Если $r^* = p$, поиск по лучу возвращает другое выравнивание \mathcal{A}^* , которое оптимально для некоторого ненулевого расстояния вдоль h . Предположим, что случилось первое. Тогда нужно определить, будет ли \mathcal{A} оптимально внутри многоугольника или только на его стороне вдоль h . Для этого выберем точку p^* внутри того сегмента h , где \mathcal{A} заведомо оптимально, и поиск по лучу, выходящему из p^* перпендикулярно h . Этот поиск либо подтвердит, что \mathcal{A} оптимально внутри многоугольника, либо предъявит выравнивание, которое оптимально внутри многоугольника. В любом случае мы получим первое выравнивание, которое можно использовать для построения первого многоугольника декомпозиции.

) Здесь есть небольшая неточность: плоскости находятся в трехмерном пространстве и их пересечение тоже, тогда как сторона e должна лежать в пространстве параметров. По-видимому, удобно считать, что линия l^ является проекцией пересечения упомянутых плоскостей на пространство параметров. — Прим. перев.

Теперь объясним, как алгоритм находит последовательные многоугольники, составляющие искомую декомпозицию. Найдя первый многоугольник $\mathcal{P}(\mathcal{A})$ (и затем каждый следующий), алгоритм включает в список L по одному вектору $(smt_{\mathcal{A}}, sms_{\mathcal{A}}, id_{\mathcal{A}}, gp_{\mathcal{A}})$ для каждого выравнивания \mathcal{A}^* , которое оказалось оптимальным внутри многоугольника, граничащего с $\mathcal{P}(\mathcal{A})$ (если такого вектора в списке еще нет). Когда построение $\mathcal{P}(\mathcal{A})$ завершается, алгоритм находит и помещает в списке L один из непомеченных векторов, скажем, для \mathcal{A}' , и затем находит многоугольник $\mathcal{P}(\mathcal{A}')$, в котором оптимально \mathcal{A}' . Продолжая действовать таким образом, пока все векторы в L не будут помечены, мы полностью разобьем пространство параметров. Так как алгоритм никогда не берет помеченный вектор и не включает в список одинаковых векторов, а при построении очередного многоугольника находит одно выравнивание, которое оптимально внутри каждого из соседних ему многоугольников, отсюда следует, что каждый многоугольник в полном многоугольном разбиении будет найден только один раз.

13.1.5. Временной анализ и следующая идея

Сказанное выше приводит к следующему временному анализу. Пусть R , E и V обозначают, соответственно, количество многоугольников, сторон и вершин в декомпозиции, а время на вычисление одного выравнивания для последовательностей длиной n и $m > n$ при фиксированных значениях параметров имеет порядок $O(mn)$. Сколько будет выполнено поисков по лучу при построении одного многоугольника $\mathcal{P}(\mathcal{A})$, заданного \mathcal{A} и p ? Пусть d — количество сторон в $\mathcal{P}(\mathcal{A})$. Тогда $3d$ поисков по лучу достаточно для нахождения сторон $\mathcal{P}(\mathcal{A})$ и в сильно вырожденном случае, когда выбранные лучи проходят через все вершины $\mathcal{P}(\mathcal{A})$, может быть сделано еще $3d$ (напрасных) поисков. Следовательно, для описания $\mathcal{P}(\mathcal{A})$ потребуется не более $6d$ поисков по лучу. Каждая сторона принадлежит не более чем двум многоугольникам, так что алгоритм для полного описания декомпозиции выполняет не более $12E$ поисков по лучу. Далее, по третьему закону Ньютона каждый поиск по лучу требует не более R вычислений выравнивания с фиксированными параметрами, так что полная декомпозиция требует не более $12RE$ выравниваний с фиксированными параметрами, которые можно выполнить за время $O(ERnm)$. Эта неудовлетворительная граница может быть улучшена благодаря одной дополнительной идее.

Новая идея заключается в такой модификации алгоритма Ньютона (при заданном \mathcal{A}), чтобы выбирать начальную точку r достаточно далеко по h , возможно ближе к r^* , насколько позволяет об этом судить имеющаяся информация. Рассмотрим любое выравнивание \mathcal{A}' , полученное до текущего выполнения алгоритма Ньютона. Вычислим пересечение плоскостей для \mathcal{A} и \mathcal{A}' и спроектируем эту линию на плоскость y - b . Если проекция пересекает h , скажем, в точке r' , то начальная точка r не должна быть дальше от p , чем r' , так как значение \mathcal{A}' за r' больше, чем у \mathcal{A} . Если проекция не пересекает h , то значение \mathcal{A} больше, чем у \mathcal{A}' , во всех точках h . Пересечение и проекцию можно вычислить за константное время. Повторяя это действие для всех предварительно вычисленных выравниваний, мы возьмем в качестве начального r ближайшую к p среди точек r' .

Модифицированный метод Ньютона, очевидно, снижает число требуемых выравниваний, но насколько? И сколько дополнительного времени уходит на его

реализацию? Рассмотрим сначала второй вопрос. На протяжении всего вычисления алгоритм декомпозиции поддерживает список L' , который является расширением списка L . После вычисления любого выравнивания его вектор помещается в L' (если такого вектора там еще нет) независимо от того, оптимально это выравнивание для многоугольника, стороны или только для вершины. Когда начинаем поиск по лучу, мы используем L' , как сказано выше, при нахождении начальной точки r . Так как каждая точка r' вычисляется за константное время, время на однократную обработку всего списка L' пропорционально размеру L' . Мы утверждаем, что этот размер не превосходит $V + E + R$, и в процессе работы алгоритма будет вычислено не более $V + E + R$ различных векторов. Причина в том, что динамическое программирование детерминировано, так что независимо от того, сколько раз оптимальное выравнивание вычисляется в конкретной вершине, всегда будет получаться то же выравнивание (и тот же вектор). Аналогично, даже если выравнивания вычисляются во многих точках стороны многоугольника, детерминированная процедура ДП будет возвращать каждый раз одно и то же выравнивание, так как любое выравнивание, которое оптимально для любой точки стороны, оптимально для всех ее точек. Наконец, из всякого выравнивания внутри многоугольника возвращается один и тот же вектор, так как все оптимальные выравнивания внутри многоугольника имеют один и тот же вектор. Таким образом, алгоритм возвращает только один вектор для каждой вершины, стороны или многоугольника разложения или в общей сложности $V + E + R$ векторов. Поэтому добавляемое учетное время за использование L' равно $O(V + E + R)$ за один поиск по лучу или всего $O(12E(V + E + R))$.

При анализе числа выравниваний с фиксированными параметрами, вычисляемых модифицированным алгоритмом, назовем вычисление выравнивания *избыточным*, если оно возвращает выравнивание с вектором, который уже записан в L' . Мы утверждаем, что при любом поиске по лучу (используя метод Ньютона с \mathcal{A}) только последнее вычисление выравнивания может быть избыточным. Рассмотрим два случая: \mathcal{A} либо оптимально в начальной точке r , либо нет. Если оптимально, то в поиске по лучу вычислено только одно выравнивание, и утверждение становится бесодержательным. В противном случае, если \mathcal{A} не оптимально в начальной точке r , любое избыточное выравнивание, скажем, \mathcal{A}' , вычислено в точке, более близкой к p , чем начальная точка r . Это означает, что пересечение \mathcal{A} и \mathcal{A}' проектируется в точку на h , более близкую к p , чем начальная точка r . Так как по третьему закону Ньютона все векторы, вычисленные во время поиска по лучу, различны, \mathcal{A}' должно было быть в L' до начала текущего поиска по лучу. Но это противоречит выбору начальной точки r .

Теперь можно проанализировать время на поиск многоугольной декомпозиции. Использование модифицированного метода Ньютона не меняет того факта, что будет выполнено не более $12E$ поисков по лучу. Только одно вычисление выравнивания в каждом поиске будет избыточным, так что все остальные вычисления выравниваний должны находить новые векторы, включаемые в L' . Поскольку размер L' не превосходит $V + E + R$, то отсюда следует, что при полном вычислении многоугольной декомпозиции вычисляется не более $V + 13E + R$ выравниваний. Это приводит к общей временной оценке $O(12E(V + E + R) + (V + 13E + R)nm)$. Многоугольная декомпозиция может рассматриваться как связный планарный граф, и поскольку каждая вершина инцидентна не менее чем трем сторонам (так как это

многоугольная декомпозиция), то $V \leq E \leq 3R$. Это легко показать, используя теорему Эйлера о планарных графах, хотя для произвольного планарного графа утверждение и не верно. Таким образом, все члены $12E$, $V + E + R$ и $V + 13E + R$ имеют порядок $O(R)$. Следовательно, приведенная оценка времени превращается в $O(R^2 + Rnm)$, или $O(R + nm)$ на многоугольник.

Граница $O(R + nm)$ на многоугольник верна независимо от того, какие параметры выбраны для изменения. Более того, если мы применяем какую-либо форму выравнивания, где вычисление с фиксированными параметрами занимает время $O(C)$, а не $O(nm)$, то оценкой становится $O(R + C)$ на многоугольник. Однако мы покажем ниже, что если не использовать алфавитно-зависимые оценочные матрицы, то $R = O(nm)$. На самом деле когда вычисляются глобальные выравнивания без обращения к оценочным матрицам, то $R < n^{2/3}$. Когда оценочные матрицы используются, но γ и δ являются переменными параметрами, то опять же $R = O(nm)$. Собрав все вместе, получаем:

Теорема 13.1.2. Для большинства (важных) выборов параметров полную многоугольную декомпозицию можно найти за время $O(nm)$ на многоугольник (т. е. оно будет пропорционально времени, необходимому для вычисления одного выравнивания с фиксированными параметрами).

13.1.6. Граница для числа многоугольников в декомпозиции

В этом пункте мы увидим, сколько может быть многоугольников в декомпозициях разного типа. Хотя этот вопрос интересен и сам по себе, он имеет два практических следствия. Во-первых, он касается опасения, что многоугольников может оказаться слишком много и декомпозиция не даст никакой интересной информации. Во-вторых, так как метод построения многоугольной декомпозиции, изложенный в предыдущем пункте, работает за время $O(R + nm)$ на многоугольник, мы должны ограничить R , чтобы получить утверждение теоремы 13.1.2. Рассмотрим два результата, подробно изложенные в [203].

Теорема 13.1.3. Рассмотрим следующую схему оценки, не использующую оценочные матрицы: $\alpha \times mt - \beta \times ms - \gamma \times id - \delta \times gp$. Независимо от того, какие два из четырех параметров выбраны для варьирования, и от того, какой тип выравнивания вычисляется (глобальный, локальный и др.), многоугольная декомпозиция может содержать не более $O(nm)$ многоугольников.

Доказательство. Для иллюстрации предположим, что γ и δ переменные, а α и β фиксированы со значениями α_0 и β_0 . Тогда для любого выравнивания \mathcal{A} значение $\alpha_0 \times mt_{\mathcal{A}} - \beta_0 \times ms_{\mathcal{A}}$ будет постоянным, назовем его $C_{\mathcal{A}}$, и значение \mathcal{A} будет равно $C_{\mathcal{A}} - \gamma \times id_{\mathcal{A}} - \delta \times gp_{\mathcal{A}}$. Сопоставим выравниванию \mathcal{A} тройку $(C_{\mathcal{A}}, id_{\mathcal{A}}, gp_{\mathcal{A}})$. Если \mathcal{A}' — какое-то другое выравнивание, имеющее $id_{\mathcal{A}'}$ пробелов и $gp_{\mathcal{A}'}$ пропусков, и $C_{\mathcal{A}} < C_{\mathcal{A}'}$, то \mathcal{A} не сможет быть оптимальным во всех точках плоскости γ, δ . Поэтому среди всех троек, у которых последние два элемента равны $id_{\mathcal{A}}$ и $gp_{\mathcal{A}}$, не более одной может быть связано с выравниванием, оптимальным в какой-либо точке. Если n и $m > n$ — длины двух строк, то может быть не больше $n + 1$ пробелов и $m + n$ пробелов в выравниваниях этих строк, так что не более $O(nm)$ троек ассоциируется с оптимальными выравниваниями. Из этого следует утверждение теоремы, так как любые два выравнивания, ассоциированные с одной и той же тройкой, оптимальны в одних и тех же точках. \square

Очевидно, что когда для оценки совпадений и несовпадений применяется оценочная матрица, то по-прежнему, пока переменными параметрами являются γ и δ , возможно только $O(nm)$ многоугольников. Однако если при использовании оценочных матриц будут меняться α и β , то оценка $O(nm)$ из приведенного рассуждения уже получаться не будет. В действительности мы не знаем никакой осмысленной границы для числа многоугольников в этом случае, а опыт показывает, что это число в декомпозициях значительно больше, чем при изменении γ и δ .

Границы $R = O(nm)$ достаточно для завершения доказательства теоремы 13.1.2. Однако, когда оценочные матрицы не используются и выравнивания глобальные, R получается значительно меньше, чем nm , что мы и покажем дальше.

Частный случай глобального выравнивания

Мы хотим ограничить число многоугольников, которые могут быть получены при декомпозиции параметров, когда оценочные матрицы не используются и выравнивания глобальные. Без потери общности рассмотрим двухпараметрическую целевую функцию: максимизировать $mt_{\mathcal{A}} - \beta \times ms_{\mathcal{A}} - \gamma \times id_{\mathcal{A}}$. Все решается следующей леммой.

Лемма 13.1.3. Для любого выравнивания \mathcal{A} с соответствующим вектором (mt, ms, id) имеет место равенство $2mt + 2ms + id = N$, где $N = n + m$ — сумма длин последовательностей. Следовательно, $mt + ms + id/2 = N/2$ для любого глобального выравнивания.

Доказательство. Совпадение и несовпадение затрагивают по два символа из со-поставляемых строк. Таким образом, полное число символов, которые в них участвуют, равно $2(mt + ms)$. Индел использует один символ из одной строки, их число равно id . Так как рассматривается глобальное выравнивание, то учитывается каждый из N символов, причем строго один раз. \square

Следствие 13.1.2. В точке $\beta = -1$, $\gamma = -1/2$ все глобальные выравнивания имеют одно и то же значение $N/2$.

Доказательство. Подставляя значения $(-1, -1/2)$ в целевую функцию, видим, что при любом глобальном выравнивании $C_{\mathcal{A}}$ имеет значение $mt_{\mathcal{A}} + ms_{\mathcal{A}} + id_{\mathcal{A}}/2$, которое равно $N/2$ по лемме 13.1.3. \square

Так как все выравнивания имеют одно и то же значение в точке $(-1, -1/2)$, то границы между соседними многоугольниками декомпозиции также должны проходить через эту точку. Значит, каждый многоугольник должен иметь форму угла с вершиной в $(-1, -1/2)$. Получим теперь границу для числа многоугольников.

Теорема 13.1.4. В случае глобального выравнивания без оценочных матриц в декомпозиции пространства параметров может быть не более $O(n)$ многоугольников, если $n \leq m$.

Доказательство. Так как каждая граница многоугольника выходит из точки $(-1, -1/2)$, то, входя в положительный квадрант плоскости β , γ (область интереса), она должна пересекать либо горизонтальную, либо вертикальную координатную ось. Покажем, что число пересечений вертикальной оси не может превосходить n .

Вдоль вертикальной оси (оси γ) мы имеем $\beta = 0$, так что параметрическая задача вдоль этой оси будет максимизировать $mt_{\mathcal{A}} - \gamma \times id_{\mathcal{A}}$ как функцию от одного параметра γ . Ясно, что при возрастании γ в оптимальных выравниваниях mt должно убывать (т. е. при каждом переходе из зоны в зону на оси γ). Но так как число

совпадений может меняться только от 0 до n , то может быть не более n границ многоугольника, пересекающих ось γ .

Та же верхняя граница в n переходов верна (по аналогичным причинам) и для горизонтальной оси. Это доказывает теорему. \square

На самом деле можно доказать более сильные утверждения. Прежде всего, легко показать, что никакая граница не пересекает горизонтальной оси (оси β). Мы оставляем это упражнением для читателя. Далее, в [203] показано, что число многоугольников не превосходит $O(n^{2/3})$. Таким образом, в случае глобального выравнивания без оценочных матриц число многоугольников в декомпозиции пространства параметров невелико.

13.1.7. Применения параметрического выравнивания

Обсудим два приложения параметрического выравнивания.

Анализ чувствительности

Первое применение — это обычный *анализ чувствительности*, который показывает устойчивость, или робастность, выравнивания. Когда оптимальное выравнивание вычисляется при одном наборе значений параметров, следует проверить, насколько оно чувствительно к изменениям параметров. Пирсон говорит об этом так:

“Следует быть очень осмотрительным при оценке “существенных” выравниваний, которые на самом деле пригодны только при одной-двух комбинациях оценочных матриц и штрафов за пропуски” [359].

Параметрическое выравнивание не меняет непосредственно оценочных матриц (хотя и может изменить весами вклад оценок), но штрафы за пропуски — начальный и за увеличение — могут меняться непосредственно. Чувствительность выравнивания в точке p можно проверить, определив, как далеко находится точка p от границы многоугольника в декомпозиции пространства параметров. Если p содержится в многоугольнике \mathcal{P} , то дополнительную информацию можно получить из размера \mathcal{P} и возможных изменений значения выравнивания на \mathcal{P} и в соседних многоугольниках.

Эффективное вычисление всех кооптимальных выравниваний

Второй пример исходит из исследования Бартона и Стернберга [51]. Пример иллюстрирует мысль, что зачастую важно вычислять все кооптимальные выравнивания, а не какое-либо одно. Он показывает, как параметрические выравнивания могут сделать это приемлемой задачей.

Суть в том, что Бартон и Стернберг выравнивали пары белков, используя (в основном) параметрическую целевую функцию: $V(\mathcal{A}, \gamma, \delta) = smt_{\mathcal{A}} + sms_{\mathcal{A}} - \gamma \times id_{\mathcal{A}} - \delta \times gp_{\mathcal{A}}$. Они вычисляли по одному оптимальному выравниванию для каждой из 121 комбинаций γ , δ , где γ и δ принимали целочисленные значения от 0 до 10, и хотели найти для каждого из этих 121 наборов параметров оптимальные выравнивания, обладающие некоторым конкретным свойством (о котором мы говорить не будем). Ни одно из вычисленных 121 выравниваний искомым свойством не обладало. Они сделали вывод, что стандартная модель выравнивания (закодированная в приведенной выше целевой функции) недостаточна для нахождения выравниваний с желаемым свойством и что требуется другая модель выравнивания.

Однако впоследствии было показано [205], что в допустимом диапазоне y и δ имеются целочисленные значения, для которых оптимальное выравнивание обладает нужным свойством. Более того, есть также нецелочисленные значения параметров, при которых этим свойством обладает большинство оптимальных выравниваний. Дело в том, что для любой точки пространства параметров p есть много кооптимальных выравниваний. Все они имеют в p одно и то же значение, но по другим свойствам существенно различаются. Если хочется узнать, существует ли оптимальное выравнивание \mathcal{A} (оптимальное относительно данной целевой функции) с некоторым дополнительным свойством, то недостаточно вычислять по одному оптимальному выравниванию в каждой точке. Нужно смотреть все кооптимальные выравнивания. Иначе, как это и случилось в упомянутом исследовании [51], вам может не повезти и нужные кооптимальные выравнивания вы пропустите. Аналогичное наблюдение было сделано в [486], где отмечалось, что в большом эмпирическом исследовании влияния различных весов, штрафов и матриц на качество выравнивания [448] часто пропускались существенные выравнивания, которые позднее были найдены полным параметрическим поиском.

Хороший совет — вычислять все кооптимальные решения — легко принять, но трудно исполнить, так как вычисление всех кооптимальных выравниваний во всех точках пространства параметров требует больших ресурсов. Именно в этом месте может помочь параметрическое выравнивание.

Чтобы найти все выравнивания, которые оптимальны в одной или более (неизвестных) точках данного параметрического пространства, построим сначала разложение этого пространства. Затем вычислим все кооптимальные выравнивания в каждой *вершине* этого разложения. Так будут найдены все выравнивания, оптимальные в какой-либо точке пространства параметров. Теперь кооптимальные выравнивания можно вычислять, тратя время $O(n + m)$ на выравнивание (см. упражнение 12 на с. 304). Для большей эффективности и устойчивости предпочтительнее выбирать по одной точке внутри каждого многоугольника и вычислять кооптимальные выравнивания только в этих точках. Таким способом будут найдены все устойчивые выравнивания, оптимальные на ненулевой площади пространства параметров, и каждое такое выравнивание будет вычислено лишь единожды.

Этот подход к вычислению устойчивых кооптимальных выравниваний очень эффективен эмпирически. В случае, рассмотренном в [51], в разложении плоскости y - δ оказалось только 17 многоугольников (частично показанных на рис. 13.1). Более того, только 120 выравниваний оптимальны в каких-либо внутренних точках этих многоугольников. Параметрические выравнивания делают приемлемым вычисление всех 120 устойчивых кооптимальных выравниваний, каждое строго по одному разу, сначала идентифицируя набор из 17 внутренних точек, где нужно перечислять кооптимальные выравнивания. Это сильно отличается от очевидного желания вычислять их в 121 целочисленной точке, выбранной в [51], где придется обрабатывать 1105 выравниваний. Почти десятикратная избыточность возникает из-за того, что многие целочисленные точки попадают в один и тот же многоугольник (так как многоугольников всего 17, а целочисленных точек 121). Хуже того, многие из этих точек на рис. 13.1 попадают на вершины и стороны разложения (по причинам, объясненным в [205]), где будет оптимальным и, следовательно, будет вычислено каждое выравнивание, оптимальное в соседнем многоугольнике. Избыточное генерирование выравниваний снижает привлекательность перечисления кооптимальных выравниваний. Но параметрическое разложение точно нацеливает на точки запросов, в которых

будут перебираться все устойчивые кооптимальные выравнивания и только они, причем каждое строго по одному разу.

13.2. Вычисление субоптимальных выравниваний

Оптимальное выравнивание, даже при широком размахе моделей и выборов параметров, не всегда распознает те биологические феномены, которые оно призвано отразить. Здесь проблема состоит из пяти частей: доступные исследователю целевые функции могут не отражать весь диапазон биологических причин, вызывающих различия строк; эти целевые функции могут быть неспособны строить оптимальные выравнивания, из которых составляется желаемая форма (вспомним обсуждение совпадений кДНК); данные могут содержать ошибки, сбивающие с толку алгоритмы; могут быть ограничения на оптимальное выравнивание; наконец, может быть много выравниваний, близких к оптимальным, которые биологически более значимы, чем оптимальные *).

Одной из реакций на эту проблему может быть признание того, что оптимальное выравнивание очень грубо отражает биологическую значимость и не должно рассматриваться как единственно возможное. Вместо этого желательно создавать больший набор выравниваний, впрочем, достаточно управляемый, который может содержать что-либо более биологически интересное. И даже хотя оптимальность может не соответствовать “истинной биологической значимости”, но если целевая функция была сконструирована для биологического моделирования, то набор выравниваний, близких к оптимальному, следует предпочесть случайному набору выравниваний. Следуя этой философии, оптимальное выравнивание можно представить как часть фильтрующего механизма для нахождения набора выравниваний терпимого размера, которые можно оценивать каким-либо дополнительным критерием или личной экспертизой. В предыдущих пунктах мы видели один способ, как это делать, варьируя параметры, находя результирующие многоугольники и получая одно или больше оптимальных выравниваний на многоугольник. Другим конкретным подходом является генерирование “близких к оптимальным” выравниваний для некоторой фиксированной установки параметров. Этот подход был первоначально развит в [490, 443, 445, 460]. Наше изложение следует работе Наора и Брутлега [346]; версия этого подхода, более эффективная по памяти, появилась в [96].

13.2.1. Первые определения и первые результаты

В этом изложении мы будем сильно опираться на концепцию графов выравнивания, введенных в п. 11.6.3 (с. 284), и на соответствие между выравниваниями строк и путями в графе выравнивания. Излагаемые результаты относятся к модели *глобального выравнивания с максимальным значением*, и поэтому вид целевой функции дальше не уточняется. Просто обозначим через $s(u, v)$ значение дуги (u, v) графа выравнивания, а через $V(S_1, S_2)$ — значение оптимального выравнивания строк S_1 и S_2 .

*) Здесь важно подчеркнуть, что термин “оптимальный” относится только к данной целевой функции. Его не следует рассматривать как синоним слов “корректный” или “желательный”, хотя в литературе по этому вопросу часто возникают неясности. Все в этом параграфе держится на том, что “оптимальный” не всегда “наиболее биологически значимый” или даже “адекватный биологическим интересам”.

для некоторой фиксированной, но не уточняемой дальше максимизируемой целевой функции. Перенос этих результатов на другие модели выравнивания или на редакционные расстояния оставляется на упражнения. Мы начнем с некоторых необходимых определений, связанных с оптимальными и близкими к оптимальным путями в графе выравнивания.

Определение. Пусть R — путь в графе выравнивания от стартовой вершины $s \equiv (0, 0)$ до терминальной вершины $t \equiv (n, m)$. Путь R соответствует некоторому глобальному выравниванию (не обязательно оптимальному) строк S_1 и S_2 .

Определение. Для любой пары вершин x, y графа выравнивания пусть $l(x, y)$ — наибольшая длина пути из x в y . Пусть R^* — путь, соответствующий оптимальному глобальному выравниванию, так что длина R^* равна $l(s, t) = V(S_1, S_2)$.

Определение. Для пути R пусть $\delta(R)$ равно разности длин R^* и R . $\delta(R)$ называется “отклонением” R (от оптимального пути), а R — путем, $\delta(R)$ -близким к оптимальному. Путь R называется δ -близким к оптимальному, если $\delta(R) = \delta$.

Определение. Для дуги $e = (u, v)$ пусть

$$\delta(e) = l(s, t) - [l(s, u) + s(u, v) + l(v, t)].$$

Значит, $\delta(e)$ равно разности длин R^* и наибольшей длины пути из s в t , который заставили идти через дугу e .

Вычисление оптимального выравнивания S_1 и S_2 дает $l(s, u)$ для всех вершин u , а вычисление оптимального выравнивания этих строк в обращенном виде дает $l(v, t)$ для всех вершин v . Следовательно:

Лемма 3.2.1. $\delta(e)$ можно вычислить для всех дуг за время, используемое для вычисления двух оптимальных выравниваний, плюс время, пропорциональное числу дуг.

Выравнивания, Δ -близкие к оптимальным

Можно изучать пути (или выравнивания), близкие к оптимальным, задав граничное значение Δ и затем вычисляя информацию о множестве всех путей, отклонение которых от оптимального не превосходит Δ , а значит, о множестве всех путей, которые δ -близки к оптимальному для некоторого $\delta \leq \Delta$. Мы увидим, как эффективно подсчитать и перечислить это множество путей. Менее очевидная альтернатива рассматривается в упражнении 11 в конце этой главы.

13.2.2. Полезное изменение весов

Следующие идеи способствуют как подсчету, так и перебору путей, близких к оптимальным.

Определение. Для дуги $e = (u, v)$ положим $\varepsilon(e) = l(u, t) - [s(u, v) + l(v, t)]$.

Здесь $\varepsilon(e)$ интерпретируется как “дополнительный штраф за использование дуги e в пути из u в t вместо следования оптимальным (самым длинным) путем” [346]. Следующая теорема показывает важность этих значений.

Теорема 13.2.1. Для любого пути R из s в t верно $\delta(R) = \sum_{e \in R} \varepsilon(e)$.

Доказательство. Пусть $R = v_0, v_1, \dots, v_{k-1}, v_k$, где $v_0 = s$ и $v_k = t$. Пусть также $|R|$ обозначает длину R . Тогда

$$\begin{aligned} \sum_{e \in R} &= \sum_{i=0}^{k-1} \varepsilon(v_i, v_{i+1}) = \\ &= \sum_{i=0}^{k-1} [l(v_i, t) - s(v_i, v_{i+1}) - l(v_{i+1}, t)] = \\ &= \sum_{i=0}^{k-1} [l(v_i, t) - l(v_{i+1}, t)] - \sum_{i=0}^{k-1} s(v_i, v_{i+1}) = \\ &= \left(\sum_{i=1}^{k-1} [l(v_i, t) - l(v_i, t)] + l(v_0, t) - l(v_k, t) \right) - |R| = \\ &= l(s, t) - |R| = |R^*| - |R| = \delta(R). \end{aligned}$$

□

Следствие 13.2.1. Рассмотрим путь R' из s в u . Пусть $\delta = \sum_{e \in R'} \varepsilon(e)$. Тогда путь R из s в t , состоящий из пути R' и наибольшего пути из u в t , является δ -близким к оптимальному.

Доказательство. По определению $\varepsilon(e)$ имеем $\varepsilon(e) = 0$ для любой дуги e наибольшего пути из u в t . Следовательно, $\delta(R) = \delta$ по теореме 13.2.1. □

Это следствие станет основой эффективного подсчета и перечисления путей, близких к оптимальным. Оно позволяет определить, будет ли начальная часть пути из s в t принадлежать полному пути, δ -близкому к оптимальному, не строя весь путь.

13.2.3. Подсчет и перечисление путей, близких к оптимальным

Зачем подсчитывать?

Одна из главных проблем в использовании моделей выравнивания в биологии заключается в том, что выравнивание с большой оценкой сходства или большой статистической значимостью (по сравнению с выравниваниями случайных строк) могут не иметь наибольшей биологической значимости. В результате ряд исследователей имеет выглядящие эффективными “кустарные” способы распознания выравниваний с полезной биологической информацией (см. [73, 127]). Подход, предложенный в [346], использует число выравниваний, близких к оптимальным, как кустарное правило. Эмпирически исследовав конкретные молекулярные последовательности, авторы обнаружили, что выравнивания с большим числом субоптимальных, близких к оптимальным, обычно не отражают правильно важную биологическую информацию, а при малом числе таких субоптимальных — отражают ее гораздо

лучше. Поэтому число субоптимальных решений “вблизи” от оптимального может быть эффективным кустарным правилом для отделения биологически информативных и неинформативных выравниваний. Таким образом, возможность эффективно подсчитать число путей, близких к оптимальным, может быть самым важным результатом, который мы получим в связи с выравниваниями, близкими к оптимальным.

Как подсчитывать?

Определение. Пусть $N(v, \delta)$ обозначает число δ -близких к оптимальному путей из s в t , проходящих через вершину v .

Для данного значения Δ число путей из s в t , отклонение которых от R^* не превосходит Δ , равно

$$\sum_{\delta \leq \Delta} N(t, \delta).$$

Вычислим эту сумму по следующему рекуррентному соотношению для каждой вершины v и для каждого “нужного” значения δ :

$$N(v, \delta) = \sum_{(u, v) \in E} [N(u, \delta - \varepsilon(u, v))].$$

Значит, для вершины v мы будем вычислять и хранить значения $N(v, \delta)$ при всех $\delta = \delta' + \varepsilon(u, v)$, таких что $N(u, \delta')$ вычислено для некоторой вершины u , предшествующей вершине v . Практически это означает сканирование значений N , полученных для каждого предшественника v и использование этих значений для изменения соответствующих значений N вершины v .

Заметим, что по следствию 13.2.1 имеется на самом деле $N(v, \delta)$ путей через v , отклонение которых от R^* равно в точности δ , так как каждый из $N(v, \delta)$ путей до v может быть дополнен самым длинным путем из v в t . Без использования значений $\varepsilon(e)$ (или сходной идеи) пришлось бы считать все пути до v , даже те, отклонение которых будет безусловно больше Δ . Без значений $\varepsilon(e)$ (или чего-нибудь аналогичного) алгоритм не знал бы, можно ли путь до v продолжить до полного пути в t с отклонением, не превышающим Δ . Мы вернемся к этому вопросу в упражнении 13.

Перебор

Пути, δ -близкие к оптимальным, можно перечислять в порядке возрастания δ и закончить, когда $\delta = \Delta$ или когда будет найдено некоторое фиксированное число путей. Здесь используется идея, общая почти для всех методов перебора. Поддерживается дерево, перебирающее частичные пути. Каждая внутренняя вершина дерева x соответствует пути R' из s в вершину u и графа выравнивания. В вершине x хранится значение $d(x) = \sum_{e \in R'} \varepsilon(e)$, и по следствию 13.2.1 существует путь из s в t с точно таким отклонением от R^* . Каждая внутренняя вершина дерева x продолжается по одному разу для каждой дуги, выходящей из u в графе выравнивания. Продолжение добавляет эту дугу к пути R' и создает новую внутреннюю дугу дерева. Чтобы перебрать пути, δ -близкие к оптимальному, в порядке возрастания δ , алгоритм поддерживает упорядоченный список (или приоритетную очередь) известных отклонений для

путей, которые уже найдены, и выбирает для продолжения вершину x с наименьшим значением $d(x)$; в этот момент он выводит в результат путь, у которого отклонение равно в точности $d(x)$. Метод становится проще, если пути с отклонением, не превосходящим Δ , не обязательно перечислять в порядке возрастания отклонений. Дополнительное улучшение схемы перебора путей в порядке возрастания предложено в [346].

13.2.4. Альтернативный подход к субоптимальным выравниваниям

Перечисление всех субоптимальных выравниваний с фиксированным отклонением от оптимального может быть очень полезным, но оно способно также произвести чрезмерное количество “похожих” выравниваний, а это может оказаться очень емким по времени. Уотермен и Эггерт [462] предложили другой, более эффективный путь к перечислению широкого спектра субоптимальных выравниваний. Их метод пропустит часть выравниваний, генерируемых описанным выше процессом перебора, но для многих целей это будет достаточный набор, который можно анализировать и выбирать подходящие выравнивания.

Для объяснения идеи, предложенной в [462], вновь воспользуемся графом выравнивания. Оптимальное выравнивание соответствует наибольшему пути из левого верхнего угла в правый нижний угол графа. Программа ДП, которая обнаруживает одно оптимальное выравнивание, находит также все дуги графа, содержащиеся в каком-либо наибольшем пути. Метод Уотермена ищет наилучшее следующее выравнивание (возможно, также оптимальное), которое отличается “существенным” образом от первого оптимального (а не просто незначительное отклонение от первого выравнивания). После того как он находит первый самый длинный путь, алгоритм удаляет все дуги этого пути и дуги, с ними соприкасающиеся. Затем он находит самый длинный путь в оставшемся графе, и процесс повторяется. Пути, полученные таким образом, определяют совершенно различные выравнивания. Этот метод работает и для глобальных, и для локальных выравниваний.

13.3. Сцепление различных локальных выравниваний

Пусть уже найдено много оптимальных или субоптимальных пар сильно похожих подстрок из двух строк, S и S' . Как выбрать из этих пар “хорошее” подмножество, которое показало бы соотношение между строками S и S' по всей их длине? Задача сцепления участков высокого сходства, заданных набором различных локальных выравниваний, — это общая важная проблема, которая часто встречается в вычислительной молекулярной биологии. В рамках биологического моделирования она в основном не решается, поскольку редко имеется в наличии хороший критерий для оценки качества выбранного подмножества. Тем не менее в наиболее простой модели, используемой в настоящее время (которую мы здесь и рассмотрим), есть эффективный алгоритм выбора наилучшего (в смысле этой модели) подмножества пар. Начнем с одномерного варианта задачи.

Задача об одномерном сцеплении

Рассмотрим множества из r (возможно) перекрывающихся интервалов на прямой R , где каждому интервалу j сопоставлено некоторое значение $v(j)$. Требуется выбрать подмножество неперекрывающихся интервалов с максимально возможной суммой этих значений (рис. 13.3).

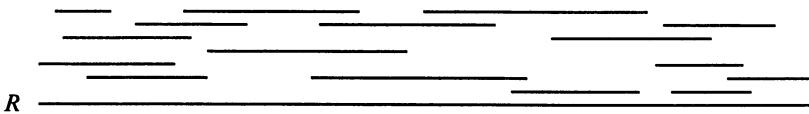


Рис. 13.3. Линия R и r меньших интервалов в R . Каждому интервалу сопоставлено значение, и задача в том, чтобы выбрать поднабор неперекрывающихся интервалов с наибольшим полным значением

Такая задача встречается при *сборке гена* (см. п. 18.2), где R является последовательностью ДНК эукариотного гена, а каждый интервал — предполагаемый экзон этого гена. Значение $v(j)$ отражает “хорошесть” или “правдоподобие” того, что интервал j действительно экзон. Обычно (исключение составляет сращивание участков) настоящие экзоны в гене не перекрываются, а предполагаемые — перекрываются. Таким образом, задача сборки гена заключается в выборе неперекрывающегося набора предполагаемых экзонов для сборки в один создаваемый ген.

Алгоритм одномерного сцепления

Пусть I — список всех $2r$ чисел, задающих координаты концов имеющихся интервалов. Отсортировать числа из I и сохранить при каждом элементе из I информацию об интервале и о том, какой это конец — левый или правый. Для удобства будем считать I одномерным массивом.

Положить \max равным 0.

```
for  $i$  from 1 to  $2r$  do begin
    if  $I[i]$  — левый конец интервала  $j$  then
         $V[j] := v(j) + \max$ ;
    if  $I[i]$  — правый конец интервала  $j$  then
         $\max := \max \{\max, V[j]\}$ 
end;
```

Значение \max в конце алгоритма — это значение оптимального решения. Время, расходуемое на алгоритм, определяется временем сортировки чисел в I и равно $O(r \log r)$. Задачу вывода оптимального решения мы оставляем на упражнения.

Двумерная задача сцепления

Обратимся теперь к задаче выбора пар сильно схожих строк. Пусть заданы две строки, S и S' , и известно несколько пар подстрок S и S' , обладающих высоким сходством. Эти пары можно получать, используя технику, описанную в пп. 13.2.4 и 11.7.3. Однако найденные подстроки из S (или S') могут перекрываться, и задача в том, чтобы выбрать “хорошее” подмножество пар так, чтобы выбранные подстроки из S

(и соответственно из S') в этих парах не перекрывали друг друга (рис. 13.4.). Пример этой задачи, относящийся к выравниванию строк ДНК с белковыми строками, будет рассмотрен в п. 18.1.

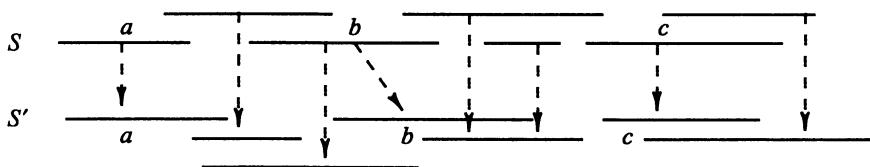


Рис. 13.4. Пары сильно схожих подстрок из S и S' . Каждая пара соединена линией со стрелкой. Пары, обозначенные a , b и c , образуют цепь. Значения пар не показаны

Исходные данные в задаче можно абстрактно представить набором прямоугольников на плоскости со сторонами, параллельными координатным осям. Если записать S по оси y , а S' по оси x , то вертикальная сторона прямоугольника представляет подстроку S , а горизонтальная — подстроку S' . Каждому прямоугольнику i сопоставлено значение $v(i)$, которое, например, может быть значением сходства этих двух подстрок.

Определение. Набор прямоугольников называется *цепью*, если никакая горизонтальная или вертикальная линия не пересекает более одного из этих прямоугольников и если прямоугольники можно упорядочить так, чтобы каждый из них был ниже и правее предшествующего (рис. 13.5). Значение цепи является суммой значений прямоугольников цепи.

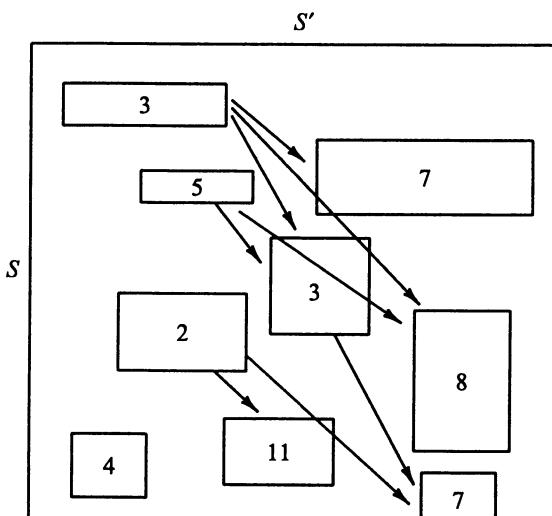


Рис. 13.5. Стрелками показаны возможные переходы в цепи от одного прямоугольника к другому. Две стрелки, существование которых следует из транзитивности, опущены. В центре каждого прямоугольника записано его значение. Оптимальная цепь содержит прямоугольники со значениями 5, 3 и 7. Ее полное значение равно 15

Задача сцепления. Выбрать из всех цепей ту, значение которой максимально.

Эскизно решение задачи о цепи заключается в построении ациклического графа G , в котором каждому прямоугольнику соответствует одна вершина, а дуга из вершины i в вершину j идет в том и только том случае, если прямоугольник j может следовать за i в какой-нибудь цепи. Граф G содержит также стартовую дугу s с дугами, идущими из s в каждую вершину G , и терминальную вершину t , в которую входят дуги из всех вершин G . Длина каждой дуги, ведущей в вершину i , отличную от t , равна $v(i)$. Цепь из прямоугольников с максимальным значением соответствует самому длинному пути из s в t . Если имеется r прямоугольников и e дуг, то самый длинный путь можно найти за время $O(e) = O(r^2)$.

Более быстрое решение

Мы решим задачу о цепи с помощью алгоритма, который требует времени $O(r \log r)$. Он обобщает подход, использованный при решении одномерной задачи. Этот метод, впервые явно описанный для задачи поиска неперекрывающихся пар подстрок с высоким сходством [251], типичен в области *разреженного динамического программирования*.

Рассмотрим отрезки — проекции r прямоугольников на горизонтальную ось. Пусть I — упорядоченный список концов этих отрезков. Снабдим каждый элемент списка I информацией о прямоугольнике и о том, левый это или правый конец.

Алгоритм обрабатывает элементы I по порядку (слева направо), как в алгоритме для одномерной задачи. Но так как прямоугольники двумерны, то нужно для каждого прямоугольника рассматривать и координаты y . Обозначим через h_j верхнее значение координаты y прямоугольника j , а через l_j — нижнее значение. Алгоритм будет работать со списком L троек $(l_j, V(j), j)$. Третий элемент тройки, j , определяет сам прямоугольник, а $V(j)$ — максимальное значение любой цепи, которая заканчивается прямоугольником j . Тройки в L будут упорядочены по убыванию значения элемента l . Грубо говоря, прямоугольники, дно которых находится на плоскости выше, записаны в списке L раньше.

Список L будет обновляться по мере обработки элементов из I . Очевидно, после обработки точки \bar{x} из I , элемент $(l_j, V(j), j)$ в L показывает, что лучшая цепь, полностью содержащаяся в верхнем левом квадранте с началом в (\bar{x}, l_j) (т. е. слева от \bar{x} и выше l_j), заканчивается в j и имеет значение $V(j)$.

Полностью алгоритм выглядит так:

Двумерный алгоритм для цепи

Начать с пустого списка L .

for i from 1 to $2r$ do begin

if $I[i]$ левый конец прямоугольника k then begin

найти в L последнюю тройку, в которой $l_j > h_k$.

{То есть найти ближайший (по вертикали) прямоугольник j с тройкой в L ,

нижняя точка которого строго выше

самой высокой точки прямоугольника k .}

Положить $V(k) := v(k) + V(j)$.

end

```

else if  $I[i]$  правый конец прямоугольника  $k$  then begin
    найти в  $L$  первую тройку, у которой  $l_j \leq l_k$ .
    if  $l_j < l_k$  or  $l_j = l_k$  and  $V(k) > V(j)$  then
        включить тройку  $(l_k, V(k), k)$  в список  $L$ , поддерживая список
        упорядоченным по  $l$  значениям троек.
        Удалить из  $L$  тройки, соответствующие прямоугольникам  $j'$ ,
        где  $l_{j'} \leq l_k$  и  $V(k) > V(j')$ .
    end
end;

```

В конце работы алгоритма последний элемент L дает значение оптимальной цепи и указывает последний ее прямоугольник. Мы оставим на упражнения изменения этого алгоритма, обеспечивающие нахождение оптимальной цепи.

Корректность и реализация

Доказательство корректности оставим на упражнения. Прежде чем заняться временным анализом, отметим, что после каждой итерации если тройка для прямоугольника j появилась в L после тройки для прямоугольника j' , то $V(j) \geq V(j')$. Значит, тройки в L расположены в порядке неубывания их значений V , несмотря на то что алгоритм не занимается явно их сортировкой по значениям V .

Теперь рассмотрим время, расходуемое на поиск в списке L (либо последней тройки с $l_j > h_k$, либо первой тройки с $l_j \leq h_k$). Если хранить список L в сбалансированном дереве поиска, таком как АВЛ-дерево [10], каждый поиск можно выполнить за время $O(\log r)$. Более того, каждый элемент может быть вставлен или удален также за время $O(\log r)$. Далее, рассмотрим удаления после того, как в L включена новая тройка $(l_k, V(k), k)$. Так как тройки в L появляются в неубывающем порядке по значениям V , все удаляемые тройки находятся в сплошном блоке, начинающемся сразу после новой тройки $(l_k, V(k), k)$. Таким образом, удаления можно выполнить, просматривая список после тройки для k до тех пор, пока не будет найдена первая тройка, для которой $V(j') > V(k)$. Этим способом будут удалены все проверенные тройки, кроме последней. Время на эти удаления будет пропорционально числу удаляемых троек, умноженному на $\log r$. Каждой удаляемой тройке сопоставляется прямоугольник j' , и после удаления никакая дополнительная тройка для j' не будет включена в L . Поэтому за все время работы алгоритма удалений может быть не более r . В итоге:

Теорема 13.3.1. *Оптимальная цепь может быть найдена за время $O(r \log r)$.*

13.4. Упражнения

1. Докажите три закона Ньютона, сформулированные в лемме 13.1.2.
2. Покажите, что в случае параметрического глобального выравнивания, как оно определено в п. 13.1.6, никакая граница многоугольника не касается горизонтальной оси (оси β) в положительном квадранте.
3. Покажите, что все результаты п. 13.1.2 выполняются, когда в качестве целевой функции выравнивания используется алфавитно-зависимая оценочная матрица.

4. Проведите анализ числа и формы многоугольников для глобального параметрического выравнивания без оценочных матриц, когда целевая функция имеет вид $\alpha \times mt_{\mathcal{A}} - \beta \times ms_{\mathcal{A}} - id_{\mathcal{A}}$, т.е. цена инделов 1, а штраф за пропуски нулевой.
 5. Изучите ситуацию, когда в целевую функцию включена плата за пропуски и для варьирования выбираются два параметра (из четырех). Рассмотрите снова случай глобального выравнивания без оценочных матриц.
 6. Ответьте на тот же вопрос, что в упражнении 5, но теперь пусть изменяются три параметра.
 7. Рассмотрите следующую однопараметрическую целевую функцию для выравнивания \mathcal{A} : $v_{\mathcal{A}}(\lambda) = mt_{\mathcal{A}} - \lambda \times [ms_{\mathcal{A}} + id_{\mathcal{A}}]$. Для любого фиксированного значения λ_0 может быть много различных выравниваний, которые при этом значении оптимальны. Если их больше, чем одно, то только два из них могут быть оптимальны для других значений λ_0 . Объясните это. Затем покажите, как найти за время $O(nm)$ выравнивание, которое оптимально для λ_0 , а также и для некоторого значения λ , большего, чем λ_0 .
 8. Эта задача имеет теоретико-графический уклон. Используя знаменитую теорему Эйлера о планарных графах, покажите, что когда каждая вершина графа имеет степень не меньше трех, то число областей (граней) графа пропорционально числу его вершин. (Напомним, что число вершин и число ребер в планарном графе пропорциональны друг другу.)
 9. Тейлор [432] высказал предположение, что может оказаться полезным такой вариант параметрического анализа: вместо одной задаются две оценочные матрицы, обозначим их через MD и ID и сопоставим им множители x и y . Чтобы оценить выравнивание \mathcal{A} при некотором выборе x и y , умножим оценку выравнивания, данную матрицей MD, на x , оценку, данную матрицей ID, на y и сложим эти оценки. Вариацией значений x и y можно изменить значимость MD относительно ID. Покажите, что этот тип параметрического анализа можно проводить средствами, развитыми в тексте главы.
 10. В биологической литературе принято добавлять константу к каждому из значений оценочной матрицы для аминокислот. Однако, какую именно константу добавлять, остается предметом дискуссий. Хотя рассматривается задача *добавления* чего-то к элементам матрицы, а не *умножения* их, изучение зависимости выравнивания от выбора аддитивной константы можно провести описанными средствами параметрического анализа. Покажите это в деталях.
 11. Для данного значения Δ обозначим через G_Δ совокупность всех дуг во всех δ -близких к оптимальным путях по всем $\delta \leq \Delta$. Граф G_Δ дает косвенное отражение множества путей, которые δ -близки к оптимальным для всех $\delta \leq \Delta$. Докажите следующую теорему.
- Теорема 13.4.1.** Для любого заданного Δ пусть $E_\Delta = \{e : \delta(e) < \Delta\}$. Тогда $E_\Delta = G_\Delta$.
12. Постройте пример пути в каком-либо G_Δ , который не будет δ -близким к оптимальному ни для какого $\delta \leq \Delta$. С учетом этого примера предложите варианты использования G_Δ .
 13. Подробно сравните эффективность методов подсчета и перебора путей, близких к оптимальным, с применением значений $\varepsilon(e)$ и таких же методов, использующих истинные значения дуг. Что нужно сделать, чтобы истинные длины дуг давали ту же эффективность, что и значения $\varepsilon(e)$?
 14. Покажите, как упростить метод перебора путей, близких к оптимальным, если нужно перебирать решения, отклоняющиеся не более чем на Δ , а соблюдение их упорядоченности по отклонению от оптимального не требуется.

15. Докажите корректность метода трудоемкости $O(r \log r)$ для двумерной задачи о цепях из п. 13.3.
16. Рассмотрим задачу вычисления оптимальной цепи в случае, когда каждой дуге сопоставлен вес, вдобавок к весам прямоугольников. Можно ли получить тот же результат со временем $O(r \log r)$, если веса произвольны? Что будет, если веса монотонно возрастают при росте евклидова расстояния между прямоугольниками?
17. **Нахождение неперекрывающихся приближенных повторов.** В упражнении 31 главы 11 рассматривалась задача нахождения двух подстрок с максимальным сходством среди всех пар подстрок строки S . При этом допускалось перекрытие подстрок. В некоторых ситуациях мы накладываем дополнительное условие — запрет перекрытий. Например, рассмотрим поиск в строке ДНК пары подстрок, одна из которых получилась копированием другой, потом была вставлена в полную ДНК, а затем модифицировалась отдельно от первой. В этом процессе две подстроки физически различны. Формальная строковая задача такова: пусть задана строка S ; найти две неперекрывающиеся подстроки S с наибольшим сходством. Задача изучалась в ряде работ [60, 257, 321, 396] и может быть решена за время $O(n^2 \log n)$ [396], где n — длина S . Этот результат слишком сложен для данного изложения. Вместо него мы набросаем первый подход, принадлежащий Миллеру [321]. Вы должны восполнить детали.

В основе подхода лежит заполнение таблицы ДП для вычисления локального выравнивания строки S с ней самой, но с некоторыми изменениями, уточняемыми ниже. Требование неперекрытия означает, что, когда алгоритм рассматривает подстроки, кончающиеся в позициях i и $j > i$, вторая подстрока должна начинаться *после* позиции i . Как нам встроить это в схему вычислений ДП для локального выравнивания? Очевидный способ — иметь в каждой клетке таблицы (i, j) значение наилучшего пути, начинающегося в каждой клетке слева сверху от нее. Тогда наилучший неперекрывающийся путь, который кончается в (i, j) , является наилучшим из этих верхне-левых путей, начинающихся в столбце $k > i$. Это значит, что каждой клетке сопоставляется $\Theta(n^2)$ значений, и требуемое время имеет порядок $\Theta(n^4)$. Объясните подробно сам метод и его корректность.

Граница времени $\Theta(n^4)$ может быть уменьшена. Небольшое размышление показывает, что в клетке (i, j) нужно хранить только значение наилучшего пути, начинающегося в каждом *столбце* слева от нее. Таким образом, в каждой клетке достаточно хранить только $\Theta(n)$ значений, и время уменьшается до $\Theta(n^3)$. Поясните это.

Метод Миллера является улучшением описанного метода. Его оценка времени в наихудшем случае составляет $O(n^3)$, но реальное время, скорее всего, $\Theta(n^2)$. Чтобы понять идею, рассмотрим наилучшие пути в клетку (i, j) , начинающиеся в столбцах k и $k' > k$. Если путь из k' лучше, чем из k , то путь из k не может быть продолжен так, чтобы он стал лучшим неперекрывающимся путем для любой клетки, находящейся слева вверху от (i, j) . Следовательно, столбец k не является больше для этих клеток кандидатом, и, значит, никакая клетка справа вниз от (i, j) не должна вычислять путь, начинающийся в столбце k . Это уменьшает размер списков, хранимых в клетках. Полностью объясните и обоснуйте эту идею.

Для эффективности реализации каждая клетка хранит упорядоченный список столбцов-кандидатов (тех, которые могут предложить лучший путь до (i, j)). Эти кандидаты упорядочены слева направо по номерам столбцов и обладают тем свойством, что значения соответствующих им путей убывают. Используя эти списки, алгоритм можно реализовать со временем, пропорциональным общей длине списков. Предложите детали этой реализации. Объясните, почему естественно ожидать, что общая длина списков будет меньше чем $\Theta(n^3)$. На самом деле практика показывает $\Theta(n^2)$.

18. Объясните, в чем сходство алгоритма Миллера, алгоритма для вогнутых весов за пропуски (п. 12.6) и алгоритма для склеивания локальных выравниваний (п. 13.3).
19. Метод Миллера гарантированно находит наиболее схожие неперекрывающиеся пары подстрок, но он сложноват. Более простой подход — использовать алгоритмы из п. 13.2.4, которые находят многие (но не все) хорошие локальные выравнивания в порядке уменьшения сходства, и останавливаются, как только будет найдена пара без перекрытия. При каких обстоятельствах этот метод может пропустить лучшую неперекрывающуюся пару?

Сравнение многих строк — Святой Грааль^{*)}

В этой главе мы начнем обсуждение *множественного сравнения строк*, одного из самых важных методологических вопросов и области наиболее активных исследований в современном биологическом анализе последовательностей. Сначала мы обсудим значимость множественного сравнения строк для молекулярной биологии. Затем займемся множественным *выравниванием* строк как одним общим способом, который формализует множественное сравнение строк. Мы дадим точные постановки задачи множественного выравнивания в трех вариантах и подробно рассмотрим те алгоритмы, с помощью которых эти задачи можно действительно атаковать. Другие подходы в этой главе только затрагиваются; дополнительные аспекты множественного выравнивания будут обсуждаться в части IV.

14.1. Зачем нужно множественное сравнение строк?

Задача множественного сравнения строк может сначала показаться компьютерщику, обобщением ради обобщения — “две строки хорошо, а четыре лучше”. Но в контексте молекулярной биологии множественное сравнение строк (строк ДНК, РНК или белков) — значительно больше, чем техническое упражнение. Это наиболее острый инструмент для *выделения и представления* биологически важных, хотя незначительных или сильно разбросанных, проявлений общности в наборе строк. Слабые

^{*)} В средневековых рыцарских легендах Святым Граалем назывался “тайный сосуд, ради приближения к которому и приобщения его благим действиям рыцари совершают свои подвиги... Некоторая неясность, что же такое Г. — конструктивно необходимая черта этого образа: Г. — это табуированная тайна, невидимая для недостойных, но и достойным являющаяся то так, то иначе, с той или иной мерой «прикосновенности»” (*Мифы народов мира*. М.: Сов. энцикл., 1980. Т. 1. С. 317). — Прим. перев.

	vvvvv*	vvvv	vvvv	vvvv	vvvv	vvvv	vvvv	
HUMA	VLSPADTKVKAHGKVGAHAEYGAELERMFLSPTKTYFPHF DLSH	GSAQKGHGKVADALTNAV	PNALSAISLDAHKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	AHYDDM	PNALSAISLDAHKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	GHDMM	PNALSAISLDAHKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	
HAOR	MLTDAEKETVTLKGKAHGEGGAELERLFIAPPTKTYFSHF DLSH	GSAQIKHGKGKVADALSTAAK	AGAKSLSPSLHAKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	MHDDDI	AGAKSLSPSLHAKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	AHDNL	AGAKSLSPSLHAKLRLVDWNFKLSSCLVTLAHLPAEFTPAWASLDKFASVSTVLTSKYR	
HADK	VLSADDTWKVFSKGHGEEYGTLEMFLAYPAKTYFPHF DLSH	GSAQIKHGKGKVAMALTEAV	KOTFATLSELCHDKLWDPEFRLLGWUVCVLAHFFKEFTPPVUAVYQKVWAGMALAHKHN	KNLDDL	KETFAKSLSELHDODLVDPEFRLLGWUVCVLAHFFKEFTPPVUAVYQKVWAGMALAHKHN	KHDN1	KETFAKSLSELHDODLVDPEFRLLGWUVCVLAHFFKEFTPPVUAVYQKVWAGMALAHKHN	
HBIU	VHLTEPEKSVTALHGKV NYDEGEALGRLLWVPTQRFFESFGDLSTPDAWNGPKVKAHGKVGLFSDGL	GSAGGGESAVTHLGKV NINEGGEGALGRLLWVPTQRFFEGDLSAGAGVGPKVKHGKVGLTSFGDL	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKGGHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKGGHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	
HBOR	VHLSGGEASVTHLGKV NYDEGEALGRLLWVPTQRFFEGDLSAGAGVGPKVKHGKVGLTSFGDL	YHTEFEKQLITLGKV NYADGEGALARLLWVPTQRFFESFGDLSTPDAWNGPKVKAHGKVGLTSFGDL	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG
HBKX	YHTEFEKQLITLGKV NYADGEGALARLLWVPTQRFFESFGDLSTPDAWNGPKVKAHGKVGLTSFGDL	YHTEFEKQLITLGKV NYADGEGALARLLWVPTQRFFESFGDLSTPDAWNGPKVKAHGKVGLTSFGDL	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG	KKKHHH	EAEKPLASHTAKHPKYLEFISECIIWVLSKHPDFGDAGAMMKALEFRDMASYKELFGQG
HYUH	GLSGDGERQLVNHGKVEADIPGHGAEVLRFLFGHPELETFDJKFKHKSEDEMASEDJKHGKTYLTALGTEL	GLSGDGENQLVNHGKVEADIPGHGAEVLRFLFGHPELETFDJKFKHKSEDEMASEDJKHGKTYLTALGTEL	1610B SPLTDEASVASSK AVSHHEWEILANFAYDQKKSQAFAGKLSIQTGAKATHTRIVSEJTA15GNTSMANV	NSLYSKLGDNKAGKGSAAKJFGEFRVLYQIAWNS	NSLYSKLGDNKAGKGSAAKJFGEFRVLYQIAWNS	ENMPAT	NSLYSKLGDNKAGKGSAAKJFGEFRVLYQIAWNS	
HYOR	GLSGDGENQLVNHGKVEADIPGHGAEVLRFLFGHPELETFDJKFKHKSEDEMASEDJKHGKTYLTALGTEL	GLSGDGENQLVNHGKVEADIPGHGAEVLRFLFGHPELETFDJKFKHKSEDEMASEDJKHGKTYLTALGTEL	1610B SPLTDEASVASSK AVSHHEWEILANFAYDQKKSQAFAGKLSIQTGAKATHTRIVSEJTA15GNTSMANV	TELROQHAWDINTLKLSIG LNKDTPRHEWNGALLGTTIKEIKENSDMRQNTAEYQWVATKMEKE	TELROQHAWDINTLKLSIG LNKDTPRHEWNGALLGTTIKEIKENSDMRQNTAEYQWVATKMEKE	1QELENGAVASATLKSLSGVHSVSKVDA HPRVUEALKTKEVGBKISEELNTNTIADELAIIKKEOMAA	1QELENGAVASATLKSLSGVHSVSKVDA HPRVUEALKTKEVGBKISEELNTNTIADELAIIKKEOMAA	
6671B	MUDQTINIKATPUKEHENBTITTFKNUFAKHPFPRPF	DNGRE	SLEAKKALANTVMAAMN	L RANKKANKIC QASBUAHHYUQGEELSAIKEVBMATDOLWKGKUWTFWVENDUWAE	L RANKKANKIC QASBUAHHYUQGEELSAIKEVBMATDOLWKGKUWTFWVENDUWAE		L RANKKANKIC QASBUAHHYUQGEELSAIKEVBMATDOLWKGKUWTFWVENDUWAE	

Рис. 14.1. Множественное выравнивание нескольких аминокислотных последовательностей глобиновых белков, заимствованное с модификацией из статьи Мак-Клуга, Вейси и Фитча [316]. Аббревиатуры слева показывают организмы, из которых выделены эти последовательности. Столбцы выравнивания, содержащие высокую концентрацию сходных остатков в участках известной вторичной структуры, помечены буквой *v*, а столбцы с одинаковыми остатками — звездочкой. Два остатка считаются родственными (сходными), если они принадлежат одному и тому же классу следующего разбиения: (*F, Y*), (*M, I, V*), (*A, G*), (*T, S*), (*Q, N*), (*K, R*) и (*E, D*)

проявления общности могут показать историю эволюции, важные консервативные мотивы или символы в ДНК или белке, общую двух- и трехмерную молекулярную структуру или подсказать общую биологическую функцию строк. Они используются также для характеристики семейств или суперсемейств белков. Затем полученные характеристики применяются при поиске в базах данных для опознания других потенциальных членов того же семейства. Так как многие значимые проявления общности слабо выражены или сильно разбросаны, они могут отсутствовать при сравнении только двух строк и стать очевидными, когда сравнивается набор родственных строк. В качестве примера см. рис. 14.1.

Основная процедура множественного сравнения строк предполагает *множественное выравнивание*. Главные его алгоритмические вопросы будут обсуждаться позднее, но для того, чтобы представить некоторые его применения, дадим определение.

Определение. Глобальное множественное выравнивание $k > 2$ строк $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ является естественным обобщением выравнивания для двух строк. Выбранные пробелы вставляются в каждую из k строк (в том числе в любом из концов) таким образом, чтобы получающиеся строки имели одинаковую длину, обозначаемую через l . Затем эти строки выравниваются в k строчек по l столбцов так, что каждый символ или пробел каждой строки оказывается в единственном столбце.

Например, см. рис. 14.1 и 14.2.

<i>a</i>	<i>b</i>	<i>c</i>	<i>_</i>	<i>a</i>
<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>a</i>
<i>a</i>	<i>c</i>	<i>c</i>	<i>b</i>	<i>_</i>
<i>c</i>	<i>b</i>	<i>_</i>	<i>b</i>	<i>c</i>

Рис. 14.2. Множественное выравнивание, которое будет использовано для создания профиля

Множественное сравнение может быть обратным к сравнению двух строк

С одной стороны, множественное сравнение строк является естественным обобщением сравнения двух строк (с помощью выравнивания), когда интерес нацелен на слабо сохраненные, а не на сильно сохраненные образцы. Цитируем Артура Леска: “Одна или две гомологические последовательности шепчут... полное множественное выравнивание кричит во весь голос” [233].

Но с другой стороны, множественное сравнение строк используется принципиально иначе, чем сравнение двух строк. Его можно применять в задачах биологического вывода, *обратных* задачам, к которым адресуется сравнение двух строк. В контексте поиска в базах данных двухстрочное выравнивание находит строки, имеющие общие подобразцы (подстроки или подпоследовательности), но не обязательно биологически родственные, судя по тому, что о них известно. Действительно, самое большое значение поиск в базах данных получает от использования *ясно видных* проявлений сходства строк для распознания биологических связей, о которых не подозревали.

Обратная задача — идти от известных биологических связей к неизвестным проявлениям сходства подстрок или подпоследовательностей. Это направление использует множественное сравнение для выявления неизвестных консервативных подобразцов из множества строк, биологическое родство которых уже известно.

14.1.1. Биологическая основа множественного сравнения строк

Напомним *первый факт биологического сравнения последовательностей* из главы 10 (с. 265). Он подчеркивает эффективность сравнения пар последовательностей и поиска в биологических базах данных. Но при всей мощи этого *первого факта* он не охватывает всех биологических закономерностей из сравнения последовательностей. Есть еще один факт, который вынуждает заниматься *множественным сравнением строк*.

Второй факт сравнения биологических последовательностей. Молекулярные строки, родственные эволюционно и функционально, могут *значительно различаться* на протяжении большей части строки и тем не менее сохранять те же трехмерные структуры, или двумерные подструктуры (мотивы, домены), или активные сайты, или те же либо родственные дисперсные остатки (ДНК или аминокислоты).

Этот *второй факт* уже участвовал неявно в обсуждении локальных выравниваний (п. 11.7). Что добавляется здесь, так это градация выраженности. Две строки, определяющие “один и тот же” белок в различных биологических видах, могут различаться настолько, что наблюдаемые сходства можно приписать случаю. Обычно в большом наборе эволюционно родственных белков наиболее сохраняющимся свойством является вторичная (трехмерная) структура. Сохранение функции встречается реже, а сохранение аминокислотной последовательности — реже всего. “Одно из самых интригующих наблюдений из недавних анализов белковых структур заключается в том, что сходная складчатость повторяется даже при отсутствии сходства первичных последовательностей” [428]. (Этот момент отмечался в цитате из Коэна на с. 268.)

Например, гемоглобин — это почти универсальный белок, содержащий четыре цепи примерно по 140 аминокислот в каждой. Он синтезируется в столь различных организмах, как млекопитающие и насекомые, и функционирует в основном одинаково (связывает и переносит кислород). Однако за то время, когда происходила дивергенция насекомых и беспозвоночных (около 600 миллионов лет тому назад), в аминокислотных последовательностях гемоглобинов у насекомых и беспозвоночных появились многочисленные замены аминокислот, — в среднем около 100 мутаций*) [11]. Попарное выравнивание цепей гемоглобина от двух млекопитающих может привести к предположению о функциональном сходстве этих белков (последовательности у человека и шимпанзе просто одинаковы), но выравнивание двух строк от млекопитающего и насекомого даст очень мало информации. Более того, многие белки мутируют еще быстрее, чем гемоглобин, что еще больше усложняет задачу.

*) Это число не равно числу позиций, в которых наблюдаются аминокислотные различия в гемоглобинах млекопитающих и насекомых, поскольку в каждой конкретной позиции могло произойти больше одной мутации.

В качестве еще одного, более свежего, примера можно привести трехмерную структуру части молекулы клеточной адгезии, называемой *E-кадерином*, которая была приведена в [354, 451]. Эта структура неожиданно оказалась “замечательно похожей на складчатую структуру Ig (хотя гомологии последовательностей не было)” [451]. Ig является общей складчатой структурой, впервые обнаруженной в суперсемействе иммуноглобулинов (антител), но впоследствии найденная и во многих других белках. Термин “гомология” здесь следует интерпретировать как “сходство”, а “отсутствие гомологии” — как “сходство не большее, чем ожидается между случайными строками”. Таким образом, найденная складчатая структура является консервативной физической структурой, сходной у Ig и Е-кадерина, несмотря на отсутствие сходства их последовательностей. Относящееся к тому же вопросу рассмотрение другого белка в [451] (рецептора гормона роста) иллюстрирует тот же момент: “Ранее было обнаружено, что в структуре рецептора гормона роста имеются домены, относящиеся к этому эволюционно последовательному мотиву (Ig); в этом случае также не существует гомологий последовательностей”.

Способность многих белков сохранять структуру и функцию несмотря на массовые подстановки аминокислот — эмпирический факт, хотя и может казаться, что он противоречит интуиции. Интуитивно можно ожидать, *) что каждая деталь и в строке аминокислоты, и в ее кодировке в ДНК столь существенна, что малейшие нарушения в них разрушают структуру или функцию белка. Действительно, многие генетические болезни вызываются именно такими малыми мутациями. Часто даже единичная замена *конкретной* аминокислоты, которая может быть вызвана мутацией одного нуклеотида в ДНК, приводит к серьезному заболеванию, такому как серповидноклеточная анемия, рак (онкоген *ras*, в частности, активируется единичной мутацией нуклеотида [403, 110]) или прионовая инфекция, например болезнь *коровьего бешенства* [377]. Но в целом замены аминокислот на другие во множестве позиций белковых молекул, происходящие в ходе эволюции, не являются разрушительными. Таким образом, хотя *первый факт* биологического анализа последовательностей, утверждающий, что сходство последовательностей влечет структурное или функциональное сходство, очень силен и широко эксплуатируется, обратное ему утверждение просто неверно.

Распространение допустимых мутаций в структурно или функционально сохраняемых молекулах может быть таким, что попарное сравнение строк даст мало информации о критических консервативных образцах или критических аминокислотах. Консервативных аминокислот может быть так мало и они могут быть так широко разбросаны, что наилучшее выравнивание двух родственных белковых строк будет статистически неотличимо от наилучшего выравнивания двух случайных аминокислотных строк. Проблема может прийти и с другого конца. При сравнении двух белков близко родственных видов можно не найти биологически важных консервативных образцов из-за того, что критические совпадения теряются в большом числе совпадений, появившихся в ходе общей эволюции. Таким образом, при выполнении попарных сравнений строк для нахождения критических общих образцов самое интересное — отобрать виды, у которых уровень различия “наиболее информативен” (это трудная и неясная задача).

*) И некоторые ранние работы по эволюции белков даже формулируют это ожидание как “факт”. Например, см. [121].

Множественное сравнение строк (часто через множественное выравнивание) является естественным ответом на перечисленные выше вопросы. При множественном сравнении строк не так существенно (хотя и по-прежнему полезно) отбирать виды, у которых уровень различия “наиболее информативен”. Зачастую биологически важные образцы, которые не могут быть выявлены при сравнении только двух строк, становятся очевидными при одновременном сравнении многих строк. Более того, при множественном выравнивании иногда удается расставить набор родственных строк (часто в дереве) так, что видны *непрерывные* изменения вдоль пути, соединяющего две крайние строки, сходство между которыми видно гораздо меньше. В частности, множественные выравнивания такого рода очень полезны в выяснении истории эволюции.

14.2. Три “крупномасштабных” биологических применения множественного сравнения

Множественное сравнение последовательностей имеет важнейшее значение для трех проблем, которые относятся к “крупномасштабным” биологическим вопросам: составление семейств и суперсемейств белков, идентификация и представление свойств консервативных последовательностей в ДНК и белках, которые коррелируют с их структурой и/или функцией, и восстановление истории эволюции (обычно в форме дерева) на основании последовательностей ДНК или белка.*)

Хотя такая систематика и полезна, она дезориентирует нас, поскольку эти три проблемы зачастую взаимосвязаны. Объединяющая их идея заключается в том, что общая структура или функция строк обычно является результатом общей эволюции. Так что, даже если нет явного интереса к установлению хода эволюции и семейство белков составлено на основе их общей функции или структуры либо если консервативные сайты разыскиваются для прояснения механики структуры или функции, в основе всего этого лежит эволюция. Поэтому знание хода эволюции может помочь в решении первых двух вопросов, и напротив, их решение помогает понять историю эволюции.

В следующих пунктах мы обсудим первые два из этих трех “крупномасштабных” применений множественного сравнения строк. Третье применение, касающееся хода эволюции, будет обсуждаться в пп. 14.8 и 14.10.2, а затем снова в главе 17 части IV.

14.3. Представление семейств и суперсемейств

Часто набор строк (семейство) определяется по биологическому родству, и хочется найти сходные черты последовательностей, которые характеризуют или представляют это семейство. Общие консервативные сходства последовательностей в семействе могут дать ключ к лучшему пониманию функции или структуры членов семейства. Более того, представление семейства может оказаться полезным при идентификации

*) В добавок к этим трем главным приложениям необходимость во множественном сравнении строк (и особенно выравнивании) часто возникает в специфических лабораторных протоколах, таких как создание праймера PCR, клонирование гена и дробовая сборка последовательности. Последнее из этих трех приложений будет рассматриваться в пп. 16.14 и 16.15.

новых его потенциальных членов, когда исключаются строки, не характерные для семейства. Белковые семейства (или суперсемейства) дают характерные примеры, в которых семейство — это набор белков, имеющих сходные биологические функции, или сходную двух- либо трехмерную структуру, или известную общую эволюционную историю.*¹) Конкретные примеры семейств белков включают знаменитые *глобины* (гемоглобины, миоглобины) и *иммуноглобины (антитела)*. Следующая цитата из Ч. Чотия подчеркивает центральное место изучения семейств в молекулярной биологии:

“Белки группируются в семейства, члены которых происходят от общего предка и имеют поэтому общие складчатости; они имеют также общие последовательности и функции. Эта классификация занимает центральное место в нашем понимании развития жизни и делает определение таких семейств одной из главных проблем молекулярной биологии” [101].

Слово “складчатости” означает “дву- и трехмерные (под)структуры”.

Представление семейств очень полезно

При установлении семейства или суперсемейства, которому принадлежит вновь идентифицированный белок, исследователь получает в награду огромную информацию относительно его возможной физической структуры или биологической функции. Этот подход очень эффективен, так как по имеющимся оценкам около 100 000 человеческих белков можно сгруппировать всего в тысячу значимых семейств белков [101]. По другим оценкам еще меньше — всего в сотню.

При определении того, принадлежит ли белок данному семейству, и при поиске новых членов семейства белков в базе данных обычно гораздо эффективнее выравнивать испытуемую строку с некоторым *представлением* известных членов семейства, а не с отдельными его членами.^{**)} Естественно спросить, какие представления наиболее полезны для идентификации новых членов семейства (и исключения нечленов) и как получать хорошие представления. Это открытая область, в которой ведутся активные исследования, но часто в ней используется своего рода множественное выравнивание белковых строк семейства, и представление семейства извлекается из

*¹) В биологической литературе имеется опасная двусмысличество в использовании слов “семейство” и “суперсемейство”. Для биолога-практика семейство (суперсемейство) означает набор белков, родственных по структуре, функции или известной истории эволюции. Задача заключается в нахождении достаточного сходства последовательностей, которое бы это семейство характеризовало. Это понимание явно выражено в цитате из Чотия, и именно его мы развиваем в книге. Однако иногда термины “семейство” и “суперсемейство” используются совершенно противоположным образом. Дейхоф предлагает относить белки к семействам и суперсемействам исключительно по степени сходства их последовательностей. В этой классификации не требуется доказательств какого-либо родства членов семейства, и задача заключается в том, чтобы определить, в каком биологическом соотношении находятся (и находятся ли) члены семейства. Конечно, общая история эволюции считается объяснением сходства последовательностей, а недавняя история, грубо говоря, обеспечивает общую структуру и функцию. С этой точки зрения оба определения семейства приводят к одним и тем же представлениям и к двойственному использованию термина. Но это теоретическое различие между двумя использованиеми термина “семейство” не так удобно применять на практике, так что при чтении биологической литературы нужно тщательно определять, в каком смысле употреблен термин. Известна ли биологическая общность членов семейства или только сходство последовательностей?

**) Это представление оказывается особенно эффективным, когда получается в итеративном процессе. Начинают с представления, основанного на нескольких известных членах семейства, которое используется для нахождения дополнительных членов в базе данных. Расширенный ими набор участвует затем в создании нового представления, которое в свою очередь используется для поиска и т.д.

него. В работе Мак-Клура, Вэйси и Фитча [316] сравниваются различные методы множественного выравнивания, подходящие для решения все того же вопроса: насколько успешно различные методы находят и определяют принадлежность образцов белковым семействам?

Три общих типа представлений семейств

Есть три общих типа представлений семейств, которые получаются из множественного сравнения строк: *профильные* представления, представления *консенсусной последовательностью* и *сигнатурные* представления. Первые два типа представлений базируются на множественном выравнивании и рассматриваются в пп. 14.3.1 и 14.7. Сигнатурные представления не всегда основываются на множественном выравнивании, они обсуждаются в п. 14.3.2.

14.3.1. Представления семейств и выравнивания с профилями

Определение. Пусть задано множественное выравнивание набора строк. *Профиль* для этого выравнивания задает для каждого столбца *частоту* появления каждого символа в этом столбце. Иногда в биологической литературе профиль называется также *матрицей весов*. Профили были введены и применены в молекулярной биологии Грибковым, Эйзенбергом и их соавторами [194, 193, 192].

Например, рассмотрим множественное выравнивание четырех строк, показанное на рис. 14.2. Профиль для него имеет пять столбцов, как показано на рис. 14.3.

	C1	C2	C3	C4	C5
a	.75		.25		.50
b		.75		.75	
c	.25	.25	.50		.25
-			.25	.25	.25

Рис. 14.3. Профиль для множественного выравнивания с предыдущего рисунка

Часто значения в профиле преобразуются в *логарифмы относительных различий*. Это значит, что если $p(y, j)$ — частота появления символа y в столбце j , а $p(y)$ — частота появления y во всех выравниваемых последовательностях, то в качестве элемента профиля или весовой матрицы на месте y, j используется $\log p(y, j)/p(y)$. Осмысленная часть этой формулы — отношение. Логарифм взят для удобства (причем годится логарифм по любому основанию). Для наших целей, когда мы используем термин “профиль”, не будет разницы, идет ли речь о самой частоте или о логарифме относительного различия.

Выравнивание строки по профилю

Пусть задан профиль \mathcal{P} и новая строка S . Возникает вопрос, насколько S или некоторая ее подстрока соответствует профилю \mathcal{P} . Так как пробел является в профиле законным символом, соответствие S профилю \mathcal{P} должно допускать включение в S пробелов, и следовательно, вопрос о соответствии S профилю \mathcal{P} естественно

<i>a</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>c</i>
1	2	3	4	5

Рис. 14.4. Выравнивание строки *aabbc* по столбцам предыдущего выравнивания

формализуется как небольшое обобщение чисто строкового выравнивания. Рассмотрим строку *C* из позиций столбцов профиля и выровнем *S* около *C*, вставляя пробелы в *S* и *C* и т.д., как в чисто строковом выравнивании. Например, выравнивание строки *aabbc* и вышеприведенного профиля показано на рис. 14.4. Ключевыми вопросами здесь являются оценка выравнивания строки и профиля и вычисление оптимального выравнивания при заданной схеме оценки.

Как оценивать выравнивание строки и профиля

Предположим, что известна схема оценки чисто строкового выравнивания с алфавитными весами. При использовании этой схемы обычный подход к оценке выравнивания символа из *S* со столбцом из *C* заключается в том, что символ строки выравнивается с каждым символом столбца и вычисляется взвешенная сумма оценок с частотами символов столбца в качестве весов. Оценка полного выравнивания определяется как сумма оценок столбцов. Например, предположим, что схема алфавитных весов дает оценку 2 для выравнивания (*a*, *a*), оценку -1 — для выравнивания (*a*, *b*) или (*a*, *_*) и оценку -3 — для выравнивания (*a*, *c*). Тогда первый столбец приведенного выравнивания строки *aabbc* с профилем *C* получает оценку $0.75 \times 2 - 0.25 \times 3$. Оценка второго столбца оказывается равной -1, так как символ *a* соответствует пробелу.

Как оптимально выровнять строку и профиль

Мы трактуем выравнивание в терминах максимального сходства и вычисляем оптимальное выравнивание с помощью динамического программирования. Все очень похоже на тот способ, с помощью которого отыскивалось сходство чистых строк (см. п. 11.6.1). Однако обозначения немного отличаются. Напомним, что для двух символов *x* и *y* через *s(x, y)* обозначен алфавитный вес выравнивания *x* и *y* в чисто строковой задаче выравнивания.

Определение. Для символа *y* и столбца профиля *j* обозначим через *p(y, j)* частоту появления *y* в этом столбце, а через *S(x, j)* сумму $\sum_y [s(x, y) \times p(y, j)]$, оценку выравнивания *x* и столбца *j*.

Определение. Пусть *V(i, j)* обозначает значение оптимального выравнивания подстроки *S[1..i]* с первыми *j* столбцами *C*.

При таких обозначениях рекуррентные соотношения для вычисления оптимального выравнивания строки и профиля таковы:

$$V(0, j) = \sum_{k \leq j} S(_, k)$$

$$V(i, 0) = \sum_{k \leq i} S(S_1(k), _).$$

Для строго положительных i и j общее соотношение таково:

$$V(i, i) = \max \left\{ \begin{array}{l} V(i - 1, j - 1) + S(S_1(i), j), \\ V(i - 1, j) + s(S_1(i), _), \\ V(i, j - 1) + S(_, j) \end{array} \right\}.$$

Корректность этих соотношений очевидна и доказательство оставляется читателю. Как обычно в динамическом программировании, само выравнивание получается в фазе обратного просмотра, после вычислений по соотношениям. Время этого расчета равно $O(\sigma nm)$, где n — длина S , а σ — размер алфавита. Проверка временной оценки также оставляется читателю. Отсюда видно, что выравнивание строки с профилем стбит немного дороже, чем выравнивание двух строк. Одно специфическое использование выравнивания строк с профилями рассматривается в первой части п. 14.10.2. Более глубокое обсуждение профилей в вычислительной биологии см. в [193].

Выравнивание профиля с профилем

Еще одно использование профили находят при сравнении одного набора белков с другим. В этом случае сравниваются профили этих наборов. Выравнивания такого рода иногда используются как подзадача в некоторых методах множественного выравнивания (см. п. 14.10.1). Формализация оптимального выравнивания двух профилей и выписывание требуемых рекуррентных соотношений выполняются непосредственно. Это также оставляется на упражнения.

14.3.2. Сигнатурные представления семейств

Обратимся теперь ко второму главному представлению семейств — *сигнатуре* или *мотивной сигнатуре*. Главные коллекции сигнатур в белках — это база данных PROSITE и полученная из нее база данных BLOCKS. О них пойдет речь в пл. 15.8 и 15.9. Здесь же мы покажем сигнатурный подход и его обычное использование на примере исследования, выполненного Горбаленей и Коонином и опубликованного в статье “Геликазы: сравнение аминокислотных последовательностей и структурно-функциональные соотношения” [185].

Сигнатуры для геликазных белков

ДНК является двунитевой молекулой, которая имеет форму двойной спирали — два (сахарных) остова нитей скручиваются вокруг друг друга и соединяются внутри (комплементарными)арами нуклеотидов. Так как сама скелетная основа одинакова по всей длине молекулы, информация, сохраняемая в ДНК (например, кодировка белков в генах), содержится в последовательности нуклеотидов внутри двойной спирали. Чтобы извлечь эту информацию, спираль нужно слегка “раскрутить” и “открыть”. Геликазы — это ферменты, которые помогают раскрутить двойную спираль ДНК так, чтобы ее можно было прочесть для дупликации, транскрипции, рекомбинации или репарации. Считается, что недавно открытый ген синдрома Вернера кодирует белок геликазу [365]. Индивидуум, пораженный синдромом Вернера, стареет гораздо быстрее, чем здоровый человек. Получены аминокислотные последовательности нескольких сотен геликаз от различных организмов.

“Большая часть имеющейся информации о структуре и возможных функциях геликаз была получена используя компьютер сравнительным анализом их аминокислотных последовательностей. Этот подход привел к обрисовке мотивов и образцов, которые сохраняются в различных подмножествах геликаз” [185].

Такие мотивы применялись для поиска ранее неизвестных геликаз в базах данных белков и ДНК (переведенных в аминокислотный алфавит). Как утверждается в [185], “эти мотивы широко использовались для предсказания новых геликаз...”.

В нашей терминологии геликазы образуют семейство белков, объединенных общей биологической функцией — все они помогают раскручивать ДНК. Выделяя и представляя общности в последовательности, можно лучше понять механику работы геликаз и идентифицировать новые неизвестные геликазы. На уровне последовательностей все известные геликазы содержат некоторые консервативные “последовательные сигнатуры” или мотивы,* но эти вездесущие мотивы могут не определять уникальные свойства геликаз. Поэтому они не могут быть признаком для различения геликаз и негеликазовых белков, хотя и в состоянии идентифицировать функционально наиболее важные аминокислоты.

Задачей в работе [185] была оценка утверждения, что “дополнительные мотивы являются уникальными идентификаторами различных групп (предполагаемых) геликаз”. Горбаленя и Коонин нашли, что “подмножества геликаз обладают дополнительными консервативными мотивами...”. Некоторые из этих мотивов можно использовать в качестве надежных идентификаторов соответствующих групп геликаз при поиске в базах данных. Две самые большие группы**) имеют похожие образцы из семи консервативных мотивов последовательностей, иногда разделенных длинными мало консервативными прокладками. Приведем один пример:

$$[&H][\&A]D[DE]x_n[TSN]x_4[QK]Gx_7[\&A],$$

здесь & обозначает любую аминокислоту из группы (*I, L, V, M, F, Y, W*), *x* — любую аминокислоту, а альтернативные аминокислоты заключены в квадратные скобки. Индекс при *x* задает разрешенную длину подстроки; *n* показывает, что допустима любая длина. Ясно, что мотив в этом контексте является не какой-то сохраняемой подструктурой, а скорее *регулярным выражением*. Следовательно, входжение образца в строку можно искать точным поиском по шаблону регулярного выражения (см. п. 3.6).

Геликазные мотивы могут быть функциональными единицами

Мотивы бывают не только хорошими показателями принадлежности семейству. Они могут быть на самом деле важными функциональными единицами белков: “...распознание мотивов, которые сохраняются в широких группах геликаз... привело к очевидной гипотезе, что эти мотивы содержат аминокислотные остатки, наиболее важные для функций геликаз” [185]. Эти гипотезы можно проверить *направленным мутагенезом*, при котором так изменяют выбранную позицию в генном коде геликазы, что это влияет только на определенную ее аминокислоту. После того как модифицированная ДНК вводится обратно в организм и модифицированная

* Имеется подсемейство геликаз, известных как геликазы DEAD, так как все они содержат эту четырехбуквенную подстроку.

**) К сожалению для непрофессионального читателя, они часто называют группы, связанные только сходством последовательностей, “суперсемействами” и “семействами”, следуя в этом традиции Дейхоф, а часто также “группами” или “подмножествами”, что соответствует нашей терминологии.

геликаза экспрессируется, экспериментатор может увидеть, отражаются ли результаты этого изменения в изменении функциональности фермента. Эксперименты такого типа позволяют определить роль аминокислот, входящих в консервативные мотивы в функционировании геликазы.

Еще одна хорошая иллюстрация сигнатурного подхода, а также идентификации и использования сигнатур, содержится в статье Посфаи и др. [374].

14.4. Множественное сравнение последовательностей для выводов о структуре

Как сказано в недавней работе по множественному выравниванию, “одним из наиболее успешных приложений множественного выравнивания последовательностей было повышение точности предсказания вторичной структуры” [306].

Проиллюстрируем теперь второе “крупномасштабное” использование множественного выравнивания, а именно получение биомолекулярной структуры. Мы сосредоточимся на специфической задаче изучения вторичной структуры тРНК. Другая иллюстрация использования множественного выравнивания в предсказании структуры будет рассматриваться в п. 14.10.2 (с. 438). Еще одно применение выравнивания последовательностей для выводов о структурах развивалось в упражнении 36 главы 11.

Задача вычисления вторичной (планарной) структуры молекулы тРНК по ее первичной последовательности была введена в упражнениях 41–44 в конце главы 11.* Там подход динамического программирования использовался для нахождения максимальной мощности набора *вложенных паросочетаний* нуклеотидов. Напомним, что вложенное паросочетание не допускает пересечения линий в его (планарном) представлении.

Совсем недавно Кэри и Стормо [89] для вычисления вторичной структуры тРНК обратились к методу, который основан на взвешенных оценках совпадений в произвольных (не обязательно двудольных) графах. При этом не вводятся явно ограничения непересечения, которые были так важны в подходе с вложенными паросочетаниями (и в общем согласуется с известными структурами тРНК).

Совпадения, допускающие пересечения

В методе, предложенном в [89], каждый нуклеотид молекулы тРНК представляется вершиной графа, в котором каждые две вершины соединяются ребром и вес каждого ребра представляет “хорошесть” (мы не хотим здесь вдаваться в технические подробности) выбора пары нуклеотидов, соединенных этим ребром. Метод находит наибольший вес *паросочетания* в графе. Паросочетание в графе — это такое подмножество его ребер E' , что любая вершина графа соприкасается не больше чем с одним из ребер E' . Весом паросочетания считается сумма весов ребер из E' . Ясно, что паросочетание в графе задает паросочетание (pairing) нуклеотидов (как оно определено в упражнении 41 главы 11).

*) Читателю следует просмотреть заново основные определения и биологические факты в тексте упражнений 41 и 42 главы 11.

Нахождение в графе паросочетания максимального веса является классической комбинаторной задачей, которая имеет относительно эффективное решение [294], и поэтому в [89] проверялось, насколько предлагаемый подход к разбиению нуклеотидов на пары пригоден для определения вторичной структуры молекул тРНК. Так как этот подход не запрещает перекрещивающихся пар (именуемых псевдоузлами), он может работать при распознании вторичной структуры истинных тРНК лучше, чем подход со вложенными парами. Однако так как вложенные пары являются общим правилом, а псевдоузлы исключением, то можно подумать, что метод, основанный на паросочетаниях общего типа, даст слишком много пересечений по сравнению с тем, что можно ожидать при нахождении действительной вторичной структуры. Таким образом, при обосновании этого подхода нужно задаться вопросом, какая структурная информация закодирована в весах ребер. Здесь-то в нашу картину и входит *множественное выравнивание строк*.

Веса ребер

Не стараясь быть точными (и даже вполне правильными) в деталях, мы можем получить веса ребер следующим образом. Большое число родственных последовательностей тРНК, включая рассматриваемую, множественно выравниваются (как именно, пусть останется за пределами обсуждения), так что каждый нуклеотид в каждой последовательности попадает в некоторый столбец выравнивания. Затем проверяется каждая пара столбцов (i, j) , с тем чтобы найти протяженность, на которой изменения символа в столбце i коррелируют с изменениями символа в столбце j (мы снова оставляем в стороне точное определение этой корреляции). Например, если бы половина последовательностей имела A в столбце 16 и U в столбце 92, а другая половина имела C в 16 и G в 92, то столбцы 16 и 92 считались бы сильно коррелированными. Сильная коррелированность объясняется тем, что два нуклеотида, занимающие в тРНК позиции 16 и 92 (определенные выравниванием), являются спаренными друг с другом, даром что конкретные нуклеотиды в этих позициях могут муттировать и быть различными в различных последовательностях. Дело в том, что для сохранения структурного единства точечные мутации в парных нуклеотидах должны коррелироваться.*¹ С другой стороны, если изменения в столбце 16 были вполне независимы от изменений в столбце 92, то можно сделать вывод, что нуклеотиды в этих двух позициях вероятно не спарены. В первом случае вес ребра (16, 92) в графе должен быть большим, а во втором — низким. Конечно, два этих крайних случая служат лишь иллюстрацией; большая часть корреляционных данных выглядит более причудливо, и поэтому для решения об общем паросочетании нуклеотидов нужно выбирать паросочетание наибольшего веса.

Мы не уточняли, каким образом попарные корреляции столбцов переводятся в веса ребер [89], а лишь представили основные интуитивные соображения. Если подвести итог, то кажется, что множественное сравнение (выравнивание) строк последовательностей тРНК дает достаточно информации относительно вторичной структуры тРНК, так что паросочетание нуклеотидов, основанное на взвешенном паросочетании, “вполне” успешно в распознании подлинной вторичной структуры исходя лишь из информации о последовательностях.

*¹) Однако это объяснение не охватывает всех случаев высокой корреляции, так как имеются высоко коррелированные пары позиций, которые не соответствуют парным нуклеотидам в тРНК. Кроме того, этот метод плохо распознает позиции, в которых нуклеотид не мутирует.

14.5. Введение в вычисление множественных выравниваний строк

Мотивировав *множественное сравнение строк* предыдущими рассуждениями, перейдем к чисто техническим вопросам его вычисления. Определение *глобального множественного выравнивания* было дано на с. 405 (примеры см. на рис. 14.2 и 14.6, б). Естественно ввести также и локальное множественное выравнивание. Локальное выравнивание для двух строк было определено как глобальное выравнивание подстрок, аналогично можно определить и множественное локальное выравнивание.

Определение. Пусть задан набор из $k > 2$ строк $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$. *Локальное множественное выравнивание* \mathcal{S} получается выбором по одной подстроке S'_i из каждой строки $S_i \in \mathcal{S}$ и последующим глобальным выравниванием этих подстрок.

Все биологические обоснования предпочтительности локального выравнивания двух строк перед глобальным (см. п. 11.7) применимы и к множественным выравниваниям. Эти обоснования только усиливаются рассмотренным выше *вторым фактом* анализа биологических последовательностей. Однако наилучшие теоретические результаты (с точки зрения информатики) получены для глобальных множественных выравниваний — на них мы и будем делать ударение. Локальные множественные выравнивания вкратце обсуждаются в п. 14.10.3.

14.5.1. Как оценить множественные выравнивания

Хотя само понятие множественного выравнивания легко переносится с двух строк на много, оценка качества выравнивания обобщается не так легко. К настоящему времени еще не найдено целевой функции, которая бы так же хорошо подходила для множественного выравнивания, как (взвешенное) редакционное расстояние или сходство для выравнивания двух строк. В действительности, как мы увидим ниже, некоторые популярные методы нахождения множественного выравнивания обходятся без задания явной целевой функции. Качество этих методов оценивается по биологическому смыслу выравниваний, которые они находят, так что решающим становится понимание биологии оценивающим лицом.

Не будучи биологами, мы сосредоточимся на методах множественного выравнивания, которые используют явные целевые функции, хотя вкратце будут описаны и другие методы. В этой главе обсуждаются три типа целевых функций: *функции попарных сумм*, *консенсусные функции* и *функции дерева*. Несмотря на то что эти функции совершенно различны, приближенные алгоритмы, используемые для решения, очень похожи. В части IV мы обсудим, как задачи множественного выравнивания соотносятся с задачами построения деревьев, описывающих историю эволюции.

Определение. Пусть задано множественное выравнивание \mathcal{M} . *Индукционное парное выравнивание* двух строк S_i и S_j получается из \mathcal{M} удалением всех строк, кроме тех, которые получились из S_i и S_j . Значит, индуцированное выравнивание — это множественное выравнивание \mathcal{M} , ограниченное до S_i и S_j . Любая пара пробелов, стоящих в этом выравнивании напротив друг друга, при желании может быть удалена.

Определение. Оценка индуцированного парного выравнивания определяется по любой выбранной схеме оценки выравнивания двух строк в стандартном подходе. В качестве примера см. рис. 14.5.



Рис. 14.5. Множественное выравнивание трех строк \mathcal{M} показано выше горизонтальной линии. При использовании схемы попарных оценок $ms + id$, т. е. число несовпадений плюс число пробелов, три получающихся попарных выравнивания имеют оценки 4, 5 и 5 с полной SP-оценкой 14. Отметим, что пробел напротив пробела имеет нулевую оценку. Использование правила большинства дает для выравнивания \mathcal{M} консенсусную строку \mathcal{S}_m (см. определение в п. 14.7.2), которая показана ниже горизонтальной черты. Ее ошибка выравнивания равна семи

Отметим, что в определении этой “оценки” не уточняется, является ли она взвешенным расстоянием или сходством. Как и ранее, для локальных выравниваний естественнее использовать сходство. Большая часть этой главы относится к глобальному выравниванию, где алгоритмы и предстоящие теоремы требуют, чтобы оценка была взвешенным расстоянием.

14.6. Множественное выравнивание с целевой функцией типа суммы пар (*SP*)

Оценка типа суммы пар (*SP*)

Определение. Оценка множественного выравнивания \mathcal{M} суммой пар (*SP*) — это сумма оценок парных глобальных выравниваний, индуцированных выравниванием \mathcal{M} (см. рис. 14.5).

Хотя можно привести “разговорные” аргументы в пользу *SP*-оценки, теоретическое обоснование для нее (или какой-либо другой схемы оценивания множественного выравнивания) дать трудно. Однако с *SP*-оценкой легко работать, и это ценят исследователи, изучающие множественные выравнивания. Впервые *SP*-оценка была введена в [88] и впоследствии использовалась в [33, 334, 187]. Похожая оценка встречается в [153]. *SP*-оценка используется также в подзадаче в пакете множественного выравнивания MACAW [398], разработанном в Национальном институте здоровья (NIH), в Национальном центре биотехнической информации.

Задача *SP*-выравнивания. Вычислить глобальное множественное выравнивание \mathcal{M} с минимальной оценкой по суммам пар.

14.6.1. Точное решение задачи *SP*-выравнивания

Как и следует ожидать, *SP*-задачу можно решить точно (оптимально) с помощью динамического программирования [334]. К сожалению, если задано k строк, каждая из которых длиной, скажем, n , то ДП требует времени $\Theta(n^k)$ и, следовательно, целесообразно только для малого числа строк. Более того, доказано, что точная задача *SP*-выравнивания *NP*-полнна [454]. Поэтому мы рассмотрим рекуррентные соотношения ДП только для случая трех строк. Обобщение на любое большее число строк выполняется непосредственно, хотя доказано, что оно непрактично даже для пяти строк, длины которых составляют по несколько сотен символов (что типично для белков). Мы предложим также способ ускорения основного ДП-подхода, и этот способ несколько увеличит число строк, которые можно оптимально выровнять.

Определение. Пусть S_1 , S_2 и S_3 обозначают три строки, длины которых равны, соответственно, n_1 , n_2 и n_3 , а $D(i, j, k)$ — оптимальная *SP*-оценка выравнивания $S_1[1..i]$, $S_2[1..j]$ и $S_3[1..k]$. Оценки совпадения, несовпадения и пробела задаются, соответственно, переменными *smatch*, *smis* и *sspace*.

Таблица динамического программирования D , используемая для выравнивания трех строк, образует трехмерный куб. Каждая клетка (i, j, k) , которая не лежит на границе таблицы (т. е. не имеет равных нулю индексов), окружена семью соседями, которых нужно опрашивать при определении $D(i, j, k)$. Общие рекуррентные соотношения для вычисления стоимости неграничной клетки по духу схожи с соотношениями для двух строк, но технически немного сложнее. Эти соотношения закодированы в следующем псевдокоде.

Рекуррентные соотношения для неграничной клетки (i, j)

```

for  $i := 1$  to  $n_1$  do
    for  $j := 1$  to  $n_2$  do
        for  $k := 1$  to  $n_3$  do begin
            if ( $S_1(i) = S_2(j)$ ) then  $cij := smatch$ 
            else  $cij := smis$ ;
            if ( $S_1(i) = S_3(k)$ ) then  $cik := smatch$ 
            else  $cik := smis$ ;
            if ( $S_2(j) = S_3(k)$ ) then  $cjk := smatch$ 
            else  $cjk := smis$ ;
             $d1 := D(i - 1, j - 1, k - 1) + cij + cik + cjk;$ 
             $d2 := D(i - 1, j - 1, k) + cij + 2 * sspace;$ 
             $d3 := D(i - 1, j, k - 1) + cik + 2 * sspace;$ 
             $d4 := D(i, j - 1, k - 1) + cjk + 2 * sspace;$ 
             $d5 := D(i - 1, j, k) + 2 * sspace;$ 
             $d6 := D(i, j - 1, k) + 2 * sspace;$ 
             $d7 := D(i, j, k - 1) + 2 * sspace;$ 
             $D(i, j, k) := \min\{d1, d2, d3, d4, d5, d6, d7\};$ 
        end;
    
```

Осталось только определить, как вычисляются значения D для граничных клеток трех начальных граней таблицы (т. е. когда $i = 0$, $j = 0$ или $k = 0$). Для этого обозначим через $D_{1,2}(i, j)$ знакомое нам парное расстояние между подстроками $S_1[1..i]$

и $S_2[1..j]$ и аналогично через $D_{1,3}(i, k)$ и $D_{2,3}(j, k)$ — расстояния для пар S_1, S_3 и S_2, S_3 . Они вычисляются стандартным образом. Тогда

$$D(i, j, 0) = D_{1,2}(i, j) + (i + j) * sspace,$$

$$D(i, 0, k) = D_{1,3}(i, k) + (i + k) * sspace,$$

$$D(0, j, k) = D_{2,3}(j, k) + (j + k) * sspace,$$

и

$$D(0, 0, 0) = 0.$$

Проверка корректности соотношений, а также утверждения что они работают за время $O(n_1 n_2 n_3)$, оставляется читателю. Рекуррентные соотношения можно обобщить на случай алфавитно-зависимых оценок; этот вопрос также оставляется читателю.

Ускорение для точного решения

Карилло и Липман [88] предложили способ некоторого уменьшения работы, необходимой для практического нахождения оптимального *SP*-множественного выравнивания. Обобщение их идеи было реализовано в программе MSA [303], но само обобщение в статье не описано. Здесь мы объясним идею варианта ускорения, использованного в MSA. О дополнительных улучшениях MSA (в частности, о сокращении требуемой памяти) сообщалось в [197]. Мы снова ограничимся случаем трех строк.

Показанная выше программа множественного выравнивания служит примером использования рекуррентных соотношений в *обратном* направлении. В момент, когда алгоритм находит $D(i, j, k)$, программа смотрит назад для получения значений D в (не более) семи клетках, влияющих на искомое значение.

При альтернативном впередсмотрящем ДП (см. также п. 12.6.1), когда находится значение $D(i, j, k)$, оно посыпается *вперед* (не более чем) семи клеткам, значения D в которых могут зависеть от клетки (i, j, k) . Можно сказать иначе, так как оптимальное выравнивание является задачей о кратчайшем пути во взвешенном редакционном графе, соответствующем таблице множественного выравнивания. В этих терминах, когда вычисляется кратчайший путь из источника s (клетка $(0, 0, 0)$) в вершину v (клетка (i, j, k)), корректируются наилучшие расстояния от s до всех вершин, непосредственно следующих за v . Другими словами, пусть $D(v)$ — кратчайшее расстояние от s до v , (v, w) — дуга веса $p(v, w)$, а $p(w)$ — кратчайшее из имеющихся расстояний от s до w . После вычисления $D(v)$ значение $p(w)$ немедленно заменяется значением $\min\{p(w), D(v) + p(v, w)\}$. Более того, истинное кратчайшее расстояние от s до w , т. е. $D(w)$, должно равняться $p(w)$ после того, как $p(w)$ будет исправлено каждой вершиной v , из которой идет дуга в w . Нужно уточнить еще одну деталь. Реализация прямого ДП поддерживает *очередь* из вершин (клеток), для которых окончательные значения D еще не получены. Алгоритм будет устанавливать значение D в вершине v , стоящей в голове очереди, и удалять ее из очереди. Когда это будет сделано, он обновит $p(w)$ для каждой вершины, непосредственно следующей за v , и, если w еще не попало в очередь, добавит w в конец очереди. Легко проверить, что, когда вершина доходит до головы очереди, все вершины графа, из которых есть переходы в данную, уже будут из очереди удалены.

Для задачи выравнивания трех строк, по n символов в каждой, редакционный граф имеет примерно n^3 вершин и $7n^3$ дуг, и вычисления по обратной схеме ДП обрабатывают каждую из этих дуг. Однако если удастся опознать большое число вершин как не лежащих ни на каком оптимальном (кратчайшем) пути из $(0, 0, 0)$ в (n, n, n) , то упомянутую обработку дуг, выходящих из этих вершин, можно будет не проводить. Ускорение Карилло и Липмана исключает некоторые вершины до основной работы схемы ДП, причем здесь и прямая и обратная схемы равно удобны. В общении, используемом в MSA, вершины исключаются в ходе работы основного алгоритма, и принципиально, чтобы это было прямое ДП.

Определение. Пусть $d_{1,2}(i, j)$ обозначает редакционное расстояние между суффиксами $S_1[i..n]$ и $S_2[j..n]$ строк S_1 и S_2 . Определим аналогично $d_{1,3}(i, k)$ и $d_{2,3}(j, k)$.

Ясно, что все эти d можно вычислить за время $O(n^2)$, обратив строки и найдя три парных расстояния. Далее, кратчайший путь из вершины (i, j, k) в вершину (n, n, n) редакционного графа для S_1 , S_2 и S_3 должен иметь длину не меньше $d_{1,2}(i, j) + d_{1,3}(i, k) + d_{2,3}(j, k)$.

Теперь предположим, что известно некоторое множественное выравнивание S_1 , S_2 и S_3 (возможно, из эвристического метода с ограниченной ошибкой, типа того, который рассматривается ниже) и что это выравнивание имеет SP -оценку z . Идея эвристического ускорения такова.

Основная идея. Напомним, что $D(i, j, k)$ — оптимальная SP -оценка для выравнивания $S_1[1..i]$, $S_2[1..j]$ и $S_3[1..k]$. Если $D(i, j, k) + d_{1,2}(i, j) + d_{1,3}(i, k) + d_{2,3}(j, k)$ превосходит z , то вершина (i, j, k) не может лежать ни на каком оптимальном пути, и следовательно, (в прямом алгоритме) $D(i, j, k)$ не нужно посыпать вперед ни в какую клетку.

Когда эта эвристическая идея применяется к клетке (i, j, k) , она экономит работу, поскольку значение D не посыпается вперед (семи) соседям клетки. Но еще важнее, что, когда эта эвристика прилагается ко многим клеткам, из рассмотрения можно автоматически исключить целый водопад дополнительных клеток, так как они никогда не появятся в очереди. Благодаря этому эвристика может избавить нас от значительной части графа выравнивания и существенно уменьшить число клеток, значения D в которых определяются для вычисления $D(n, n, n)$. Это стандартная эвристика при вычислениях кратчайшего пути. Отметим, что вычисления остаются точными и оптимальное выравнивание будет найдено.

Если никакого начального значения z не известно, то эта эвристика может быть реализована как эвристика A^* [230, 379], когда самые “многообещающие” выравнивания вычисляются первыми. Тогда наилучшее выравнивание, вычисляемое в каждый момент, обеспечивает (убывающее) значение z . Можно доказать [230], что этот подход чистит график выравнивания не хуже, чем метод Карилло и Липмана, но обычно он чистит значительно лучше. В [303] сообщается, что MSA может выравнивать шесть строк длиной около 200 символов за “практически” приемлемое время. Однако не похоже, чтобы подход годился для оптимального выравнивания десятков или сотен строк, если только не начать с предельно удачного значения z . Но и тогда практичность MSA остается под вопросом.

14.6.2. Приближенный метод с ограниченной ошибкой для SP -выравнивания

Так как точное решение задачи SP -выравнивания допустимо только для малого числа строк, самые практические (эвристические) методы множественного выравнивания не достигают оптимального SP -выравнивания. Однако обычно не известно, насколько выравнивания, выработанные этими эвристиками, могут отличаться от оптимального. В этом пункте рассмотрен один из тех редких методов, в которых возможен анализ такой ошибки. Предлагается метод построения приближенного SP -выравнивания с ограниченной ошибкой из работы [201]. У этого метода доказуемо хорошая скорость (он требует в худшем случае полиномиального времени), и он находит выравнивания, у которых SP -оценка не более чем вдвое превосходит оценку оптимального SP -выравнивания. Он будет первым из нескольких представленных в этой книге приближенных методов с ограниченной ошибкой, которые включают в себя дополнительные приемы для различных задач множественного выравнивания.

Исходная идея: выравнивания, совместные с деревом

Представляемая нами SP -аппроксимация, ее улучшения и некоторые другие методы для других проблем, — все они используют основную идею, которая связывает множественные выравнивания с деревьями.*¹⁾ Мы дадим сначала общее изложение этой идеи, а затем конкретизируем ее для SP -выравнивания. Напомним, что $D(S_i, S_j)$ обозначает (оптимальное) взвешенное редакционное расстояние между строками S_i и S_j .

Определение. Пусть \mathcal{S} — набор строк, а T — неориентированное дерево, каждая вершина которого помечена отдельной строкой из \mathcal{S} . Тогда множественное выравнивание \mathcal{M} набора \mathcal{S} называется *совместным* с T , если индуцированное парное выравнивание S_i и S_j имеет оценку $D(S_i, S_j)$ для любой пары строк (S_i, S_j) , помечающих смежные вершины в T . См. пример на рис. 14.6.

Теорема 14.6.1. Для любого набора строк \mathcal{S} и для любого дерева T , вершины которого помечены различными строками \mathcal{S} , мы можем эффективно найти множественное выравнивание $\mathcal{M}(T)$ набора \mathcal{S} , совместное с T .

Отметим, что роль T в этой теореме очень специальна и оценка индуцированного выравнивания двух строк S_i и S_j , не помечающих смежные вершины, обычно выше, чем $D(S_i, S_j)$.

Доказательство. Будем строить множественное выравнивание $\mathcal{M}(T)$, добавляя по одной строке за раз, и покажем по индукции, что утверждение теоремы 14.6.1 выполняется после добавления к выравниванию каждой новой строки. Для начала выберем любые две строки S_i и S_j , помечающие смежные вершины в T , и сформируем их двухстроковое выравнивание с расстоянием $D(S_i, S_j)$. Для такого дерева теорема тривиально верна. Предположим, что утверждение выполнено после того, как к множественному выравниванию добавлено некоторое число строк. Продолжая

*¹⁾ Не следует путать соотношение между деревьями и множественными выравниваниями с соотношением между деревьями *эволюции* и множественными выравниваниями, которые будут обсуждаться в пп. 14.8 и 14.10.2.

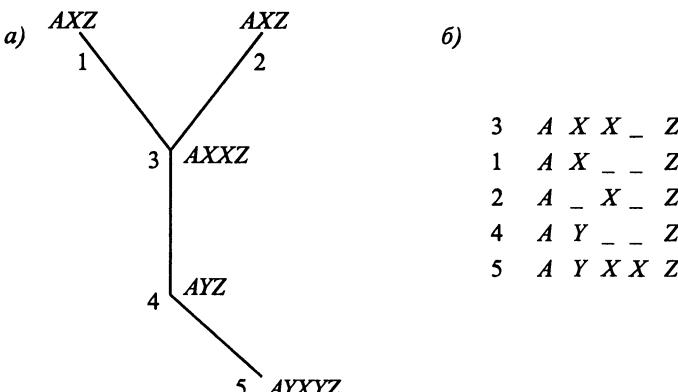


Рис. 14.6. *а* — дерево, вершины которого помечены строками из набора; *б* — множественное выравнивание этих строк, совместное с деревом. Схема парной оценки оценивает нулем совпадение, а единицей — несовпадение или пробел. Читатель может проверить, что каждое из четырех индуцированных выравниваний, отвечающих ребрам деревя, имеет оценку, равную соответствующему оптимальному расстоянию. Однако индуцированное выравнивание двух строк, помечающих несмежные вершины, может иметь оценку, превышающую их оптимальное парное расстояние

процесс, выберем любую строку S' , еще не включенную в выравнивание и помечающую вершину, смежную с вершиной, метка которой, скажем, S_i , уже включена в выравнивание. В этом существующем множественном выравнивании в S_i могло быть добавлено некоторое количество пробелов; обозначим через \bar{S}_i получившуюся строку. Отметим, что для каждой строки существующего множественного выравнивания каждый ее символ соответствует вполне определенному столбцу с одним символом строки \bar{S}_i .

Далее, оптимально выровняем строку S' с \bar{S}_i , используя оценочную схему двухстрокового выравнивания, с дополнительным правилом нулевой оценки стоящих напротив друг друга пробелов (они считаются совпадением). Очевидно, что оценка получающегося парного выравнивания в точности равна $D(S_i, S')$. Добавим теперь S' к существующему множественному выравниванию, так что индуцированное выравнивание S_i и S' будет иметь оценку $D(S_i, S')$, а все индуцированные оценки строк, уже входящих в выравнивание, не изменятся. Пусть \bar{S}' — строка S' со всеми пробелами, добавленными при выравнивании S' и \bar{S}_i . Если парное выравнивание S' и \bar{S}_i не добавляет новых пробелов в \bar{S}_i , то добавим \bar{S}' к существующему множественному выравниванию. Результатом будет множественное выравнивание с еще одной строкой, в котором индуцированная оценка S_i и S' равна $D(S_i, S')$, а все индуцированные оценки из предыдущего множественного выравнивания не изменились.

Однако если парное выравнивание включило в \bar{S}_i новый пробел, например между символами l и $l + 1$, то нужно включить пробел между символами l и $l + 1$ в любой строке существующего множественного выравнивания. Это создаст в выравнивании новый столбец, где стоят только пробелы, но существовавшие столбцы останутся без изменений. (Например, предположим, что первые четыре строки на рис. 14.6 уже выровнены и что строка \bar{S}_4 в этот момент равна AY_Z . Выравнивание S_5 с \bar{S}_4 добавляет в \bar{S}_4 пробел в четвертой позиции. Первые четырьмя строчки на рис. 14.6

показывают результирующее множественное выравнивание после добавления этого пробела в строки S_1 , S_2 и S_3 .) Результатом добавления столбца, содержащего только пробелы, будет множественное выравнивание с теми же оценками всех индуцированных парных выравниваний. Добавление \tilde{S}' к этому множественному выравниванию создает множественное выравнивание с еще одной строкой, где утверждение теоремы сохраняется. Существование $\mathcal{M}(T)$ получается по индукции.

Время, необходимое для вычисления $\mathcal{M}(T)$, доминируется временем на вычисление $k - 1$ парных выравниваний. Если каждая строка имеет длину n , то все парные выравнивания занимают время $O(n^2)$, и время на построение $\mathcal{M}(T)$ равно $O(kn^2)$. \square

Хотя теорема 14.6.1 стала частью “фольклора”, она, наверное, впервые была сформулирована в [153]. Мы можем теперь вернуться к приближенному методу для задачи SP -выравнивания.

Метод центральной звезды для SP -выравнивания

Мы опишем этот метод в терминах алфавитно-взвешенной схемы оценки для двухстроковых выравниваний. Пусть $s(x, y)$ — оценка выравнивания символа x (возможно, пробела) с символом y (также, возможно, пробелом).

Определение. Схема оценки удовлетворяет *неравенству треугольника*, если для любых трех символов, x , y и z , выполняется неравенство

$$s(x, z) \leq s(x, y) + s(y, z).$$

Неравенство треугольника открывает двери для правильной интуиции при работе с редакционным расстоянием. Оно говорит, что “цена” преобразования x в z не превосходит цены преобразования x в промежуточный символ y с последующим преобразованием y в z . Следует заметить, однако, что не все оценочные матрицы, применяемые в вычислительной биологии, этому неравенству удовлетворяют.

Определение. Пусть задан набор \mathcal{S} из k строк. Определим *центральную строку* $S_c \in \mathcal{S}$ как строку из \mathcal{S} , минимизирующую $\sum_{S_i \in \mathcal{S}} D(S_i, S_c)$. Пусть M обозначает это минимальное значение. Определим *центральную звезду* как звездообразное дерево из k вершин, в котором центральная вершина имеет метку S_c , а остальные $k - 1$ вершин помечены различными оставшимися строками из \mathcal{S} . См. пример на рис. 14.7.

Определение. Определим множественное выравнивание \mathcal{M}_c набора строк \mathcal{S} как множественное выравнивание, *совместное* с центральной звездой.

Определение. Определим $d(S_i, S_j)$ как оценку парного выравнивания строк S_i и S_j , *индуцированного* выравниванием \mathcal{M}_c . Обозначим оценку выравнивания \mathcal{M} через $d(\mathcal{M})$.

Ясно, что $d(S_i, S_j) \geq D(S_i, S_j)$ и $d(\mathcal{M}_c) = \sum_{i < j} d(S_i, S_j)$. Кроме того, так как \mathcal{M}_c совместно с центральной звездой и центром звезды является строка S_c , то $d(S_i, S_c) = D(S_i, S_c)$ для каждой строки S_i . Покажем, что $d(\mathcal{M}_c)$ не более чем вдвое превосходит оценку оптимального SP -множественного выравнивания \mathcal{S} в предположении, что используемая оценочная схема для парных выравниваний удовлетворяет неравенству треугольника.

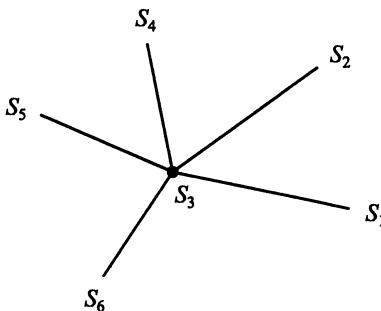


Рис. 14.7. Типичная центральная звезда для шести строк, в которой центральной строкой S_c является S_3

Лемма 14.6.1. Предположим, что двухстроковая схема оценки удовлетворяет неравенству треугольника. Тогда для любых строк S_i и S_j из \mathcal{S} выполняется неравенство

$$d(S_i, S_j) \leq d(S_i, S_c) + d(S_c, S_j) = D(S_i, S_c) + D(S_c, S_j).$$

Доказательство. Возьмем какой-либо столбец множественного выравнивания \mathcal{M}_c . Пусть x , y и z — три символа этого столбца из трех строк S_i , S_c и S_j соответственно. По неравенству треугольника $s(x, z) \leq s(x, y) + s(y, z)$, и требуемое неравенство следует из определения d . Равенство получается из-за того, что индуцированное \mathcal{M}_c парное выравнивание S_i и S_c оптимально для S_i и S_c , и то же верно для выравнивания S_c и S_j . \square

Определение. Пусть \mathcal{M}^* — оптимальное множественное выравнивание k строк \mathcal{S} . Пусть $d^*(S_i, S_j)$ — оценка парного выравнивания строк S_i и S_j , индуцированного \mathcal{M}^* . Тогда $d(\mathcal{M}^*) = \sum_{i < j} d^*(S_i, S_j)$.

Теперь можно сформулировать и доказать основную теорему этого пункта.

Теорема 14.6.2. $d(\mathcal{M}_c)/d(\mathcal{M}^*) \leq 2(k - 1)/k < 2$.

Доказательство. Во-первых, определим $v(\mathcal{M}_c) \equiv \sum_{(i,j)} d(S_i, S_j)$ и $v(\mathcal{M}^*) \equiv \sum_{(i,j)} d^*(S_i, S_j)$, где пара (i, j) в каждом случае упорядочена. Ясно, что $v(\mathcal{M}_c) = 2d(\mathcal{M}_c)$ и $v(\mathcal{M}^*) = 2d(\mathcal{M}^*)$, и поэтому отношения $d(\mathcal{M}_c)/d(\mathcal{M}^*)$ и $v(\mathcal{M}_c)/v(\mathcal{M}^*)$ равны. Будет удобнее работать со вторым отношением. Напомним, что минимальная сумма расстояний M определена как $\sum_j D(S_c, S_j)$. Далее,

$$v(\mathcal{M}_c) = \sum_{(i,j)} d(S_i, S_j) \leq \sum_{(i,j)} [D(S_i, S_c) + D(S_c, S_j)],$$

по лемме 14.6.1. Для любого фиксированного j значение $D(S_c, S_j)$ (которое равно $D(S_j, S_c)$) превосходит это выражение ровно в $2(k - 1)$ раз. Поэтому

$$v(\mathcal{M}_c) \leq 2(k - 1) \times \sum_j D(S_c, S_j) = 2(k - 1)M.$$

С другой стороны,

$$v(\mathcal{M}^*) = \sum_{(i,j)} d^*(S_i, S_j) \geq \sum_{(i,j)} D(S_i, S_j) = \sum_i \sum_j D(S_i, S_j) \geq k \times \sum_j D(S_c, S_j) = kM$$

(по выбору S_c). Таким образом,

$$\frac{d(\mathcal{M}_c)}{d(\mathcal{M}^*)} = \frac{v(\mathcal{M}_c)}{v(\mathcal{M}^*)} \leq \frac{2(k-1)M}{kM} = \frac{2(k-1)}{k} = 2 - \frac{2}{k} < 2. \quad \square$$

Заметим, что при $k = 3$ гарантированная верхняя граница равна 4/3. Значит, для трех строк множественное выравнивание, полученное по методу центральной звезды, имеет оценку, превосходящую оптимальную не более чем на 34 %. В переводе на нижние оценки это значит, что для $k = 3$ верно $d(\mathcal{M}^*) \geq 0.75d(\mathcal{M}_c)$. Для $k = 4$ верхняя граница равна только 1.5, а для $k = 6$ (такой размер задачи считается слишком большим для эффективного точного решения с длиной строк 200) граница всего лишь 1.67.

Следствие 14.6.1. $kM \leq \sum_{i < j} D(S_i, S_j) \leq d(\mathcal{M}^*) \leq d(\mathcal{M}_c) \leq [2(k-1)/k] \sum_{i < j} D(S_i, S_j)$.

На практике качество \mathcal{M}_c лучше оценивать отношением $d(\mathcal{M}_c) / \sum_{i < j} D(S_i, S_j)$. По следствию 14.6.1 это отношение всегда меньше двух, но проведенный анализ относится к худшему случаю, так что можно ожидать, что это отношение будет значительно меньше двух. Аналогично следует ожидать, что значительно меньше двух будет отношение $d(\mathcal{M}_c) / d(\mathcal{M}^*)$, так как обычно сумма $\sum_{(i,j)} D(S_i, S_j)$ будет значительно больше, чем kM ; что $d(\mathcal{M}^*)$ не будет вообще-то близко к $\sum_{i < j} D(S_i, S_j)$ для любых строк, кроме очень похожих, и что $D(S_i, S_j)$ будет меньше, чем $D(S_i, S_c) + D(S_c, S_j)$, для большинства обычных строк.

Следствие 14.6.1 полезно также в ускорении MSA (п. 14.6.1) для точного решения задачи оптимального SP-выравнивания, так как этот метод требует знания эффективно вычисляемой верхней границы z для оценки оптимального SP-выравнивания.

14.6.3. Взвешенное SP-выравнивание

Альтшуль и Липман [20] предложили обобщение SP-выравнивания, в котором индуцированная парная оценка каждой пары (S_i, S_j) умножается на вес $w(i, j)$. При этом оценкой множественного выравнивания \mathcal{M} становится $\sum_{i < j} w(i, j)d(S_i, S_j)$, где $d(S_i, S_j)$ — оценка парного выравнивания, индуцированная \mathcal{M} . Веса изменяют значение различных пар строк и часто предназначаются для отображения известного эволюционного расстояния между организмами, из которых получены эти строки. Используя веса, можно пытаться более аккуратно отразить во множественном выравнивании уже известную историю эволюции.

Оптимальное взвешенное SP-выравнивание также можно вычислить за экспоненциальное время с помощью ДП, но приближенные методы изучены мало. Однако веса прежде всего нужны для отображения эволюционного расстояния, и для задачи множественного выравнивания есть другая целевая функция, которая служит той же цели — использовать историю эволюции, чтобы повлиять на результирующее

множественное выравнивание. Эта целевая функция присутствует в задаче *филогенетического выравнивания*, для решения которой есть приближенный метод, очень близкий методу центральной звезды. Мы обсудим эти вопросы в п. 14.8. Но до этого нам нужно рассмотреть консенсусную целевую функцию и приближенный алгоритм, еще более близкий методу центральной звезды.

14.7. Множественное выравнивание с консенсусными целевыми функциями

Многие методы множественного выравнивания или сравнения строк, используемые в биологии, предназначены для получения *консенсусного представления* важных общих черт набора строк. Поскольку нет всеобщего консенсуса в том, как определить “консенсус”, мы рассмотрим три определения, передающие дух большинства этих методов. Мы покажем затем, что эти три определения приводят к одной и той же строке и одному и тому же множественному выравниванию, которое можно аппроксимировать методом центральной звезды. Для ясности изложения будем использовать некоторые понятия консенсуса, которые не появлялись в литературе по вычислительной биологии и, следовательно, определим некоторые новые термины.

14.7.1. Штейнеровские консенсусные строки

Первое определение консенсусной последовательности дается без явной связи с множественным выравниванием. Как и раньше, предположим, что существует двухстроковая схема оценки, удовлетворяющая неравенству треугольника. Напомним, что $D(S_i, S_j)$ обозначает взвешенное редакционное расстояние между строками S_i и S_j .

Определение. Пусть задан набор строк \mathcal{S} и еще одна строка \bar{S} . Консенсусной ошибкой строки \bar{S} относительно набора \mathcal{S} называется $E(\bar{S}) = \sum_{S_i \in \mathcal{S}} D(\bar{S}, S_i)$. Заметим, что от строки \bar{S} не требуется, чтобы она принадлежала \mathcal{S} .

Определение. Пусть задан набор строк \mathcal{S} . Оптимальной *штейнеровской строкой* S^* для \mathcal{S} называется строка, минимизирующая консенсусную ошибку $E(S^*)$ на множестве всех возможных строк.

Заметим, что строка S^* не должна входить в набор \mathcal{S} , и обычно она не входит. “Штейнеровскую консенсусную строку” мы будем называть “штейнеровской строкой”.

Штейнеровская строка S^* пытается вобрать и отразить в одной строке общие характеристики набора строк \mathcal{S} . Эффективные методы нахождения S^* неизвестны, но есть приближенный метод, который находит строку \bar{S} , такую что отношение $E(\bar{S})/E(S^*)$ никогда не превосходит двух.

Лемма 14.7.1. Пусть \mathcal{S} состоит из k строк и двухстроковая схема оценки удовлетворяет неравенству треугольника. Тогда существует такая строка $\bar{S} \in \mathcal{S}$, что $E(\bar{S})/E(S^*) \leq 2 - 2/k < 2$.

Доказательство. Пусть \bar{S} — любая строка из \mathcal{S} . По неравенству треугольника парной схемы оценки, $D(\bar{S}, S_i) \leq D(\bar{S}, S^*) + D(S^*, S_i)$ для любой строки S_i . Поэтому

$$E(\bar{S}) = \sum_{S_i \in \mathcal{S}} D(\bar{S}, S_i) \leq \sum_{S_i \neq \bar{S}} [D(\bar{S}, S^*) + D(S^*, S_i)] = (k - 2)D(\bar{S}, S^*) + E(S^*).$$

Теперь возьмем в качестве \bar{S} строку из \mathcal{S} , ближайшую к оптимальной штейнеровской строке S^* . Это значит, что нужно выбрать \bar{S} так, чтобы $D(\bar{S}, S^*)$ не превосходило $D(S_i, S^*)$ для любой $S_i \in \mathcal{S}$. (Конечно, строка \bar{S} конструктивно неизвестна, так как неизвестна S^* , но \bar{S} существует.) Тогда $E(S^*) = \sum_{S_i \in \mathcal{S}} D(S^*, S_i) \geq kD(\bar{S}, S^*)$. Поэтому

$$\frac{E(\bar{S})}{E(S^*)} \leq \frac{k-2}{k} \cdot \frac{D(\bar{S}, S^*)}{D(\bar{S}, S^*)} + 1 = \frac{k-2}{k} + 1 = 2 - \frac{2}{k} < 2. \quad \square$$

Напомним, что *центральная строка* S_c определена как строка из \mathcal{S} , которая минимизирует $\sum_{S_i \in \mathcal{S}} D(S_c, S_i)$ по всем строкам из \mathcal{S} .

Теорема 14.7.1. В предположении, что схема оценки удовлетворяет неравенству треугольника, справедлива оценка $E(S_c)/E(S^*) \leq 2 - 2/k$.

Доказательство. Утверждение следует немедленно из леммы 14.7.1 и того факта, что по определению $E(S_c) \leq E(\bar{S})$. \square

Таким образом, центральная строка имеет консенсусную ошибку, которая не более чем в $2 - 2/k$ раз превосходит ошибку оптимальной штейнеровской консенсусной строки. Этот результат является частным случаем приближенного метода, используемого в задаче о филогенетическом выравнивании, которая будет рассмотрена в п. 14.8.

14.7.2. Консенсусные строки из множественного выравнивания

Хотя определение оптимальной штейнеровской строки не упоминает о множественном выравнивании, задача нахождения S^* эквивалентна более традиционным консенсусным задачам в терминах множественных выравниваний. Дадим точные формулировки этих задач.

Определение. Пусть задано множественное выравнивание \mathcal{M} набора строк \mathcal{S} . Консенсусным символом столбца i из \mathcal{M} называется символ, на котором достигается минимум суммарного расстояния от него до всех символов в столбце i . Пусть $d(i)$ обозначает эту минимальную сумму в столбце i .

Так как алфавит конечен, то консенсусный символ для каждого столбца в \mathcal{M} существует и может быть найден перебором. В частном случае, когда парная схема оценки оценивает совпадение нулем, а несовпадение или пробел единицей, этот консенсусный символ в столбце i будет символом *большинства* (т. е. символом, наиболее часто встречающимся в этом столбце). Заметим, что символом большинства может быть и пробел. См. пример на рис. 14.5.

Определение. Консенсусной строкой $S_{\mathcal{M}}$, полученной из выравнивания \mathcal{M} , называется конкатенация консенсусных символов всех столбцов \mathcal{M} .

В литературе по вычислительной биологии принято получать множественное выравнивание \mathcal{M} набора строк и затем представлять эти строки консенсусной строкой $S_{\mathcal{M}}$. Поэтому естественно использовать качество (“хорошесть”) строки $S_{\mathcal{M}}$ для оценки качества множественного выравнивания \mathcal{M} . Один способов лежит в определении качества \mathcal{M} как $\sum_i D(S_{\mathcal{M}}, S_i)$. Это значит, что качество \mathcal{M} определяется тем, насколько удачно $S_{\mathcal{M}}$ в роли *штейнеровской строки* набора \mathcal{S} . Другой подход к оценке качества множественного выравнивания \mathcal{M} через его консенсусную строку $S_{\mathcal{M}}$ связан с \mathcal{M} более тесно.

Определение. Пусть \mathcal{M} — множественное выравнивание набора строк \mathcal{S} , и $S_{\mathcal{M}}$ — его консенсусная строка, состоящая из q символов. Тогда *ошибка выравнивания* строки $S_{\mathcal{M}}$ равна $\sum_{i=1}^q d(i)$, а ошибка выравнивания \mathcal{M} определяется как ошибка выравнивания $S_{\mathcal{M}}$ (см. рис. 14.5).

Определение. *Оптимальным консенсусным множественным выравниванием* называется множествоное выравнивание \mathcal{M} для набора строк \mathcal{S} , консенсусная строка которого $S_{\mathcal{M}}$ имеет наименьшую ошибку выравнивания среди всех возможных множественных выравниваний \mathcal{S} .

Итак, мы имеем три определения консенсуса для набора строк \mathcal{S} и два связанных с ними способа оценки множественного выравнивания \mathcal{M} . Первое определение консенсуса — это *штейнеровская строка* S^* , задаваемая по \mathcal{S} без какого-либо упоминания о множественном выравнивании. Второе определение — это консенсусная строка $S_{\mathcal{M}}$, которая получается из множественного выравнивания \mathcal{M} , но качество которой характеризуется тем, насколько хорошо она в качестве штейнеровской строки для \mathcal{S} . В этом случае качество множественного выравнивания \mathcal{M} устанавливается как консенсусная ошибка консенсусной строки $S_{\mathcal{M}}$. Третье определение связано с множественным выравниванием наиболее тесно. В нем строка $S_{\mathcal{M}}$ и формируется по \mathcal{M} , и в ее вычислении учитывается, насколько хорошо она отражает свойства отдельных столбцов \mathcal{M} . В этом случае также качество \mathcal{M} основывается на качестве $S_{\mathcal{M}}$, именно на ошибке выравнивания $S_{\mathcal{M}}$. Эти три представляющиеся различными определения эквивалентны в том смысле, что они приводят к одному и тому же множественному выравниванию.

Определение. Пусть задан набор \mathcal{S} из k строк и T — звездное дерево с корнем в штейнеровской строке S^* , а все k строк \mathcal{S} находятся в различных листьях T . Тогда множественное выравнивание $\mathcal{S} \cup S^*$, совместное с T , называется совместным с S^* .

Теорема 14.7.2. Пусть S обозначает консенсусную строку оптимального консенсусного множественного выравнивания. Тогда удаление из S пробелов приводит к оптимальной штейнеровской строке S^* . Обратно, удаление строчки для S^* из множественного выравнивания, совместного с S^* , приводит к оптимальному консенсусному множественному выравниванию для \mathcal{S} .

Доказательство. Пусть \mathcal{S} имеет k строк, а \mathcal{M} — какое-либо множественное выравнивание \mathcal{S} . По определению каждому символу консенсусной строки $S_{\mathcal{M}}$, полученной из \mathcal{M} , сопоставлен отдельный столбец \mathcal{M} , и это сопоставление порождает некоторые попарные выравнивания $S_{\mathcal{M}}$ и каждой строки $S_i \in \mathcal{S}$. Ясно, что оценка

этого выравнивания не меньше $D(S_i, S_{\mathcal{M}})$. Далее, ошибка выравнивания $S_{\mathcal{M}}$ в точности равна сумме оценок этих k индуцированных парных выравниваний, так что ошибка выравнивания $S_{\mathcal{M}}$ не меньше чем $\sum_i D(S_i, S_{\mathcal{M}})$, а это консенсусная ошибка $S_{\mathcal{M}}$ для \mathcal{S} . Но по определению S^* имеет минимальную консенсусную ошибку для \mathcal{S} , так что ошибка выравнивания для $S_{\mathcal{M}}$ не меньше консенсусной ошибки S^* .

Теперь рассмотрим множественное выравнивание \mathcal{M}^* набора $\mathcal{S} \cup S^*$, совместное с S^* . Для любой строки $\alpha \in \mathcal{S} \cup S^*$ обозначим через $\bar{\alpha}$ строку из строчки \mathcal{M}^* , соответствующей α . По совместности, оценка индуцированного выравниванием \mathcal{M}^* парного выравнивания \bar{S}^* и \bar{S}_i равна $D(S^*, S_i)$ для любого $S_i \in \mathcal{S}$. Обозначим через \mathcal{M} выравнивание \mathcal{M}^* с удаленной строчкой для S^* . Тогда ошибка выравнивания \bar{S}^* с \mathcal{M} в точности равна $\sum_i D(S^*, S_i)$, т. е. консенсусной ошибке S^* для \mathcal{S} . Следовательно, используя заключение из предыдущего абзаца, мы видим, что ошибка выравнивания любой другой консенсусной строки $S_{\mathcal{M}}$ для любого другого множественного выравнивания \mathcal{M} должна быть не меньше, чем ошибка выравнивания S^* для \mathcal{M} . Отсюда следует, что \mathcal{M}^* является оптимальным консенсусным множественным выравниванием для \mathcal{S} и что \bar{S}^* является его консенсусной строкой. Поэтому, так как S^* получается из \bar{S}^* удалением пробелов, теорема доказана. \square

Таким образом, оптимальное консенсусное множественное (КМ) выравнивание определяет оптимальную штейнеровскую строку S^* , и обратно, S^* можно использовать для построения оптимального КМ выравнивания. И штейнеровская строка, и консенсусная строка для оптимального КМ выравнивания находят одни и те же общие места в \mathcal{S} , и оптимальное КМ выравнивание позволяет нам отражать эти общие места, располагая их по столбцам выравнивания.

14.7.3. Аппроксимация оптимального консенсусного множественного выравнивания

Из доказательства теоремы 14.7.2 и утверждения теоремы 14.7.1 должно быть ясно, как найти множественное выравнивание \mathcal{M} , ошибка выравнивания которого не более чем в $2 - 2/k$ раз превосходит ошибку выравнивания оптимального КМ выравнивания (в предположении, что неравенство треугольника для двухстроковой схемы оценки выполняется). Нужно только, как и раньше, найти центральную строку S_c , поместить ее в центр k -вершинной звезды, пометить каждый лист одной из оставшихся строк \mathcal{S} и построить множественное выравнивание \mathcal{M} , совместное с этим деревом.

Описанное множественное выравнивание \mathcal{M} в точности совпадает с выравниванием \mathcal{M}_c , которое использовалось для приближения целевой функции SP . Таким образом, мы получили следующий, возможно неожиданный, результат:

Теорема 14.7.3. *В предположении, что верно неравенство треугольника, множественное выравнивание \mathcal{M}_c , построенное по методу центральной звезды, имеет SP -оценку, которая не более чем в $2 - 2/k$ раз превосходит SP -оценку оптимального SP -выравнивания, и это выравнивание имеет (консенсусную) ошибку выравнивания, которая не более чем в $2 - 2/k$ раз превосходит ошибку выравнивания оптимального консенсусного множественного выравнивания.*

14.8. Множественное выравнивание по (филогенетическому) дереву

Как отмечалось при рассмотрении взвешенных *SP*-выравниваний, желательно использовать известную историю эволюции, чтобы влиять на вычисляемое множественное выравнивание набора строк. Метод должен более тесно выравнивать последовательности, полученные от более близких организмов. История эволюции чаще всего представляется с помощью *эволюционного дерева*, в котором известные последовательности из (дошедших до нас) организмов представлены листьями, а их неизвестные предшественники — внутренними вершинами. Когда дерево установлено (из предыдущих данных и выводов), нужно вывести последовательности для внутренних вершин так, чтобы оптимизировать определенную ниже целевую функцию. Если эти последовательности у нас на руках, то можно найти множественное выравнивание, совместное с помеченным деревом, и затем удалить эти выведенные последовательности из множественного выравнивания. В качестве альтернативы само помеченное дерево дает картину эволюции, которая приводит к известным последовательностям (в листьях). Это последнее использование филогенетического выравнивания будет рассматриваться в п. 17.6.

Определение. Пусть задано исходное дерево T , в каждом листе которого записана своя строка (из множества строк \mathcal{S}). *Филогенетическим выравниванием* для T называется сопоставление каждой внутренней вершине T по одной строке. Заметим, что строки, сопоставляемые внутренним вершинам, не должны быть различными и не должны принадлежать исходному набору строк \mathcal{S} .

Филогенетическое дерево T предназначено для изображения “установленной” истории эволюции набора таксонов (читай “объектов исследования”) с тем условием, что каждый нынеживущий таксон (“объект”) обозначается в T своим листом. Каждая дуга (u, v) представляет какую-то историю мутаций, которая преобразует строку вершины u (предполагая, что u — предок v) в строку вершины v . Цена преобразования задается редакционным расстоянием между строками, так что цена филогенетического выравнивания равна сумме всех цен на дугах. Изложим это более формально.

Напомним, что $D(S, S')$ обозначает редакционное расстояние между строками S и S' . Единственное предположение, сделанное о функции D , заключается в том, что она удовлетворяет обычному неравенству треугольника.

Определение. Если строки S и S' соответствуют началу и концу дуги (i, j) , то (i, j) имеет *дуговую длину* $D(S, S')$. Длиной пути является сумма дуговых длин пути. Длина филогенетического выравнивания равна сумме дуговых длин всех дуг дерева.

Задача филогенетического выравнивания для T . Найти сопоставление строк внутренним вершинам T (одна строка для каждой вершины), минимизирующее длину выравнивания.

Задача филогенетического выравнивания, называемая также *задачей древовидного выравнивания*, развивалась в основном Санковым [387, 388]. Мы используем термин “филогенетическое выравнивание”, так как термин “древовидное выравнивание” занят в литературе о строках под совершенно иную задачу [117].

Заметим, что задача о консенсусной строке (см. п. 14.7) представляет собой частный случай задачи филогенетического выравнивания (т. е. случай, когда дерево T — звезда). Связь между филогенетическим выравниванием и консенсусной задачей впервые была отмечена в [20]. Сходным образом предлагаемый здесь алгоритм для задачи филогенетического выравнивания будет естественным обобщением алгоритма, предложенного для SP - и консенсусной целевых функций.

14.8.1. Эвристика для филогенетического выравнивания

Общая задача филогенетического выравнивания NP -полна [454], но для нее есть метод решения с экспоненциальным временем, использующий динамическое программирование [387]. Эвристический подход к ограниченному варианту этой задачи сформулирован в [457]. Здесь мы рассмотрим эффективный эвристический метод, который был предложен и исследован Вангом, Янгом и Лаулером [249]. Он строит филогенетические выравнивания, длина которых не более чем вдвое превосходит длину минимального выравнивания. Однако наше изложение следует упрощению исходного доказательства, предложенному Майком Патерсоном. Для получения этого результата мы предположим, что функция редакционного расстояния D удовлетворяет неравенству треугольника. Результат, гарантирующий множитель два, дает также полиномиальную приближенную схему с возможностью различного баланса времени вычислений и обеспечиваемой точности [249]. Новый результат в этом направлении дает практическую приближенную схему, у которой отклонение, не превышающее 1.58, обеспечивается для строк с длиной, характерной для белков [453].

Определение. Филогенетическое выравнивание называется *поднятым выравниванием*, если любой внутренней вершине v сопоставляется строка, соответствующая одному из ее потомков.

Например, решение, представленное на рис. 14.8 является поднятым выравниванием. Ясно, что каждой вершине v в поднятом выравнивании сопоставлена строка, которая помечает один из листьев в поддереве с корнем в v .

Мы покажем, что лучшее поднятое выравнивание в T имеет полную длину, менее чем вдвое превышающую длину оптимального филогенетического выравнивания.

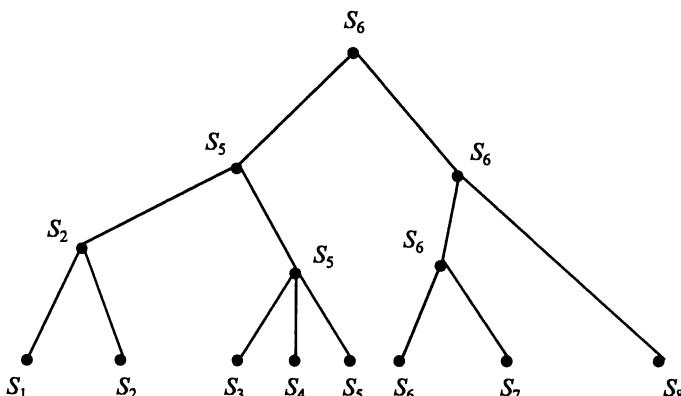


Рис. 14.8. Абстрактное поднятое выравнивание. Каждая вершина имеет абстрактное имя — записанную под ней строку

Для этого построим обладающее этим свойством конкретное поднятое выравнивание T^L . Оно получается преобразованием оптимального филогенетического выравнивания. Позднее покажем, что наилучшее поднятое выравнивание можно вычислить эффективно с помощью динамического программирования. Так как наилучшее поднятое выравнивание имеет полную длину, не превосходящую длины T^L , то можно эффективно найти поднятое выравнивание с длиной, менее чем вдвое превосходящей длину оптимального филогенетического выравнивания.

Преобразование, создающее T^L

Пусть T^* — оптимальное филогенетическое выравнивание для дерева T . Будем преобразовывать T^* в поднятое выравнивание T^L последовательностью замен строк во внутренних вершинах T . Это преобразование важно для нас только концептуально, так как T^* мы не знаем, но оно поможет определить поднятое выравнивание T^L .

Скажем, что вершина была *поднята*, после того как пометим ее строкой из множества меток для листьев \mathcal{S} . По определению первоначально каждый лист поднят. Процесс подъема “поднимает” последовательно все внутренние вершины T в любом порядке, соблюдая лишь условие, что вершина поднимается после того, как подняты все ее дети. Получающееся в конце этого процесса дерево и является поднятым выравниванием T^L .

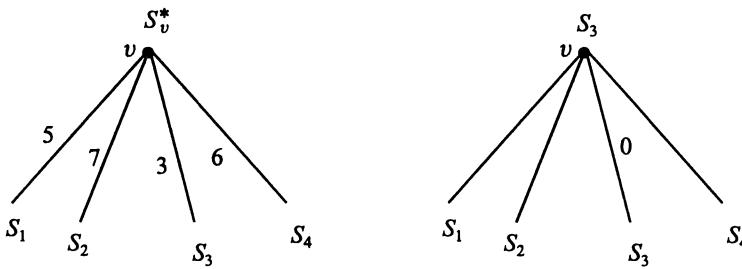


Рис. 14.9. Операция подъема в вершине v . Номера при дугах показывают их длины — расстояния от S_v^* до поднятых строк, помечающих детей v . Заметьте, что после подъема длина одной дуги станет нулевой

Мы поднимаем вершину v следующим образом. Пусть S_v^* — строка, помечавшая внутреннюю вершину v из T^* . Не умаляя общности, предположим, что дети v помечены поднятыми строками S_1, S_2, \dots, S_k из \mathcal{S} , и мы ссылаемся на детей v , используя их метки. Пусть S_j — та из этих меток, которая ближе всего к S_v^* . Значит, $D(S_v^*, S_j) \leq D(S_v^*, S_i)$ для любого i от 1 до k . Чтобы поднять v , заменим строку S_v^* на S_j (рис. 14.9). Это изменит длину каждой дуги, выходящей из v , на $D(S_j, S_i)$. В частности, станет равной нулю длина дуги из v в ту вершину, которая помечена строкой S_j .

Анализ ошибок

Теорема 14.8.1. *Поднятое T^L имеет полную длину, не превосходящую удвоенной длины оптимального филогенетического выравнивания T^* дерева T .*

Доказательство. Пусть $e = (v, w)$ — любая дуга в T , так что v — родитель w . Предположим, что в T^L вершина v помечена строкой $S_j \in \mathcal{S}$, а вершина w —

строкой $S_i \in \mathcal{S}$. Если $S_j = S_i$, то дуге e соответствует в T^L дуга нулевой длины, так что о ней беспокоиться не нужно. Но если $S_j \neq S_i$, то длина дуги e в T^L равна $D(S_j, S_i) \leq D(S_j, S_v^*) + D(S_v^*, S_i) \leq 2 \times D(S_v^*, S_i)$. Первое неравенство верно в силу неравенства треугольника. Чтобы проверить второе неравенство, заметим, что в той точке процесса подъема, когда помечается вершина v , S_i помечает одного из детей (w) вершины v и следовательно, является кандидатом на пометку v . Поэтому $D(S_v^*, S_j)$ не должно превосходить $D(S_v^*, S_i)$, и значит, $D(S_j, D_i) \leq 2 \times D(S_v^*, S_i)$. Теперь рассмотрим путь в T^* из v к листу, помеченному S_i . Обозначим этот путь через P_e . По неравенству треугольника $D(S_v^*, S_i)$ не превосходит суммы длин дуг P_e в T^* . Таким образом, длина дуги e в T^L не превосходит удвоенной длины пути P_e . Отметим, что путь P_e определяется только для дуги e , имеющей ненулевую длину в T^L .

Для дуги ненулевой длины $e = (v, w)$ путь P_e — это путь, вдоль которого листовая строка S_i была поднята до вершины w в T^L (рис. 14.10). Следовательно, каждая вершина на P_e , кроме v , помечена строкой S_i , и никакая вершина вне P_e строкой S_i не помечена. Так что если $e' = (v', w')$ — другая дуга ненулевой длины и $P_{e'}$ — путь, вдоль которого поднималась до w' ее метка, то P_e и $P_{e'}$ не имеют общих дуг (снова см. рис. 14.10). Это определяет отображение дуг ненулевой длины e дерева T^L в пути P_e в T^* , такое что 1) длина дуги e в T^L не более чем вдвое превосходит полную длину дуг P_e в T^* и 2) ни в какую дугу из T^* не отображается более одной дуги из T^L . Далее, если корень T^L помечен строкой $S \in \mathcal{S}$, то ни в какую дугу пути от корня до листа S в дереве T^* не отображается ни одна дуга T^L . Поэтому полная длина поднятого выравнивания T^L не превосходит удвоенной полной длины оптимального филогенетического выравнивания T^* . \square

Доказательство можно изменить так, чтобы неравенство было строгим. Теперь сформулируем главный результат.

Следствие 14.8.1. *Наилучшее поднятое выравнивание имеет полную длину, менее чем вдвое превышающую длину оптимального филогенетического выравнивания T^* .*

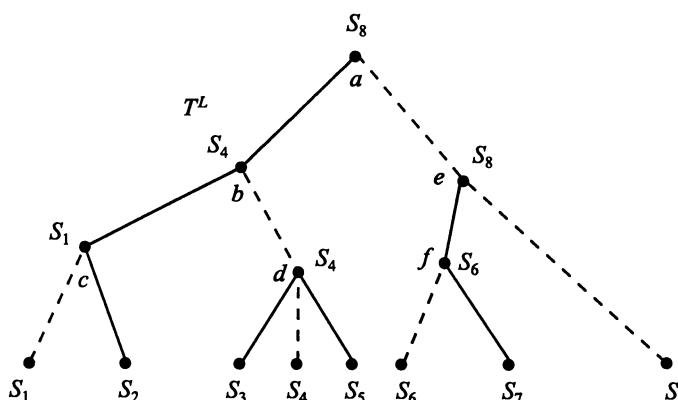


Рис. 14.10. Поднятое дерево T^L . Штриховые линии показывают пути, вдоль которых листовые строки поднимаются к внутренним вершинам. В T^L каждая из этих штриховых дуг имеет нулевую длину. Например, путь $P_{(a,b)}$ — это путь b, d, S_4 , вдоль него поднимается строка, помечающая b . Дуга (a, b) имеет в T^L длину, не более чем вдвое превышающую длину пути $P_{(a,b)}$ в T^* .

Вычисление поднятоого выравнивания минимальной длины

Наилучшее поднятое выравнивание вычисляется с помощью динамического программирования.

Определение. Пусть T_v — поддерево T с корнем в вершине v . Пусть $d(v, S)$ обозначает длину наилучшего поднятоого выравнивания T_v при условии, что вершине v сопоставлена строка S (предполагая, конечно, что S является строкой листа из T_v).

По завершении вычисления если r представляет корневую вершину, то значение наилучшего поднятоого выравнивания равно минимуму $d(r, S)$ по всем S из строк, соответствующих листьям T . Алгоритм ДП вычисляет значения $d(v, S)$, двигаясь от листьев наверх. Каждая вершина обрабатывается только после обработки всех ее детей. Алгоритм начинает с предположения, что все листья уже обработаны. Если v — внутренняя вершина, все дети которой — листья, то алгоритм полагает $d(v, S) = \sum_{S'} D(S, S')$, где S' — строка, сопоставленная одному из детей v .

Для произвольной внутренней вершины v рекуррентное соотношение ДП будет выглядеть так:

$$d(v, S) = \min_{v'} \min_{S'} \{D(S, S') + d(v', S')\},$$

где v' — ребенок v и S' — листовая строка в дереве $T_{v'}$.

Очевидно, что эти соотношения корректны и что алгоритм корректно находит длину наилучшего поднятоого филогенетического выравнивания. Чтобы найти само сопоставление строк внутренним вершинам, нужно еще выполнить обратный проход по рекуррентным соотношениям (или указателям), как это обычно делается в динамическом программировании.

Для временного анализа обозначим через k число листьев в T и предположим, что все C_k^2 расстояний между парами строк вычислены в препроцессной фазе. Эта фаза занимает время $O(N^2)$, где N — полная длина всех k строк. Тогда обработка любой внутренней вершины будет порядка $O(k^2)$, а полная работа алгоритма — $O(N^2 + k^3)$. В упражнениях мы предложим улучшение, которое уменьшит оценку времени счета до $O(N^2 + k^2)$. В итоге:

Теорема 14.8.2. *Оптимальное поднятое выравнивание можно вычислить за время, полиномиально зависящее от размера дерева и длин исходных строк.*

Следовательно, филогенетическое выравнивание, длина которого лежит в пределах двукратной длины оптимального, может быть вычислено за полиномиальное время. Это филогенетическое выравнивание будет выравниванием с центральной звездой, рассмотренным ранее для SP- и консенсусной целевых функций.

14.9. Замечания о приближениях с ограниченной ошибкой

В предыдущих пунктах мы рассмотрели три приближенных метода с ограниченной ошибкой. Другие аналогичные методы будут рассмотрены в части IV. Для читателей, незнакомых с приближениями с ограниченной ошибкой, следует подчеркнуть несколько моментов.

Во-первых, теорема 14.6.2 говорит, что выравнивание \mathcal{M}_c будет иметь оценку, которая никогда не превзойдет оптимальную SP -оценку более чем вдвое. Это не означает, что оценка \mathcal{M}_c будет вдвое больше оптимальной (всегда или обычно) или даже что она может быть вдвое больше оптимальной. На самом деле в небольшом числе тестов метода центральной звезды отклонение оценки \mathcal{M}_c от границы $\sum_{i < j} D(S_i, S_j)$ (и следовательно, от оптимальной оценки) составляло от 2 до 16 % [201].

Во-вторых, приближенные методы с ограниченной ошибкой не всегда наиболее эффективны или практичны и не всегда претендуют на использование в качестве “автономных библиотечных” методов. Скорее они рассматриваются как теоретические средства, используемые для получения *доказательных* утверждений о поведении алгоритмов при решении трудных задач. Однако методы с ограниченной ошибкой могут привести к очень эффективным методам, поведение которых не удается легко доказать, или стать их частью. Одним из способов является комбинирование с методами локальных улучшений, которые берут какое-либо решение и пытаются его улучшить небольшими изменениями. Другой способ — комбинация с методами ветвей и границ, где результат приближенного метода можно использовать как для верхней границы, так и для создания нижней границы оптимального решения.*.) Сходным образом, приближенные методы можно иногда применять для получения гарантированных практических границ для методов, которые таких границ не имеют, но считаются превосходными. Предположим, что приближенный метод на самом деле “плох”, имея в виду, что его результаты часто близки к гарантированной границе наихудшего случая, а некоторая другая эвристика на самом деле “хороша”, имея в виду, что ее результаты часто близки к оптимальным. Тогда на конкретных наборах данных отношение плохого значения к хорошему будет близко к гарантированной приближенной границе, *доказывая*, что хорошее решение на самом деле мало отличается от оптимального. В этом смысле приближенный метод с гарантированной границей ошибки будет всегда победителем — если он дает решения, которые лучше, чем у других методов, то он выигрывает в соревновании за верхнюю границу; если он дает решения много хуже, чем другие методы, то он также выигрывает, обеспечивая *доказанную* границу качества того решения, которое лучше. Пример для случая филогенетического выравнивания предложен в упражнениях и развит более глубоко в [206].

В-третьих, методы с фиксированными границами ошибки часто доводятся до методов, которые имеют *доказуемый баланс* между границей ошибки и временем счета. Применяя такие методы, называемые *приближенными схемами с полиномиальным временем*, пользователь знает время счета, достаточное для достижения любого конкретного уровня доверия (т. е. любого выбранного предела для максимального возможного отклонения от оптимума). Один из них был разработан для задачи SP -выравнивания Бафна, Лаулером и Певзнером [37]. Этот метод выравнивает k строк и строит множественное выравнивание для k строк, SP -оценка которого не может превысить оптимальную SP -оценку более чем в $2 - q/k$ раз для любого заданного q . Если q фиксировано, время счета полиномиально по n (предполагаемой длине каждой из строк) и по k . Когда q возрастает, то возрастает и гарантированная

*.) Некоторые методы с ограниченной ошибкой обладают весьма желательным свойством: для любой конкретной задачи алгоритм вычисляет неочевидную *нижнюю границу* (так же как и верхнюю границу) значения оптимального решения. Это не верно для метода центральной звезды, где используемая нижняя граница получается непосредственно и не связана с вычислениями верхней границы.

точность результата, но и время счета в наихудшем случае увеличивается вместе с ними. Такие приближенные схемы были предложены и для задачи филогенетического выравнивания [249, 453].

Наконец, результаты по приближениям с ограниченной ошибкой иногда пробуждают реакцию типа “невежество-это-счастье”. Эта реакция характеризуется убеждением, что доказанная граница максимальной возможной ошибки представляет собой выпад против алгоритма (а не довод в его пользу), если эта граница недостаточно мала. В своей крайней форме это убеждение принимает форму того, что эвристический метод, у которого максимальное возможное отклонение от оптимума неизвестно, лучше, чем такое, у которого оно известно, кроме тех случаев, когда доказанное отклонение очень мало. Это убеждение может также быть результатом того, что установление предела максимальной возможной ошибки смешивается с утверждением, что *обычная* ошибка находится на таком уровне. Кажется, что в этом крайнем карикатурном виде убеждение “невежество-это-счастье” навряд ли бытует, но на самом деле оно встречается (даже среди специалистов). Напротив, наша точка зрения заключается в том, что всегда хорошо получить алгоритм с доказанной границей ошибки, даже если такой алгоритм не будет самым эффективным в расчетах.

14.10. Обычные методы множественного выравнивания

Алгоритм выравнивания с центральной звездой был предложен как эвристика с ограниченной ошибкой и применяется нечасто. В этом пункте мы сосредоточимся на тех идеях, которым следуют практически используемые методы множественного выравнивания.

В литературе по вычислительной биологии опубликованы многочисленные методы (и программы) нахождения множественных выравниваний, и сотни (если не тысячи) статей сообщают о результатах множественных выравниваний по молекулярным данным. Две статьи, [92] и [316], служат начальной точкой для анализа используемых методов. Мы не будем пытаться делать полный обзор. Но почти все они являются вариантами одной из двух идей (или их смеси), так что целесообразно обсудить сами идеи в достаточно общих терминах, обращаясь к некоторым подробностям только для конкретных иллюстраций. Затем мы упомянем еще две совсем новые идеи.

14.10.1. Итеративное парное выравнивание

Большинство усилий в глобальном множественном выравнивании следует общей стратегии итеративного слияния двух множественных выравниваний двух подмножеств строк в одно множественное выравнивание объединения подмножеств. В своей простейшей форме этот подход использует оценки парного выравнивания для итеративного добавления строк по одной в растущее множественное выравнивание. Методы различаются по критериям выбора для слияния пар выравниваний (и следовательно, порядка выполнения слияний) и стилем, используемым для создания нового выравнивания.

Простейший пример такого подхода начинает с выравнивания двух строк с наименьшим редакционным расстоянием между ними, а затем последовательно добавляя к ним каждый раз строку с наименьшим редакционным расстоянием до какой-либо

из строк, уже находящихся в множественном выравнивании. Новая строка добавляется к растущему множественному выравниванию точно так же, как описано в доказательстве теоремы 14.6.1. Ясно, что метод может рассматриваться как нахождение множественного выравнивания, совместного с *минимальным оствовым деревом* (см. [112]), построенным по парным редакционным расстояниям. Когда вместо редакционных расстояний используются оценки сходства, такие методы можно рассматривать как нахождение множественных выравниваний, совместных с *максимальными оствовыми деревьями*, построенными по данным о сходстве.

Поскольку имеется несколько внешне различных способов построения минимального (или максимального) оствового дерева [112], то существует и несколько методов множественного выравнивания, которые кажутся различными, но в действительности создают одно и то же выравнивание. Метод, описанный в предыдущем абзаце, следует алгоритму Прима, а метод, основанный на алгоритме Краскала, сливал бы множественные выравнивания для подмножеств строк. Тем не менее даже этот метод определял бы, какую пару множественных выравниваний слить на основе парных редакционных расстояний (или оценок сходства) исходных строк, и использовал бы две исходные строки для слияния двух множественных выравниваний, как в доказательстве теоремы 14.6.1.

Более реалистический вариант предыдущего подхода мог бы выбирать для слияния два множественных выравнивания, основываясь на *среднем* редакционном расстоянии (или сходстве) между парами строк в двух подмножествах. Это называется *методом средней связи*, или UPGMA от Unweighted Pair-Group Method using arithmetic Averages (метод невзвешенных групповых пар, использующий арифметические средние). На каждом шаге слияния новое множественное выравнивание получалось бы выравниванием некоторого представления двух меньших выравниваний (например, профилей или консенсусных последовательностей), а не конкретных строк из подмножеств. В действительности если дать на вычисления больше времени, то на каждой итерации можно сначала вычислить все парные выравнивания существующих множественных выравниваний, а затем выбрать для слияния ту пару, у которой выравнивание получится лучше всего. (Другой вариант подхода с максимальным оствовым деревом будет рассмотрен в первой части п. 14.10.2.)

Итеративное выравнивание и кластеризация

Очевидно, что методы множественного выравнивания, основанные на парном итеративном выравнивании, являются прямым приложением *алгоритмов кластеризации* [217]. Другие методы глобального множественного выравнивания могут подобным же образом рассматриваться как перенос методов кластеризации, примененных к данным о редакционных расстояниях. По существу, каждый конкретный метод кластеризации определяет историю слияний, строя большие кластеры из меньших. Следовательно, почти любой метод, появляющийся в чисто кластерной литературе, можно адаптировать, чтобы он стал методом множественного выравнивания.

Вопрос о том, какой из этих кластерных вариантов приведет к более эффективному методу множественного выравнивания и привел ли к такому хотя бы какой-нибудь, остается открытым, и вероятно, ответ сильно зависит от конкретного биологического приложения. В результате без сомнения многие дополнительные варианты выравниваний этого типа будут разработаны и использованы на практике.

14.10.2. Две конкретные иллюстрации итеративного парного выравнивания

Целью приведенного выше обсуждения было передать дух методов итеративного парного выравнивания. Как читатель может себе представить, у изложенной идеи есть много конкретных вариантов. Здесь мы вкратце рассмотрим два варианта, появившихся в литературе. Один метод предназначен для распознания вторичной структуры белков, другой — для частичной реконструкции истории эволюции.

Итеративное множественное выравнивание для распознания вторичной структуры белков

Бартон и Стернберг [52] разработали и испытали метод множественного выравнивания, являющийся вариантом подхода максимального оствового дерева, который мы рассмотрели в предыдущем пункте. Целью их метода было выравнивание родственных последовательностей белков для обнаружения в них консервативных β -нитей (см. упражнение 36 главы 11).

Они испытали свой метод на наборе последовательностей иммуноглобулина, в которых истинное расположение β -нитей было известно по исследованиям с помощью трехмерной рентгеновской кристаллографии. Данные рентгеноанализа позволяют легко сопоставлять соответствующие области вторичной структуры. В этом приложении биологически информативный метод множественного выравнивания должен был бы, не зная истинного расположения β -нитей, предложить множественное выравнивание, в котором расположение β -нитей согласуется с данными рентгеноанализа. Значит, метод множественного выравнивания должен быть “настроен” так, чтобы наиболее важной характеристикой выравнивания была трактовка им (неизвестных) регионов вторичной структуры.

Метод множественного выравнивания в [52] сначала вычисляет парные оценки сходства строк для всех возможных пар, но не применяет их прямо для множественного выравнивания. Вместо этого он использует оценки, которые представляются более биологически информативными и получаются из оценок сходства. Для каждой пары строк S_i, S_j он случайным образом переставляет символы обеих строк и вычисляет оценку сходства для переставленных строк. Это делается для каждой пары сто раз, и затем вычисляется среднее и стандартное отклонение этой сотни оценок (прием называется в [127] “взбалтыванием”). Затем для каждой пары строк S_i, S_j оценка $sd(i, j)$ определяется как оценка сходства исходных строк S_i, S_j , деленная на стандартное отклонение, вычисленное по переставленным строкам S_i, S_j .

Интуитивное обоснование оценок $sd(i, j)$ очевидно. Если S_i и S_j содержат неслучайные структуры (предположительно, искомую вторичную структуру белка), перемежающиеся случайными последовательностями, то оценка выравнивания для S_i, S_j должна быть много больше, чем средняя оценка выравнивания переставленных копий этих двух строк. В переставленных строках исходная неслучайная структура разрушена. В [52] сообщается, что есть очень хорошая корреляция между оценками $sd(i, j)$ и степенью, до которой оптимальное выравнивание S_i и S_j корректно воспроизводит известные вторичные структуры в строках. Сообщается, что когда для изучаемых строк оптимальное парное выравнивание имеет оценку sd больше пяти, то выравнивание более чем на 70 % соответствует выравниванию, определенному рентгеноанализом. Таким образом, значения $sd(i, j)$ можно использовать, чтобы

обеспечить некоторое доверие к биологической информативности оптимального выравнивания, даже когда выравнивание получается из белков, где вторичная структура неизвестна.

Определив оценки sd по оценкам сходства, можно описать итеративный метод множественного выравнивания, предложенный в [52]. Этот метод начинает с нахождения пары строк с максимальной оценкой sd и оптимально выравнивает две строки. Затем последовательно он находит наибольшую оценку $sd(i, j)$ среди всех пар строк S_i, S_j , где S_i уже включена в выравнивание, а S_j еще нет, и присоединяет S_j к растущему множественному выравниванию. Присоединение S_j выполняется выравниванием S_j по профилю (см. п. 14.3.1) существующего выравнивания.

Ясно, что конкретные оценки sd , выбранные описанным методом, ведут себя точно так же, как любые другие оценки в задаче о максимальном остовном дереве. Таким образом, это просто метод кластеризации на основе максимального остовного дерева. Однако мы не можем описать получающееся множественное выравнивание как совместное с некоторой пометкой вершин в максимальном остовном дереве, потому что строка S_j выравнивается по профилю предыдущего выравнивания, а не по отдельной строке S_i .

В [52] вычисленные множественные выравнивания оценивались сопоставлением индуцированных попарных выравниваний (индуцированных множественным выравниванием) с оптимальными парными выравниваниями, для того чтобы увидеть, насколько хорошо проявились вторичные структуры (известные по рентгеноанализу). Результаты были отчасти противоречивы. Индуцированное парное выравнивание было существенно лучше в выравнивании вторичной структуры, чем оптимальное парное выравнивание, когда последнее отражало вторичную структуру очень слабо. Однако когда оптимальное парное выравнивание проявляло вторичную структуру очень хорошо, то индуцированное парное выравнивание делало это хуже. Таким образом, по сравнению с оптимальными индуцированные выравнивания сильно улучшали качество в трудных случаях и незначительно ухудшали в легких.

Итеративное множественное выравнивание в построении эволюционных деревьев

Мы показали значение множественного сравнения строк и выравнивания для характеристики семейств белков и при распознании важной молекулярной структуры, но, как утверждает Дулиттл:

“Самое интересное для нас — это *историческое* выравнивание. Оно должно как можно точнее отражать ряд расхождений, который привел к современным последовательностям” [127].

Это последнее из трех “крупномасштабных” использований множественного выравнивания.

Методы итеративного выравнивания определяют последовательность слияний дизъюнктных подмножеств строк. Следовательно, история этих слияний может быть описана двоичным деревом T . Каждый лист T представляет отдельную строку исходного набора, а каждая внутренняя вершина — слияние строк, соответствующих листьям ее поддерева (или, что эквивалентно, двух подмножеств, соответствующих ее потомкам). Каждая внутренняя вершина представляет также множественное выравнивание, созданное при слиянии, породившем эту вершину.

Если целью алгоритма является главным образом нахождение множественного выравнивания исходных строк, то дерево T является побочным продуктом, который может не представлять самостоятельного интереса. Но если считать, что критерии отбора слияний отражают историю эволюции (например, следуя той идее, что более похожие строки более родственны), тогда T представляет также построенное дерево эволюции таксонов, строки которого были исходными в задаче множественного выравнивания. Поэтому методы итеративного (кластерного) множественного выравнивания играют свою роль не только в создании представлений семейств и идентификации важных молекулярных структур, но и в выводе истории эволюции. Мы вернемся к соотношению между построением дерева и множественным выравниванием в п. 17.7.

Прогрессивное выравнивание

Возможно, самый прозрачный аргумент в пользу взаимоотношения между итеративным множественным выравниванием и выводом эволюционных деревьев появился в классической статье Фенга и Дулиттла [153] (см. также [127]), озаглавленной “Прогрессивное выравнивание последовательностей как предпосылка к созданию корректных филогенетических деревьев”.

Основная идея в том, что пара строк с минимальным редакционным расстоянием (и наибольшим сходством), скорее всего, получается из пары таксонов, которые дивергировали в эволюции позднее всего, и что парное выравнивание этих двух конкретных строк дает самую “надежную” информацию, которую можно извлечь из исходных строк. Поэтому любые пробелы (пропуски), появляющиеся в оптимальном парном выравнивании таких строк, должны сохраняться в общем множественном выравнивании по принципу “один раз пропуск — всегда пропуск” [153]. Идея неудаления пропусков повторяется многократно во всей последовательности слияний. Как и в доказательстве теоремы 14.6.1, последовательно сливать выравнивания, не удаляя пробелов, — дело простое. В действительности, если слитое выравнивание определено выравниванием профилей двух существующих выравниваний, то имеющиеся парные пропуски сохраняются автоматически.

Метод прогрессивного выравнивания Фенга и Дулиттла явно предназначен для построения по молекулярным данным эволюционного дерева при одновременном формировании эволюционно информативного множественного выравнивания. Это хороший пример итеративного парного выравнивания. Однако используемые в [153] критерии выбора того, какие два множественных выравнивания сливать на каждом шаге, несколько сложны, и интересующемуся читателю следует обратиться за подробностями к первоисточнику. Мы вернемся к взаимоотношению между эволюционными деревьями и множественным выравниванием в части IV этой книги.

14.10.3. Методы повторяющихся мотивов

Обратимся теперь ко второму важному подходу, обычно используемому в методах множественного выравнивания (как локального, так и глобального). Этот подход заключается в том, что сначала ищется подстрока или небольшая подпоследовательность, которая присутствует во многих строках набора. Этой общей подпоследовательности (часто требуется, чтобы она была подстрокой) даются разные имена,

такие как *мотив*, *якорь*, *ядро*, *блок*, *центральный блок*, *область*, *идентифицирующий сегмент*, *модель*, *колено*, *q-грамма*, *примитивное окно*, *консенсусный шаблон* и т. д. Мы будем использовать термин “мотив”; его “ширина” относится к длине общей подпоследовательности (или подстроки), а “кратность” — к числу строк, его содержащих.

Если найден “хороший” мотив (широкий и с высокой кратностью), содержащие его строки сдвигаются так, чтобы вхождения в них этого мотива расположились рядом друг с другом. Тогда завершение множественного выравнивания этих строк разбивается на две подзадачи меньшего размера, по одной для подстрок с каждой стороны от мотива. Эта рекурсия продолжается, пока не исчерпаются достаточно широкие или частые мотивы; с остающимися подзадачами можно справиться методами итеративного выравнивания. Строки, не содержащие первого хорошего мотива, выравниваются отдельно, и получившиеся два выравнивания сливаются в одно. В случае локального множественного выравнивания результат работы алгоритма может состоять просто из лучших мотивов, найденных по ходу работы, поскольку они являются интересующими нас подстроками.*)

Этот набросок представляет почти точное описание метода множественного выравнивания, предложенного в [374]. Другое удачное представление этого подхода с повторным выделением мотивов предложено Уотерменом и др. [455, 456, 467]. Широко доступная программа коммерческого уровня MACAW [398], разработанная Национальной медицинской библиотекой в NIH, использует подход мотивов на верхнем уровне, хотя и переключается на целевую функцию *SP* при выравнивании строк между якорями.

Есть множество способов, с помощью которых существующие методы стараются найти хорошие мотивы, но исходные идеи при этом близки. Обычно все (перекрывающиеся) подстроки фиксированного размера (окна) собираются и ищется подстрока, которая входит во многие из них. Иногда такое сравнение выполняется техникой хеширования, иногда более непосредственным сравнением. Оно может быть выполнено с помощью суффиксных деревьев (см. упражнения 32 и 33 главы 7) или с помощью сортировки и т. п. Иногда в каждом окне изменяется небольшое число символов, образуя множество сходных подстрок каждой исходной подстроки. Эти новые подстроки трактуются, как ранее, и таким способом можно делать мотив не из одинаковых, а из похожих подстрок. Когда мы находим подстроку, которая содержится во многих различных строках, обычные методы пытаются расширить изучаемый мотив добавлением дополнительных символов с каждой его стороны.

Как и в случае итеративных парных методов, существует большое число способов, которыми этот общий подход может быть превращен в конкретный метод, и процесс создания новых методов можно продолжать и дальше. Представляется, что теоретического способа для выбора лучшего варианта нет, так что сравнивать эти методы приходится на эмпирической основе. Однако есть одно теоретическое улучшение, которое можно добавить во многие опубликованные методы, следующие подходу повторяющихся мотивов. Этому улучшению будет посвящено упражнение 26. А сейчас мы обсудим вариант подхода, который имеет более точный, комбинаторный оттенок, чем многие другие методы повторяющихся мотивов.

*) Отметим, однако, что Мак-Клур и др. [316] нашли, что хорошие мотивы лучше выравниваются (и поэтому легко распознаются) существующими методами глобального выравнивания, которые не ищут мотивов явно, чем существующими методами повторяющихся мотивов. Отсюда вывод, что, возможно, локальное выравнивание лучше вычислять, извлекая упорядоченные мотивы из хорошего глобального выравнивания. Это замечание может быть оспорено.

Метод Вингрона—Аргос

Мартин Вингрон и Пэт Аргос [446] предложили метод множественного выравнивания с повторяющимися мотивами, который для случая трех строк можно кратко описать в терминах теории графов. На языке, отличном от использованного в [446], метод выглядит так.

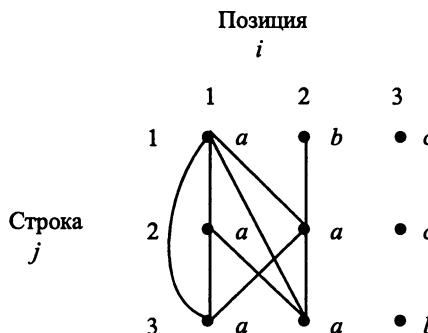


Рис. 14.11. Граф D для трех строк, каждая длины три. В этом примере две вершины соединяются, если они представляют начальные позиции подстрок длины два, имеющих не меньше одного совпадающего символа

Во-первых, образуем $3n$ вершин неориентированного графа D , по одной вершине на каждую пару (i, j) , где i представляет позицию в строке j (рис. 14.11). Каждой такой вершине соответствует подстрока длины l , где l выбрано разработчиком модели. Для каждой пары вершин (i, j) и (i', j') определим, не будут ли “достаточно близки” соответствующие им строки. Критерии достаточной близости также находятся в руках разработчика модели. Соединим вершины (i, j) и (i', j') , если их строки достаточно близки. Затем найдем и удалим из этого графа все ребра, которые не входят в клики размера три. Клика размера три представляет собой три подстроки трех исходных строк, которые все “достаточно близки”. Эти три подстроки формируют **мотив**. Теперь представим клику (мотив), образованную из вершин $(i, 1)$, $(i', 2)$ и $(i'', 3)$ тройкой (i, i', i'') . Скажем, что клика (i, i', i'') находится *слева* от клики (z, z', z'') в том и только том случае, если $i < z$, $i' < z'$ и $i'' < z''$. Две клики *не пересекаются*, если одна находится слева от другой. Теперь алгоритм пытается найти набор непересекающихся клик, “хорошо расположенных” в смысле, который сейчас будет уточнен.

Если клика (i, i', i'') находится слева от клики (z, z', z'') , то алгоритм назначает этой паре клик *вес расположения*, зависящий от того, насколько разности $i' - i$ и $i'' - i'$ совпадают с $z' - z$ и $z'' - z'$. Это снова находится в руках разработчика модели. Например, хорошо, если $i' - i = z' - z$ и $i'' - i' = z'' - z'$, поскольку тогда множественное выравнивание, которое выравнивает позицию i в строке 1 с позициями i' и i'' в строках 2 и 3, а позицию z — с z' и z'' , будет иметь совместное заполнение пробелами между этими двумя столбцами выровненных мотивов. Мотивы в этом случае заметно “параллельны”, такой паре клик нужно сопоставить большой вес. Паре плохо расположенных клик (представляющих мотивы, которые не параллельны) следует сопоставить малый вес.

Чтобы найти набор *хорошо расположенных непересекающихся клик*, построим ориентированный взвешенный граф G . Каждая вершина G представляет клику в D . Включим в G дугу из v в w в том и только том случае, если v представляет в графе D клику, находящуюся слева от клики, представленной w . Вес размещения этих двух клик сопоставляется дуге (v, w) . Каждой вершине (представляющей клику/мотив) сопоставляется также (большой) вес, отражающий качества мотива. Эти веса вершин опять-таки определяются разработчиком модели, но общая схема позволяет использовать любую желаемую оценочную матрицу и целевую функцию множественного выравнивания. Теперь найдем в G путь *максимального веса*. Предположим, что в нем k вершин. Эти вершины представляют k непересекающихся клик графа D , и следовательно, они могут задавать набор из k непересекающихся мотивов, которые образуют “якоря” трех выравниваемых строк. Это использование самого длинного пути в множественном выравнивании было впервые предложено в [444]. Подстроки между этими k якорями можно выровнить любым методом. Например, меняя относительные веса дуг по сравнению с весами вершин G , разработчик модели может перенести внимание с нахождения множественного выравнивания с большим числом хороших непересекающихся мотивов на меньшее число хорошо расположенных мотивов. Как изучать компромисс между этими двумя целями? Конечно, параметрической оптимизацией.

Важность формы

Возможность явно ввести понятие формы или расположения в целевую функцию множественного выравнивания очень важна, так как “хорошая” форма — это один из индикаторов “биологической осмысленности” выравнивания. При поиске структурной информации множественное выравнивание, показывающее несколько коротких областей высокого сходства, перемежающихся длинными областями низкого сходства, более правдоподобно, чем выравнивание, у которого совпадения широко разбросаны по последовательностям [433]. Для эволюционных выводов, когда родственные последовательности “хорошо выровнены”, строчки зачастую можно разбить на несколько классов так, что в каждом классе наборы пробелов в строчках будут похожи. Примером служит выравнивание на рис. 14.1. Множественное выравнивание с большим числом пробелов, где пробелы в каждой строчке совершенно не такие, как в других строчках, будет считаться подозрительным в биологических приложениях. Это ожидание эксплуатируется в работе, о которой сообщается в [45] (см. п. 17.3.2).

При переносе метода Винграна–Аргос на $r > 3$ последовательностей в идеале можно было бы основываться на нахождении клик размером r в графе D , содержащем rn вершин. Однако вычисления при таком поиске трудны; вместо этого строится надмножество ребер, содержащихся в кликах размера r . В [369] предложен алгоритм построения такого надмножества в терминах теории графов. Когда надмножество ребер построено, плотные подмножества ребер играют роль клик в нахождении хороших непересекающихся мотивов для выравнивания. Подробности слишком сложны для воспроизведения здесь.

14.10.4. Два более новых подхода к множественному сравнению строк

Существуют еще два новых опубликованных подхода к множественному сравнению строк, которые имеют совсем другой привкус, чем методы, рассмотренные до сих пор. Оба метода идут от более статистического, или стохастического, подхода

к анализу последовательностей. Первый из них, называемый *выборочным методом Гиббса* [295], является методом локального выравнивания, или локального сравнения. Интересующийся читатель отсылается к работе [295].

Второй метод (первоначально попавший в вычислительную биологию через работу Дэвида Хаусслера из Калифорнийского университета в Санта-Круз) основан на *скрыто марковских моделях* (hidden Markov model — HMM) [285]. В принципе метод HMM пытается построить марковскую модель (сеть), которая могла бы генерировать семейство требуемых молекулярных последовательностей (например, глобинов), используя неопределенные вероятности перехода по дугам и выхода из вершин. Метод пытается определить хорошие вероятности для вершин и дуг, находя наилучшее приближение данных выбранной последовательности к марковской модели. Эта подзадача является сложной задачей статистической оптимизации, которая полностью лежит вне области внимания данной книги. Оптимизированную HMM можно использовать для выяснения, не будут ли дополнительные последовательности хорошими кандидатами на членство в семействе. Стока-кандидат оценивается по вероятности того, что ее могла бы сгенерировать HMM. Это вычисление является простым упражнением на динамическое программирование.

Если HMM построена и оптимизирована, ее можно также использовать для построения множественного выравнивания набора строк. Прежде всего, для каждой строки вычисляется в HMM наиболее вероятный путь, генерирующий ее. Этот путь размещает в каждой последовательности пробелы. Затем последовательности с их пробелами выравниваются для получения множественного выравнивания.

14.11. Упражнения

1. Докажите корректность рекуррентных соотношений для профильного выравнивания из п. 14.3.1.
2. Формализуйте целевую функцию для выравнивания двух профилей и напишите рекуррентные соотношения для оптимального вычисления этого выравнивания.
3. Джон Кесесиоглу [267] заметил, что многие специальные строковые задачи можно рассматривать как задачи множественного выравнивания, выбирая подходящий способ оценки столбцов выравнивания. Например, *кратчайшую общую надпоследовательность* набора строк \mathcal{S} можно искать как кратчайшую строку, которая содержит каждую строку из \mathcal{S} в качестве подпоследовательности. Если каждый столбец множественного выравнивания \mathcal{S} оценивать числом различных символов, в нем содержащихся, то множественное выравнивание, минимизирующее суммарную оценку столбцов, определяет кратчайшую общую надпоследовательность \mathcal{S} . Подробно объясните это.
4. Рассмотрите, насколько *SP*-оценки пригодны для вычисления множественного выравнивания. То есть почему и/или когда следует ожидать, что множественные выравнивания, полученные при целевой функции *SP*, будут выдавать ту информацию о последовательности, которая нужна биологам? Рассмотрите, насколько пригодны производство по множественному выравниванию консенсусной строки и оценка качества множественного выравнивания по консенсусной строке.
5. Докажите корректность рекуррентных соотношений ДП для множественного выравнивания, приведенных в п. 14.6.1.

6. Объясните подробнее, почему прямое динамическое программирование так важно при реализации ускорения, описанного в п. 14.6.1.
7. В методе центральной звезды множественное выравнивание строится последовательным пристраиванием каждой новой строки к центральной строке S_c . Однако порядок пристраивания этих строк не уточнялся. Покажите, что при любом порядке в результате получится одно и то же множественное выравнивание.
8. В доказательстве теоремы 14.6.1 (строящем множественное выравнивание $\mathcal{M}(T)$, совместное с помеченным деревом T) порядок обработки ребер T может меняться. На первый взгляд может показаться, что разные порядки обработки дадут различные множественные выравнивания. Докажите, что это не так и независимо от порядка обработки получится одно и то же множественное выравнивание.
9. Интуитивно кажется, что в методе центральной звезды можно получить выравнивание с лучшей SP -оценкой, выравнивая каждую следующую строку с профилем существующего выравнивания. Конечно, сначала результирующие выравнивания будут лучше. Но если это делать для каждой новой строки, то будет ли окончательное выравнивание обязательно лучше того, которое получается в исходном методе центральной звезды? Сохранится ли множитель два, если присоединения делать новым способом? Как порядок присоединения будет влиять на качество выравнивания?
10. Вспомним оценки sd , рассмотренные в п. 14.10.2. Нам бы хотелось использовать их в предложенных в этой главе ограниченных приближенных методах. Однако оценки sd не обязательно удовлетворяют неравенству треугольника, которое нужно для гарантии ограниченности ошибки. Похоже ли, что на практике оценки sd удовлетворяют неравенству треугольника? Проведите эмпирическое исследование этого вопроса.
11. Вспомним взвешенное SP -значение (п. 14.6.3), в котором индуцированная оценка каждой пары (S_i, S_j) умножается на вес $w(i, j)$. Попробуйте распространить приближенный метод центральной строки на этот взвешенный вариант задачи.
12. Если мы ослабим утверждение леммы 14.7.1, сказав, что существует такая строка $\bar{S} \in \delta$, что $E(\bar{S}) \leq 2$ (а не $2 - 2/k$), то ее доказательство станет проще. Сделайте это упрощение.
13. Доказательство леммы 14.7.1 можно модифицировать так, чтобы установить, что если строка \bar{S} — ближайшая к S^* в δ , вторая ближайшая или третья ближайшая, то $E(\bar{S})/E(S^*) \leq 2$. Докажите это. (Задача предложена Майком Патерсоном.)
14. Предположим, что в SP -задаче вместо M_c в качестве центральной строки случайно выбирают одну из исходных строк S_i . Покажите, что при усреднении по всем выборам S_i ожидаемое SP -значение результирующего множественного выравнивания будет не больше $2M$. Далее, используйте это среднее и тот факт, что минимальное значение равно M , чтобы показать, что медиана значений не превосходит $3M$. Наконец, установите, что при помещении случайной строки S_i в центр звезды результирующее множественное выравнивание находится в трехкратной границе оптимального значения SP с вероятностью не меньше одной второй. Проведите аналогичный анализ для консенсусной целевой функции.
15. SP -оптимальное множественное выравнивание должно иметь значение, не меньшее $\sum_{i < j} D(S_i, S_j)$. Эта нижняя граница оптимального значения полезна не только для получения гарантированной границы $2 - 2/k$ для приближения центральной строки, но и для получения лучшей эмпирической границы ошибки, когда метод применяется к любому конкретному набору строк. При рассмотрении консенсусных критериев для множественного выравнивания никаких похожих нижних границ получено не было, не было

и метода для вычисления эмпирических границ ошибки. Ниже мы предлагаем такой метод. Докажите, что он дает нижнюю границу цены оптимальной штейнеровской строки.

Для простоты предположим, что набор \mathcal{S} содержит четное число строк. Построим полный взвешенный граф G_D , где каждая вершина представляет одну из строк набора \mathcal{S} . Назначим вес $D(S_i, S_j)$ ребру между вершинами, представляющими S_i и S_j . Полным паросочетанием C в G_D называется набор ребер, в которые каждая вершина входит ровно по одному разу как концевая вершина. Весом C называется сумма весов ребер из C . Максимальный вес полного паросочетания в G_D может быть найден эффективно. Этот вес является нижней границей цены оптимальной штейнеровской строки.

16. Дайте подробное обоснование того, что оптимальное поднятое выравнивание можно корректно найти за время $O(N^2 + k^3)$.

17. Время на вычисление оптимального поднятоого выравнивания можно уменьшить до $O(N^2 + k^2)$, используя следующее определение и наблюдение: упорядоченная пара последовательностей (S, S') называется *законной* для дуги (v, v') , если S — строка листа в T_v , а S' — строка листа в $T_{v'}$. Предположим, что v является родителем v' . При вычислении рекуррентных соотношений $d(v, S) = \sum_{v'} \min_{S'} \{D(S, S') + d(v', S')\}$ нужно учитывать для каждой пары (v, v') только законные пары.

Покажите, что упорядоченная пара строк может быть законной парой для не более чем одной дуги дерева. Предложите алгоритм трудоемкости $O(k^2)$ для нахождения законных пар каждой дуги. Затем оденьте подробностями метод трудоемкости $O(N^2 + k^2)$.

18. В задаче филогенетического выравнивания мы бы хотели иметь эффективный алгоритм, который по любой конкретной задаче вычисляет *нижнюю границу* значения оптимального решения. Докажите, что для любой конкретной задачи цена оптимального филогенетического выравнивания не меньше половины веса минимального оствового дерева T , образованного по данным о парных расстояниях, как описано в п. 17.5.2.

Покажите, например, что полная цена минимального оствового дерева не является нижней границей, и объясните, почему.

19. **Однородные поднятые выравнивания** [453]. В этой задаче мы рассматриваем другой способ преобразования оптимального филогенетического выравнивания T^* в поднятое выравнивание, называемое T'' . Сначала разобьем вершины T на уровни по числу дуг от корня. Будем подниматься снизу вверх, по одному уровню за раз. Для простоты предположим, что T — двоичное дерево, так что каждая внутренняя вершина имеет двух детей, но полноты двоичного дерева T не предполагается. Для каждой внутренней вершины v произвольно назовем одного из потомков красным — $r(v)$, а другого синим — $b(v)$. Теперь поднятое выравнивание назовем *однородным поднятым* выравниванием, если на каждом уровне все внутренние вершины получают их поднимаемые метки от потомков одного цвета.

Преобразование T^* в T'' выполняется снизу вверх, по одному уровню за раз. Пусть $V(i)$ — внутренние вершины уровня i . Чтобы поднять метки на уровень i (с уровня $i+1$), рассмотрим две суммы: $\sum_{v \in V(i)} D(S_v^*, S_{r(v)})$ и $\sum_{v \in V(i)} D(S_v^*, S_{b(v)})$, где $S_{r(v)}$ и $S_{b(v)}$ — соответственно, метки красного и синего потомка v . Если первая сумма не превосходит второй, то назначим каждой внутренней вершине уровня i поднятую метку ее красного потомка, а в противном случае — синего. Когда все уровни будут пройдены, то получится однородное поднятое выравнивание T'' .

Доказать, что однородное поднятое выравнивание T'' имеет полную длину, превосходящую не более чем вдвое длину оптимального филогенетического выравнивания T^* . Доказательство можно моделировать анализом поднятоого выравнивания T^L , представленного в тексте.

20. Снова предполагая дерево двоичным, предложите алгоритм с полиномиальным временем для нахождения однородного поднятого выравнивания наименьшей длины. Его можно найти за время $O(n^2)$, а не за $O(n^3)$, как оптимальное поднятое выравнивание по алгоритму, описанному в тексте.
21. Перенесите определения, методы и анализ однородного поднятого выравнивания на произвольное дерево, сняв предположение, что дерево двоичное.
22. Ясно, что наилучшее однородное поднятое выравнивание будет иметь длину не меньшую, чем лучшее поднятое выравнивание. Кроме эффективности по времени, какая еще может быть польза от нахождения однородного поднятого выравнивания с границей ошибки два? Совет. Отметьте, что назначение цветов потомкам произвольно, и вспомните комментарии в пункте 14.9 о пользе приближений с (плохой) границей ошибки.
23. Снова предположим, что T — двоичное дерево, которое не обязательно сбалансировано. Раскладка T определяется выбором для каждой внутренней вершины v , кого из потомков считать левым, а кого правым. Как много есть различных раскладок у T ? Как много у T есть поднятых выравниваний?
24. Ясно, что каждое поднятое выравнивание является однородным поднятым выравниванием для некоторой раскладки T (или раскладок). Пусть d — глубина T (максимальное число дуг в пути от корня до листа). Покажите, что каждое поднятое выравнивание является однородным поднятым выравниванием для 2^d раскладок T .
25. Используя ответы предыдущих двух задач, установите, что в любом наборе из 2^d различных поднятых выравниваний по крайней мере одно имеет длину, которая не более чем вдвое превосходит длину оптимального филогенетического выравнивания T^* . Далее улучшите это утверждение до двух различных поднятых выравниваний. Следовательно, в сбалансированном двоичном дереве с n листьями не менее $2/n$ из всех поднятых выравниваний хуже оптимального не более чем вдвое.
26. Задачи о множественных подстроках, рассмотренные в пп. 7.6 и 9.7, допускают ясную формализацию фазы поиска точного совпадения, используемой в нескольких методах нахождения мотива и множественного выравнивания. Таким образом, решение за линейное время множественной задачи о подстроке (и ее вариантов) дает мощное средство, которое можно использовать в самом центре многих программ биологического анализа последовательностей. Рассмотрим для конкретности одну частную программу, разработанную Леунгом и др. [298], которая находит множественное совпадение с ошибками в наборе длинных строк.
- Эта программа ищет подстроки, которые похожи и входят в “достаточное” число исходных строк. На фазе точного совпадения программа сначала опознает некоторые “центральные блоки”. Блок *кратности* m — это набор из m подстрок, каждая из отдельной строки (в наиболее интересном случае), которые *идентичны* и максимальны для этого свойства. То есть любое расширение строк на один символ в любую сторону сделает различными по крайней мере две строки из этих m . В программе пользователь устанавливает m и параметр c_m , задающий минимально приемлемую длину блока, о котором стоит сообщать. Интуитивно, если m и c_m достаточно велики (как решил пользователь), то блок является нетривиальным сохранившимся точным образцом, который проявился в значительном числе строк. Локальное множественное выравнивание строится в программе окружением ядра дополнительными блоками *тождества* (блоками из одинаковых подстрок, которые не так длинны, как центральный блок), находящимися не дальше предписанного расстояния от предыдущего блока тождества. На результат работы программы существенно влияет выбор m и c_m . Предложенные в [298] выборы m и c_m основаны на статистических результатах для случайных строк и не определялись по настоящим данным.

Опишем общую задачу нахождения центральных блоков в терминах множественной задачи об общей подстроке.

Напомним, что алгоритм с линейным временем из п. 9.7 строит таблицу, показывающую для каждого возможного значения k длину наибольшей подстроки, общей для по меньшей мере k исходных строк. Таблица содержит *все* компромиссы между m и c_m для данных исходных строк. Объясните это подробно и укажите преимущества использования таблицы для выбора m и c_m по сравнению с предопределеными значениями. Покажите, что после того, как выбор m и c_m сделан, строки центрального блока могут эффективно локализоваться с помощью того суффиксного дерева, которое применялось при создании таблицы.

Покажите, что, используя алгоритм множественной общей подстроки, можно выбрать значения m и c_m для любых исходных данных, обращая внимание либо на длину подстроки, либо на ее кратность. Значит, можно сначала выбрать желаемую длину строки c , а затем найти наибольшее m , для которого подстрока длины c найдется в m из q строк, а можно идти в обратном направлении — от m к c .

Другую попытку формализовать определения и вычисления, относящиеся к методам повторяющихся мотивов, см. в [385].

Базы данных для последовательностей

Обратимся теперь к доминантному, наиболее созревшему и наиболее успешному применению строковых алгоритмов в вычислительной биологии: формированию и использованию баз данных, содержащих молекулярные последовательности.*¹) Начнем с иллюстрации некоторых применений баз данных последовательностей и с небольшого описания существующих баз данных. Эффект поиска в базах данных (и ожидание еще большего эффекта в будущем) объясняет большую часть интереса биологов к алгоритмам поиска, манипулирования и сравнения строк. В свою очередь деятельность биологов стимулировала дополнительный интерес к строковым алгоритмам у специалистов по информатике.

После описания “почему” и “что такое” базы данных для последовательностей мы обсудим некоторые вопросы, “как” работают строковые алгоритмы, которые применяются при организации базы данных и при поиске в ней.

Почему базы данных для последовательностей?

Всеобъемлющие базы данных/архивы, содержащие последовательности ДНК и белков, заняли прочное место среди главных средств современной молекулярной биологии — “электронные базы данных быстро становятся живительной силой в этой области” [452]. Основная причина — мощь биомолекулярного *сравнения последовательностей*. Это проявилось в *первом факте анализа биологических последовательностей* (глава 10, с. 265). При такой эффективности сравнения последовательностей в молекулярной биологии естественно *накапливать и систематизировать* биопоследовательности, которые нужно сравнивать, что естественно приводит

*¹) Некоторые считают, что в биологии нет настоящих *баз данных* молекулярных последовательностей, а существуют только *библиотеки* или *архивы*. Они полагают, что настоящая база данных должна интегрировать данные продуктивнее и допускать более сложные запросы, чем это сейчас делается. Однако здесь мы будем пользоваться терминами “база данных” и “архив” как равнозначными.

к росту баз данных. Мы начнем эту главу с нескольких иллюстраций моши сравнения последовательностей в виде поиска в базах данных.

15.1. Истории успешного поиска в базах данных

15.1.1. Первая успешная история

Организация баз данных и поиск в них приобрели промышленный характер, и “открытия, основанные исключительно на гомологии последовательностей, стали рутиной” [360]. При этом только четырнадцать лет назад появилось сообщение о первом важном успехе этого подхода [132].

В начале 1970-х гг. было твердо установлено, что инфицирование некоторыми вирусами может вызвать безграничный рост клеток в культуре (*in vitro*). Канцеро-подобная *трансформация* клеток культуры вирусами вызвала предположение, что вирусная инфекция может быть причиной рака животных, но механизмы развития рака были неизвестны. Предположили, что некие гены в инфицирующих вирусах (названные *онкогенами*) кодируют *клеточные факторы роста* — белки, необходимые для стимулирования или продолжения роста клеточной колонии. Инфицированные вирусами клетки, таким образом, синтезируют неконтролируемые количества фактора роста, позволяя клеточной колонии расти больше своих нормальных размеров. Эта гипотеза теперь широко принята. “Однако связь между онкогенами и факторами роста не обнаружилась при прямой проверке этой гипотезы — она обнаружилась как неожиданный результат сопоставления двух независимых наборов данных при компьютерном поиске” [110].

Вирус саркомы Симиана является *ретровирусом* (под этим понимается, что его геном состоит из двойной спирали РНК, которая в инфицированной клетке должна быть конвертирована в ДНК прежде, чем вирус сможет копироваться); о нем с начала 1970-х гг. известно, что он вызывает рак у некоторых видов обезьян. Ответственный за это онкоген, именуемый *v-sis*, был изолирован и расшифрован в 1983 г. Примерно в то же время была разработана и опубликована частичная аминокислотная последовательность важного фактора роста — *тромбоцитарного фактора роста* (platelet-derived growth factor — PDGF). Р. Ф. Дулиттл поддерживал доморощенную базу данных опубликованных аминокислотных последовательностей (вводимых вручную с частичным привлечением домашней рабсилы) и загрузил в нее транслированную аминокислотную последовательность онкогена *v-sis*. Когда была опубликована частичная аминокислотная последовательность PDGF, он сравнил ее со своей базой данных и обнаружил область из 31 аминокислотного остатка с 26 точными совпадениями между последовательностью PDGF и последовательностью, кодируемой геном *v-sis*. В другой области с 39 остатками он нашел 35 точных совпадений [132]. Этот поиск и аналогичный компьютерный поиск другой группы “впервые помогли объяснить физиологическое действие онкогенов — неопластическая трансформация была бесспорным следствием неограниченной экспрессии белка, стимулирующего пролиферацию нормальных клеток” [110]. Для наших целей “неопластическая трансформация” может быть переведена как “рак”.

Эта впервые полученная взаимосвязь между онкогеном и нормальным белком указала способ, с помощью которого была обнаружена и с этого времени понята сущность онкогенеза. Теперь уже показано, что многие дополнительные онкогены

очень похожи (но не совпадают) в своих последовательностях с генами, которые кодируют белки регуляции роста в нормальных клетках. Некоторые из этих связей между онкогенами и белками были получены при поиске сходства в белковых базах данных. Нормальные гены для ростовых факторов были названы *protoонкогенами*, и теория считает, что безобидный вирус становится онкогенным после включения в его геном protoонкогена его хозяина. В вирусном геноме этот protoонкоген затем мутирует или оказывается близко к гену-усилителю или далеко от гена-репрессора, в результате, когда этот вирус (с его онкогеном) инфицирует нормальную клетку, производится избыточное количество продукта protoонкогена (например, фактора роста).

15.1.2. Более свежий пример успешного поиска в базе данных

Первая полная последовательность ДНК свободноживущего организма была опубликована летом 1995 г. [162]. В последовательности были идентифицированы в общей сложности 1743 предполагаемые кодовые области (в основном предполагаемых генов). Это было сделано с помощью средств, которые мы здесь обсуждать не будем (один подход вкратце рассматривается в п. 18.2). Каждая из этих 1743 строк была затем переведена в аминокислотную последовательность (или несколько, если есть варианты установки рамки считывания) и использована для поиска “достаточно похожих” последовательностей в базе данных белковых последовательностей Swiss-Prot. При этом 1007 предполагаемых кодирующих генов не просто совпали с записями в базе данных, но совпали таким безусловным образом, что для каждой из них можно было получить (или предположить) конкретную биохимическую функцию. Эти заключения оказались возможными благодаря связи Swiss-Prot с другой базой данных (базой данных Riley), где расшифрованные белки разделены на 102 группы с подробной детализацией биохимических ролей. Таким образом, когда предполагаемая аминокислотная последовательность, кодируемая одним из 1743 генов, хорошо совпадает с известной последовательностью, о биохимической функции нового белка можно делать конкретные предположения. И такие предположения оказались возможны в 1007 случаях. Полный список 102 биохимических ролей появился в [162], и поразительно, как много подробной информации о предполагаемых генах было получено (или предположено) в результате поиска в базе данных по одной лишь последовательности.

15.1.3. Косвенные приложения поиска в базе данных

Два предыдущих примера иллюстрировали самое прямое использование баз данных биологических последовательностей для поиска похожих генов, регуляторных последовательностей или родственных белков в полученных новых последовательностях. Теперь это обычная практика. Когда клонируется и дешифруется новый ген (или новый белок), посещение соответствующих баз данных — обязательный следующий шаг. Но базы данных для последовательностей используются и по-другому. Значительная роль принадлежит поиску в базе данных в связи с *кластеризацией* сходных последовательностей в семейства последовательностей. Такие семейства могут обнаруживать важные сохраненные биологические феномены, которые не наблюдаются в лабораторных исследованиях и которые трудно распознать, глядя только на пару последовательностей. Придумано много умных способов применения баз данных для

последовательностей в преодолении и биологических и биотехнических проблем. Мы приведем два примера. Первый — это техническая задача, а второй — проблема, решение которой открыло важные биологические факты. Другие примеры косвенного использования поиска в базах данных были представлены в пп. 7.3 и 7.5. Дополнительные иллюстрации появятся в пп. 16.9 и 18.2.1.

Сборка последовательности в бактериях

Некоторые аспекты сборки последовательности рассматривались в предыдущем изложении, и она еще будет подробно рассматриваться в п. 16.15. Здесь мы рассмотрим только следующую абстрактную задачу, встречающуюся при сборке. Предположим, что очень длинная подстрока ДНК от конкретной (изучаемой) бактерии была разрезана на много более мелких подстрок (называемых *фрагментами*) и вам эти фрагменты достались в полном беспорядке.*.) То есть фрагменты переставлены так, что их первоначальный порядок в длинной подстроке совершенно потерян. Задача сборки последовательности заключается в восстановлении исходного порядка фрагментов.

Понадобятся два ключевых факта. В большинстве бактерий основная часть ДНК (около 85 %) содержится в генах, и бактериальные гены изредка разрываются инtronами (прерывающими последовательностями между генными). Таким образом, случайный участок бактериальной ДНК, скорее всего, участвует в кодировании белка, производимого бактерией. Этот факт наряду с мощью сравнительного анализа молекулярных последовательностей использовался Флейшманом и др. [162] для облегчения сборки бактериальной последовательности.

Один элемент их метода сборки заключается в том, что каждый фрагмент дешифруется, переводится в аминокислотную последовательность (на самом деле в три, так как рамка считывания не установлена) и затем для каждого фрагмента ведется поиск в базе данных для белков подстрок известных белков, похожих на этот фрагмент. Мы должны идентифицировать пары фрагментов ДНК, которые либо соседствуют в искомой бактериальной последовательности, либо близки друг другу. Поиск в базе данных помогает найти некоторые из таких пар. Два фрагмента, которые оказались очень похожи на два интервала в одной белковой последовательности (из базы данных), могут считаться пришедшими из одного интервала искомого бактериального генома, кодирующего один белок. Предполагается, что этот белок в искомой бактерии гомологичен тому белку в базе данных, которому подошли эти два фрагмента. Вдобавок если эти два фрагмента совпали со смежными областями в белке из базы данных, то они предполагаются смежными и в бактериальном геноме. Таким образом можно получить некоторые куски (контиги) правильного порядка фрагментов. Конечно, этих совпадений с базой данных недостаточно для того, чтобы собрать полный порядок, но их использование является важной частью метода сборки в [162].

Рассеянный склероз и поиск в базе данных

Четкий пример того, как поиск в базе данных можно комбинировать с лабораторной молекулярной биологией, был упомянут в газете *New York Times* [79]. Рассеянный склероз — истощающее неврологическое заболевание, которое еще недостаточно изучено. Однако уже понятно, что это *автоиммунное заболевание*, т. е. иммунная система неправильно воспринимает клетки собственного организма как инородные.

*.) На самом деле разрезается много экземпляров длинной подстроки и куски перекрываются. Эта деталь важна для общего случая, но не для иллюстрируемого здесь конкретного использования баз данных

При рассеянном склерозе *миелиновая оболочка*, окружающая нервные клетки, атакуется иммунной системой, что нарушает нормальную передачу сигналов по нервам. Первой линией атаки в иммунной системе являются *T*-клетки, которые распознают постороннее вещество. Однажды распознанный антиген другими элементами иммунной системы атакуется и уничтожается. Организм производит специфические *T*-клетки в качестве реакции на выявление того или иного постороннего (чужеродного) вещества (антигена).

Недавно были найдены *T*-клетки, которые идентифицируют белки или белковые фрагменты (полипептиды), появляющиеся на поверхности миelinовых клеток. Естественно предположить, что эти *T*-клетки (которые распознают белки на миелиновой поверхности как посторонние, некорректно разрушая их) были ранее созданы иммунной системой, чтобы (правильно) идентифицировать очень похожие белки на поверхности бактерий или вирусов. Значит, иммунная система атакует миелиновую оболочку потому, что спутала некоторые белки на ее поверхности с белками на поверхности каких-то бактерий или вирусов, которые ранее инфицировали организм. Но как можно проверить это предположение? Какие бактерии и вирусы здесь участвуют?

Для последовательностей белков миелиновой поверхности выполнили поиск в белковой базе данных похожих белков в бактериях и вирусах. Было найдено около сотни белков. Лабораторное исследование установило, что специфические *T*-клетки, которые атакуют миелиновую оболочку, атакуют также и те белки, которые были найдены в базе данных. Этот подход, комбинирующий поиск данных и лабораторное исследование, не только подтвердил сформулированное выше предположение, но и указал конкретные бактериальные и вирусные белки, которые можно перепутать с белкам на миелиновой поверхности. Теперь есть надежда, что при помощи анализа сходства этих бактериальных и вирусных белковых последовательностей (пример множественного сравнения последовательностей) можно лучше понять, какие свойства миelinовых белков использованы *T*-клетками при ошибочной идентификации миelinовых клеток как посторонних.

15.2. Промышленность баз данных

Основные архивы ДНК содержат последовательности суммарной длиной более 500 000 000 пар оснований, полученных из частей 300 000 различных генов многих организмов. Основные архивы белковых последовательностей содержат последовательности порядка 25 000 000 аминокислот, полученных из около 100 000 различных белков. Эти числа растут экспоненциально; по стандартной прикидочной оценке число пар оснований в архивах ДНК возрастает ежегодно на 75 % [479]. И конечно, когда различные геномные проекты будут в разгаре расшифровки ДНК, приведенные цифры покажутся допотопными.

Растут не только размеры баз данных, но и их число [1]. Дедушкой современных архивов ДНК является GenBank, на который была возложена функция хранения и обслуживания всех когда-либо опубликованных последовательностей ДНК.*¹⁾ Большую часть своей жизни GenBank содержался лос-аламосскими национальными

*¹⁾ В прежние времена, впрочем не так уж давно, сотрудники GenBank просматривали журналы и вводили найденные в них последовательности ДНК в базу данных вручную. Сейчас большинство данных предоставляется банку в электронном виде, независимо от того, опубликованы последовательности или нет, и своевременное представление данных часто оговаривается спонсирующими организациями.

лабораториями (LANL), а теперь содержится Национальным центром биотехнологической информации (NCBI) в Национальной медицинской библиотеке (NLM), которая входит в систему Национального института здоровья (NIH). Европейцы ведут по существу эквивалентный архив ДНК, библиотеку данных EMBL; еще один архив существует в Японии (DNA DataBase of Japan — DDBJ). Имеются еще новая обширная база данных по ДНК, Genome Sequence DataBase (GSDB), которая является ответвлением GenBank, и проект Национального центра геномных ресурсов (NCGR), поддерживаемый Министерством энергетики США (DOE). Эти четыре полные базы данных ДНК делят информацию между собой, и представление информации в одну из них равносильно представлению во все четыре. Что касается белков, то главными архивами белковых последовательностей являются Protein Information Resource (PIR) в США и Swiss-Prot в Европе.

Хотя может показаться, что акронимов уже много, их на самом деле значительно больше. За последние несколько лет было создано около сотни специализированных баз данных, содержащих кроме последовательностей еще и другие данные (см. в [1] обзор некоторых из них). Некоторые из этих баз данных и их использование уже упоминались в п. 3.5.1. Базы данных с такими названиями, как dbEST, dbSTS, FLYBASE, HAEMA, HAEMB, HLA, p53, REBASE, REPBASE, TRANSFAC, SGD, YPD, EcoCyc, XREFdb, CGSC, ECDC, NRBP, EGAD, HCD, SST, ALU, BERLIN, EPD, KABAT, LISTA, SBASE, YEAST, GOBASE, появились в последнее время.* Базы данных отличаются по некоторым показателям. Некоторые специализируются на конкретном организме или типе клеток, некоторые сосредоточены на отдельных биологических функциях, некоторые используют специальную терминологию и стиль систематики, характерные для особой области биологии, некоторые пытаются записать все мутации и различия (полиморфизмы), открытые в данном гене или наборе генов, и наконец, некоторые базы отличаются тем, каким образом последовательности хранятся и интегрируются с другой биологической информацией, а также способами выдачи информации и предлагаемыми услугами обработки запросов. (GenBank традиционно поддерживается как плоский файл, тогда как более новые базы данных — реляционные или объектно-ориентированные, и многие из них пытаются интегрировать различные типы биологических данных.)

Все базы данных функционируют как хранилища последовательностей, допуская вызов любой последовательности по названию или номеру доступа. Но более важный аспект, чем эта функция хранения и передачи данных, заключается в том, что базы данных работают на “создание нового знания” [130], допуская запрос последовательностей, основанный на *поиске сходства последовательностей* того или иного типа. Более того, многие из этих баз данных доступны через Интернет [76, 215, 452] и существует целый ряд программных пакетов, позволяющих пользователю искать и просматривать в архивах последовательности и манипулировать с ними (наиболее широко используемый пакет — это GCG (Genetics Computer Group), первоначально разработанный в университете Висконсина).

Использование баз данных для последовательностей растет так же быстро, как сами эти базы. Swiss-Prot насчитывает около 400 000 запросов в месяц. Аналогично, GenBank сейчас обслуживает около 10 000 запросов в день, но полное использование

*) Пусть те, у кого мысли не заняты полностью миром биологических баз данных или акронимов, попробуют хотя бы просто запомнить различные архивы, не говоря уже о попытках понять различия в биологических сообществах и политической истории, которые вложили свою лепту в этот взрыв появления новых особей в семействе баз данных.

GenBank (или EMBL) нельзя правильно оценить, так как многочисленные копии этих баз данных распределены по серверам во всем мире, и можно перекачать себе текущие версии и вести поиск на своих собственных машинах. Такая возможность на самом деле удобна для биотехнических и фармацевтических компаний, которые хотят сохранять в тайне свои исследования, чтобы никто по их запросам к базе не мог понять, над чем они работают.

Со всеми этими различными базами данных для последовательностей (не говоря уже о базах данных для карт ДНК), серверами и программными пакетами для доступа, просмотра, сравнения и перекачки данных о биологических последовательностях пользователям трудно работать так, чтобы получить как можно больше пользы от хранимой информации (достаточно сложно стало даже запомнить имена всех этих баз данных). Вследствие этого главное усилие направляется на связь и интеграцию, или на *федерацию*, различных баз данных для того, чтобы разделить информацию и автоматически обрабатывать сложные запросы, которые требуют взаимодействия нескольких баз данных [76, 452, 479, 480]. Сейчас существуют и являются стандартными многобазовые программы поиска (*Entrez* и его сетевой родственник *nEntrez* из NCBI [397]) и создаются другие браузеры и краулеры. Некоторые думают уже (возможно, фантазируя) о “*knowbots*” [55] — программах, которые бы автоматически “разрабатывали” базы данных и добывали из них информацию, используя неожиданные совпадения и зависимости для формулирования и сообщения интересных биологических гипотез.

Итак, базы данных для последовательностей и программное обеспечение для поиска в них стали жизненно важными средствами современной молекулярной биологии. “Было бы трудно найти недавнее открытие, связанное с ДНК и не использующее этих средств” [55].

15.3. Алгоритмические вопросы поиска данных

15.3.1. А есть ли они?

Поиск в базе данных для последовательностей доминирует среди приложений строевых алгоритмов в молекулярной биологии, и на это приложение были направлены гигантские усилия. Но почему здесь потребовались какие-то особые усилия? При всех доступных способах попарного сравнения строк, когда есть строка запроса и нужно найти похожую строку в базе данных, почему бы не сравнить строку запроса по очереди с каждой строкой в базе, а затем сообщить “наилучшие” результаты? Кроме того, можно использовать различные усовершенствованные методы, с которыми мы встретились в пп. 12.1–12.4, они помогут ускорить вычисления и уменьшить затраты памяти. Конечно, остаются важные детали (локальное или глобальное выстраивание, модель пропусков, выбор параметров, оценочные матрицы, пороги для установления сходства, оценки статистической значимости и т. п.), но все это — вопросы модельные и статистические, а не алгоритмические. Итак, неужели после четырнадцати глав сравнения и выстраивания нужно еще что-нибудь говорить о том, как искать в базах данных?

И действительно, некоторые утверждают, что придет день, когда почти не останется дополнительных алгоритмических вопросов, требующих обсуждения, и подход, обрисованный выше, будет работать прекрасно. Дело в том, что, хотя базы данных увеличиваются, белки не изменяются. Размер строки запроса и базы данных (по

крайней мере, в белковом поиске) редко будет больше пятисот, а обычно меньше. Поэтому время вычислений при увеличении числа строк в базе данных будет расти линейно. При росте скорости работы компьютеров, падении цены на процессоры и возможности тривиально распределять поиск в базе по многим процессорам можно дождаться до того дня, когда будет практически искать данные в базе, многократно оптимизируя точную целевую функцию какой-нибудь версией динамического программирования. Даже сегодня предоставляются возможности поиска в базе с использованием компьютера MASPAC с 4000 процессорами и имеются специализированные чипы и другое "железо" для очень быстрой реализации выравнивания последовательностей. Часть этого "железа" продаётся как коммерческая продукция (например, машины Time Logic, Inc. и CompuGen Ltd. из Израиля), а часть создаётся в академических учреждениях (например, проект Kestral в Калифорнийском университете в Санта-Круз [235]).

Да, может настать день, когда поиск в базе для последовательностей будет решаться только за счет многократного точного выстраивания двух строк. Но этот день еще не пришел. Более того, вопросы, отличные от вычислительной эффективности, диктуют применение стратегий, отличных от динамического программирования. При поиске известных мотивов и при неполноте заданных характеристиках важности в строке запроса можно добиться более чувствительного, селективного и эффективного по времени поиска, используя специализированную стратегию или (очищенную) базу данных вместо поиска сходства общего типа в сырой базе. Следовательно, обсудим теперь, что делается на практике, когда у вас есть новая последовательность и вы обращаетесь к базе данных для поиска важных свойств этой последовательности или последовательностей с высокой степенью сходства. Как обычно, основное внимание уделяется алгоритмическим аспектам, а не конкретным программам и базам данных, которые сейчас широко используются.

15.4. Реальный поиск в базе данных для последовательностей

Дуг Братлэг описал действия специалиста по поиску в базах данных, когда идентифицируется новая белковая последовательность.*¹) Прежде всего, эта новая последовательность сравнивается с базами PROSITE и BLOCKS для обнаружения типичных мотивов. Затем идет поиск в базах данных для последовательностей ДНК и белков (GenBank, Swiss-Prot и др.) с целью найти последовательности, существенно похожие на новую (обычно это локальное сходство). При этом вначале не пытаются найти оптимальное сходство (как оно определено в п. 11.6) между новой последовательностью и каждой записью базы, а используют приближенные (эвристические) методы BLAST и FASTA. Только затем, если требуется, передаются ресурсы, необходимые для вычисления оптимального сходства с помощью ДП. В качестве дальнейшего усовершенствования, когда используются матрицы подстановки аминокислот, обычно делаются попытки применить вариант матрицы PAM Маргарет Дейхоф и матрицу BLOSUM.

Подведя этот итог, обсудим вкратце алгоритмические и модельные идеи, стоящие за каждым из слов предыдущего абзаца: FASTA, BLAST, PAM, PROSITE, BLOCKS

*¹) Другие описывают иной порядок, но с использованием того же или сходного набора средств

и BLOSUM. Эти шесть слов (наряду с “динамическим программированием”, “сходством”, “GenBank” и “Swiss-Prot”) составляют основной словарь любого серьезного специалиста по поиску в базе данных для последовательностей.

FASTA и BLAST

FASTA (сокращение от fast-all, произносится “ФАСТ-Эй”, а не “ФАСТ-А”) и BLAST (сокращение от basic local alignment search tool) — две родственные программы, которые почти универсально используются для приближения локального выстраивания и локального сходства. В этом контексте термин “приближенный” имеет нормальный разговорный смысл, а не тот математический смысл, который ему придавался в приближенном алгоритме с ограниченной ошибкой. Обе эти программы на самом деле составлены из программ, которые содержат варианты, настроенные на применение в различных приложениях (поиск ДНК или белка, поиск в базе или выстраивание двух строк и т. п.). Мы не будем касаться техники использования этих программ, а ограничимся их алгоритмическими идеями.

Обе программы основаны на одних и тех же идеях, лежащих в основе алгоритмов *исключения*, описанных в п. 12.3. Однако в отличие от них FASTA и BLAST более комплексны и более эвристичны (и более эффективны) поэтому они не допускают корректного анализа их скорости и точности. Наше изложение обоих методов будет кратким, но передаст их основной дух и позволит читателю увидеть отношение “в принципе” методов FASTA и BLAST к описанным ранее методам исключения. FASTA был разработан Д. Липманом и У. Пирсоном [304, 361], а BLAST позднее С. Альтшулем, У. Гишем, У. Миллером, Ю. Майерсом и Д. Липманом [19].

15.5. FASTA

FASTA — это эвристический метод исключения, который мы опишем в терминах стандартной таблицы ДП для локального (взвешенного) выстраивания проверяемой строки с одной строкой текста. Он применяется по очереди к каждой строке базы данных, в которой ведется поиск. Есть несколько вариантов FASTA, и метод продолжает развиваться. Предлагаемое описание “порождающее”, оно основано на исходных работах по FASTA [304, 361] и на некоторых более свежих замечаниях о текущем положении дел [360]. Дополнительные комментарии и анализ эффективности FASTA см. также в [356, 357, 358].

В самом начале пользователь выбирает значение параметра, именуемого *ktup*, и на первом шаге метода в таблице ДП выделяются “горячие точки” длины *ktup*, которые определяются следующим образом. FASTA находит такие пары (i, j) , что подстрока длины *ktup* (*k*-отрезок, а по-английски *k*-tuple), начинающаяся в позиции *i* анализируемой строки, *точно* совпадает с *k*-отрезком текста, начинающимся с позиции *j*. Такая пара и называется *горячей точкой*. Стандартно для *ktup* рекомендуются значения 6 в случае поиска в ДНК и 2 — для белка. Для таких малых значений *ktup* совпадения горячих точек можно эффективно находить хешированием *k*-отрезков анализируемой и/или текстовой строки. Например, *k*-отрезки в базе данных могут быть организованы в хеш-таблицу перед каждым выпуском базы, если для этого достаточно места. В этом случае при появлении исследуемой строки метод ищет в готовой таблице каждый ее *k*-отрезок. Если памяти требуется слишком

много, то можно делать хеш-таблицу для k -отрезков исследуемой строки и искать в ней k -отрезки текста. Последний подход наиболее часто отождествляется с FASTA.

Каждую горячую точку (i, j) , найденную на первом шаге, следует рассматривать как интервал длины $ktyp$ на диагонали $(i - j)$ полной таблицы ДП. (Главная диагональ имеет номер 0; диагоналям выше ее сопоставлены положительные числа, а ниже — отрицательные.) Затем FASTA находит в таблице десять лучших *диагональных полос* горячих точек. Диагональная полоса — это последовательность подряд идущих горячих точек на одной диагонали. Полоса не обязана содержать все горячие точки данной диагонали, и диагональ может содержать больше одной из десяти выбранных полос. Чтобы оценить диагональные полосы, FASTA сопоставляет каждой горячей точке положительную оценку, а каждому пробелу между последовательными горячими точками — отрицательную, уменьшающуюся с ростом расстояния. Оценка диагональной полосы равна сумме оценок горячих точек и пробелов между ними. FASTA находит десять диагональных полос с самыми высокими оценками по этой схеме.

При запросе длины n и базе данных длины m может показаться, что нахождение лучших диагональных полос потребует времени $\Theta(nm)$, что уничтожит декларированную эффективность FASTA. Однако диагональный поиск требует времени, пропорционального числу горячих точек, найденных во время просмотра хеш-таблицы. Предположим, что хеш-таблица содержит k -отрезки строки *запроса*. Тогда метод должен просмотреть слева направо k -отрезки текста. При таком просмотре можно поддерживать цепной список горячих точек каждой диагонали и расширять его за константное время. Более того, при корректировке списков метод может поддерживать набор кандидатов на лучшие диагональные полосы, следя правилу продолжения кандидатской полосы до тех пор, пока оценка полосы остается положительной.

Каждая из выбранных выше десяти диагональных полос определяет пару выстроенных подстрок. Теперь каждое такое выстраивание может содержать и совпадения (от горячих точек), и несовпадения (от междуточечных областей), но не содержит пробелов, так как оно получено из одной диагонали. На шаге два в каждом из этих десяти выстраиваний разыскивается пара выстроенных подстрок (подвыравнивание) с максимальной оценкой, причем оценка определяется по матрице подстановок аминокислот (в случае выравнивания белковых последовательностей). Первоначально в качестве матрицы подстановки рекомендовалась матрица PAM Марграт Дейхоф, но теперь используется BLOSUM50 (про эти матрицы см. пп. 15.7 и 15.10). FASTA выводит найденное на этом шаге единственное лучшее подвыравнивание и называет его *init1*.

На шаге три FASTA пытается комбинировать “хорошие” подвыравнивания (взятые из шага два, оценка которых выше некоторого заданного уровня) в одно большее выравнивание с высокой оценкой, в котором допускается сколько-то пробелов. Чтобы объяснить общую стратегию, предположим, что каждое из десяти подвыравниваний с оценкой выше определенного уровня представлено взвешенной вершиной в ориентированном графе, причем вес этой вершины равен оценке (из шага два) представляющего ею подвыравнивания. Пусть u представляет одно из выбранных подвыравниваний, начинающееся в клетке (i, j) и кончающееся в клетке $(i + d, j + d)$ таблицы ДП, а v — другое подвыравнивание, начинающееся в клетке (i', j') . В графе проводится дуга из u в v в том и только том случае, если $i' > i + d$. Значит, v должна начинаться в строчке таблицы ниже той, где кончается u . Припишем этой дуге некоторый вес,

который штрафовал бы за пропуски в том возможном выравнивании, которое ставит подвыравнивание v сразу после u . Итак, на дуге (u, v) нужно поставить большой штраф (отрицательный вес), если i' много больше, чем $i + d$, т. е. велико расстояние между двумя диагоналями, содержащими u и v . В другом варианте [360] используется постоянный штраф за пропуск, не зависящий от его длины. Затем FASTA находит в этом графе путь максимального веса. Выбранный путь определяет одно локальное выравнивание двух строк. Результат такого шага называется *initn*. Это локальное выравнивание может не совпадать с оптимальным локальным выравниванием, которое указал бы полный алгоритм ДП (т. е. алгоритм Смита–Уотермана), но стоящая за FASTA интуиция (подкрепленная опытом) говорит за то, что оно будет иметь хорошую оценку по сравнению с оптимумом из ДП.

Шаг три можно рассматривать как маленький пример более общей задачи, которую мы обсуждаем в п. 13.3, — комбинирования конфликтующих подвыравниваний в одно большее выравнивание.

Окончательно, на шаге четыре FASTA вычисляет альтернативную оценку локального выравнивания, в добавок к *initn*. Он возвращается к *init1*, результату шага два, и образует в таблице ДП полосу вокруг диагонали, содержащей *init1*. Для белка, когда $ktyp = 2$, полоса составлена из 16 диагоналей вокруг диагонали, содержащей *init1*. При $ktyp = 1$ полоса состоит из 32 диагоналей. Теперь FASTA использует алгоритм Смита–Уотермана и вычисляет оптимальное локальное выстраивание в подтаблице, соответствующей полосе. Результат этого шага называется *opt*.

В текущей версии FASTA, когда строка запроса сравнивается со строками из базы, по умолчанию строки из базы ранжируются по оценкам, соответствующим *opt*. Раньше по умолчанию они ранжировались по оценкам *initn*. Какие бы оценки ни использовались для выбора (нескольких) многообещающих строк из базы, FASTA в заключение выравнивает эти строки со строкой запроса, используя полную схему ДП (т. е. алгоритм локального выравнивания Смита–Уотермана).

Когда FASTA применяется к последовательным строкам в базе данных (при фиксированной строке запроса), он накапливает статистику об оценках *initn*, *init1* и *opt*, хотя текущая версия сообщает только статистику об оценке, применяемой для ранжирования строк базы. Накопленные данные используются для оценки статистической значимости наилучших локальных выравниваний, найденных для строки запроса и всех строк базы.

15.6. BLAST

Почти немедленно после выпуска в 1990 г. BLAST стал доминантной поисковой машиной для баз данных биологических последовательностей. Причинами его успеха были скорость, тот факт, что он вырабатывает некоторый диапазон ответов, и вычисление для каждого совпадения оценки статистической значимости (фактически вероятности того, что совпадение с таким или лучшим значением найдется в случайной строке). После появления BLAST стали совершенствовать и FASTA, и теперь он также выдает множество решений и оценивает статистическую значимость. Тем не менее BLAST продолжает доминировать.

BLAST возник из соединения трех усилий. Первым было стремление Дэвида Липмана, Уоррена Гирша и др. в NCBI увеличить скорость FASTA за счет более

строгих правил определения (меньше и лучше) горячих точек выравнивания. Вторым была работа Майерса по сублинейному поиску [337, 342] (см. п. 12.3.5), которая использовала идею окрестностей подстроки и конечных автоматов для определения начальных горячих точек выравнивания. Третьим была работа Карлина, Альтшуля и Дембо [124, 262], которые получили вероятностные результаты, используемые в BLAST для оценки статистической значимости совпадений, включенных в ответ.

BLAST сосредотачивается на нахождении областей высокого локального сходства в выравниваниях без пропусков, оцениваемых алфавитно-весовой матрицей. Выравнивания с некоторым числом пропусков можно строить, сцепляя по нескольку локально похожих областей, найденных с помощью BLAST. Основные объекты, с которыми работает BLAST, — это *сегментные пары*, локально максимальные сегментные пары и максимальные сегментные пары.

Определение. Пусть заданы две строки, S_1 и S_2 . Сегментной парой называется пара их подстрок равной длины, выровненная без пробелов. Локально максимальной называется сегментная пара, оценка выравнивания которой (без пробелов) уменьшается при удлинении и укорачивании сегментов с любой стороны. Максимальная сегментная пара (MSP) в S_1 , S_2 — это сегментная пара с максимальной оценкой среди всех сегментных пар в S_1 , S_2 .

При сравнении всех последовательностей базы данных с искомой последовательностью P , BLAST пытается найти все такие последовательности, которые в паре с P образуют MSP с оценкой выше заданного уровня C . При выборе C можно руководствоваться теоремами из [262, 124], исходя из оценочной матрицы и характеристик P и последовательностей из базы данных. Эти теоремы определяют наименьшее значение C , превышение которого в оценке MSP будет неправдоподобно для случайно выбранной последовательности. Следовательно, любая последовательность с оценкой MSP выше C рассматривается как “существенная” и о ней сообщается. BLAST также сообщает о последовательностях, у которых оценка MSP не превосходит C , но они имеют по несколько сегментных пар, которые в комбинации статистически значимы. Статистический метод, используемый для идентификации таких множественных сегментных пар, описан в [263].

15.6.1. Стратегия хитов (горячих точек) в BLAST

BLAST — это совокупность программ, каждая из которых настроена на свою группу задач. Мы рассмотрим отдельно BLASTP, версию BLAST для поиска в базах данных белков и применение BLAST к ДНК.

Чтобы найти в белковой базе данных последовательности с оценкой MSP выше C , BLASTP использует следующую стратегию. При фиксированной длине w и фиксированном пороге t BLAST находит все подстроки длины w строки S (называемые в BLAST “словами”), которые выравниваются с какой-либо подстрокой длины w образца P с оценкой выше t . Каждая такая горячая точка (называемая в BLAST “хитом” — удачей) затем продолжается, чтобы посмотреть, не содержится ли она в сегментной паре с оценкой выше C .

Чтобы найти хиты длины w , BLASTP следует идее, использованной в сублинейном методе совпадений Майерса (см. п. 12.3.5). Для каждой подстроки α длины w из P BLAST конструирует все слова длины w , сходство которых с α не меньше t . Альтшуль и др. [19] утверждают, что “если немного тщательнее программировать,

то список слов можно сгенерировать за время, фактически пропорциональное длине списка*. Во время генерирования эти слова помещаются в *дерево ключей* (на самом деле — в детерминированный конечный автомат) для использования в алгоритме, похожем на алгоритм *Ахо-Корасика* (см. п. 3.4), хотя и не совпадающем с ним. Этот алгоритм находит точное вхождение в базу данных одного из слов списка. Таким образом, за время, фактически линейно зависящее от числа сгенерированных слов, а значит, и от длины базы данных, BLASTP находит в базе все подстроки длины w , которые имеют оценку выравнивания не меньше t с какой-либо подстрокой длины w из P .

Чтобы найти все последовательности с MSP выше C в базе данных ДНК, BLAST следует более простой стратегии. Он использует список всех подстрок длины w в строке запроса P . На практике w около двенадцати.

В любом случае (белок или ДНК), когда хит найден, в идеале он расширяется, чтобы узнать, не содержится ли он в локально максимальной сегментной паре с оценкой выравнивания не меньше C . На практике эти расширения быстро обрываются, если текущая оценка выравнивания падает слишком низко по сравнению с найденной ранее наилучшей оценкой для более короткого расширения. Этот обрыв и использование хитов длины w означают, что BLAST не гарантирует нахождения всех сегментных пар с оценкой не менее C .

Отметим, что BLAST для ДНК — это фактически стандартный метод *типа исключения*. Он локализует первоначально найденные выжившие элементы (горячие точки, хиты) фиксированной длины, разыскивая *точные* совпадения с подстроками P фиксированной длины. Затем проверяется область вокруг каждого хита. Однако в отличие от конкретного метода исключений, описанного в п. 12.3, BLAST не привлекает динамического программирования для проверки этих областей, потому что оно не допускает пробелов при выравниваниях.

15.6.2. Эффективность BLAST

Ясно, что выбор матрицы оценок, значений w и t , относительно C , принципиален для эффективности BLAST во всех ее смыслах. Снижение t уменьшает шанс того, что будет пропущена последовательность с оценкой MSP выше C , но увеличивает количество требуемых вычислений. Возможности этого выбора активно изучались опытным путем, и рекомендуемые значения все время изменялись. Мы не будем пытаться описать или обосновать текущие рекомендации и стандартные установки, ограничившись замечанием, что при поиске белков w лежит в диапазоне от трех до пяти аминокислотных остатков, а при поиске ДНК — около двенадцати нуклеотидов.

Первоначально сообщалось [19], что BLASTP считает на порядок быстрее, чем FASTA, и во много раз быстрее, чем вычисление оптимального локального выравнивания с помощью динамического программирования. Однако сравнения BLASTP с современными версиями FASTA [359] показали существенно меньшее различие в скорости, хотя обе программы остаются гораздо быстрее, чем ДП.

Вопрос биологической эффективности — чувствительности и селективности методов*) — также остается активной областью экспериментального исследования.

*.) В эксперименте, где известно, какие последовательности биологически схожи с запросом, а какие нет, “чувствительность” — это доля случаев, когда методу не удается распознать схожие последовательности (неправильный отрицательный ответ), “селективность” — доля случаев, когда метод принимает за схожую последовательность, которая биологически схожей не является (неправильный положительный ответ).

Эти эмпирические результаты будут продолжать варьироваться с изменением методов, данных и биологических взглядов, но текущая точка зрения такова: BLAST, возможно, немного менее эффективен, чем FASTA, в определении важных совпадений последовательностей, особенно когда наиболее биологически значимые выравнивания содержат пробелы, но в целом BLAST выдерживает сравнение с FASTA. В некоторых ограниченных обстоятельствах и FASTA и BLAST ощущаются менее эффективны, чем оптимальное локальное выравнивание, но при обычном использовании они с ним вполне сравнимы. Рекомендация пользоваться и BLAST и FASTA вполне осмыслена, как и рекомендация вычислять локальное выравнивание по Смиту–Уотерману по всей длине для любой пары последовательностей, которые эвристические методы сочли локально схожими. Пирсон [359] утверждает, что “в то время, как BLASTP эффективен в обнаружении удаленных соответствий, выравнивание Смита–Уотермана следует всегда использовать, когда анализируются и показываются совпадения, найденные BLASTP”.

15.7. PAM: первые главные матрицы подстановки аминокислот

Выравнивания последовательностей, найденные при поиске в белковой базе данных, почти всегда являются *взвешенными*, и выбор используемой *матрицы оценок* (матрицы подстановки аминокислот) может влиять на результаты поиска очень сильно. Иногда считают, что правильная матрица оценок является наиболее важным техническим элементом для успешного поиска в белковых базах данных. В идеале оценки этой матрицы должны отражать те биологические феномены, для демонстрации которых ищется выравнивание. В случае различия последовательностей из-за эволюционных мутаций, числа в матрице оценок следует получать из эмпирического наблюдения последовательностей прародителей и их современных потомков. А в случае сохраняемых мотивов или хорошо видных корреляций последовательностей со структурами эти числа нужно извлекать из коллекции последовательностей, содержащих такие мотивы или обладающих такими структурными свойствами.

15.7.1. Единицы и матрицы PAM

Термин “PAM”, который является акронимом для “point accepted mutation” или для “percent accepted mutations”, имеет два родственных применения. Первое — как единица измерения количества эволюционных различий (или эволюционного расстояния) между двумя аминокислотными последовательностями. В этом контексте можно сказать, что последовательность S_1 отличается (или отстоит) от S_2 на 5 PAM. Второе — для указания на некоторые матрицы подстановки аминокислот (матрицы оценок), оценки которых имеют отношение к единицам PAM. Методология единиц PAM и первые конкретные матрицы PAM были предложены Маргарет Дейхоф и др. [122, 400].

При обсуждении единиц и матриц PAM полезно различать *идеальные* объекты PAM, определяемые в терминах недоступных нам данных, и *реальные* объекты PAM, вычисляемые по данным, менее чем идеальным.

15.7.2. Единицы РАМ

Определение. В идеале последовательности S_1 и S_2 считаются *отстоящими на одну единицу РАМ*, если серия приемлемых точковых мутаций (без вставок и удалений) превратила S_1 в S_2 с одной приемлемой точковой мутацией в расчете на один процент аминокислот.

Здесь термин “приемлемый” означает мутацию, которая изменила белок и передалась по наследству. Следовательно, либо эта мутация не изменила функции белка, либо изменение было благотворно для организма (или, по крайней мере, нелетально).

Приведенное определение единицы РАМ звучит как “при одной единице РАМ расхождения между S_1 и S_2 получается расхождение между этими последовательностями на один процент”. Это не так. Причина в том, что одна позиция может подвергнуться более чем одной мутации, так что (например) если было восемь точковых мутаций в аминокислотной последовательности длины 800, то может случиться, что какая-то позиция мутировала дважды. Может даже случиться так, что позиция мутировала обратно к исходной аминокислоте. Поэтому если мы выравниваем S_1 и S_2 (помня, что вставки и удаления невозможны), то ожидаемое число позиций, где эти последовательности отличаются, будет (немного) меньше восьми.

Различие между корректным определением единицы РАМ и (обычным) недопониманием его смысла невелико для последовательностей, расходящихся на одну единицу РАМ. Но это различие растет с ростом числа единиц. Например, неправильно ожидать, что две последовательности с расхождением в РАМ различаются во всех позициях. В действительности даже аминокислотные последовательности, различающиеся на 200 РАМ, должны совпадать примерно в 25% позиций, и последовательности с различием в 250 РАМ можно вообще-то отличить от пары случайных последовательностей.

Вся концепция РАМ-единиц вызывает ряд возражений. Но обойдем принципиальные проблемы и спросим: как определить число РАМ-единиц, разделяющих последовательности S_i и S_j ? В идеальном случае, когда одна нынешняя последовательность отличается от предшествовавшей ей благодаря только точковым мутациям, нужно просто подсчитать число наблюдаемых позиций, в которых эти две последовательности различаются. Затем применяется простая вероятностная формула, которая связывает ожидаемое число наблюдаемых различий с ожидаемым числом точковых мутаций.*)

На практике реализация этого идеального сценария встречает две проблемы. Первая в том, что все последовательности, которыми мы располагаем, взяты из *ныне живущих* организмов. Мы не знаем ни одной пары белковых последовательностей, где бы одна получалась из другой. Вторая проблема в том, что в эволюции белков участвуют вставки и удаления, так что иногда нельзя быть уверенным в корректном

* Вопрос дополнительно усложняется тем, что мутации происходят на уровне ДНК, а наблюдения — на уровне аминокислот. Вырождение генетического кода плюс факт “тихих мутаций” означает, что для перевода числа наблюдаемых изменений аминокислот в ожидаемое число истинных их мутаций нужно делать некоторые предположения о частотах кодонов. Например, TCT — один из кодонов аминокислоты серин, а TAT и AGC — два кодона тирозина. TCT мутирует в TAT при единичной (ДНК) точковой мутации, а для преобразования его в AGC нужны три мутации. Таким образом, если наблюдается преобразование серина в тирозин, вопрос о том, как много произошло мутаций, должен учитывать детали на уровне ДНК. Есть некоторые споры о правильном способе связи уровня ДНК с аминокислотным уровнем [477].

соответствии между позициями последовательностей. Дополнительные возражения против концепции единиц РАМ включают тот факт, что не все позиции мутируют с равной частотой.

Первая проблема — нехватка предков последовательностей белков — разрешается обращением к теории молекулярных часов (см. п. 17.1.4). Две последовательности, S_i и S_j , считаются полученными от некоторого общего предшественника S_{ij} , и эта теория исходит из того, что ожидаемое расстояние в РАМ между S_{ij} и S_i равно ожидаемому расстоянию между S_{ij} и S_j . Поэтому можно считать, что половина числа разностей в выравнивании S_i и S_j будет исходной цифрой для вычисления РАМ-расстояния между S_{ij} и двумя выведенными из нее последовательностями S_i и S_j . Чтобы это рассуждение было корректным, нужно предположить также, что мутации аминокислот обратимы и равновероятны в любом направлении.

Вторая проблема, со вставками и удалениями, более сложна. Чтобы определить правильное соответствие между позициями двух последовательностей, нужно бесспорно распознать истинные исторические пропуски или по крайней мере большие интервалы, где соответствие корректно. Это не всегда можно сделать достаточно определенно, особенно когда последовательности различаются на большое число единиц РАМ.

15.7.3. Матрицы РАМ

Матрицы РАМ — это матрицы подстановки аминокислот (матрицы оценок), которые кодируют и суммируют ожидаемое эволюционное изменение на аминокислотном уровне. Каждая матрица РАМ создается для использования при сравнении пар последовательностей, которые различаются на заданное число единиц РАМ. Например, матрица РАМ120 в идеале используется для сравнения двух последовательностей, о которых известно, что они различаются на 120 единиц РАМ. Для любой конкретной пары символов аминокислот, A_i и A_j , элемент (i, j) матрицы РАМ n отражает частоту случаев, когда A_i заменяет A_j в двух последовательностях, расходящихся на n единиц РАМ.

Матрица РАМ n может быть получена в *идеале* (но не практически) следующим образом. Соберем много различных пар гомологичных последовательностей, о которых известно, что они расходятся на n единиц РАМ (только в точковых мутациях). Выровняем каждую пару последовательностей, для каждой аминокислотной пары A_i , A_j подсчитаем число случаев, когда A_i стоит напротив A_j , и разделим это число на полное число пар во всех выровненных данных. Пусть $f(i, j)$ обозначает получившуюся частоту. Пусть f_i и f_j — соответственно, частоты появления аминокислот A_i и A_j в последовательностях. Значит, f_i — число раз, когда A_i появляется во всех последовательностях, деленное на полную длину последовательностей. Тогда элемент (i, j) идеальной матрицы РАМ n равен

$$\log \frac{f(i, j)}{f_i f_j}.$$

Это еще один пример отношения *log-odds* — логарифмического различия. Смысл деления $f(i, j)$ на $f_i f_j$ в том, чтобы нормализовать истинную (историческую) частоту замен частотой замен, ожидаемой только от случайности. Логарифм используется потому, что сложение легче, чем умножение.

15.7.4. Как на самом деле получаются матрицы PAM?

Приведенное описание матрицы PAM n относится к идеальному случаю. Но при наличии включений и удалений истинное соотношение между позициями не известно, если только не найдены правильные пропуски. Иными словами, прежде чем собирать статистику, необходимую для построения матрицы PAM, нужно выровнять последовательности так, чтобы правильно выяснить историю их расхождения. Однако для получения выравнивания обычно используется матрица оценок. Получается порочный круг. Как решает эту проблему Дейхоф?

Дейхоф взяла очень схожие последовательности, относительно которых была уверена, что они получились из общего предка с расхождением в малое число единиц PAM. “Для того чтобы минимизировать появление изменений, вызванных последовательными приемлемыми мутациями в одном и том же месте, последовательности... отличались друг от друга меньше чем на 15%” [122]. Так как различия малы, последовательности в каждой паре оказались примерно равной длины, и немногие включения и исключения, которые могли встретиться, были легко опознаны и учтены. Поэтому получение корректного соответствия между символами двух последовательностей не было проблемой, и накопление статистики следовало в основном идеальному случаю.*.) Это решает проблему для малого числа единиц PAM, но с большим числом нужно еще что-то делать.

Для построения матрицы PAM в случае большого числа единиц PAM Дейхоф поступила следующим образом (говоря просто, но не точно): предположим, что данные состоят из пар выровненных последовательностей, расходящихся на одну единицу PAM. По этим данным вычислим частоту мутаций аминокислоты A_i в аминокислоту A_j в одну единицу PAM; обозначим ее через $M(i, j)$. Итак, для позиции, содержащей аминокислоту A_i (не существенно, часто или редко это происходит), $M(i, j)$ — это частота мутации позиции в A_j при одной единице PAM. Пусть M обозначает матрицу этих частот**.) 20 на 20. Тогда матрица M^n (результат перемножения n экземпляров матрицы M) дает вероятность того, что одна конкретная аминокислота мутирует в другую за n единиц PAM. Элемент (i, j) матрицы PAM n равен поэтому

$$\log \frac{f(i)M^n(i, j)}{f(i)f(j)} = \log \frac{M^n(i, j)}{f(j)},$$

где $f(i)$ и $f(j)$ — наблюдаемые частоты аминокислот A_i и A_j . Такой подход предполагает, что частоты аминокислот не изменяются во времени и что мутационные процессы, вызывающие замены в интервале одной единицы PAM, действуют так же и в более длинных периодах. Эти предположения, которые делаются из соображений практичности, еще дальше отклоняют модель от идеального описания матриц PAM.

Значения в матрицах PAM обычно умножают на 10 и округляют до целых, и зачастую в них добавляется константа (от 2 до 8) к каждому элементу, кроме перехода от символа аминокислоты к пробелу и обратно. Это дает эффект смягчения разностей элементов, и в некоторых контекстах наблюдалась благотворность использования такой смягченной матрицы при получении “лучших” выравниваний.

*.) На самом деле в методе Дейхоф есть дополнительная подробность, включающая использование филогенетических деревьев. Мы обсудим эту деталь в п. 17.6.2 части IV, после того как филогенетические деревья будут рассмотрены более полно.

**) Фактически считается, что процесс мутаций в каждой позиции образует марковскую цепь, M — матрица переходных вероятностей, а единица времени выбрана так, чтобы число мутаций в единицу времени равнялось единице PAM. — Прим. перев.

15.7.5. Использование матрицы PAM

Хотя матрицы PAM определялись и выводились из выровненных пар последовательностей, содержащих небольшое число несовпадений (точковых мутаций) и небольшое число вставок и удалений (инделов), они в основном используются во взвешенном выравнивании с аффинными весами пропусков, когда несовпадений и инделов много. Выравнивание двух последовательностей, расходящихся на 250 единиц PAM, обычно не будет иметь много совпадений даже после оптимальной вставки пропусков. Более того, при выравнивании двух последовательностей в идеале нужно было бы использовать матрицу PAM, соответствующую их PAM-расстоянию (числу единиц PAM, на которое эти две последовательности расходятся), но PAM-расстояние между двумя последовательностями, вообще говоря, неизвестно. Действительно, одна из целей вычисления этого выравнивания в том, чтобы включить пропуски для обнаружения сплошных сегментов, которые можно использовать для расчета PAM-расстояния. Ввиду этих двух проблем из первых принципов неясно, насколько эффективна будет матрица PAM при поиске “биологически информативных” выравниваний.

Тем не менее из общей практики складывается убеждение, что матрицы PAM очень эффективны в нахождении выравниваний, которые высвечивают важные биологические феномены. В частности, о матрице PAM250 сообщалось как об очень эффективной в выравнивании последовательностей, использованных при изучении эволюции, и до самого последнего времени она рассматривалась как “каноническая” матрица оценок для белков.*) Однако матрица BLOSUM62 неумолимо вытесняет PAM250 с места матрицы, принимаемой по умолчанию.

15.8. PROSITE

PROSITE — это “словарь белковых сайтов и шаблонов” [41, 42], который присоединен к базе данных белковых последовательностей Swiss-Prot. Некоторые особенности PROSITE упоминались ранее в п. 3.6 и упражнении 32 главы 3. Целью создателя PROSITE, Амоса Бейроха, было опознание и представление *биологически значимых* шаблонов в семействах белков (в частности, тех, которые считаются важными для функций белков), позволяющих надежно относить новые белковые последовательности к правильному семейству. Выделение биологически значимых шаблонов отличает PROSITE от других попыток поиска хороших *дискриминаторов*, когда стаются лишь надежно распознавать известных членов семейства при исключении известных не-членов. В PROSITE шаблон (pattern) представляется либо сигнатурным

*) Я однажды присутствовал на семинаре, устроенном для информатиков, математиков и биологов, посвященном совместной работе в вычислительной биологии. Предполагалось, что большинство присутствующих небиологов знает в вычислительной биологии мало, поэтому большинство выступлений носило вводный характер. Однако один оратор (специальность которого я забыл) забрался в густые дебри, говоря о специфических особенностях работы с матрицами PAM. Он предполагал, что все знают основные термины матриц PAM, и поэтому не определял их и не говорил об их функции, происхождении и формировании. Наконец в середине его выступления, когда он уже упомянул “матрицу PAM” раз тридцать, один разочарованный слушатель (ни разу до того о матрицах PAM не слышавший) поднял руку и сказал что-то вроде этого: “Вы упоминали несколько раз матрицу PAM, но не сказали нам, что это. Так что же такое эта матрица PAM, о которой вы говорите?” Докладчик быстро ответил: “Это 250” — и без тени смущения продолжил.

мотивом (см. п. 14.3.2), либо профилем (см. п. 14.3.1), полученным из множественного выравнивания членов семейства. В настоящее время PROSITE содержит около тысячи шаблонов. Каждый из них сопровождается перекрестными ссылками на записи в Swiss-Prot, в которых найден этот шаблон, а также известными неправильными положительными и отрицательными совпадениями в Swiss-Prot.

Сигнатурные шаблоны в PROSITE записаны как *регулярные выражения* (см. п. 3.6). Например, $G - [GN] - [SGA] - G - x - R - x - [SGA] - C - x(2) - [IV]$ — это шаблон PROSITE. Каждая заглавная буква определяет конкретную аминокислоту, заглавные буквы внутри скобок показывают, что возможна любая одна из представленных ими аминокислот, x показывает, что возможна любая аминокислота, а $x(2)$ разрешает любую пару аминокислот. При поиске вхождения шаблона PROSITE в новую последовательность можно использовать методы поиска совпадений с регулярными выражениями (п. 3.6). Если полная длина всех регулярных последовательностей равна n , а длина новой последовательности равна m , то этот подход требует времени $\Theta(nm)$. Однако шаблон PROSITE распознает только подстроки конечной длины. Более того, как отмечено в упражнении 5 главы 4 (с. 116), метасимволы не вызывают трудностей для таких методов, как *Shift-And* и *agrep*. Таким образом, как указывается в упражнении 7 главы 4, шаблоны PROSITE, не содержащие большого числа спецификаций диапазона, можно более эффективно обрабатывать, используя методы в стиле *Shift-And* или *agrep*.

Шаблоны PROSITE, представленные как *профили*, получаются из множественного выравнивания членов семейства. Они используются, когда семейственного сходства для получения эффективных сигнатур мотивов недостаточно. Дополнительная возможность распознания новых членов семейства получается при “экстраполяции” профиля, т. е. при использовании знания о возможности подстановки аминокислот для получения их оценок в некоторых позициях, где эти аминокислоты не наблюдались. Например, если аминокислота лейцин наблюдается в данной позиции множественного выравнивания часто, а изолейцин нет, он все же может получить в этой позиции высокую оценку, потому что эти аминокислоты имеют сходные химические свойства и часто заменяют одна другую.

15.9. BLOCKS и BLOSUM

BLOCKS — это база данных белковых мотивов, полученная из библиотеки PROSITE, а BLOSUM — матрица подстановки аминокислот, произведенная из BLOCKS. И BLOCKS и BLOSUM были созданы Стивеном и Жорией Хайнайкоф [220–222].

База данных BLOCKS пытается представить наиболее высоко сохраненные подстроки (мотивы) в аминокислотных последовательностях родственных белков. На этом уровне описания такая цель ставится и перед библиотекой мотивов PROSITE. Однако мотивы PROSITE собирались с особым вниманием к известным функциям или известным структурам белков. От каждого мотива в PROSITE ожидается, что он имеет известный биологический смысл. Напротив, мотивы в BLOCKS основаны на сохраняющем сходстве последовательностей, даже если никакой функции этих мотивов не известно.

Определение. *Блок* — это короткий сплошной интервал в множественном выравнивании аминокислотных последовательностей.

База данных BLOCKS содержит примерно 3000 блоков коротких, высоко консервативных последовательностей, полученных из примерно 800 групп родственных белков. Система PROTOMAT, которая строит BLOCKS, обрабатывает отдельно каждую группу родственных белков из библиотеки PROSITE. Для каждой группы PROTOMAT многократно выравнивает ее последовательности и затем ищет интервалы вплоть до шестидесяти позиций, где выровненные аминокислоты высоко сходны в "критическом числе" последовательностей (не менее 50 %). Подробности намерено не уточняются, и описываются только базовые идеи, так как используемые методы и конкретные параметры весьма эвристичны и получены из обширного опыта.

После того как PROTOMAT находит индивидуальные блоки в множественном выравнивании, он отбирает из них подмножество для формирования "пути" (т. е. упорядоченного набора неперекрывающихся блоков, которые встречаются в "критическом числе" последовательностей). Путь идентифицирует в множестве белков области высокого локального сходства безотносительно к длине пропусков между составляющими его блоками. При заданной оценке каждого отдельного блока (исходящей из его ширины, уровня сходства, числа содержащих его последовательностей и др.) можно найти путь, у которого сумма оценок блоков наибольшая. Это в точности задача *цеплени*, рассмотренная в п. 13.3. Получающийся путь может определить эффективный набор мотивов для характеристики этой белковой группы. Однако критерии оценки пути из [221] сложнее, чем суммарная оценка блоков, и "наилучший путь" в BLOCKS находится грубым методом — перебором всех возможных путей. База данных BLOCKS сохраняет найденные лучшие пути всех белковых групп PROSITE.

Использование BLOCKS для классификации новых последовательностей

Блоки (или пути) из BLOCKS используются для идентификации потенциальных новых членов белковой группы, из которой выведен этот блок (или путь). Для каждого блока создается *профиль* (см. п. 14.3.1), и новая последовательность выравнивается с профилем каждого блока. Последовательности, которые хорошо выравниваются с многими блоками одного пути, считаются весьма правдоподобными членами этой белковой группы.

15.10. Матрицы подстановки BLOSUM

Матрицы подстановок аминокислот BLOSUM*) производятся из базы данных BLOCKS [222] и сейчас являются главным соперником матриц PAM. Основная оценка подстановки пары аминокислот i, j в BLOSUM, грубо говоря, вычисляется следующим образом. Определим P как множество пар позиций в блоках, содержащихся в BLOCKS, таких что позиции пары содержатся в одном и том же столбце одного и того же блока. Например, если во всех блоках BLOCKS содержится n позиций и каждый блок состоит из ровно k строчек, то P состоит из nC_k^2 пар позиций. Пусть теперь $n(i, j)$ — число таких пар из P , в которых одна позиция содержит аминокислоту i , а другая — аминокислоту j . Очевидно, $n(i, j)$ возрастает, если аминокислоты i и j стремятся появляться в одной и той же позиции блоков, и убывает, если они стремятся не появляться. Чтобы нормализовать $n(i, j)$, определим $f(i)$ и $f(j)$

*) Слово BLOSUM получено от blocks substitution matrix.

как доли позиций в BLOCKS, занятых аминокислотами i и j соответственно, и положим $e(i, j) = |P|f(i)f(j)$. Далее положим $s(i, j) = \log_2(n(i, j)/e(i, j))$. Наконец, умножим $s(i, j)$ на два и округлим до ближайшего целого. В результате получится элемент i, j основной матрицы BLOSUM. Интуитивно, $s(i, j)$ — это отношение числа появлений аминокислот i и j в одном и том же столбце, к числу ожидаемых появлений, если бы места аминокислот i и j выбирались в BLOCKS случайно (точнее, его логарифм). Подобно значениям в матрице PAM, значение BLOSUM служит примером отношения лог-разности.

Важным дополнительным улучшением основной методологии BLOSUM является уменьшение влияния пар строчек, которые в BLOCKS “слишком” сильно похожи. Например, если две строчки, которые участвуют в данном блоке, идентичны, то одна удаляется, чтобы уменьшить влияние этой пары строчек в итоговых оценках BLOSUM. Развивая эту идею, матрицу BLOSUM x (для x обычно между 50 и 80) создают после последовательного удаления по одной строчке из каждой пары строчек в блоке, в которой больше x процентов совпадает. В результате любые две из оставшихся строчек данного блока будут совпадать менее чем на x процентов.

Уменьшить влияние сильно связанных последовательностей нужно для того, чтобы лучше отразить более далекие, хотя и сохраняемые, мотивы из множества сильно расходящихся последовательностей. Идеально последовательности этого множества должны равномерно выбираться из полного эволюционного диапазона изучаемых видов. Но часто эта равномерность не обеспечивается и кластеризуется в подмножества более сильно схожих последовательностей. Итеративное удаление из блока строчки, которая “слишком похожа” на некоторую остающуюся строчку, — это попытка создать блок, отражающий менее статистически смещенный набор данных. Эффективность этого подхода — вопрос чисто эмпирический. Сейчас сложилось мнение, что матрица BLOSUM62 особенно эффективна как менее смещенное отражение сохраненности важных аминокислот.

Чем матрицы BLOSUM отличаются от матриц PAM

Оценки в матрицах PAM экстраполируются из данных, полученных из очень похожих последовательностей. На самом деле, как уже объяснялось, эти последовательности намеренно отбирались как очень похожие. Однако “наиболее важной задачей, включающей матрицы подстановки, является определение существенно более далеких отношений” [222], и матрицы BLOSUM были предложены как способ явно учесть эти важные далекие отношения. Есть убеждение, что высоко сохраненные сегменты последовательностей из сильно отличающихся в других местах аминокислотных последовательностей (из белков данного семейства) приводят к оценкам, которые более эффективно помогают алгоритмам локального выравнивания находить выравнивания, высвечивающие биологически важные сходства. Общее мнение таково, что матрицы BLOSUM достигают этой цели.

15.11. Дополнительные вопросы поиска в базе данных

Эта глава концентрировала внимание на использовании строковых алгоритмов поиска в базах данных и на поиске биологически значимых подстрок (мотивов). Однако мы обсуждали также родственные, но не алгоритмические вопросы оценочных

матриц PAM и BLOSUM, так как их использование влияет на биологическую пользу поисковых алгоритмов. В этом параграфе мы кратко обсудим другие неалгоритмические вопросы, влияющие на практическую эффективность поиска в базах данных.

15.11.1. Статистическая значимость

Начнем с цитаты из Уильяма Пирсона:

“Главным техническим усовершенствованием в сравнении белковых последовательностей за последние пять лет было включение в широко используемые программы поиска статистических оценок” [359].

Улучшение оценок статистической значимости привело к улучшению чувствительности и селективности и поэтому — к более эффективным программам поиска. Эти оценки получены благодаря теоремам о статистике MSP [124, 262], первоначально включенным в BLAST, и более свежей работе Карлина и Альтшуля [263] и Уотермена и Вингрона [459] о выравниваниях, допускающих пропуски. Обсуждение этих статистических результатов выходит за рамки нашей книги. Мы отсылаем читателя к оригиналным статьям, к тексту Уотермена [461] и к легко читаемой статье Альтшуля и др. [17], ориентированной на широкую аудиторию.

После появления BLAST метод FASTA также включил теоремы из [262, 124] в свои оценки значимости. Однако, так как FASTA находит выравнивания с пропусками, статистические результаты об MSP к результатам FASTA непосредственно не применимы. Были проведены теоретические и эмпирические исследования [21, 263, 459], показавшие, как естественным образом перенести полученные в [262, 124] результаты на локальные выравнивания с пропусками, но статистическая значимость выравнивания с пропусками часто оценивается с помощью своего рода *перемешивающего* вычисления [127, 359]. При таком вычислении строка запроса выравнивается со случайной строкой, которая имеет ту же среднюю длину и состав символов, что и строки в базе данных. Статистика, основанная на этих оценках выравниваний, используется для оценки значимости любого найденного выравнивания с высокой оценкой, установленного между строкой запроса и какой-либо исходной строкой из базы данных.

Использование вероятностей: поучительная история про BRCA1

Желательно сделать поиск в базе данных, насколько возможно, упражнением на “нажатие кнопок”, не требующим, чтобы пользователь был крупным экспертом. Но сегодня эффективный поиск еще требует разумной смеси понимания биологического и статистического аспектов. Эта мысль иллюстрируется недавним опытом с геном BRCA1.

Семейные формы рака молочной железы и яичников были связаны мутациями в двух генах, BRCA1 и BRCA2; эти гены были тщательно локализованы, клонированы и расшифрованы. Открытие и клонирование этих генов было важным шагом в морфологических исследованиях, но лечение и профилактика этих заболеваний требуют понимания того, какие белки кодируют эти гены. В случае BRCA1 вначале была надежда, что это выяснится после того, как при поиске в базе данных PROSITE нашелся мотив (см. п. 15.8) в аминокислотной последовательности, выведенной из

гена BRCA1 [245, 417]. Подстрока последовательности BRCA1 совпала с консенсусной последовательностью семейства белков, именуемых *границами* (эта консенсусная последовательность показана в п. 3.6). Границы прежде не связывались с раком молочной железы, но ввиду указанного совпадения были выполнены эксперименты, подтвердившие роль границ в развитии этого типа рака. Однако результаты других экспериментов противоречили таким заключениям. Было поразительно и вызывало особое внимание то, что границы — это белки, которые секрециируются из клеток, где они произведены. Большинство белков, связанных с раком, работает внутри клетки (или внутри ядра клетки), и трудно до них добраться. Если функция белка BRCA1 похожа в действительности на функцию известных границ и он секрециируется из клетки, то успешные способы борьбы против онкогенных мутантов белка BRCA1 кажутся гораздо более правдоподобными.

К сожалению, более глубокая проверка исходного поиска в PROSITE привела к выводу, что “мотив границы не имел статистической поддержки” [282]. Первая работа [245] сообщила, что “вероятность случайного появления в BRCA1 консенсусной строки границы равна примерно... 0.00175”, и поэтому авторы заключили, что совпадение с границой отражает важный биологический феномен. Однако без предварительных свидетельств об участии границы в развитии рака молочной железы вычислительное значение (которое является хорошим приближением вероятности того, что границовая последовательность появилась в BRCA1 случайно) не отвечает на правильный вопрос. Более информативный вопрос такой: какова вероятность того, что некоторый мотив из PROSITE (такой же сложный, как мотив границы) случайно появится в BRCA1? В [282] сообщается, что эта вероятность равна 0.87. Такое большое число появилось благодаря трем факторам: PROSITE содержит около 1000 мотивов, мотив границы имеет длину всего в десять позиций, из которых три — это метасимволы, а белок BRCA1 относительно длинен (1863 остатков). При такой высокой вероятности случайного совпадения с каким-то мотивом из PROSITE для эффективной работы с последовательностью и базой данных мотивов нужна дополнительная работа. Например, следовало бы поискать более важные сходства между последовательностью белка BRCA1 и последовательностью, содержащей совпадающий мотив, вне самой области мотива. Или следовало бы сначала распознать в последовательности белка BRCA1 короткие отрезки, где совпадающий мотив, скорее всего, отражает реальный биологический феномен и где вероятностные вычисления были бы более информативны.

Упомянутому подходу последовали Коонин, Альтшуль и Борк [282]. Они сначала нашли в последовательности BRCA1 шесть областей с предсказанной глобуллярной структурой, так как глобуллярные области наиболее правдоподобно содержат сохраняемые функциональные мотивы. Сравнивав каждую из этих шести областей с базой данных, они нашли умеренное сходство глобуллярной области в 202 остатков и аналогично расположенной области (на C-конце белков) в человеческом белке 53BP1, известном способностью прицепляться к универсальному супрессору (подавителю) опухолей p53. Более того, это совпадение состояло из двух отдельных сегментов, разделенных в каждом белке почти одинаковым числом остатков (100 и 97). Сходная позиция и форма совпадающих подстрок позволили предположить, что совпадение имеет какое-то биологическое значение. Использование таких аргументов иногда важно для подкрепления числа вероятностных рассмотрений. Биологическое правдоподобие связи между BRCA1 и p53 (который ассоциируется примерно с половиной всех видов рака) усиливается тем фактом, что вызывающие рак мутации в гене

BRCA1 часто случаются в 3'-конце, который соответствует C-концу белка BRCA1. Эта линия рассуждений и дополнительные свидетельства привели авторов к предположению, что ген BRCA1 ассоциируется не с гринином, а с хорошо известным онкологическим агентом.*)

15.11.2. Теория оценок лог-различия

Мы подробно рассмотрели две схемы оценки аминокислот (матрицы PAM и BLOSUM) и отметили, что в обеих матрицах элементами являются отношения лог-различия. Другие матрицы подстановок, например представленные в [57, 183], также содержат значения лог-различия. Аналогично, и профили, характеризующие важные мотивы в биологических последовательностях, часто переводятся в отношения лог-различия (см. п. 14.3.1). Недавно была развита теория [16, 262], которая обосновывает превалирование отношения лог-различия в некоторых специфических контекстах. Эта теория наиболее естественно прилагается к задаче характеризации подстрок (без пропусков), выбранных в некотором семействе, и к задаче нахождения локальных выравниваний без пропусков, которые образуют семейство выбранных выравниваний. Легко читаемый обзор этой теории, включающий в себя также другие неалгоритмические вопросы поиска в базах данных, появился в [17].

Чтобы объяснить простейшую форму этой теории, рассмотрим задачу поиска подстроки фиксированной длины t , обладающей некоторым заданным свойством (например, свойством быть промоторной последовательностью для *E. coli*). Первоначально задано множество \mathcal{M} выбранных подстрок, обладающих требуемым свойством. Затем будет задано множество последовательностей, каждая из которых может содержать или не содержать подстроку с этим свойством. Биологический интерес представляет выяснение, для каких последовательностей правдоподобно, что они содержат подстроку длины t с этим свойством, а для каких нет. Теория, развитая в [16, 262], вводит методы, которые смотрят на эти подстроки следующим образом. Пусть задана последовательность S , каждая ее подстрока длины t явно проверяется и оценивается суммой оценок входящих в нее символов. Оценки символов могут зависеть и от самих символов, и от их места в этой подстроке.

Если схема оценки эффективна, то подстроки, обладающие неким свойством, должны получить высокую оценку, а не обладающие — низкую. Эта неясная цель может быть уточнена требованием, что подстроки из \mathcal{M} имеют статистически значимо более высокую оценку, чем случайные подстроки с тем же глобальным распределением символов. Какая оценочная схема будет наилучшей при подобном сценарии, чтобы отличить членов \mathcal{M} от случайных подстрок? Именно на этот точный вопрос и был дан ответ.

Пусть $p(x, j)$ — частота появления символа x в позиции j для строк из \mathcal{M} , а $p(x)$ — частота появления символа x где угодно в исходном множестве. Было доказано [16, 262], что оценки, оптимально отличающие подстроки \mathcal{M} от случайных подстрок (сгенерированных по частотам $p(x)$ для каждого символа x), имеют вид $\log[p(x, j)/p(x)]$, где основание логарифма не существенно. Эта теорема

*) Они выдвинули еще одно удивительное предположение. Проводя в базе данных поиск по профилю, сконструированному из наиболее сохраненных частей BRCA1 (от человека и мыши) и 53BP1, они нашли в дрожжах белок, который может быть структурно родственным p53.

согласуется с тем способом, который часто используется для формирования значений профиля (п. 14.3.1).*)

В контексте локальных выравниваний без пропусков пусть \mathcal{M} обозначает множество интересующих нас локальных выравниваний, $p(x, y)$ — частоту выравнивания символов x и y напротив друг друга в \mathcal{M} , а $p(x)$ — частоту появления символа x в локально выравниваемых последовательностях. Тогда оценки, которые наилучшим образом отличают интересующие нас локальные выравнивания от случайных локальных выравниваний, имеют вид $\log(p(x, y)/[p(x)p(y)])$. Снова оценка лог-различия.

15.11.3. Важность поиска белка среди белков

Сейчас в подавляющем большинстве случаев новые аминокислотные последовательности получаются расшифровкой исходного ДНК гена (или мРНК), который кодирует белок. Поэтому большинство записей в “белковой” базе данных являются на самом деле “выведенными аминокислотными последовательностями”, а исходные последовательности ДНК для них содержатся в базе данных ДНК. Поэтому, когда исследователь хочет найти сходную белковую последовательность, он часто может использовать либо последовательность ДНК для интересующего его белка и искать в базе данных ДНК, либо переведенную последовательность для поиска в базе белков. Есть много причин для поддержания и использования исходных последовательностей ДНК, но при распознании сходств в сильно различающихся последовательностях стандартный совет таков (по Дулиттлу):

“Переводите эти последовательности ДНК! Некоторые начинающие сравниватели строк находятся под впечатлением, что можно много выиграть за счет поиска в исходной последовательности ДНК, а не в выведенной из нее аминокислотной последовательности. Такое впечатление совершенно ошибочно...” [127].

Аргумент в пользу перевода следующий. Между аминокислотными последовательностями возможны более чувствительные и информативные сравнения, чем между последовательностями ДНК, благодаря общим методам выравнивания и целевым функциям типа описанных в этой книге.**) Вся “информация”, содержащаяся в выведенной аминокислотной последовательности (и даже больше), содержится в ее исходной последовательности ДНК. Однако эта информация по ДНК находится в форме, не позволяющей прямо применять общие методы выравнивания и целевые функции, делающие использование этой информации наиболее эффективным.

Выведенные строки аминокислот допускают более осмысленное выравнивание за счет использования матриц оценок, которые отражают эволюционное, биологическое или химическое сходство конкретных пар аминокислот. Например, замена лейцина на изолейцин обычно рассматривается как небольшое изменение, и этот

*) В этих теоремах, по-видимому, не учитывается закономерность появления цепочек символов. Если множество \mathcal{M} состоит из подстрок, содержащих цепочку букв, например *quick*, посимвольные оценки такого свойства распознать не смогут. — Прим. перев.

**) Перевод часто ошибочно обосновывается утверждениями вроде того, что “сходства у аминокислот лучше сохраняются в эволюции, чем у исходных ДНК”. Правду сказать, это утверждение было и у меня в первоначальном варианте этой книги, до того, как Стивен Альтшулер отметил, что главным здесь является не то, что информация сохраняется в последовательности, а то, насколько хорошо могут использовать эту форму информации общие методы выравнивания и статистической оценки. И действительно, есть случаи, где последовательности ДНК оказываются лучше сохраненными, чем переведенные аминокислотные последовательности [314].

факт отражается в матрицах PAM и BLOSUM. Но оценки выравнивания, полученные суммированием совпадений и несовпадений соответствующих пар нуклеотидов, не отразят близкого сходства лейцина и изолейцина. Сходным образом, большинство аминокислот закодировано больше чем одним кодоном, и два кодона, отличающиеся только в третьей позиции, часто кодируют одну и ту же аминокислоту. Этот тип корреляции, эффект третьей позиции, трудно корректно смоделировать в выравниваниях ДНК, использующих методы, описанные в книге. Оценка несовпадения пары нуклеотидов должна отражать появления этих нуклеотидов в кодах многих разных аминокислот. И так как есть только шестнадцать различных пар нуклеотидов, сумма оценок пар кодона редко сможет отразить специфичность оценок подстановок аминокислот из матрицы для 400 пар. Выведенная аминокислотная последовательность кодирует зависимости этих позиций, позволяя стандартным алгоритмам быть эффективными (вместе с матрицами подстановок). Есть, однако, некоторые попытки получить все возможное из обоих представлений, используя выведенные аминокислотные последовательности, но помня конкретные кодоны исходной ДНК. Это в идеале требует матрицы подстановок размера 64×64 .

Хотя искать белковые последовательности в переведенных последовательностях ДНК стандартно, в этом подходе есть важная техническая проблема, которая рассматривается в п. 18.1.

15.12. Упражнения

1. В некоторых базах данных реализовано хранение всех поступающих пользовательских запросов (строк запросов) и адресов пользователей. При поступлении в базу новых последовательностей они проверяются на сделанные ранее запросы, и при установлении существенного сходства автору запроса посылается сообщение. До появления такой службы было несколько случаев, когда пользователи упускали важные находки потому, что они обращались к обновленному экземпляру базы (GenBank сейчас обновляет текущую версию шесть раз в год), а их конкуренты пользовались более свежей версией и находили важные совпадения.

Какие структуры данных и поисковые алгоритмы лучше всего подошли бы для реализации этой услуги? Ответ может зависеть от таких параметров, как количество возможных запросов и новых последовательностей, добавляемых при каждом обновлении базы.

2. Вернемся к ситуации, описанной в задаче 1. Некоторые базы данных используют коллекцию предыдущих запросов для анализа будущих запросов. Значит, когда появляется очередной запрос, он сопоставляется с предыдущими, чтобы выяснить, не появлялся ли такой или очень похожий запрос ранее. Каждый запрос в коллекции указывает на строки базы с сильным сходством, так что при получении повторяющегося запроса можно избежать полного поиска. В некоторых базах данных повторы запросов являются обычным делом.

Рассмотрите возможности выбора структур данных и алгоритмов поиска для реализации этой услуги.

3. Когда в качестве основы значений подстановки аминокислот используются оценки (лог-)различия, может случиться, что оценка точного совпадения будет меньше, чем совпадения с отдельным несовпадением. Объясните, как это может случиться. Является ли эта особенность матрицы подстановок нежелательной?

Часть IV

Другие задачи:
текущие,
родственные
и просто изящные

В предыдущих трех частях книги изложены общая техника и специальные строковые алгоритмы, важность которых либо уже установлена, либо похоже, что будет установлена. Мы ожидаем, что этот материал будет отвечать области строковых алгоритмов и анализа молекулярных последовательностей еще многие годы. В заключительной части книги мы уходим в сторону от проверенной техники и от проблем, строго определенных на строках. Мы делаем это тремя способами.

Во-первых, мы рассматриваем подходы, которые сейчас очень актуальны (*current*), но могут не выдержать проверки временем, хотя и в состоянии привести к более мощным и эффективным методам. Равным образом, мы рассматриваем строковые задачи, которые важны для существующей технологии в молекулярной биологии, но могут потерять свое значение при изменении технологии.

Во-вторых, мы рассматриваем такие задачи, как физическое картирование, сборка фрагментов и построение филогенетических (эволюционных) деревьев. Они хотят и имеют отношение к строковым задачам, но сами таковыми не являются. Эти дальние родственники строковых задач (*cousins*) побуждают к исследованию либо конкретных чисто строковых задач, либо общей проблематики строковых задач вообще, обеспечивая более полную картину того, как биологические последовательности получаются и как в них *используются результаты работы* чисто строковых алгоритмов.

В-третьих, мы представляем несколько важных изящных миниатюр (*cameo topics*), не вдаваясь в подробности в отличие от других частей книги.

Конечно, некоторые из вопросов заключительной части входят во все три категории и являются одновременно актуальными, родственными и изящными (*and are simultaneously currents, cousins, and cameos*).

Карты, картирование, упорядочение и надстроики

16.1. Взгляд на некоторые задачи картирования и секвенирования ДНК

В этой главе мы рассмотрим ряд теоретических и практических вопросов создания и использования геномных карт и крупномасштабной (геномной) расшифровки ДНК. Мы обращаемся к ним по двум причинам. Во-первых, чтобы более подробно обсудить происхождение данных о молекулярных последовательностях, поскольку строковые задачи с такими данными в значительной мере побуждают к изучению строковых алгоритмов. Во-вторых, чтобы более подробно рассмотреть специфические строковые задачи, возникающие при получении этих данных.

Мы начнем с общего обсуждения картирования и различий между *физическими* и *генетическими* картами. Это приведет нас к рассмотрению нескольких методов физического картирования, таких как *STS-картирование* и *радиационно-гибридное картирование*. Мы выделим общие для этих методов комбинаторные и вычислительные аспекты. Дальше перейдем к обсуждению задач о плотнейшей укладке и дадим краткое введение в *сравнение карт* и *выравнивание карт*. Затем перейдем к *крупномасштабной расшифровке* и ее отношению к физическому картированию. Мы обратим особое внимание на *дробовое секвенирование* и на строковые задачи, связанные со *сборкой последовательности* при дробовой стратегии. Дробовое секвенирование вполне естественно приводит к изящной *чистой* строковой задаче о *кратчайшей общей надстроке*. Эта чистая и точная строковая задача мотивируется практической задачей дробовой сборки последовательности и заслуживает внимания хотя бы из элегантности полученных в связи с ней результатов. Дальше мы обратимся к расшифровке ДНК, обсуждая *секвенирование гибридизацией*, — это высоко структурированный частный случай задачи о кратчайшей надстроке.

16.2. Картирование и геномный проект

Конечной целью проекта “Геном человека” является расшифровка всего человеческого генома, создание полной записи ДНК — строки длиной в три миллиарда нуклеотидов. Хотя эта цель — наиболее афишируемый аспект проекта, на самом деле, расшифровка человеческой ДНК в промышленном масштабе только началась.*¹) Менее известной исходной целью проекта человеческого генома и других геномных проектов является создание карт генома, показывающих местонахождение “ориентиров” (таких, как STS и EST, рассмотренные в п. 3.5.1) на регулярных расстояниях вдоль генома. Геномные карты имеют решающее значение в охоте за конкретными генами; они применяются на первых шагах большинства, хотя и не всех, проектов крупномасштабной расшифровки ДНК (см. п. 16.13), они полезны и для дальнейшего физического исследования ДНК, которое требуется в других частях геномного проекта.

Мы рассмотрим несколько вопросов картирования и использования карт, которые переплетаются со строковыми задачами или родственны им по их участию в стратегиях расшифровки ДНК, обсуждаемых далее. Мы подчеркиваем комбинаторные аспекты этих задач, поскольку они лучше соответствуют стилю этой книги. Однако мы не будем стремиться к исчерпывающему обзору различных карт и стратегий картирования или к подробному обсуждению того, как карты создаются и используются, так как это увелено бы нас далеко от основной темы.

16.3. Физические и генетические карты

В целом есть два типа геномных карт: *генетические* и *физические*. Задачи получения и использования физических карт ближе к строковым, чем задачи, связанные с генетическими картами. Поэтому основное внимание будет уделяться задачам физического картирования.

Термин “физическое картирование” относится к получению правильных физических координат ориентиров (маркеров) или интересующих нас объектов (часто генов или известных шаблонов) в геноме. Используемая метрика основывается на наблюдаемых или подсчитываемых физических элементах, обычно на числе нуклеотидов между двумя точками. Самые ранние физические карты состояли из *шаблонов поперечной исчерченности*, которые появлялись на хромосомах при использовании метода дифференциальной окраски. Исчерченность хорошо воспроизводится и видна под микроскопом, так что она обеспечивает физическую карту низкого разрешения. Эта карта использовалась для предварительной локализации генов, удаление или развитие которых вызывает заболевание. При этом изменения хромосомы настолько велики, что меняется поперечная исчерченность в районе одной из полос. Например, ген-супрессор *ретинобластомы* был первоначально локализован в районе

*¹) Однако гены нескольких небольших бактерий с длинами порядка 2 миллиона пар оснований были полностью расшифрованы. Кроме того, весной 1996 г. была опубликована полная последовательность генома (пивных) дрожжей *Saccharomyces cerevisiae* (длиной около 12 миллионов пар оснований), геном *E. coli* (длиной 4.7 миллиона пар оснований) был закончен 16 января 1997 г. и было завершено секвенирование большой части генома круглого червя *C. elegans* (длиной 100 миллионов пар оснований). Кроме того, согласие о крупномасштабной расшифровке человеческого генома достигнуто быстрее, чем планировалось [309, 352, 176].

одной из полос хромосомы 13, когда было обнаружено исчезновение такой полосы у некоторых больных этим заболеванием [472]. И напротив, изолированные фрагменты хромосомы можно зачастую локализовать в полной хромосоме, зная характер ее поперечной исчерченности. В человеческой хромосоме число полос, проявляющихся при дифференциальной окраске, около сотни, и таким образом наблюдаемые полосы обеспечивают только физическую карту низкого разрешения. В других организмах, таких как дрозофилы, число полос значительно больше (около 5000), что повышает их полезность.*)

Целью современного физического картирования высокого разрешения является локализация в хромосоме изучаемых особенностей в терминах отсчетов пар оснований. Это значит, что положения измеряются и описываются в терминах числа нуклеотидов (пар оснований) между особенностью и некоторой другой имеющейся опорной точкой. Одной из таких особенностей, представляющей значительный интерес, являются локализации, маркированные последовательностями (STS, см. п. 3.5.1). Картирование STS на расстояниях порядка 100 000 оснований вдоль всего генома было одной из основных задач проекта “Геном человека” [234, 399]. В сравнении с генетическим картированием, физическое картирование высокого разрешения в большом масштабе относительно ново, сильно зависит от технологии рекомбинирования ДНК и является центральным компонентом проектов исследования человеческого и других геномов. Основные принципы физического картирования и других техник изучения человеческого генома можно найти в [111].

В отличие от физического картирования *генетическое картирование* (или *картирование групп сцепления*) устанавливает относительный порядок локусов в хромосоме или утверждает, что две особенности похожи в разных хромосомах. В генетическом картировании используют единицу, называемую *сантиморганидой*; и расстояния, измеренные в ней, зависят от физических расстояний монотонно.**) Генетическое картирование основано на учете частот совместного наследования аллелей двух генов в мейозе. Чем выше частота совместного наследования, тем ближе считаются эти два локуса. Поэтому для получения карты сцеплений нужно найти набор локусов, равномерно распределенных по ДНК, аллели которых были бы высоко изменчивы (или “полиморфны”). Первоначально это были локусы, изменения которых заметны на уровне организма (на “фенотипическом” уровне), но в последнее десятилетие стали использовать базовые маркеры, изменения которых проявляются прямо на уровне ДНК. Первыми маркерами этого типа стали *полиморфизмы с ограниченной длиной фрагмента* (*restriction-fragment-length polymorphisms* — RLFP) [476], но

*.) В слюнных железах личинки дрозофилы хромосомы удваиваются, но не разделяются затем, как это происходит обычно. Вместо этого тысяча или более копий хромосомы остаются соединенными (сшитыми) по всей длине, создавая очень толстую *политенную хромосому*, полосы которой хорошо видны даже в световой микроскоп. Эти политенные хромосомы можно было легко физически картировать и тем самым изучать генетику дрозофилы еще до появления технологии рекомбинирования ДНК. Например, знаменитая мутация дрозофилы, которая вызывает *изогнутые крылья*, была обнаружена благодаря *инверсии* определенного участка одной из хромосом этой мушки.

**.) В элементарных учебниках повсеместно сообщается, что одна сантиморганида примерно соответствует миллиону пар оснований ДНК. Но несмотря на все повторения этого утверждения, известно, что прямой линейной зависимости между физическим и генетическим расстояниями нет. Например, та ограниченная зависимость, которая существует, различна для различных видов (корреляция очень слаба у томатов и относительно велика у *Arabidopsis*), различна в различных частях генома, различается у мужских и женских особей. Более того, корреляция физического и генетического расстояний ухудшается с ростом физического расстояния.

все чаще в качестве маркеров выбираются микросателлиты (см. п. 7.11.1). Генетическое картирование посредством анализа сцеплений подробно описано в [426] и более глубоко рассмотрено в [353].

Генетическое картирование значительно старше и более развито, чем физическое картирование высокого разрешения. Оно сохраняет большое значение для приблизительной локализации нужных генов и в создании тестов, обнаруживающих носителей некоторых дефектных генов. Оно очень важно также для таких прикладных областей, как селекция растений и животных, когда пытаются сделать так, чтобы нужный ген располагался близко к гену, эффект которого в организме легче наблюдать. Селекционер пытается в этом случае получить особей, имеющих аллель (состояние) этого гена, который сильно коррелирует с состоянием более глубоко спрятанного нужного гена.

Генетическое картирование полезно и в физическом картировании — тем, что выделяет нужную область, в которой можно потом работать более дорогими методами. Такое сочетание генетического и физического картирования получило важный импульс и будет получать их в дальнейшем в связи с необходимостью картировать гены наследственных заболеваний, даже когда дефектный белок, связанный с этим заболеванием, неизвестен. Этот подход к нахождению дефектных генов называется *позиционным клонированием*, — он противоположен старому подходу, где прежде всего распознается и изучается вызывающий заболевание белок. Благодаря позиционному клонированию за последние несколько лет было локализовано около сотни генов (наиболее известные случаи — это муковисцидоз, хорея Гентингтона и наследственные формы рака молочной железы). Грубо говоря, идея состоит в использовании генетической карты для анализа образцов наследования заболевания в нескольких поколениях наблюдаемой семьи. Анализируя, какие локусы имеют аллели, часто *наследуемые вместе* с заболеванием, можно поймать болезнетворный ген между двумя позициями на генетической карте. Затем физические карты этой области позволяют вытянуть подстроки ДНК для более точного анализа и/или расшифровки. Далее эти подстроки изучаются, чтобы в них “увидеть и понять” особенности гена (например, открытую рамку считывания, высокую концентрацию нуклеотидов G и C или кодоны, типичные для генов этого организма). Дальнейшие детали см. в п. 18.2. Целью является отыскание в ДНК маленькой подстроки из предполагаемого гена (часто лишь отдельного нуклеотида), в которой проявляется систематическое различие между индивидуумами — подверженными болезни и здоровыми. Ген, содержащий эту подстроку, является первым кандидатом на роль болезнетворного гена. По мере того как генетические и физические карты станут более точными, подход позиционного клонирования к локализации генов будет быстрее, дешевле и более рутинным.

Создание генетических карт высокого разрешения формирует центральную часть геномных проектов. Однако мы не будем дальше обсуждать генетическое картирование, потому что связанные с ним вычислительные задачи менее “строковые” и менее комбинаторны, чем задачи физического картирования.

16.4. Физическое картирование

Хотя крупномасштабное физическое картирование относительно ново, в нем используется уже несколько различных методов и существует много перспективных предложений. Этим ситуация отличается от расшифровки ДНК, где широко используемых

методов мало, и от генетического картирования, где есть только один общий подход (основанный на кроссинговере в мейозе) и несколько родственных методов, работающих в специальных ситуациях.

Мы рассмотрим несколько вычислительных, комбинаторных вопросов, возникающих при создании физических карт, и один вычислительный вопрос, связанный со сравнением карт (физических и генетических). Эти задачи в строгом смысле слова не строковые, но близки к ним, и техника их решения часто близка к чисто строковым задачам. Более того, в контексте картирования возникают некоторые специфические строковые задачи, и наоборот, крупномасштабную расшифровку ДНК обычно начинают с получения ее физической карты.

16.5. Физическое картирование: STS-картирование и упорядоченные библиотеки клонов

Начнем обсуждение физического картирования с конкретного случая одной важной стратегии картирования: с *STS-картирования* [189, 234].

Напомним из п. 3.5.1 (с.), что STS — это в общем случае (хотя не вполне точно) короткая подстрока ДНК, которая встречается только в одном месте генома. EST — это STS, полученная путем транскрибирования последовательности (кДНК генов). Наиболее общий процесс получения STS не позволяет определить ее местонахождение в геноме, — он определяет лишь, что STS встречается только единожды [111, 317]. Чтобы эффективность была наибольшей, физическое место STS должно быть определено по физической карте STS.

Библиотека клонов представляет собой набор коротких фрагментов ДНК (они-то и называются *клонами*), которые можно сохранять в лабораторных условиях и которые получены из нужной ДНК. Важно отметить, что клоны библиотеки могут перекрываться. Таким образом, любое конкретное место ДНК может быть накрыто двумя или более клонами. В случае физического картирования клоны библиотеки в идеале должны покрывать рассматриваемую ДНК. Однако исходное положение клонов в ДНК и даже их относительный порядок могут быть неизвестны. Задача построения *упорядоченной библиотеки клонов* подразумевает определение порядка и физической локализации клонов в рассматриваемой строке ДНК.

STS-картирование — это подход, при котором одновременно определяют упорядочение STS и строят физическую карту для упорядоченной библиотеки клонов.*^{*)} Процедура вначале определяет содержание STS в клонах библиотеки (рис. 16.1). Это можно сделать, пытаясь гибридизовать каждую STS к каждому клону. Можно также использовать PCR **), чтобы определить, содержится ли конкретная STS в конкретном клоне из библиотеки. Концы каждой STS включают в себя известные и уникальные праймерные последовательности PCR. Поэтому, чтобы определить, содержится ли конкретная STS в конкретном клоне библиотеки, используют известные праймеры для этой STS и определяют, генерирует ли PCR большие количества новой ДНК из ДНК в клоне. Конкретная STS находится в клоне в том и только том случае,

^{)} Некоторые исследователи больше заботятся об упорядочении STS, чем о раскладке клонов. Причина в том, что многие существующие библиотеки клонов имеют ошибки, которые снижают интерес к самим клонам, хотя их можно использовать для получения порядка STS.

**) Краткую информацию о PCR — реакции полимеризации цепи — см. в словаре.

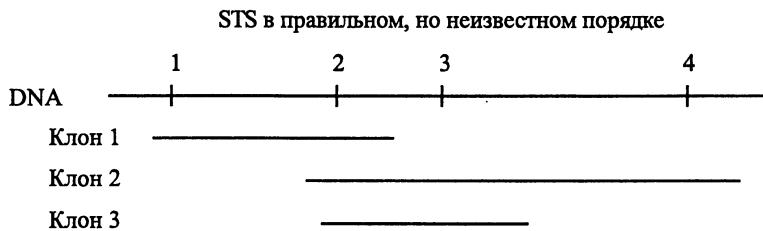


Рис. 16.1. Схема с четырьмя клонами и тремя STS. Рисунок показывает реальное расположение STS и клонов, которое, однако, неизвестно и должно быть установлено алгоритмом

		STS			
		2	4	1	3
Клон	1	1	0	1	0
	2	1	1	0	1
	3	1	0	0	1

Рис. 16.2. Матрица данных для клонов (см. рис. 16.1). Порядок STS в записи изменен, так как их правильный порядок, вообще говоря, неизвестен

если применение PCR с участием этих праймеров синтезирует новую ДНК. Выполнение эксперимента для каждой пары STS/клон дает данные, которые можно собрать в двоичную матрицу. В ней строчки соответствуют клонам, а столбцы — STS. Элемент (i, j) этой матрицы равен 1, если клон i содержит STS j (рис. 16.2).

Конечно, настоящий протокол эксперимента сложнее описанного, и вычислительная задача всегда усложняется ошибками (которые мы обсудим позднее). Но наше рассмотрение отражает все черты, присущие комбинаторным аспектам этой задачи. Обратимся теперь к вопросу, как использовать данные о содержании STS для воссоздания порядка STS и места клонов ДНК на физической карте. За всеми подходами к получению порядка STS стоят две идеи. Во-первых, расстояние между STS i и i' находится в обратной зависимости от числа клонов, общих для i и i' . Значит, если i и i' появляются вместе во “многих” клонах, то можно заключить, что они “близки” в строке ДНК. Во-вторых, если STS i появляется в клонах j и j' , то эти клоны должны перекрывать друг друга в месте, где расположена STS i .

16.5.1. Реконструкция порядка STS

Задача реконструкции сильно упрощается предположением, что данные не содержат ошибок. Оно означает, что данные правильно отражают предположение, что каждая STS встречается только в одном месте ДНК и что если два клона перекрываются в этом месте, то процедура PCR сообщит, что оба клона содержат эту STS. При этих предположениях задача реконструкции порядка STS является приложением классической задачи о последовательных единицах [74]. Исходными данными в задаче о последовательных единицах является двоичная матрица, а целью — перестановка столбцов матрицы так, чтобы в каждой строчке все единицы располагались в блоке

		STS			
		2	4	1	3
Клон	1	1	1	0	0
	2	0	1	1	1
	3	0	1	1	0

Рис. 16.3. Когда данные о содержании STS не содержат ошибок, столбцы матрицы можно переставить таким образом, что в каждой строчке все единицы будут идти подряд. Здесь показана матрица рис. 16.2, в которой столбцы переставлены именно так

последовательных элементов (рис. 16.3). По предположению об отсутствии ошибок в данных такая перестановка, соответствующая верному, но неизвестному порядку STS в ДНК, существует.

Очаровательный алгоритм с линейным временем, предложенный Бутом и Люкетром [74], находит перестановку, обеспечивающую сплошные единицы, или устанавливает, что такой перестановки не существует. В случае если таких перестановок несколько, метод характеризует их все компактным, но ясным образом. Ясно, что упорядочение с последовательными единицами определяет перестановку, которая дает упорядочение STS, совместное с данными. Такой порядок STS фиксирует положение каждого клона относительно этого порядка. Если длины клонов известны, то их можно использовать для определения физической раскладки клонов и физической карты расположения клонов. Конечно, точную физическую раскладку клонов из данных можно и не получить, даже если порядок STS был установлен правильно.

Итак, задача воссоздания STS может быть решена подходом со сплошными единицами, если данные не содержат ошибок. К сожалению, так бывает редко. Рассмотрим несколько типов ошибок, встречающихся в STS-картировании.

Ошибки в данных о содержании STS

Есть три типа систематических ошибок (в добавок к ошибкам, вызванным загрязнением): *ложные положительные, ложные отрицательные* и ошибки из-за *химерных клонов*. Ложным положительным является заключение, что клон содержит конкретную STS, когда на самом деле он ее не содержит, а ложным отрицательным — заключение, что клон не содержит STS, когда на самом деле содержит. Химерные клоны — это более сложный тип ошибки. В процессе получения и сохранения клонов иногда случается, что два разных фрагмента ДНК объединяются и ведут себя как отдельный клон. Этот новый “отдельный” клон называется *химерным*, два образующих его фрагмента на самом деле могут отстоять друг от друга в ДНК очень далеко. Ясно, что химерные клоны усложняют процесс определения правильного порядка STS по данным PCR. В некоторых библиотеках химерных клонов больше 50 %, и похоже, что тенденция к формированию *химер* растет с ростом размеров фрагментов. Особенно серьезна эта проблема для фрагментов, включенных в *искусственные дрожжевые хромосомы* (*yeast artificial chromosomes* — YAC), которые обычно содержат включения около 500 000 оснований, и клоны мегаYAC с включениями до 2 000 000 оснований. Химеры, состоящие более чем из двух фрагментов, могут также встречаться, но гораздо реже.

Для обработки ошибок в STS-картировании предлагалось много подходов, но в этом очерке мы не будем пытаться их даже описывать. Вместо этого обсудим один

комбинаторный подход, основанный на задаче о “бродячем торговце”. Эта задача обобщает задачу о сплошных единицах, а получающийся алгоритм можно применить также к воссозданию порядка STS в другом методе картирования, называемом *радиационно-гибридным картированием*. Мы увидим, что задача обработки ошибок в STS-картировании может рассматриваться как частный случай задачи упорядочения STS в радиационно-гибридном картировании. Поэтому отложим обсуждение обработки ошибок на время после рассмотрения радиационно-гибридного картирования.

16.6. Физическое картирование: радиационно-гибридное картирование

Техника, называемая *радиационно-гибридным картированием* [113, 317, 408], стала популярной как средство картирования больших клонов, которые могут составлять от 10 до 25 % отдельной человеческой хромосомы (так что каждый клон может иметь длину в десятки миллионов оснований). Этот метод представляет собой гибрид генетического и физического картирования и использовался в нескольких геномных проектах [234, 399]. Как мы увидим, некоторые связанные с ним вычислительные задачи можно свести к задаче о бродячем торговце.

Радиационно-гибридное картирование использует способность человеческой ДНК включаться в клетки грызунов. Чтобы упростить рассмотрение, ограничимся случаем, когда метод прилагается к отдельной человеческой хромосоме, хотя его можно использовать и на уровне целого генома. Итак, вкратце метод таков. Отдельная хромосома из генома человека изолируется и облучается, что разбивает ее в случайных местах на небольшое число фрагментов. Эти фрагменты (из одного экземпляра отдельной хромосомы) внедряются затем в клетку хомяка, от чего человеческая ДНК начинает воспроизводиться вместе с ДНК хомяка. В последовательных поколениях каждая клетка хомяка воспроизводит человеческую ДНК все в меньших количествах, так что становится обычной клетка, которая содержит только какую-то долю отдельной человеческой хромосомы. В результате в последнем поколении любая отдельная клетка хомяка содержит пять–десять не связанных и не перекрывающихся фрагментов отдельной человеческой хромосомы, общая длина которых составляет 15–20 % человеческой хромосомы (рис. 16.4). Этот процесс получения набора фрагментов, который частично покрывает отдельную человеческую хромосому, повторяется с разными экземплярами одной и той же хромосомы; он достаточно случаен, так что две клетки хомяка, содержащие фрагменты из экземпляров одной и той же хромосомы, будут содержать разные наборы фрагментов. Эти наборы будут покрывать разные, но, возможно, перекрывающиеся части хромосомы (см. рис. 16.4).

В этот момент можно для каждой клетки определить, какой из набора неупорядоченных STS находится в человеческих фрагментах, содержащихся в клетке. Как в случае STS-картирования, это определение можно выполнить гибридизацией или с помощью PCR. Для каждой пары STS/клетка нужно проверить, допускают ли праймеры для этой STS генерирование больших количеств новой ДНК на основе ДНК гибридной клетки. Хотя это и напоминает STS-картирование, есть одно существенное отличие. Такой тест определяет, содержится ли STS в одном из человеческих

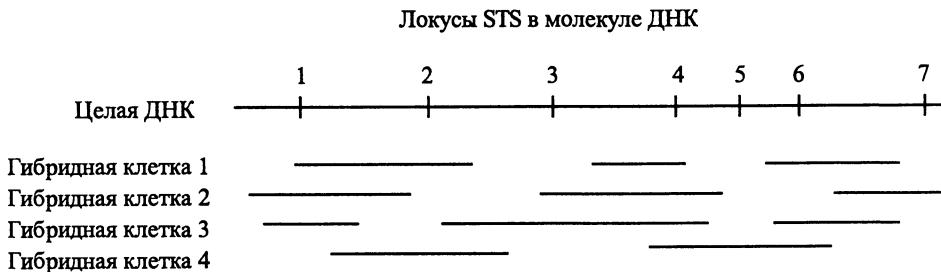


Рис. 16.4. Схема фрагментов человеческой ДНК, которые содержатся в четырех гибридных клетках хомяка. Первая линия представляет полную хромосому человеческой ДНК и показывает истинное (но неизвестное) расположение семи STS. Еще четыре следующие линии показывают истинное (но неизвестное) расположение фрагментов. Для простоты в каждой клетке показано меньше пяти фрагментов

		STS						
		2	4	5	3	1	7	6
Фрагмент	1	1	1	0	0	1	0	1
	2	0	1	0	1	1	1	0
	3	0	1	0	1	1	0	1
	4	1	1	1	0	0	0	1

Рис. 16.5. Пример данных радиационно-гибридного картирования с рис.16.4. Единица в клетке (i, j) показывает, что STS j присутствует во фрагменте, находящемся в гибридной клетке i . Так как истинный порядок STS неизвестен и должен быть определен, метки столбцов представлены

фрагментов в клетке, но не может сказать, в *каком именно*. На самом деле даже число фрагментов неизвестно и должно определяться. Данные, полученные в этих тестах, снова сводятся в двоичную матрицу, каждая строчка которой соответствует гибридной клетке, а каждый столбец — одной STS (рис. 16.5). В клетке (i, j) стоит 1, если фрагменты гибридной клетки i содержат STS j .

Теперь задача заключается в том, как из этих двоичных данных получить правильный порядок STS. За всеми подходами к получению порядка стоит интуитивное представление, что частота, с которой две STS вместе присутствуют или вместе отсутствуют в гибридной клетке, зависит от физического расстояния между этими двумя STS в хромосоме. Это следует из того, что чем они ближе, тем правдоподобнее, что они обе содержатся во фрагменте клетки (в частности, в одном и том же), если одна из них содержится в этом фрагменте. Эксперимент осуществляется над многими случайными образцами хромосомы, и каждый образец дает набор STS, найденных в случайному наборе неперекрывающихся интервалов, покрывающих только часть хромосомы.

Опять-таки, истинный экспериментальный протокол сложнее, чем описанный выше, и дополнительно усложнен ошибками. Однако комбинаторный элемент ясен, и мы обсудим теперь один подход к реконструкции порядка STS по данным гибридизации.

16.6.1. Реконструкция порядка STS в радиационных гибридах

Задача упорядочения STS по безошибочным радиационно-гибридным данным не отличалась бы от задачи воссоздания по безошибочным данным содержания STS, если бы каждая гибридная клетка содержала только один человеческий фрагмент. Но так как фрагментов в клетках больше, чем по одному, то предложенный способ [13] получения правильного порядка STS заключается в нахождении перестановки столбцов, которая минимизирует число блоков из последовательных единиц. При этом каждый блок интерпретируется как фрагмент из клетки, в которой были получены эти данные. В другом способе предлагается найти перестановку, которая минимизирует максимальное число блоков, появляющихся в отдельных строчках. Оба эти предложения обобщают задачу о последовательных единицах, но, в отличие от нее, для них нет быстрого (по оценке наихудшего случая) точного метода решения. Однако мы увидим, что первая задача может рассматриваться как задача о бродячем торговце на неориентированном графе, а вторая — как минимаксная задача о бродячем торговце. Обсудим первый вариант подробно.

16.6.2. Формулировка упорядочения STS как задачи о бродячем торговце

Пусть задана двоичная матрица радиационно-гибридных данных. Построим неориентированный граф с одной вершиной для каждой STS (столбца матрицы) и еще одной дополнительной вершиной s и ребрами, соединяющими каждые две вершины. Обход бродячего торговца (или гамильтонов цикл) в этом графе — это путь, который начинается и кончается в вершине s и заходит в каждую другую вершину только по одному разу. Поскольку все вершины, кроме s , посещаются в обходе строго по разу, порядок их обхода определяет перестановку столбцов матрицы. Следовательно, каждый обход определяет упорядочение STS.

Сопоставим теперь каждому ребру графа некоторый вес. Весом ребра (s, v) для любой вершины v будет общее число единиц в столбце v . Весом любого другого ребра (u, v) — *расстояние Хэмминга* между столбцами u и v , т. е. число строчек, где элементы в столбцах u и v различны. Задачей о бродячем торговце называется задача нахождения в графе обхода с наименьшей суммой весов ребер (рис. 16.6 и 16.7).

Каково интуитивное обоснование таких весов? Предположим сначала, что столбец u имеет единицу, а столбец v — нуль в строчке i и что ребро (u, v) участвует

		STS			
		2	4	3	1
Фрагмент	1	1	1	0	1
	2	0	1	1	1
	3	0	1	1	1
	4	1	1	0	0

Рис. 16.6. Часть радиационно-гибридных данных из примера, изображенного на рис. 16.5. Неориентированный граф для этих данных показан на рис. 16.7

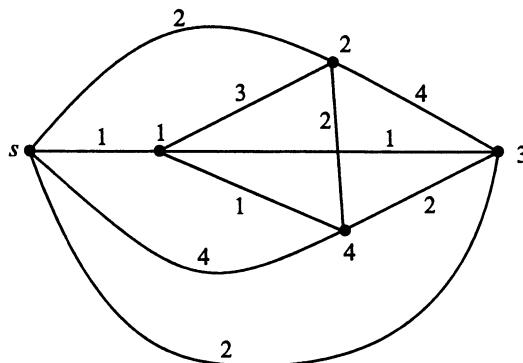


Рис. 16.7. Граф, используемый в задаче о бродячем торговце для упорядочения STS по данным рис. 16.6. К сожалению, в этом примере оптимальный обход $s, 3, 1, 4, 2, s$ минимизирует число создаваемых блоков (четыре), но не воссоздает правильно исходное упорядочение STS

в обходе. Упорядочение, порождаемое обходом, создает блок единиц строчки i , который заканчивается в столбце u . Судя по элементам матрицы, $STS\ i$ и имеется в гибридной клетке i , а $STS\ v$ отсутствует. Поэтому из упорядочения STS , соответствующего упорядочению столбцов, следует, что фрагмент в гибридной клетке i должен закончиться после $STS\ u$, но до $STS\ v$. Вес ребра (u, v) , таким образом, считает число блоков единиц, которые закончатся в столбце u (если ребро (u, v) участвует в обходе), и следовательно, он считает также число фрагментов, которые кончаются после $STS\ u$ в получающемся восстановлении фрагментов. Теперь предположим, что столбец u имеет 0, а столбец v имеет 1 в строчке i . Тогда из прохождения ребра (u, v) следует, что фрагмент в гибридной клетке i начинается в $STS\ v$ и что новый блок единиц в строчке i начинается в столбце для v .

Если обход включает ребро из начальной вершине s в вершину v , то вес ребра (s, v) равен числу блоков единиц, начинающихся в крайнем левом столбце представлена матрицы. Аналогично, если обход включает ребро из v в s , то вес ребра (s, v) равен числу блоков, которые заканчиваются в крайнем правом столбце этой матрицы. Таким образом, при такой системе весов ребер каждый блок (или получившийся фрагмент), созданный обходом, учитывается строго два раза. Итак:

Теорема 16.6.1. *Обход с суммой весов ребер w задает перестановку столбцов, создающую ровно $w/2$ блоков сплошных единиц. Следовательно, обход с минимальным суммарным весом порождает перестановку с минимальным числом блоков сплошных единиц. Эта перестановка порождает упорядочение STS , минимизирующее число фрагментов, которые должны находиться в радиационно-гибридных клетках.*

Поэтому если истинное упорядочение STS действительно минимизирует суммарное число блоков сплошных единиц (как мы предполагали), то это упорядочение можно найти, решая получающуюся задачу о бродячем торговце. К сожалению, ее оптимальное решение не всегда воссоздает исходное решение, что и показано на рис. 16.7.

Задача о бродячем торговце NP-трудна [171], но некоторые весьма впечатляющие успехи показали, что за разумное время ее можно решить точно для графов

с сотнями вершин. Вдобавок в этих методах само точное решение находится относительно быстро, и большая часть времени идет на доказательство того, что получено именно точное решение. Если доказательства не требуется, то методы становятся особенно быстрыми. В настоящее время число STS, используемых в радиационно-гибридном картировании (для отдельной хромосомы), достаточно мало, так что соответствующие задачи о бродячем торговце можно решать таким способом. Кроме того, известны очень эффективные эвристические методы. Основанные на локальном поиске или на такой технике, как денатурация ДНК, они могут очень быстро получать решения, близкие к оптимальным, хотя и не дают никакой гарантии близости к решению [13]. Чтобы получить гарантию, можно использовать приближенный метод с полиномиальным временем, который находит перестановку с числом блоков сплошных единиц, превосходящим число в оптимальной перестановке не более чем на 50 %. Значит, если оптимальная перестановка производит q блоков единиц, то перестановка из приближенного алгоритма произведет не более $1.5q$ блоков. Это следует из метода, предложенного Кристофицидисом [102, 250], который для задачи о бродячем торговце нашел обход, длина которого не более чем в 1.5 раза превосходит длину кратчайшего обхода в предположении, что длины ребер удовлетворяют неравенству треугольника. Легко установить, что расстояние Хэмминга удовлетворяет неравенству треугольника, так что метод Кристофицидиса применим. Как отмечалось ранее (см. п. 14.9), гарантированные приближенные оценки обычно слишком пессимистичны, и истинное поведение этих методов на практике много лучше, чем гарантированное.

Обычно от модели бродячего торговца для задачи упорядочения STS не ожидается, что она даст полностью порядок STS. Скорее, она дает первое приближение к порядку, которое может подтверждаться или исправляться другими данными. Например, предлагаемый порядок STS можно использовать в сочетании с некоторой информацией о содержании STS из отдельных фрагментов, с различными типами генетической дактилоскопии (рассматриваемыми ниже), с существующей генетической картой или с другими физическими картами той же ДНК. Эти дополнительные данные можно либо вводить в модель бродячего торговца как побочные ограничения, либо использовать по случаю (ad hoc) для модификации порядка STS, задаваемого путем торговца.

16.6.3. Назад к STS-картированию: случай ошибок

Мы отложили обсуждение обработки ошибок при STS-картировании до рассмотрения радиационно-гибридного картирования. Теперь следовало бы объяснить, почему это сделано. Задача STS-картирования при наличии ошибок (ложных положительных, ложных отрицательных и химер) имеет ту же комбинаторную структуру, что и задача упорядочения STS по безошибочным радиационно-гибридным данным. Поэтому работа с двумя этими задачами очень похожа.

Данные о STS-содержании в химерных клонах обладают всеми свойствами данных радиационно-гибридного картирования, кроме того, что число фрагментов (клонов) в клетке меньше. В радиационно-гибридном картировании число фрагментов на гибридную клетку часто составляет 5–10, а в химерном клоне их редко больше двух. Тем не менее химерные данные о STS-содержании можно использовать для построения неориентированного взвешенного графа, в точности как это было сделано для

радиационно-гибридных данных. Путь торговца, посещающий каждую вершину по разу, снова определяет упорядочение STS и создает блоки сплошных единиц в каждой строчке матрицы данных. Так как множество клонов содержит химеры, мы ожидаем, что будут строчки, содержащие больше одного блока сплошных единиц. Возникает вопрос: какую целевую функцию мы должны использовать при выборе обхода?

В случае химерных клонов не ожидается более двух блоков сплошных единиц на строчку, так что искать обход с минимальной *полной ценой* уже не годится. Такой обход порождает упорядочение столбцов, которое минимизирует суммарное число блоков сплошных единиц, но не ограничивает число блоков в строчке. Лучше будет минимизировать полную стоимость обхода *при условии*, что получающаяся представлена матрица не содержит ни в одной строчке больше двух или, может быть, трех блоков единиц. Можно также искать упорядочение (перестановку столбцов), минимизирующее максимальное число получающихся блоков сплошных единиц во всех строчках. Эта задача также NP-трудна, но Истраил и др. [191, 242] предложили для нее приближенный метод и основанную на нем программу.

Рассмотрим теперь случай ложных положительных и ложных отрицательных данных без усложнения химерными клонами. Когда ошибок нет, столбцы матрицы содержания STS можно переставлять так, чтобы каждая строчка сохраняла свойство сплошных единиц. Относительно этого корректного упорядочения эффект ложного положительного теста для конкретной пары STS/клон заключается в том, что либо (единственный) блок данного клона удлиняется на один столбец (с любой стороны), либо создается отдельный блок, состоящий из одного элемента. Если ложный положительный ответ встречается во всех парах STS/клон с равной вероятностью, то вероятность создания нового блока будет существенно выше вероятности расширения имеющегося. Аналогично, ложный отрицательный ответ вызовет уменьшение блока на один столбец или разрыв блока на два блока, разделенных одним нулем. И здесь, если нет систематического смещения, вероятность разбить блок выше, чем вероятность его уменьшить. В результате мы можем сосредоточиться на тех ложных ответах, которые увеличивают число блоков, а не на тех, которые меняют их размеры. С этой точки зрения, задача о фальшивых ответах может снова моделироваться как задача нахождения упорядочения столбцов, минимизирующего либо полное число блоков, либо максимум из числа блоков в одной строке. Какая из двух целевых функций окажется более подходящей, зависит от частоты ошибок. Мы оставляем читателю рассмотрение задач, в которых наряду с ложными ответами встречаются химерные клоны.

16.7. Физическое картирование: дактилограммы при общем построении карт

STS-картирование и радиационно-гибридное картирование, использующее STS, пользуются тем удобным свойством, что каждая STS встречается в ДНК в единственной позиции. Оно облегчает задачу упорядочения STS и задачу раскладки карты, на которой показаны позиции клонов или фрагментов. При всей трудности этих задач единственность STS, очевидно, дает выигрыш. Есть, однако, другие методы физического картирования, где пробы могут находиться в ДНК больше, чем в одной локализации (рис. 16.8). Такие пробы часто необходимы для получения более подробной

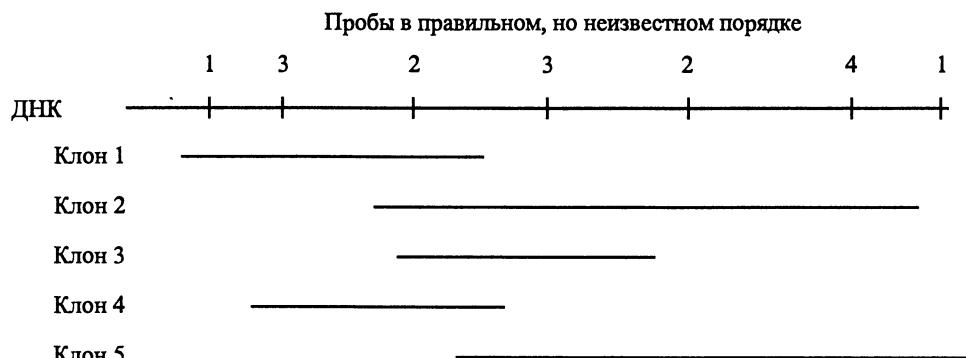


Рис. 16.8. Общая задача картирования, использующая дактилограммы для неодноместных проб. Данные пробы/клон из этого примера можно было бы представить двоичной матрицей 5×4 . В этом примере было бы трудно правильно воссоздать раскладку клонов по двоичной матрице

карты, чем карта STS. Задача воссоздания с использованием неодноместных проб часто основывается на **дактилограммах**.

Чтобы определить дактилограммы, рассмотрим библиотеку клонов и набор проб. Характерный пример пробы в этой постановке задачи — это либо олигонуклеотид (короткая строка ДНК), который гибридизуется с клонами, содержащими подстроку, комплементарную к олиге, либо рестриктаза, которая разрезает ДНК в каждом вхождении специфической короткой строки в клоне или класса коротких подстрок. Менее информативная форма “пробы” дает только длину подстроки ДНК между двумя местами рестрикции. Какова бы ни была специфика пробы, набор проб, которые дают положительные ответы для конкретного клона, образует **дактилограмму** клона. Таким образом, дактилограмма снова абстрактно представляется двоичной матрицей, в которой каждая строчка соответствует клону, а каждый столбец — пробе. Это напоминает STS-картирование, но теперь ожидается, что пробы встречаются в ДНК больше одного раза. Еще больше усложняет задачу то, что не всегда можно определить количество мест в полной ДНК или в клоне, где данная пробы дает положительный ответ.

Проблема заключается в том, чтобы расположить (или собрать) образец из перекрывающихся клонов, проверяя совпадения в их дактилограммах. Так как пробы, вообще говоря, входит в ДНК не обязательно один раз, подходы, основанные на задаче о сплошных единицах, больше не годятся. Вместо этого в большинстве подходов используется идея, в согласии с которой чем больше два клона перекрываются, тем больше можно ожидать, что их дактилограммы одновременно разделяют или отрицают одни и те же пробы. Многие из предлагаемых методов сборки первоначально проверяют клоны попарно, используя их дактилограммы для оценки длины перекрытия. Эта оценка использует число разделенных проб, число совместно отвергнутых проб и точную или оцениваемую частоту конкретных проб. После попарных сравнений клоны собираются в манере жадного алгоритма; два клона с наибольшим оцениваемым перекрытием составляются вместе, а копии их разделенных проб располагаются в области перекрытия клонов. Составной клон (называемый **контигом**)

добавляется затем к набору клонов, и шаг жадного алгоритма повторяется, хотя задача и ограничивается расположением копий проб в контиге. У нас нет возможности полностью рассмотреть этот метод и его многочисленные варианты. Основной статьей, обсуждающей аналитические и статистические аспекты работы с дактилограммами (оценка перекрытия клонов по отпечаткам, определение нужного числа клонов и др.), является работа Ландера и Уотермена [292]. Учебное изложение этих вопросов появилось в [461]. Однако приведенный выше набросок достаточен, чтобы обосновать предлагаемую в следующем параграфе подзадачу сборки, которая имеет особенно прозрачное комбинаторное решение.

16.8. Вычисление самой плотной раскладки

Этот параграф посвящен подзадаче раскладки клонов, которая решается в подходе к физическому картированию, предложенном Р. Карпом и др. [12]. Она возникает как особый фрагмент более крупной задачи картирования, но она интересна и сама по себе.

Определение. Пусть \mathcal{P} — набор проб (или имен), а C — набор из n клонов. Для каждого клона $c \in C$ пусть P_c — набор проб, о которых известно, что они встречаются в клоне c .

В этой задаче проба может встретиться больше чем в одном месте, и число появлений пробы в “истинной” раскладке неизвестно.

Определение. Пусть заданы P_c для каждого $c \in C$. Допустимой раскладкой называется такое отображение клонов (как интервалов) и появлений проб (как точек) на вещественную прямую, что интервал для каждого клона $c \in C$ покрывает по крайней мере одну копию каждой пробы из P_c и не покрывает копий проб, не принадлежащих P_c .

Так как на число местонахождений одной пробы не накладывается ограничений, то допустимая раскладка всегда возможна — нужно просто разместить каждый клон c полностью изолированно от остальных и поместить одну копию каждой требуемой пробы в интервале этого клона. Более полезная задача — найти “самую плотную” допустимую раскладку (занимающую наименьшую протяженность на вещественной прямой и/или требующую наименьшего числа копий), из которой можно вывести все другие допустимые раскладки.

Этой задачей занимаются в статье [12], но рассмотрение ограничено в двух аспектах. Во-первых, длина всех клонов предполагается одинаковой. Это предположение приближенно моделирует сегодняшние протоколы картирования, которые требуют от фрагментов клонов “отбора по размеру”, чтобы достичь достаточной равномерности. При равной длине если левый конец одного клона расположен левее левого конца другого клона в той же раскладке, то так же соотносятся и правые концы этих клонов. Поэтому порядок клонов при просмотре слева направо в любой фиксированной раскладке полностью определяется *перестановкой* имен клонов, хотя к одной и той же перестановке приводит бесконечное число раскладок. Второе сделанное в [12] ограничение заключается в том, что первоначально задается перестановка имен клонов.

Итак, решаемая в [12] задача состоит в отыскании самой плотной допустимой раскладки (клонов и проб) по данным о пробах и клонах и заданной перестановке имен клонов и в предположении, что интервалы клонов имеют равную длину. Два перекрывающихся клона в допустимой раскладке могут располагаться на прямой сколь угодно близко друг к другу, если только некоторое ненулевое расстояние разделяет их левые концы. В дальнейшем мы считаем, что зафиксирована перестановка клонов, и они уже занумерованы слева направо числами $1, 2, \dots, n$.

Нахождение плотнейшей раскладки

Ключевым является простое условие, которое исключает перекрытие двух клонов i и k в любой допустимой раскладке. Пусть $i < j < k$ — три клона, занумерованные согласно исходной перестановке. Если данные о пробах и клонах определяют, что некоторая проба p входит в клон j , но не входит ни в i , ни в k , то клоны i и k не могут перекрываться ни в одной допустимой раскладке (рис. 16.9). Более того, если i, j, k таковы, как предполагалось, и $i' \leq i < k \leq k'$, то клон i' не может перекрыть клон k' ни в какой допустимой раскладке. Действительно, так как длины всех клонов одинаковы и порядок левых концов фиксирован, перекрытие клонов i' и k' влечло бы и перекрытие i и k (рис. 16.10).

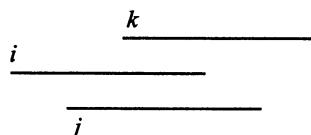


Рис. 16.9. Если клоны i и k перекрываются, то клон j (для $i < j < k$) должен полностью содержаться в их оболочке, так как левый конец клона j лежит справа от левого конца клона i , а правый конец клона j лежит слева от правого конца клона k . Следовательно, если клоны i и k перекрываются, то никакую пробу $p \in P_j$ не поместить в интервал для j так, чтобы не попасть в интервалы для i и k

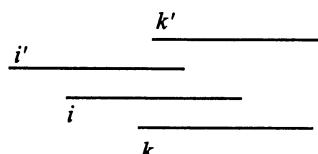


Рис. 16.10. Если клоны i' и k' перекрываются, то клоны i и k тоже, поскольку $i' \leq i < k \leq k'$

Определение. Пара клонов (i', k') называется *исключенной парой*, если $i' \leq i < j < k \leq k'$ для некоторых клонов i, j, k , где P_j содержит пробу p , которая не содержится ни в P_i , ни в P_k . Пара называется *разрешенной*, если она не исключенная.

Отметим, что определения исключенной и разрешенной пары относятся к фиксированной перестановке клонов. Пара может быть исключенной в одной перестановке и разрешенной в других. Следующая лемма вытекает прямо из определений. Она будет использоваться в дальнейшем.

Лемма 16.8.1. Если пара (i', k) исключена и $i < i' < k$, то пара (i, k) также исключена. Аналогично, если пара (i, k) разрешена и $i < i' < k$, то пары (i, i') и (i', k) также разрешены.

Чтобы пара клонов перекрывалась в какой-либо допустимой раскладке, необходимо, чтобы она была разрешенной, но мы докажем нечто более сильное. Руководствуясь леммой 16.8.1, мы построим допустимую раскладку, где *каждая* разрешенная пара (i, k) перекрывается (и конечно, не перекрывается никакая исключенная). Более того, эту раскладку можно выбрать так, чтобы обеспечить наименьшую протяженность среди всех допустимых раскладок и использовать наименьшее число копий проб. Следовательно, это будет “самая плотная” допустимая раскладка.

Жадная раскладка клонов

Поместить клон 1 в некоторой левой начальной позиции раскладки.

for k from 2 to n do begin

Найти наименьший индекс $i < k$, такой что пара (i, k) разрешена. {Отметим, что i существует, так как пара $(k - 1, k)$ всегда разрешена.} Поместить клон k так, чтобы он перекрывался с клоном i , не перекрывался с клоном $i - 1$ и его левый конец был справа от левого конца клона $k - 1$ (рис. 16.11).

end;

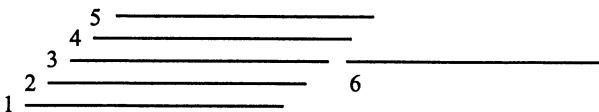


Рис. 16.11. Каждый клон от 2 до 5 образует с клоном 1 разрешенную пару. Клон 6 образует исключенную пару с клонами от 1 до 3, но пара $(4, 6)$ разрешена

Теорема 16.8.1. При жадной раскладке клонов два клона перекрываются в том и только том случае, если они составляют разрешенную пару.

Доказательство. Доказательство ведется индукцией по k . Рассмотрим алгоритм перед началом итерации k . Предположим для каждого $\bar{k} < k$ и каждого $j < \bar{k}$, что клон \bar{k} перекрывает правый конец клона j в том и только том случае, если пара (j, \bar{k}) разрешена. На итерации k , если i — наименьший индекс, для которого пара (i, k) разрешена, то пара $(i - 1, k)$ исключена. Тогда по лемме 16.8.1 пара (i', k) также исключена для любого $i' \leq i - 1$. По построению клон k помещается так, чтобы избежать перекрытия с клоном $i - 1$, и следовательно, он не перекрывается и с любым клоном меньше чем $i - 1$. Поэтому шаг индукции получается для каждой пары (i', k) с $i' \leq i - 1$.

Теперь рассмотрим каждый индекс i' между i и $k - 1$. По лемме 16.8.1 пара (i, i') также разрешена для любого индекса i' между $i + 1$ и $k - 1$. Таким образом (по индукционному предположению), каждый клон i' между $i + 1$ и $k - 1$ расположен так, что перекрывает правый конец клона i . По построению, поскольку клон k расположен так, чтобы перекрывать правый конец клона i , k будет перекрывать также каждый клон i' между i и $k - 1$, и таким образом шаг индукции доказан для всех пар клонов до индекса k . \square

Ниже мы убедимся (располагая пробы), что жадная раскладка клонов может быть сделана допустимой раскладкой, но сначала покажем, что протяженность жадной раскладки клонов может быть сделана меньше, чем у любой другой допустимой раскладки. В жадной раскладке клонов пусть i_2 — наименьший индекс, больший 1 и такой, что пара $(1, i_2)$ исключена. По построению, все клоны строго между 1 и i_2 перекрывают клон 1, и последовательные смещения их левых концов можно сделать произвольно малыми. Поэтому пропуск между правым концом клона 1 и левым концом клона i_2 можно сделать произвольно малым. Повторяя это рассуждение, предположим, что $1 < i_2 < i_3 < \dots < i_r$ — последовательность индексов, где для каждого $l < r$ индекс i_{l+1} — наименьший, больший i_l и такой, что пара (i_l, i_{l+1}) исключена. Тогда в жадной раскладке клонов пропуск между правым концом клона i_l и левым концом клона i_{l+1} можно сделать произвольно малым. Поэтому протяженность жадной раскладки клонов можно сделать сколь угодно близкой к длине r клонов. Теперь каждая последовательная пара клонов в этой последовательности исключена; следовательно, по лемме 16.8.1 каждая пара в последовательности исключена и никакая допустимая раскладка не имеет длины меньше чем r клонов. В итоге:

Теорема 16.8.2. *Протяженность жадной раскладки клонов на вещественной прямой может быть сделана меньше, чем у любой другой допустимой раскладки.*

Теперь при заданной жадной раскладке клонов мы разместим копии проб так, чтобы сделать эту жадную раскладку допустимой раскладкой. При этом количество копий проб окажется наименьшим по всем допустимым раскладкам. Предположим, что есть искусственный клон $n + 1$, который не содержит проб. Каждая проба p будет размещена отдельно по следующему алгоритму.

Жадное размещение проб для пробы p

Пусть i^* — наименьший индекс, для которого p принадлежит P_{i^*} .

repeat

Положить j равным наименьшему индексу, большему, чем i^* , и такому, что либо пара (i^*, j) исключена, либо $p \notin P_j$. (Клоны от i^* до $j - 1$ называются полосой.)

Поместить копию пробы p внутрь интервала для клона i^* в крайнюю правую позицию, в которой интервал для клона j не покрывает этой копии p .

Если $p \in P_k$ для некоторого $k \geq j$, то положить i^* наименьшему такому индексу k .

until нет индексов $k \geq j$, для которых $p \in P_k$;

Теорема 16.8.3. *Жадное расположение проб делает жадную раскладку клонов допустимой. Более того, это расположение использует наименьшее число копий проб по всем допустимым раскладкам.*

Доказательство. Для любой фиксированной пробы p возьмем ее индекс i^* , и пусть $j > i^*$ определено, как в алгоритме. В случае если пара (i^*, j) исключенная, клоны i^* и j не перекрываются, так что копия p помещена в крайний правый конец клона i^* . Поэтому эта копия p покрыта только клоном i^* и клонами, перекрывающими его правый конец. Так как все клоны имеют одну и ту же длину, это в точности те же клоны, которые входят в полосу, определенную индексом i^* . Но по выбору j пробы p находится в P_l для любого l в этой полосе.

В случае если эта пара (i^*, j) разрешена (и значит, $p \notin P_j$), клон j перекроет клон i^* , но не может быть клона $l < i^*$, который перекрыл бы левый конец клона i^* . Клон l не ограничивает расположения p в интервале для i^* , кроме случая, когда p не содержится в P_l . Но если $p \notin P_l$, то пара (l, j) исключена (из-за клона i^*); таким образом, клоны l и j не перекрываются в жадной раскладке. Поэтому при расположении копии p в крайней правой позиции в i^* , но левее j эта копия пробы p будет покрыта только клонами из полосы, определенной i^* .

По алгоритму расположения проб если $p \in P_l$ для некоторого клона l , то l принадлежит одной из полос, определенных алгоритмом. Следовательно, клон l накрывает по меньшей мере одну копию пробы p .

Далее рассмотрим число используемых копий пробы p . Пусть I^* — множество значений, принимавшихся переменной i^* во время работы алгоритма расположения проб. Число копий пробы p , размещенных алгоритмом, равно $|I^*|$. Но при любой допустимой раскладке ни одна копия p не может быть покрыта более чем одним клоном из I^* . Чтобы убедиться в этом, обозначим через i и i' два последовательных индекса из I^* . По построению либо пара (i, i') исключена, либо между i и i' есть такой индекс j , что $p \notin P_j$. В первом случае клоны i и i' не могут перекрываться ни в какой допустимой раскладке. Во втором случае они могут перекрываться, но область перекрытия должна быть покрыта интервалом для клона j , и следовательно, копия пробы p в эту область помещена быть не может. Следовательно, любая допустимая раскладка должна использовать не меньше $|I^*|$ копий пробы p , и теорема доказана. \square

Множество всех допустимых раскладок

Используя рассуждения предыдущего раздела, легко получить следующую теорему.

Теорема 16.8.4. *Пусть перестановка клонов задана. Любая раскладка клонов, у которой порядок слева направо соответствует этой перестановке, может быть преобразована в допустимую раскладку (размещением проб) в том и только том случае, если никакая исключенная пара клонов в этой раскладке не перекрывается.*

Теорема 16.8.4 показывает, что самая плотная допустимая раскладка имеет одно дополнительное желательное свойство: ее можно использовать для получения любой другой допустимой раскладки клонов. Нужно только, начав с плотнейшей раскладки, просто сдвигать клоны в любом желательном направлении, не создавая новых перекрытий и не меняя порядок клонов слева направо. В точке, где два перекрывающихся клона прекращают перекрываться, нужно проверить, не требуется ли новая копия пробы, и если так, то создать ее. Легко представить себе графическую интерактивную систему, позволяющую создавать таким образом допустимые раскладки. Заметим, что при этом не создаются все допустимые раскладки, потому что при любой допустимой раскладке клонов пробы можно располагать по-разному.

Определение. *Прослойкой (interleaving) раскладки клонов называется список пар перекрывающихся клонов этой раскладки. Прослойка называется допустимой, если она получена из допустимой раскладки клонов.*

Множество всех допустимых прослоек можно представить в сжатой графической форме, предложенной в [12]. Оно дает еще одно представление множества всех допустимых раскладок клонов.

16.9. Физическое картирование: последние замечания

Группировка

Следует упомянуть еще одно усложнение. В STS-картировании тесты обычно выполняются не для всех пар STS/клон. Это слишком дорого. На самом деле клоны группируются в *пулы*, и затем каждая STS отдельно тестируется по каждому пулу клонов. Единичный тест покажет, не встретилась ли эта STS по меньшей мере в одном из клонов данного пула, но не определит, в каких именно клонах она присутствует. Таким образом, STS-картирование с пулами клонов очень похоже на радиационно-гибридное картирование, за исключением того обстоятельства, что пулы конструируются свободно, а “пулы” в радиационно-гибридном картировании создаются случайно.

Стратегии группировки либо *статичны*, и в этом случае пулы фиксированы и тестируются единожды, либо они *адаптивны*, и результаты первого раунда группированных тестов определяют, какие пулы попробовать в следующий раз. Однако стоимость создания пулов велика, так что обычно делается не более двух раундов, и даже при двух раундах выбор производится из фиксированного набора возможностей. Ясно, что группировка приводит к задачам комбинаторного планирования экспериментов и статистического вывода, которые выходят за рамки этой книги. Краткое обсуждение группировки появилось в [111].

Физическое картирование обеспечивает исследования

Техника физического картирования, основанная на создании раскладки фрагментов, имеет еще одно важное свойство, которого нет у генетических карт: физическая карта сопровождается упорядоченной библиотекой клонов. Физическая карта — это не просто символическое представление физических позиций. Картированные фрагменты в упорядоченной библиотеке клонов могут использоваться для быстрого определения примерного положения нового куска ДНК, полученного из некоторого другого экспериментального протокола. Например, можно изучать некоторое заболевание и обнаружить, что конкретная иРНК синтезируется в организме здоровых людей и не синтезируется у заболевших. Это означает, что заболевание вызывается отсутствием конкретного белка, который не экспрессируется. Хочется найти положение гена для этого белка, чтобы точно увидеть различия между заболевшими и здоровыми индивидуумами. Имея физическую карту, можно превратить иРНК в кДНК (см. п. 11.8.3), которую затем использовать для гибридизации с фрагментами из упорядоченной библиотеки клонов. Результаты этого эксперимента в комбинации с физической картой фрагментов клонов определят примерное положение гена, кодирующего нужный белок.

Блестящим примером применения физической карты и дополнительной иллюстрацией совместного использования (специализированной) базы данных и лабораторной работы была идентификация гена рано наступающей болезни Альцгеймера [47, 300]. Ген, ответственный примерно за 80 % случаев рано наступающей болезни Альцгеймера (вариант заболевания, поражающий относительно молодых людей), был найден в четырнадцатой хромосоме и клонирован в начале 1995 г. Но еще 20 % случаев оставались необъясненными. Вскоре после этого попытки генетического картирования (сцеплений) дали возможность идентифицировать примерное

положение гена, связанного с оставшимися случаями, в первой хромосоме. Однако, поскольку генетическое картирование в общем локализует ген с точностью до миллиона оснований, для более точного поиска нужно было привлечь физические методы.

Исследователи действовали так. Они рассудили (предположили), что ген первой хромосомы должен быть похож в расшифровке (по крайней мере, локально) на ген болезни Альцгеймера в четырнадцатой хромосоме, уже клонированный и расшифрованный. Им были доступны библиотеки EST (см. п. 3.5.1), и они сравнили библиотечные последовательности с последовательностью гена болезни Альцгеймера из четырнадцатой хромосомы. Одна EST хорошо совпала. Эта EST была длиной в 475 пар оснований при переводе в аминокислотный алфавит, и при выравнивании с геном четырнадцатой хромосомы дала 80 % совпадений. В этот момент они выявили EST, которая могла быть частью искомого гена, но не знали, из какой части генома эта EST (методы нахождения EST обычно не определяют их положения). Но используя выбранную EST как пробу в карте клонов YAC, ученые узнали, что она расположена в той части первой хромосомы, о которой было предварительно установлено (анализом частот сцепления), что она содержит пропущенный ген. После этого, используя праймеры PCR, сделанные из EST (и библиотеки кДНК), они смогли изолировать отдельный фрагмент длиной около двух тысяч пар оснований, содержащий пропущенный ген. Этот фрагмент был расшифрован и (локально) выровнен с геном четырнадцатой хромосомы (с 67 % совпадений на аминокислотном уровне), что идентифицировало отдельный предполагаемый ген фрагмента. Снова используя PCR, они повторно извлекли участок ДНК с предполагаемым геном у членов семьи, которые были частью подверженны рано наступающей болезни Альцгеймера. Сравнивая последовательности для этого предполагаемого гена от больных с последовательностями от здоровых членов семьи, они нашли состоятельное различие между ними, подтверждая тем самым, что ген найден. Затем, как обычно, были проведены дополнительные поиски в базе данных, чтобы найти белки с последовательностями, похожими на белковый продукт найденного гена. Родственные белки, найденные в процессе этого поиска, могут помочь исследователям понять роль обнаруженного белка в этиологии болезни Альцгеймера.

Еще о комбинаторике

Наше рассмотрение физического картирования сосредоточилось на комбинаторных задачах картирования и комбинаторных, теоретико-графовых подходах к их решению. Это контрастирует с большой литературой, которая рассматривает задачи картирования в статистическом ключе. Комбинаторный взгляд на картирование был гораздо полнее, чем в этой главе, рассмотрен, в частности, Роном Шамиром и его соавторами [181, 260] и Ричардом Карпом и соавторами [12, 13, 247, 264]. Теоретико-графовый подход к современным задачам картирования возник в работе Уотермена и Григгса [464], где некоторые из этих задач были представлены как задачи на интервальных графах. Более ранняя работа по использованию интервальных графов в молекулярной биологии (для начальных методов физического картирования) содержится в книге *Графы и гены* [327]. Картированию и расшифровке посвящен специальный выпуск *The Journal of Computational Biology* [367], и многие из статей этого выпуска следуют комбинаторному подходу.

16.10. Введение в выравнивание карт

В п. 7.14.6 мы рассматривали вопрос поиска в строках из большого алфавита и отмечали, что *карты рестриктаз* могут рассматриваться в качестве таких строк. На самом деле карта рестриктаз — это строка, в которой каждый нечетный символ есть целое число (расстояние), а каждый четный символ представляет конкретный тип ограничивающего энзима. В более общем определении любая карта (физическая или генетическая) может рассматриваться как строка, в которой нечетные символы — целые числа, а четные берутся из некоторого конечного алфавита, представляющего собой картируемые свойства.

Мы видели в п. 7.14, что при поиске точных совпадений можно с помощью суффиксных массивов эффективно находить все вхождения меньшей карты в большую. Эта задача имеет большое практическое значение. Но обычно карты содержат ошибки, и нужны способы решения задач приближенного совпадения для карт и способы определения и вычисления редакционного расстояния между двумя картами.

Эта задача не нова, так как карты можно рассматривать как строки, и легко подобрать штрафы, чувствительные к несовпадениям чисел. Но обработка удалений — вопрос более деликатный. Например, карта *) $x\ 125\ y$ хорошо совпадает с внутренней частью $z\ 100\ x\ 126\ y$ и хуже совпадает с $z\ 100\ x\ 60\ y$ или $z\ 100\ x\ 66\ y$. Значит, несовпадение 125 со 126 накладывает маленький штраф, а несовпадение 125 с 60 или 66 — большой. Легко модифицировать методы приближенного совпадения, чтобы отразить в них такие случаи. Но совпадение $x\ 125\ y$ с внутренней частью $z\ 100\ x\ 60\ p\ 66\ y$ может рассматриваться как хорошее, даже при том, что штраф за несовпадение 125 с 60 или 66 велик. Действительно, если удалить символ p , то первая строка будет выравниваться с $x\ 126\ y$ с маленьким штрафом. Каким образом следует моделировать такие удаления при выравнивании карт с помощью динамического программирования?

Прямой подход заключается в переписывании строк (карт) с преобразованием каждого целого в *унарную* запись. То есть каждое число d заменяется d единицами. Тогда строка $x\ 125\ y$ превращается в строку из 127 символов. Эта строка может встретиться в строке $z\ 10\ x\ 60\ p\ 66\ y$ (длины 230) после всего двух редактирующих операций: удаления из длинной строки символа p и удаления одной единицы из записи 60 или 66. При таком переписывании задача поиска приближенного вхождения малой карты в большую превращается в задачу приближенного совпадения строк, и редакционное расстояние между двумя картами превращается в редакционное расстояние между двумя переписанными строками. Кажется, что идея унарного переписывания решает задачу, но ее недостаток в том, что она создает длинные строки. И поскольку вычисление редакционного расстояния с помощью ДП требует времени, пропорционального произведению длин строк, для больших карт этот подход не эффективен.

Для таких задач выравнивания карт или редакционного расстояния были развиты альтернативные — более эффективные — методы решения. Наиболее эффективный (по оценке наихудшего случая) метод считает за время $O(nm \log nm)$ [338], где n и m — соответственно, числа символов в картах.

*) По-видимому, нумерация позиций в карте рестриктаз начинается с нуля. — Прим. перев.

16.10.1. Выравнивание карт с помощью неунарного динамического программирования

В этом пункте мы опишем ранний, менее сложный подход ДП, предложенный Уотерменом, Смитом и Кетчером [465].

Определение. Рассмотрим две карты в задаче выравнивания. Пусть первая из них представлена последовательностью пар (s_i, d_i) , где s_i — символ, представляющий особенность i -го места карты, а d_i обозначает расстояние этого места от левого края карты. Пусть вторая карта представлена аналогичной последовательностью пар (s'_j, d'_j) .

Определение. Примитивные операции, которые можно использовать для преобразования одной карты в другую, — это включение или удаление особенностей, а также изменение расстояния между местами в любой из карт (т. е. изменение чисел d_i или d'_j).

Заметим, что подстановка одной особенности на место другой не допускается. Сначала может показаться, что это серьезное ограничение, но это нерезонно. При получении генетических или физических карт место может быть пропущено, или неправильно вставлено, или смещено с его правильной позиции, но очень неправдоподобно, чтобы перепутали тип особенности. В действительности во многих картах положение особенностей каждого типа находится отдельно, а потом эти карты комбинируются. Каждая отдельная карта может пропускать места, смещать их или некорректно вводить, но не может поставить место другого типа. Расстояния между местами обычно достаточно велики, так что непохоже, чтобы получающаяся комбинированная карта могла сменить особенность одного типа на другую.

Целью является перевод одной карты в другую с наименьшей ценой, где цена вставки или удаления особенности задается параметром λ , а цена изменения расстояния на величину t равна βt .

Решение с помощью динамического программирования

Пусть две карты имеют n и m мест соответственно, где $n > m$. Прежде всего добавим два фиктивных места с номером 0 слева от каждой карты и два фиктивных места с номерами $n + 1$ и $m + 1$ соответственно справа от каждой карты. Более того, предположим, что $s_0 = s'_0$ и $s_{n+1} = s'_{m+1}$.

Определение. Пусть $MD(i, j)$ — минимальная стоимость преобразования первой карты до места i во вторую карту до места j при требовании, что место i в первой карте выравнивается с местом j во второй карте.

Требование в определении $MD(i, j)$, чтобы места i и j выравнивались, меняет смысл $MD(i, j)$ по сравнению со смыслом $D(i, j)$, использованного при вычислении редакционного расстояния. Цель — вычислить $MD(n + 1, m + 1)$. Мы используем следующие рекуррентные соотношения:

$$MD(0, 0) = 0,$$

$$MD(0, j) = MD(n + 1, j) = MD(i, 0) = MD(i, m + 1) = \infty,$$

если $0 < j < m + 1$ или $0 < i < n + 1$. Это налагает ограничение, что два нулевых места должны быть выровнены и места $n + 1$ и $m + 1$ тоже. Когда $s_i \neq s'_j$,

$$MD(i, j) = \infty,$$

благодаря предположению, что особенность типа s_i не может переводиться в особенность типа s'_j . Когда $s_i = s'_j$, имеем

$$MD(i, j) = \min_{\substack{0 < k \leqslant i, \\ 0 < l \leqslant j}} [MD(i - k, j - l) + \lambda(l + k - 2) + \beta(|(d_i - d_{i-k}) - (d'_j - d'_{j-l})|)].$$

Чтобы лучше понять последнее отношение, напомним, что место i выровнено с местом j , так что k и l указывают количество мест, на которое нужно отойти назад, соответственно в первой и во второй карте, чтобы получить последнюю выровненную пару мест перед i и j . Конкретно, из первой карты удаляется $k - 1$ место перед местом i и из второй карты — $l - 1$ перед местом j . Просмотром всех значений k и l будет найдено наилучшее выравнивание i и j . Так как пара $(0, 0)$ включена в обработку случая, когда i и j — первые места для выравнивания, это соотношение найдет наилучшее выравнивание для мест i и j .

Общее время на вычисление $MD(n + 1, m + 1)$ равно $O(n^2m^2)$. Так как расстояние между местами велико (заведомо больше, чем среднее n для карты с n местами), эта оценка времени лучше, чем оценка, получаемая при переводе строк в унарную форму. Как уже отмечалось, более быстрый, но более сложный подход приводит к временной границе $O(nm \log nm)$ [338].

16.10.2. Обобщение модели выравнивания карт

Выше мы утверждали, что спутать один тип рестриктазы с другим непросто, и следовательно, подстановки в методе выравнивания не поддерживаются. Однако более реалистичная модель карт рестриктаз должна учитывать некоторые экспериментальные ошибки, не обрабатываемые приведенной моделью. Например, такая экспериментальная ошибка — сообщение об одном месте с рестриктазой в случае, когда есть два близких места с одной и той же рестриктазой. Другая экспериментальная ошибка — перестановка порядка двух близких мест для разных рестриктаз. В статье [323] Миллер, Остелл и Радд бьются над вопросом перестановки при несколько иной схеме оценки. Также в зависимости от того, как экспериментально получены расстояния на карте, может быть, более реалистично было бы позволить d_i быть расстоянием между местами i и $i - 1$, а не расстоянием до места i от левого конца.

16.11. Крупномасштабная расшифровка и сборка последовательности

Некоторые аспекты крупномасштабной (на геномном уровне) расшифровки ДНК уже появлялись в этой книге в различных обсуждениях конкретных строковых задач и задач картирования. Разумно собрать все эти разрозненные факты вместе для более содержательного обсуждения крупномасштабного секвенирования и его отношения

к физическому картированию. Это полезно не только ввиду важности крупномасштабной расшифровки, но и как способ подчеркнуть роль различных строковых алгоритмов.

Сегодня главная проблема в расшифровке большой строки ДНК заключается в том, что ДНК в конечном счете расшифровывается методом Максама–Гилберта или Сангера [111, 469], где в отдельном “прочтении” можно расшифровать только малый фрагмент ДНК. То есть ведется ли расшифровка полностью автоматизировано или с применением ручной обработки, наибольшая сплошная подстрока ДНК, которую можно надежно определить в отдельной лабораторной процедуре, имеет длину от 300 до 1000 оснований. Для простоты изложения мы будем использовать цифру в 400 оснований как стандарт длины чтения, хотя люди с “хорошими руками” и более новые машины претендуют на то, что способны надежно расшифровать существенно более длинные участки. В процедуре (в качестве исходных данных для машины или процесса) может использоваться и более длинная строка, но только начальные 400 оснований будут определены. Следовательно, для секвенирования длинных строк или целого генома нужно разделить ДНК на много коротких строк, которые индивидуально расшифровываются, а затем используются для сборки последовательности полной строки. Критическое различие между разными методами крупномасштабной расшифровки заключается в том, как задача секвенирования полной ДНК делится на разрешимые подзадачи так, чтобы исходную последовательность можно было бы собрать из последовательностей длины 400.

16.12. Направленная расшифровка

Наиболее естественный метод, *направленной расшифровки*, был введен в упражнениях 61 и 63 главы 7. Здесь мы рассмотрим два варианта направленной расшифровки: *продвижение праймера* и *вложенное удаление* (nested deletion).

При продвижении праймера (см. также обсуждение хождения по хромосоме на с. 224) сначала расшифровывают левые 400 оснований большой строки. Знание этой подстроки позволяет сделать праймеры PCR, которые используют для создания копии части исходной строки, начинающейся около правого конца расшифрованных 400 оснований. В этой точке есть подстрока исходной строки, начинающаяся около основания 400. Теперь расшифровывают самые левые 400 оснований этой подстроки, и процедуру можно повторить.

При вложенном удалении для удаления оснований с левого конца строки используют фермент (экзонуклеазу). После расшифровки первых 400 оснований экзонуклеаза удаляет расшифрованную часть (и хорошо, если только ее), так что процедура может повториться. К сожалению, использование экзонуклеазы в настоящее время не так точно, как хотелось бы, так что часто удаляется слишком много или слишком мало.

Главная проблема направленной расшифровки состоит в том, что она изначально последовательная и потому медленная. Последовательную природу метода можно преодолеть, если будут известны несколько разбросанных точек ДНК, откуда направленную расшифровку можно запустить параллельно. Недавнее предложение

(см. п. 16.13.1) может сделать такую прямую параллельную расшифровку возможной. Читатель может удивиться очевидности подхода с разрезанием длинной строки ДНК на меньшие фрагменты, которые можно прямо расшифровывать. К сожалению, процесс разрезания переставляет исходный порядок фрагментов, делая использование такой стратегии в чистом виде невозможной. Один из подходов к разрешению проблемы перестановки привел к дробовой расшифровке, главной альтернативе направленной. Мы рассмотрим дробовую расшифровку в п. 16.14.

Вторая критическая проблема направленной расшифровки заключается в том, что по одной из причин она застrevает в какой-то точке строки, и продолжение строки за этой точкой не может расшифровываться, пока не будет найдена новая начальная точка. Направленная расшифровка часто застrevает или запутывается повторяющимися подстроками, а ДНК высших организмов (в частности, млекопитающих) относятся к высоко повторяемым (см. п. 7.11.1). Если выбранный праймер находится в повторяющейся подстроке, он может гибридизоваться в нежелательной части полной строки, и в этом случае PCR будет копировать не ту часть ДНК, которая нужна. Комплементарный палиндром (см. п. 7.11.1) вызывает трудности и для вложенного удаления, и для продвижения праймера, так как два комплементарных сегмента составляют петлю (шпильку для волос), которая блокирует работу методов расшифровки Максама–Гилберта и Сангера. Если говорить на жаргоне секвенторов (тех, кто работает в этой области), направленное секвенирование не допускает “сквозного прочтения” повторяющейся ДНК. Следовательно, направленная расшифровка обычно используется для относительно малых строк, которые не содержат повторов. Направленная расшифровка применяется также, чтобы “закрыть” небольшие пропуски, остающиеся в последовательности, полученной дробовой расшифровкой. Мы вернемся к этому вопросу в п. 16.13.1.

16.13. Нисходяще-восходящая расшифровка: картина, использующая YAC

Ввиду того что с помощью направленной расшифровки не всегда можно расшифровать длинные строки ДНК, имеющие размер генома, были изобретены более иерархические или рекурсивные подходы. Хотя конкретные детали могут сильно меняться, сегодня большинство попыток крупномасштабного секвенирования следует общей базовой линии, начиная работу с *физического картирования* больших клонов и кончая дробовой расшифровкой существенно меньших клонов (обычно *космидных*). Этот подход подытожен Мейнардом Олсоном.

“После периода соревнования нескольких альтернативных технологий секвенирования выявились доминантная технология крупномасштабной геномной расшифровки: клоны размера космida или больше анализируются случайной выборкой (т. е. «дробовой» расшифровкой), реализованной на коммерческих... инструментах расшифровки. Оптимальный размер начальных клонов, уровень детализации, с которым эти клоны нужно картировать, и степень, в которой случайная выборка должна дополняться более «направленными» методами, остаются дискуссионными. Однако важная новость заключается в том, что основной подход работает в любой из нескольких испытанных вариаций” [352].

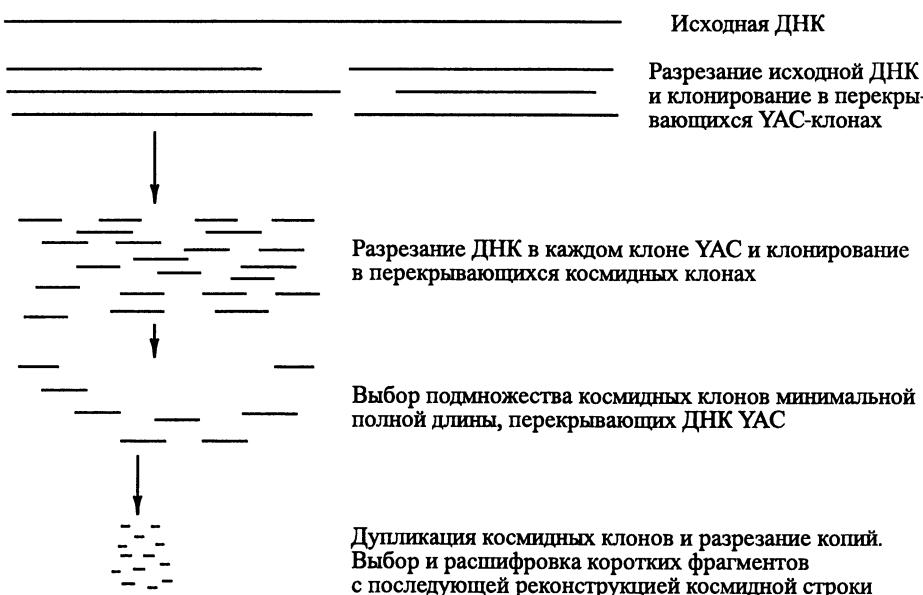


Рис. 16.12. Схема гипотетической крупномасштабной расшифровки. Нисходящая часть строит карту YAC для расшифровываемой ДНК и затем карту космид для каждой YAC в покрывающем наборе YAC. Далее выбирается для расшифровки подмножество космидных клонов и для каждого идет дробовая (восходящая) расшифровка. После того как космиды будут расшифрованы, они используются вместе с космидными картами для назначения символов YAC и, следовательно, полной ДНК. Возможное развитие этой картины включает выполнение радиационно-гибридного картирования фрагментов ДНК, больших, чем включения YAC, и картирование с помощью рестриктазы для карт, более точных, чем космидные до начала расшифровки. Рисунок не передает масштаба

Соответственно, в нашем обсуждении крупномасштабной расшифровки мы упомянем некоторые из используемых сегодня конкретных клонирующих векторов, но не будем описывать какой-либо реальный проект во всех подробностях. Это было бы слишком сложно и затемнило бы описание выявившейся “доминантной стратегии”. На рис. 16.12 показан общий вид процесса.

Наш гипотетический крупномасштабный проект, расшифровывающий, скажем, 100 000 000 пар оснований, стартовал бы с создания библиотеки достаточно больших клонов, в диапазоне от 100 000 до 1 000 000 пар оснований каждый. Эти клоны можно получить из многих копий целевой ДНК (которую нужно секвенировать), разрезая ее достаточно редко с помощью рестриктазы. Каждый клон в библиотеке поддерживается включением его в подходящий *клонирующий вектор* и включением этого вектора в некоторую биологическую систему, которая дуплицирует ДНК этого вектора вместе с ее собственной ДНК. Например, *искусственная хромосома дрожжей* (YAC) является клонирующим вектором, который может разместить включения ДНК длиной до миллиона оснований и который может быть включен и повторен в клетках дрожжей [329, 335]. YAC сделана из хромосомы *Saccharomyces cerevisiae* (обычных пивных дрожжей) экстрактацией всех компонентов хромосомы,

которые необходимы для воспроизведения. Когда эти компоненты собираются (в правильном порядке и пространственном разделении) вместе с недрожевой ДНК подходящей длины, получающуюся искусственную хромосому можно включить в нормальную дрожжевую клетку и воспроизводить наряду с нормальной дрожжевой ДНК. Таким способом каждый фрагмент библиотеки сохраняется и воспроизводится в YAC.

На следующем этапе в библиотеке выбирается достаточно большое число YAC, чтобы выбранные клоны YAC покрывали изучаемую ДНК. Затем от этих клонов получаются дактилограммы или STS-данные, и информация используется для построения физической карты фрагментов.

Когда физическая карта YAC будет закончена, процедура работает с каждым из картированных клонов YAC отдельно и создает более чистую физическую карту внутри этого сегмента ДНК. Целевая ДНК из каждого картированного клона YAC реплицируется и затем разрезается рестриктазами более часто, создавая фрагменты размера, скажем, 40 000 пар оснований. Фрагмент такого размера обычно включается в искусственный клонирующий вектор, называемый *космидным*, который поглощается вирусом (*фагом*), инфицирующим бактерии *E. coli*. Эти фаги вводят ДНК, которые они содержат, в клетки *E. coli*, которые затем реплицируют эту ДНК наряду со своей собственной. Случайно выбирается подмножество этих космидных клонов с достаточной плотностью, чтобы они полностью покрыли YAC. Выбранные космидные клоны картируются уже рассмотренными методами физического картирования.

Разрезанию YAC есть альтернатива. Зачастую существующая библиотека содержит неупорядоченный набор космид, покрывающих геном. Можно выделить их подмножество, покрывающее нужную YAC, гибридизацией этой полной YAC с фрагментами космидной библиотеки. Космиды, которые гибридизуются с YAC, в ней содержатся. Затем их используют для построения физической карты космид, покрывающих YAC.

Другие уровни

Хотя нисходящее физическое картирование не всегда является частью стратегии расшифровки, оно может стартовать выше уровня YAC (например, с радиационно-гибридной карты) и продолжаться ниже космидного уровня (например, до рестрикционной карты с частыми сайтами рестрикции или до генетической карты). За счет выбора часто режущей рестриктазы таким способом можно получить физическую карту высокой точности. Иметь физическую карту маркеров, которые используются также в генетическом картировании (например, микросателлитов), очень полезно, так как это позволяет объединять генетическую и физическую карты. Особенность, которая нашла свое место на генетической карте, может быть затем примерно обозначена и на физической карте.

YAC, или PAC, или BAC

Несмотря на то что это не влияет на общую логику нисходящего физического картирования, следует отметить существование других клонирующих векторов, популярность которых растет [329]. Это PAC и BAC — искусственные хромосомы, сделанные из плазмиды P1 в случае PAC и из *бактериальной ДНК* хромосомы в случае BAC. Размер включений в них меньше, чем у YAC, — примерно от 100 000

до 300 000 оснований вместо миллиона. Однако с ними легче работать, и, кажется, не возникает проблем химеризма и пропаданий, которые обычны для YAC. Более того, их меньший размер иногда становится преимуществом. Например, при попытке найти клон, который покрывает пропуск в карте, ищут клон, гибридизующийся с обоими концами этого пропуска. Заполняющую YAC найти, может быть, и легче, но размер пропуска определяется более точно нахождением меньшей BAC или PAC, покрывающей этот пропуск (примерно раз в десять).

16.13.1. Необходимо ли картирование для расшифровки?

В соответствии с доминантной стратегией крупномасштабной расшифровки обычно начинают с фазы картирования, однако полная последовательность ДНК для двух бактериальных геномов и одной археобактерии были получены [82, 162] с использованием подхода, который пропускал стадию картирования и приступал прямо к дробовой расшифровке. Наибольший из этих двух полных геномов был близок по длине к двум миллионам пар оснований. Успех этого подхода удивил многих [350], но еще не ясно, насколько он распространяется на фрагменты большего размера и на небактериальные ДНК, содержащие больше некодирующих областей и повторов.

С другой стороны, успех “бескартной” расшифровки ДНК бактерий размером в мегаоснования и развитие техники BAC и PAC привели к недавнему предложению миновать ступень картирования в проекте человеческого генома. Предлагаемый “метод расшифровки BAC–PAC” Вентера, Смита и Худа [442] работает следующим образом. Сначала создается библиотека человеческого генома BAC (или PAC), состоящая из около 300 000 клонов, каждый со вставкой человеческой ДНК длиной около 150 килооснований. Таким образом, библиотека обеспечивает 15-кратное покрытие генома и содержит BAC, начинающиеся примерно через каждые 5 килооснований. Следующий шаг заключается в расшифровке около 500 оснований с обоих концов каждой клонированной последовательности и в записи в компьютер каждой из этих 600 000 последовательностей (вместе с указанием на колонию BAC, из которой она получена). Каждый расшифрованный конец называется *соединением, маркованным последовательностью* (*sequence tagged connector* — STC).

Чтобы полностью расшифровать длинную область человеческой ДНК, выделяют клон BAC, человеческая вставка которого находится в нужной области (возможно, гибридизацией этой области к клонам библиотеки BAC), и расшифровывают вставку в 150 килооснований дробовой расшифровкой. Затем с помощью компьютера проверяют 600 000 STC, чтобы найти (скажем) самое правое STC, содержащееся в первой расшифрованной BAC. Это STC определяет другую BAC, которая содержит перекрытия и расширяет первую последовательность. Расшифровывая вставку второй BAC и продолжая тем же самым способом, можно расшифровать длинный сплошной интервал ДНК. Этот подход был развит Дж. Вебером и Майерсом [470a] с критикой Фила Грина [190a], которые намеревались провести дробовую расшифровку всего человеческого генома, минуя расшифровку клон-за-клоном. Это предложение было исправлено и улучшено для генома дрозофилы в Celera Genomics corp., возглавляемой Крейгом Вентером, с поддержкой в \$200 миллионов от Perkin-Elmer см. [442a]).

16.13.2. Выбор фрагмента для расшифровки

Вернемся теперь к рассмотрению расшифровки на основе физического картирования. Когда получена физическая карта перекрывающихся космидных клонов (для отдельной YAC), выбирается подмножество этих клонов для полной расшифровки ДНК. Если один клон имеет длину L , то на расшифровку клона потребуется примерно $L/400$ отдельных процедур расшифровки. Следовательно, стоимость расшифровки космидного клона (и по времени, и по деньгам) примерно пропорциональна его длине. Это приводит к небольшой вычислительной задаче выбора подмножества космидных клонов минимальной общей длины, такого, чтобы эти клоны покрывали клон YAC, из которого они получены. Такое подмножество обычно называется *путем минимальной облицовки* (*minimum tiling path*).

Один из способов эффективного поиска пути минимальной облицовки — сведение к задаче о *кратчайшем пути* в ориентированном графе. Каждый клон из космидной карты представляется вершиной, а дуга идет из вершины i в вершину j в том и только том случае, если космидные клоны i и j перекрываются; клон i начинается слева от начала клона j и кончается слева от конца клона j . Отметим, что если один клон полностью содержит другой, то дуги между ними нет. Длина дуги из i в j полагается равной длине перекрытия клонов i и j (т.е. числу общих нуклеотидов). Любой клон, который начинается в крайнем левом конце космидной карты, называется *источником*, а любой клон, кончающийся в крайней правой точке карты, называется *стоком*. Задав так взвешенный ориентированный (ациклический) граф, мы получаем следующую теорему.

Теорема 16.13.1. *Вершины любого кратчайшего пути в графе от источника до стока образуют набор клонов с наименьшей суммарной длиной, покрывающих карту. Этот набор является путем минимальной облицовки.*

Доказательство непосредственно и оставляется читателю.

Так как граф ациклический, кратчайший путь можно найти за время, линейно зависящее от числа дуг, что меньше времени, необходимого для поиска кратчайшего пути в произвольном графе.

16.13.3. Некоторые реальные цифры

Чтобы получить более реалистичную картину крупномасштабного картирования, рассмотрим положение на январь 1996 г. проекта картирования шестнадцатой хромосомы человека, выполняемого группой в Лос-аламосской национальной лаборатории [125]. Шестнадцатая хромосома состоит из примерно 100 миллионов пар оснований, около 3 % человеческого генома и содержит около 3 000 генов. Приблизительно 260 из них уже локализованы. Примерно тысяча клонов YAC и мегаYAC картированы для шестнадцатой хромосомы. Эти YAC покрывают около 99 % хромосомы, и в карте YAC осталось только несколько пропусков. Космидная карта была получена из двух библиотек клонов, содержащих вместе порядка 40 000 космидных клонов. Из них около 4 000 клонов были выбраны случайно (с некоторой фильтрацией клонов с большим содержанием G и C), обеспечив десятикратное покрытие шестнадцатой хромосомы. Было картировано около 1800 космид для хромосомы, которые были организованы в приблизительно 300 контигов, покрывающих более 80 % хромосомы. К указанному времени около 300 STS, 300 дополнительных маркеров (EST, гены, кДНК), а также 1700 мест рестрикции картированы на шестнадцатой хромосоме.

16.14. Дробовая расшифровка ДНК

Найдя множество космидных клонов, формирующих путь минимальной облицовки, процедура обращается к расшифровке ДНК. Она обычно выполняется *дробовой*, или *случайной*, расшифровкой, описываемой следующим образом. Сначала создается много копий нужной ДНК (содержащейся в космиде нашего гипотетического проекта), а затем эти копии разрезаются в случайных местах физическими, ферментативными или химическими средствами. Разрезание создает фрагменты исходной ДНК, но свойства процесса таковы, что исходный порядок теряется — получается просто набор фрагментов без информации о том, из какого места полной строки произошел каждый из них. Кроме того, фрагменты из разных копий перекрывают друг друга, так что, если расшифровано достаточное количество фрагментов, общие образцы из перекрывающихся частей фрагментов можно использовать, чтобы попытаться собрать последовательность полной строки ДНК.

На самом деле будет расшифровано совсем немного космидных фрагментов, так как космиды дуплицируются много раз до того, как эти копии были разрезаны, и большинство процедур расшифровывает только фрагменты в ограниченном диапазоне размеров. (Этот диапазон может превосходить 400, и часто превосходит, но только первые 400 оснований можно прочесть одновременно.) Итак, вместо расшифровки всех фрагментов дробовая процедура случайно выбирает из космидов фрагменты подходящей длины и затем расшифровывает первые 400 оснований каждого из них. Число фрагментов должно быть достаточным, чтобы позволить восстановить полную космидную строку по перекрывающимся расшифрованным частям фрагментов.

Сколько фрагментов нужно отобрать, чтобы удовлетворить требование достаточного перекрытия? Был проведен вероятностный анализ [292], результаты которого вполне соответствуют кустарным эмпирическим правилам. Все сходится на том, что необходимо 5–10-кратное покрытие. Значит, если полная космидная строка имеет размер L и длина читаемого участка 400, то число отбираемых фрагментов должно быть между $5L/400$ и $10L/400$. Префиксы длины 400 такого числа случайно выбранных фрагментов должны перекрываться достаточно, чтобы допускать реконструкцию полной космидной строки. На практике создание, клонирование и выбор фрагментов не всегда достаточно равномерны, чтобы полностью восстановить космидную последовательность, но такое отклонение от случайного покрытия происходит по молекулярным и биологическим причинам, а не по вероятностным. Следовательно, добавление дополнительного покрытия не всегда решает эту проблему, и в случае если дробового покрытия недостаточно, нужно использовать другие подходы, такие как направленная расшифровка.

16.15. Сборка последовательности

После отбора и расшифровки подмножества частично перекрывающихся строк (первые 400 оснований каждого выбранного фрагмента) задачей является построение расшифровки полной космидной строки. Это алгоритмическая задача дробовой *сборки последовательности*. При общем подходе, который мы опишем, далее не важно,

что каждая строка длины 400 является частью более длинного выбранного фрагмента^{*)}, и поэтому для простоты предполагаем, что фрагменты имеют длину 400.

Различные методы сборки последовательности имеют в разных реализациях специфические детали, но почти все они следуют общей схеме, впервые formalizedованной в [362, 363]. Схема содержит три шага: *поиск перекрытий, раскладка фрагментов и поиск консенсуса* (которому часто помогает множественное выравнивание). Первый и последний из этих шагов соответствуют очевидным строковым задачам.

16.15.1. Шаг первый: поиск перекрытий

Первый шаг в схеме сборки последовательности — это определение для каждой упорядоченной пары расшифровываемых строк того, насколько суффикс первой строки “совпадает” с префиксом второй. Как formalizedовать это суффиксно-префиксное совпадение? Если бы не было ошибок расшифровки, то для каждой упорядоченной пары строк S_1, S_2 мы бы вычислили самый длинный суффикс S_1 , который точно совпадает с префиксом S_2 . (В п. 7.10 был предложен алгоритм с линейным временем для нахождения суффиксно-префиксных совпадений во всех упорядоченных парах из набора строк.) Но ошибки расшифровки — это реальность (даже если их только 1–5%), и суффиксно-префиксное совпадение должно допускать приближенные совпадения. Поэтому большинство реализаций определяет суффиксно-префиксное совпадение, решая для каждой упорядоченной пары строк S_1, S_2 что-то вроде следующей задачи:

Найти суффикс S_1 и префикс S_2 , сходство которых максимально среди всех суффиксно-префиксных пар S_1 и S_2 .

Здесь *сходство* определено точно так же, как в п. 11.6, вклад совпадений в целевую функцию положителен, а несовпадений и пробелов — отрицателен. Как всегда, остается вопрос подбора наиболее полезных весов и штрафов, который мы оставим в стороне, но алгоритмический аспект нахождения наиболее похожих суффиксно-префиксных пар из S_1 и S_2 должен быть ясен.

Чтобы найти наиболее похожее суффиксно-префиксное совпадение, используем стандартные рекуррентные соотношения динамического программирования для вычисления сходства S_1 и S_2 . Однако в таблице ДП нужно инициализировать все элементы в нулевом столбце нулями (предполагая, что строка S_1 выписана в таблице по вертикали) и получить максимальное значение в строчке $n = |S_1|$. Это значение и дает максимальное сходство по всем суффиксно-префиксным парам S_1 и S_2 . Если максимальное значение строчки n находится в столбце j и обратный ход в таблице ДП из этой ячейки приводит к строчке i в столбце 1, то наилучшая суффиксно-префиксная пара состоит из суффикса $S_1[i..n]$ и префикса $S_2[1..j]$.

Если длины S_1 и S_2 порядка n , то наилучшее суффиксно-префиксное совпадение ищется за время $\Theta(n^2)$. Таким образом, если есть k фрагментов, то время на вычисление всех $k(k - 1)$ наилучших суффиксно-префиксных совпадений равно $\Theta(k^2 n^2)$. См. детали в п. 11.6.4.

Эвристическое ускорение

Вычисление всех наилучших суффиксно-префиксных совпадений (как эта задача сформулирована выше) является на самом деле узким местом вычислений при

^{*)} В некоторых недавних подходах это становится существенно. Мы рассмотрим один из них в упражнении 3.

сборке последовательностей для тех размеров задач, которые сегодня интересны. В некоторых усилиях по расшифровке на это уходит большая часть времени счета всего проекта. Но целью вычисления наилучших суффиксно-префиксных совпадений является нахождение упорядоченных пар с *большим* сходством, определяющих кандидатов на перекрытие — т. е. фрагментов, которые станут соседями. Эти возможные пары используются на втором шаге сборки для раскладки фрагментов. Следовательно, можно ускорить первый шаг сборки, если опознать пары строк, наибольшее суффиксно-префиксное сходство которых, очевидно, “недостаточно”, чтобы пара была привлекательным кандидатом^{*)}. Вычисления ДП для этих непривлекательных пар можно будет пропустить.

Каким способом можно эффективно распознать непривлекательные пары? Нетрудно придумать множество подходов, но для их сравнения нужно немного поэкспериментировать. Упомянем только одну идею, которая имеет несколько вариантов. Так как доля ошибок довольно мала (ниже 5%), а покрытие довольно высоко, две строки, обладающие “достаточным” перекрытием, будут иметь по крайней мере одну “существенную” длинную общую подстроку (см. п. 7.4). Какая длина “существенна”, снова может быть определено вероятностными рассмотрениями, лежащими вне интересов нашего обсуждения. Важно, что, вычисляя длину наибольшей общей подстроки для каждой пары выбранных строк, можно распознать и исключить многие пары строк, которые непохожи на перекрывающихся соседей в полной строке. Напомним упражнение 8 главы 7, где говорилось, что для k строк длины n каждая наибольшая общая подстрока каждой из C_k^2 пар может быть вычислена за полное время $O(k^2n)$ использованием обобщенного суффиксного дерева. Можно еще улучшить этот подход, рассматривая только общие подстроки, находящиеся в первой четверти (скажем) одной из двух строк. Этого естественно ожидать от двух соседних перекрывающихся строк, а число пар, для которых нужно будет выполнять полный алгоритм ДП, будет еще меньше. Чен и Скиена [100] провели глубокую проверку методов точного совпадения для поиска перекрытий. Они сообщают о большом ускорении в сравнении с ДП при совсем небольшой потере качества (см. п. 7.15).

Обобщение подхода точных совпадений использовало бы что-нибудь вроде BLAST для нахождения областей высокого локального сходства пар строк. Идея снова в том, что можно ожидать, что соседние строки имеют по несколько областей “приличного размера” с высоким локальным сходством. Каким бы ни был конкретный метод, высматривая неподходящих соседей, можно ограничить вычисления по ДП меньшим числом привлекательных пар строк. Техника такого типа решительным образом уменьшает полное время вычислений, необходимое для сборки последовательности.

16.15.2. Шаг второй: раскладка подстрок

Есть разные подходы к раскладке. Некоторые из них алгоритмически очень хитры, но большинство практических реализаций следует какому-либо из вариантов *жадного* алгоритма, сходного с методом, который раскладывает фрагменты в проектах физического картирования. На самом деле, можно взглянуть на последовательность ДНК подстроки как на очень точную *упорядоченную дактилограмму* и следовать манере рассуждений, применявшейся при раскладке физических карт.

^{*)} Достаточность зависит от покрытия, от длины чтения и от типа и распределения ошибок. Этот род анализа носит обычно вероятностный характер и находится вне сферы внимания этой книги.

Если говорить подробнее, пара строк с наибольшей оценкой суффиксно-префиксного совпадения выбирается первой и сливается вместе, с фиксацией расширения ее перекрытия. Затем выбирается и сливается вместе следующая пара, от чего получается либо один контиг из трех строк, либо два отдельных контига, содержащих четыре строки (рис. 16.13). Так как суффиксно-префиксные перекрытия определены по критериям сходства, в суффиксно-префиксных совпадениях разрешены пробелы. Поэтому по мере того, как последовательные космиды добавляются к контигу, в добавляемую строку могут включаться дополнительные пробелы, чтобы она была совместна с пробелами в предварительно включенных строках (рис. 16.14). Это в точности та же ситуация, которая встречается в множественном выравнивании, и в результате здесь конструируется высокоструктурированное множественное выравнивание.

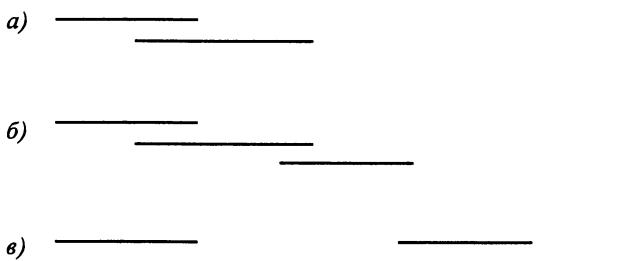


Рис. 16.13. а — первый контиг; б, в — две разные возможности после слияния следующей лучшей суффиксно-префиксной пары

a) ATC - GACTTA
CTGACTTACCCG

б) AAGGTAATC - T
ATC - GACTTA
CTGACTTACCCG

Рис. 16.14. а — первый контиг. Выравнивание пары строк содержит один добавленный пробел. б — следующее по качеству суффиксно-префиксное совпадение сопоставляет суффикс ATCT с префиксом ATCG. Это парное выравнивание не содержит пробелов, но пробел включается в третью строку, когда она добавляется к этому контигу. Новый пробел наследуется от выравнивания первых двух строк

Отметим, что здесь не вычисляется наилучшее суффиксно-префиксное перекрытие фиксированного контига с одним из фрагментов или с другим контигом. Оно могло бы улучшить множественное выравнивание, но было бы дорого в вычислительном отношении и на практике обычно не делается. Вместо этого все решения по раскладке основываются на парных суффиксно-префиксных оценках, вычисленных на шаге один, и любые несовместности, порожденные раскладкой, оставляются для

разрешения на шаге три. Следовательно, шаг два не включает в себя дополнительных вычислений ДП и на практике выполняется очень быстро.

16.15.3. Шаг третий: достижение консенсуса

Раскладка подстрок, созданная на шаге два, используется для создания единой *выведенной космидной* строки ДНК. То есть основания в космиде определяются на этом шаге. В разных методах это делается по-разному, но везде используется один принцип. Раскладка подстрок (неявно) сопоставляет каждый символ каждой расшифровываемой подстроки конкретной космидной позиции (рис. 16.15). Если все символы, сопоставленные конкретной позиции, одинаковы, то проблем не возникает. Но если есть расхождение, то нужно выбрать один конкретный символ или отметить, что расхождение слишком велико для определения основания в указанной позиции. Мы опишем для иллюстрации несколько подходов, но предостережем читателя, что каждая существующая система решает этот вопрос по-своему.

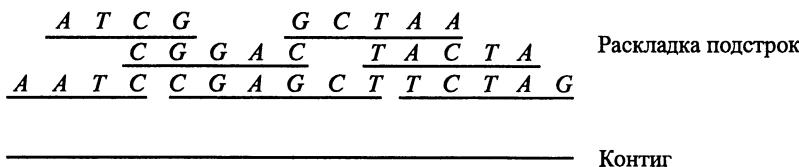


Рис. 16.15. Раскладка расшифрованных строк, созданная на втором шаге, сопоставляет каждый символ единственной космидной позиции. Следовательно, раскладка вводит столбцовую структуру символов в расшифрованных строках. На рисунке в выравниваниях нет пробелов, хотя, вообще, пробелы допустимы

Простейший подход — сообщить частоту каждого символа в каждом столбце (профиль) и предоставить пользователю решать, как использовать эту информацию. Альтернативой является *консенсусная* последовательность, которую можно скомпилировать, приняв в каждом столбце символ *большинства*, при условии, что это большинство выше некоторого установленного заранее порога. С другой стороны, можно определить *окна* раскладки подстрок, где рассогласование велико. Для каждого такого окна подстроки, в нем содержащиеся, можно множественно выровнять каким-либо методом и затем вывести консенсусную строку множественного выравнивания. (Напомним дискуссию в п. 14.7 о множественном выравнивании с консенсусной целевой функцией.) Множественное выравнивание в окне с высоким рассогласованием позволит изменить некоторые символы, а другие немного сдвинуть. Более строгий подход к подгонке раскладки и определению оснований содержится в [273, 269, 267].

Промежуточный метод заключается в *повторном слиянии* подстрок раскладки в соответствии с той же историей слияний, определенной в шаге раскладки, но теперь каждая новая подстрока вливается в *профиль* (см. первую часть п. 14.3.1) подстрок, уже находящихся в контиге (рис. 16.16). Этот тип повторного слияния не так обременителен вычислительно, как полное множественное выстраивание, и требует не больше $n - 1$ профильных выстраиваний, где n — число подстрок. Тем не менее

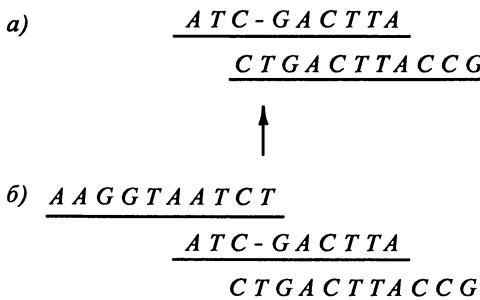


Рис. 16.16. Повторное слияние, следующее истории слияний, показанной ранее. *а* — первый контиг с одним пробелом в выравнивании. Профиль этого выравнивания содержит один столбец с пробелом и символом *T*. (Профили остальных столбцов содержат по одному символу.) Когда третья строка выравнивается по профилю (а не по отдельной строке), пробел не вставляется. *б* — получающееся выравнивание. Оно делает убедительным принятие *T* в качестве символа для столбца, содержавшего пробел

он может улучшить согласие сливаемой строки с контигом по сравнению с оригинальной раскладкой, которая основана на выравнивании новой строки с *только одной* из подстрок контига. Когда все подстроки влиты повторно, из раскладки создается одна согласительная строка.

16.16. Заключительные комментарии о нисходяще-восходящей расшифровке

Возврат наверх: компилирование полной последовательности

После расшифровки строк длины 400 и вывода последовательности для каждого из выбранных космид наш гипотетический проект крупномасштабной расшифровки входит в фазу *обратного хода*, которая вырабатывает последовательность для полной строки ДНК. Сначала в каждой выбранной YAC полученные строки из выбранных космид собираются с использованием космидной карты YAC. На этом шаге просто используется шаг сборки, определяющий основание, который мы уже рассмотрели выше. Затем полученные строки YAC используются с картой YAC для получения полной строки искомой ДНК.

На самом деле применяемая процедура подразумевает некоторый итеративный процесс, в котором чередуются фазы движения вниз и вверх.

Проблемы и будущие направления дробового подхода

В дробовой расшифровке ДНК постоянно возникают две проблемы. Первая в том, что окончательно получаемая строка часто нуждается в дополнительной корректировке. Обычно лаборанты просматривают результаты в протоколе расшифровки и программы сборки, отыскивая в полученной строке ненадежные области, которые затем либо расшифровываются повторно, либо проверяются каким-либо другим лабораторным методом. Один из предложенных методов проверки включает *расшифровку гибридизацией*, она будет рассматриваться в п. 16.18.

Пока еще активно исследуются возможности более полной автоматизации заключительных шагов отыскания и корректирования трудных мест полученной строки.

Вторая важная проблема для высших организмов — это частое появление повторяющихся подстрок. Проблема особенно серьезна, когда повтор длиннее, чем обычная длина читаемого участка (400 пар оснований). Повторы вызывают в методе дробовой расшифровки проблемы, которые похожи на те, что обсуждались (в п. 16.12) для направленной расшифровки. Однако, так как дробовая расшифровка работает параллельно над всей строкой, она может получить куски строки по обе стороны от повтора, даже если не может корректно получить все повторяющиеся области. В этом ее отличие от направленной расшифровки, которая может оказаться неспособной продолжить расшифровку за первым же встретившимся трудным местом. Кроме того, при использовании дробовых методов проблемы, вызванные повторами, зачастую могут легче определяться и разрешаться альтернативными лабораторными методами. Одним из показателей существования повторов является то, что окончательно выведенная космидная строка короче, чем известная настоящая. Более информативным указанием на повторы является то, что раскладка из шага два содержит область, в которой перекрывается значительно больше подстрок, чем ожидалось, и/или область, где перекрытий существенно меньше, чем ожидалось. Может также случиться, что пропуск в раскладке может быть закрыт за счет перераспределения некоторых подстрок, которые жадный алгоритм разместил в одной из областей с сильным перекрытием.

Другая проблема, связанная с дробовой расшифровкой в практическом ее использовании, — частая встречаемость в YAC химерных клонов. Это, строго говоря, не проблема дробовой методологии, но она часто включается в такие обсуждения потому, что дробовую расшифровку обычно используют в больших проектах, где YAC участвует как клонирующий вектор. Родственная задача встречаемости в YAC удалений также представляется очень общей.

Недавно ряд исследователей попытались внести в фазу сборки фрагментов дробовым методом использование доступной побочной информации. Побочная информация зачастую состоит из знания конкретных пар фрагментов, которые должны или не должны перекрываться, из знания того, как фрагменты соотносятся с генетической картой или с другими физическими картами низкого разрешения, из знания диапазона физических расстояний между парами фрагментов или только из знания некоторого частичного порядка в раскладке фрагментов. Есть надежда, что при включении всей побочной информации число допустимых сборок будет очень мало и получающиеся вычислительные задачи сильно упростятся. О попытках формализации в этом направлении сообщается в [343].

16.17. Задача о кратчайшей надстроке

Обсудив картирование и расшифровку, которые включают передовые проблемы молекулярной биологии, вернемся немного назад. Мы рассмотрим более абстрактную чисто строковую задачу, интерес к которой отчасти объясняется задачей сборки последовательности в дробовой расшифровке.

16.17.1. Введение в надстроки

Определение. Пусть задан набор из k строк $\mathcal{P} = \{S_1, S_2, \dots, S_k\}$. *Надстрокой* набора \mathcal{P} называется строка, содержащая каждую из строк \mathcal{P} в качестве подстроки.

Например, конкатенация строк \mathcal{P} в любом порядке дает тривиальную надстроку \mathcal{P} . В качестве более интересного примера пусть $\mathcal{P} = \{abcc, efab, bccla\}$. Тогда $bcclabccefab$ является надстрокой \mathcal{P} , а $efabccla$ — другой надстрокой, более короткой. Вообще, мы заинтересованы в нахождении надстрок малой длины.

Определение. Для набора \mathcal{P} обозначим через $S^*(\mathcal{P})$ его кратчайшую надстроку. Когда набор \mathcal{P} ясен из контекста, будем писать S^* вместо $S^*(\mathcal{P})$.

Задача нахождения кратчайшей надстроки может иметь приложения в сжатии данных (см. упражнения 25 и 26), но больше всего интерес к ней вызывается одним применением к задаче сборки последовательности с помощью дробовой расшифровки. Каждая строка набора \mathcal{P} моделирует один из расшифровываемых фрагментов ДНК, созданных протоколом дробовой расшифровки (п. 16.14). В задаче сборки требуется получить исходную строку S из набора расшифрованных фрагментов \mathcal{P} . Если ошибок расшифровки нет, исходная строка S является надстрокой \mathcal{P} , и при некоторых предположениях похоже, что S будет кратчайшей надстрокой \mathcal{P} . В этом случае кратчайшая надстрока \mathcal{P} будет хорошим кандидатом на роль исходной строки S .

На самом деле реальные данные о последовательностях всегда содержат ошибки и пропущенные места, и исследование дробовых данных показывает, что кратчайшая надстрока часто короче исходной строки, особенно, когда в исходной строке есть области повторяющихся подстрок. Следовательно, задача о кратчайшей надстроке — это абстрактная алгоритмическая задача, которая только отдаленно моделирует работу по реальной сборке последовательности. Тем не менее исследование задачи о надстроке обосновано по тем же причинам, что и исследование модельных алгоритмов или исследования *in vitro* в молекулярной биологии. Методы, развивающиеся при изучении абстрагированной, упрощенной задачи (или простого организма), часто являются важными шагами для получения практических результатов о более сложных или реалистичных системах.

Известно, что задача нахождения кратчайшей надстроки для набора строк является NP-трудной [71], и поэтому неправдоподобно, что существует эффективный алгоритм для нахождения ее точного решения. Более того, задача о кратчайшей надстроке является MAX-SNP трудной [69], и представляется, что не существует и алгоритма с полиномиальным временем, который может находить решение, отстоящее от оптимального на произвольную, но заранее заданную константу. Тем не менее имеются методы с полиномиальным временем, которые приближаются к оптимальному решению в пределах четырех- или трехкратного значения (и лучше). Мы подробно рассмотрим метод с гарантированной трехкратной границей, т. е. быстрый метод, который находит надстроку с длиной, отличающейся от длины кратчайшей надстроки не более чем в три раза. Эта граница была получена Блюном, Янгом, Ли, Тромпом и Яннакакисом [69]. Метод, который мы представляем здесь, получен из их решения, но несколько отличается деталями, так что надстроки, созданные этими алгоритмами, могут несколько отличаться.

Начиная с публикации [69], граница ошибки уменьшалась в ряде статей, хотя все алгоритмы в этих статьях следовали сходным основным идеям. Общий подход

из [69] был сначала уточнен Тенгом и Яо [434], которые нашли алгоритм, гарантирующий границу ошибки 2.89. Затем эту границу уменьшили до $2\frac{50}{63}$ Козараю, Парк и Стейн [283] и до 2.75 Армен и Стейн [30]. Уменьшение до 2.67 было достигнуто Янгом и Янгом [246] и затем до 2.596 — Бреслауером, Янгом и Янгом [77]. Другим способом уменьшение до 2.67 получили Армен и Стейн [29]. К данному моменту лучшая оценка принадлежит Свидыку, уменьшившему ее до 2.5 [427].

Улучшения множителя до уровня ниже трех выходят за рамки этой книги. Однако те результаты, которые рассмотрены далее, охватывают большую часть принципиального подхода и сам дух улучшений (хотя и не детали).

Близкая задача, также представляющая интерес, — найти кратчайшую надстроку набора строк, когда каждая из строк набора может входить в подстроку как есть или в перевернутом виде. Интерес к этой задаче о надстроке без ориентации объясняется тем фактом, что при реальной сборке последовательности фрагменты ДНК поступают с обеих нитей строки ДНК, и происхождение каждого конкретного фрагмента неизвестно (см. упражнение 2). В контексте задачи о надстроке такая ситуация приводит к задаче, где задан набор подстрок, ориентация которых неизвестна. Этот случай рассмотрен в статье Янга, Ли и Ду [248], где подробно описан приближенный метод с множителем четыре и утверждается, что можно получить множитель три.

Основные определения

На протяжении всего рассмотрения надстрок мы предполагаем, что никакая строка в \mathcal{P} не является подстрокой другой строки из \mathcal{P} . Любые такие строки могут быть эффективно опознаны и удалены заранее (см. упражнение 20 главы 7). Легко видеть, что такое удаление не изменяет кратчайшей надстроки. Следовательно, наше предположение не умаляет общности.

Определение. Для любой строки S обозначим через $|S|$ ее длину. Для набора строк \mathcal{P} обозначим через $||\mathcal{P}||$ суммарную длину его строк.

Определение. Для двух строк, S_i и S_j , обозначим через $ov(S_i, S_j)$ длину наибольшего совпадения (overlap — перекрытие) суффикса строки S_i и префикса строки S_j . Мы введем $p(S_i, S_j) = |S_i| - ov(S_i, S_j)$ и определим $pref(S_i, S_j)$ как префикс S_i длины $p(S_i, S_j)$.

Иначе говоря, $p(S_i, S_j)$ — это префикс S_i , кончающийся перед началом наибольшего суффиксно-префиксного совпадения S_i и S_j . См. пример на рис. 16.17.

S_i	$\underline{a \ b \ a \ c \ d}$
S_j	$\underline{\quad a \ c \ d \ e \ f}$

Рис. 16.17. Перекрытие строк $S_i = abacd$ и $S_j = acdef$, $pref(S_i, S_j) = ab$, $p(S_i, S_j) = 2$ и $ov(S_i, S_j) = 3$

Определение. Пусть $L = o_1, o_2, o_3, \dots, o_t$ — перестановка чисел от 1 до t . L определяет упорядоченный набор из t строк $\{S_{o_1}, S_{o_2}, \dots, S_{o_t}\}$ из \mathcal{P} , она определяет также следующую строку $S(L)$:

$$S(L) = pref(S_{o_1}, S_{o_2})pref(S_{o_2}, S_{o_3}) \dots pref(S_{o_{t-1}}, S_{o_t})S_{o_t}.$$

$S(L)$ является конкатенацией неперекрывающихся префиксов пар смежных строк в L , к которым добавлена полная строка S_{o_i} . Например, если $S_{o_1} = azwad$, $S_{o_2} = adcste$, $S_{o_3} = stee$, $S_{o_4} = eeaz$, то $S(L) = azwadcsteeaz$ (рис. 16.18). Рассмотрим два важных свойства строки $S(L)$.

$S(L)$	<u>a</u>	<u>z</u>	<u>w</u>	<u>a</u>	<u>d</u>	<u>c</u>	<u>s</u>	<u>t</u>	<u>e</u>	<u>e</u>	<u>a</u>	<u>z</u>

Рис. 16.18. Стока $S(L)$ сформирована из четырех строк S_{o_1} , S_{o_2} , S_{o_3} и S_{o_4} , определенных в тексте; $\text{pref}(S_{o_1}, S_{o_2}) = azw$, $\text{pref}(S_{o_2}, S_{o_3}) = adc$ и $\text{pref}(S_{o_3}, S_{o_4}) = st$

Лемма 16.17.1. Длина строки $S(L)$ равна $\sum_{i=1}^{t-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_t}|$.

Эта лемма иллюстрируется на рис. 16.18. Ее доказательство прямо следует из определений.

Лемма 16.17.2. Пусть L и $S(L)$ определены, как выше. Тогда строка $S(L)$ является надстрокой (не обязательно кратчайшей) строк $\{S_{o_1}, S_{o_2}, \dots, S_{o_t}\}$.

Рассмотрение рис. 16.18 делает лемму почти самоочевидной, но все же представим формальное доказательство.

Доказательство леммы 16.17.2. Доказательство ведется индукцией по i , начиная от $i = t$ с уменьшением до $i = 1$. Стока S_{o_i} является подстрокой $S(L)$, так как она явно включена в ее конец. Предположим, что утверждение верно для $i + 1$. Мы должны показать, что S_{o_i} также является подстрокой $S(L)$. Так как $\text{pref}(S_{o_i}, S_{o_{i+1}})$ явно включена в $S(L)$, нам нужно доказать только, что оставшаяся часть S_{o_i} содержится в $S(L)$ непосредственно вслед за $\text{pref}(S_{o_i}, S_{o_{i+1}})$. Это так, если $S_{o_i} = \text{pref}(S_{o_i}, S_{o_{i+1}})$ (т. е. если S_{o_i} и $S_{o_{i+1}}$ не имеют перекрытия). В противном случае пусть r_{o_i} обозначает суффикс S_{o_i} , начинающийся сразу после $\text{pref}(S_{o_i}, S_{o_{i+1}})$. Из того факта, что никакая строка в \mathcal{P} не является подстрокой другой строки, r_{o_i} является префиксом $S_{o_{i+1}}$. Теперь, по индуктивному предположению, $S_{o_{i+1}}$ является подстрокой $S(L)$, поэтому r_{o_i} содержится в $S(L)$, начинаясь непосредственно вслед за $\text{pref}(S_{o_i}, S_{o_{i+1}})$, и индукция завершена. \square

16.17.2. Целевая функция для надстрок

Пусть список L будет перестановкой Π всех чисел от 1 до k . Из леммы 16.17.2 получается:

Следствие 16.17.1. Для набора \mathcal{P} из k строк каждая перестановка Π чисел от 1 до k определяет надстроку $S(\Pi)$ набора \mathcal{P} .

Обратно, если S^* — кратчайшая надстрока набора \mathcal{P} , то S^* должна быть $S(\Pi)$ для какой-то перестановки Π . Чтобы увидеть это, отметим, что, так как \mathcal{P} не содержит подстрок своих строк, строка S_i может начинаться слева от строки S_j в надстроке S^* в том и только том случае, если S_i и кончается левее S_j в S^* . Далее, никакие две строки не могут начинаться или кончаться в одной и той же точке S^* . Следовательно, искомая перестановка Π (для которой $S^* = S(\Pi)$) получается при том порядке, в котором в S^* получаются левые концы строк \mathcal{P} . Перекрытие в каждой смежной

паре строк в S^* должно быть максимально возможным, иначе S^* можно было бы укоротить, и следовательно, $S^* = S(\Pi)$. В качестве примера рассмотрим снова строки на рис. 16.18. Если \mathcal{P} — упорядоченный набор $\{eeaz, adcste, azwad, stee\}$, то перестановка, показанная на рис. 16.18, будет $\Pi = 3, 2, 4, 1$.

По лемме 16.17.1 если перестановка Π есть $o_1, o_2, o_3, \dots, o_k$, то

$$|S(\Pi)| = \sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}|.$$

Значит, кратчайшая надстрока определяется перестановкой $\Pi = o_1, o_2, o_3, \dots, o_k$, минимизирующей

$$\sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}|.$$

Но

$$p(S_{o_i}, S_{o_{i+1}}) = |S_{o_i}| - ov(S_{o_i}, S_{o_{i+1}}),$$

так что

$$\begin{aligned} |S(\Pi)| &= \sum_{i=1}^{k-1} p(S_{o_i}, S_{o_{i+1}}) + |S_{o_k}| = \sum_{i=1}^k |S_{o_i}| - \sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}) = \\ &= ||\mathcal{P}|| - \sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}). \end{aligned}$$

Следовательно, кратчайшая надстрока может быть также задана перестановкой, максимизирующей

$$\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}}).$$

В последнем определении есть тонкие моменты, так как возможно, что в одной точке перекрываются более двух строк (как на рис. 16.18), тогда как суммирование включает только члены, соответствующие перекрытиям пар последовательных строк в перестановке. В качестве целевой функции при поиске кратчайшей надстроки мы будем в основном использовать последний максимизационный критерий.

16.17.3. Циклические строки и циклические покрытия

Определение. Пусть задана (линейная) строка S . Определим *циклическую строку* $\varphi(S)$ как копию S с той модификацией, что последний символ S считается предшествующим первому символу S . Другими словами, начало и конец S соединяются вместе, чтобы получилась циклическая строка $\varphi(S)$.

Будем использовать φ вместо $\varphi(S)$, если из контекста понятно, о какой строке S идет речь, или если это несущественно.

Определение. Длина циклической строки $\varphi(S)$, обозначаемая $|\varphi(S)|$, равна $|S|$ — длине строки S .

Определение. Пусть $inf(S)$ — бесконечное повторение строки S . Скажем, что строка α отображается в циклическую строку $\varphi(S)$, если α является подстрокой $inf(S)$. Ясно, что если α является подстрокой $inf(S)$, то одно из вхождений α начинается в какой-либо позиции от 1 до $|S|$ строки $inf(S)$.

Например, если $S = abac$, то строка $acabacabacabaca$ отображается в $\varphi(S)$.

Отметим, что строка, отображающаяся в циклическую строку φ , может содержать сколь угодно больше чем $|\varphi|$ символов. Можно представить себе колесо, на котором по кругу стоят символы, составляющие строку φ . Представьте себе, что эти символы смазали краской и колесо прокатили по бумаге. Любая отпечатанная так строка (любой длины) будет строкой, отображаемой в φ .

Определение. Определим *циклическое покрытие* $C(\mathcal{P})$ набора \mathcal{P} как набор циклических строк, такой что каждая строка из \mathcal{P} отображается по крайней мере в одну из циклических строк $C(\mathcal{P})$. Когда набор \mathcal{P} ясен из контекста, мы будем использовать C вместо $C(\mathcal{P})$.

Определение. *Длина циклического покрытия* $C(\mathcal{P})$ обозначается через $\|C\|$ и равна сумме длин ее циклических строк. Значит, $\|C\| = \sum_{\varphi \in C} |\varphi|$.

Определение. Пусть $C^*(\mathcal{P})$ обозначает минимальную длину циклического покрытия \mathcal{P} .

Любая надстрока S набора \mathcal{P} тривиально определяет циклическое покрытие той же длины (содержащее одну циклическую строку $\varphi(S)$). Отсюда:

Лемма 16.17.3. *Длина кратчайшей надсторки S^* набора \mathcal{P} не меньше, чем $\|C^*(\mathcal{P})\|$.*

Мы увидим, что задачу нахождения циклического покрытия минимальной полной длины можно решить эффективно. Этот факт лежит в основе всех опубликованных приближенных методов для задачи о кратчайшей надстроке. До обсуждения конкретных методов посмотрим, как получить надсторку из циклического покрытия.

16.17.4. Как циклические покрытия определяют надсторки

Пусть задано циклическое покрытие $C(\mathcal{P})$ набора \mathcal{P} . Нужно найти разбиение набора \mathcal{P} на подмножества, отображаемые в одну и ту же циклическую строку этого покрытия. Но так как строка из \mathcal{P} может отображаться более чем в одну циклическую строку из $C(\mathcal{P})$, при создании этого разбиения может существовать свобода выбора, о которой сейчас и пойдет речь.

Определение. Если строка $S \in \mathcal{P}$ отображается только в одну циклическую строку φ из $C(\mathcal{P})$, то мы *ассоциируем* S с φ . В противном случае мы произвольно *ассоциируем* S с одной из циклических строк $C(\mathcal{P})$, в которые отображается S .

Определение. Для циклической строки $\varphi \in C^*(\mathcal{P})$ пусть \mathcal{P}_φ обозначает подмножество строк \mathcal{P} , ассоциированных с φ . Пусть $|S_\varphi| = t$.

Для любой циклической строки $\varphi(S)$ перенумеруем символы φ , начиная с первого символа S . При этой нумерации каждая строка \mathcal{P}_φ отображается в φ , начиная с одной из позиций от 1 до $|S|$. Далее, так как никакая строка в \mathcal{P} не является началом другой, то никакие две строки в \mathcal{P}_φ не отображаются в одну и ту же начальную точку.

Определение. Пусть $L_\varphi = o_1, o_2, o_3, \dots, o_t$ — индексы строк в \mathcal{P}_φ в порядке их стартовых позиций в φ . Остерегайтесь спутать эти индексы (которые определяют упорядочение конкретного подмножества \mathcal{P}) с их действительными стартовыми позициями в φ .

Например, рассмотрим строку $S = azwadcsstee$. Циклическая строка $\varphi(S)$ показана в левой части рис. 16.19. Каждая строка упорядоченного множества $\Pi = \{eeaz, adcste, azwad, stee\}$ отображается в $\varphi(S)$. Начальные позиции каждой из этих четырех строчек на рисунке помечены и занумерованы. L_φ — это перестановка $3, 2, 4, 1$, а строка $S(L_\varphi)$ показана в правой части рисунка.

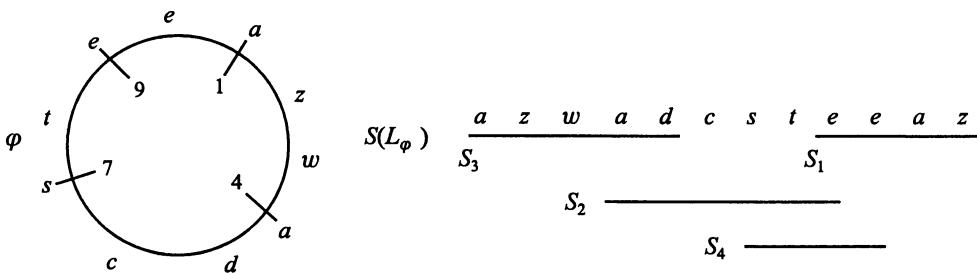


Рис. 16.19. Циклическая строка $\varphi(S)$ для $S = azwadcsstee$. Стока $S(L_\varphi)$ образована из четырех строк \mathcal{P} , ассоциированных с φ

По лемме 16.17.2 строка $S(L_\varphi) = \text{pref}(S_{o_1}, S_{o_2})\text{pref}(S_{o_2}, S_{o_3})\dots\text{pref}(S_{o_{t-1}}, S_{o_t})S_{o_t}$ является надстрокой набора \mathcal{P}_φ . Далее, если L'_φ является циклическим сдвигом списка L_φ , то строка $S(L'_\varphi)$ оказывается также надстрокой \mathcal{P}_φ . Например, если $L'_\varphi = \{4, 1, 3, 2\}$, то $S(L'_\varphi)$ — надстрока \mathcal{P}_φ , и ее длина больше, чем длина надстроки $S(L_\varphi)$, изображенной на рис. 16.19. Это произошло потому, что $ov(S_2, S_4) = 3 = ov(S_1, S_3) + 1$.

Определение. Если o_i — последний индекс в списке L'_φ (циклическом сдвиге L_φ), то строка S_{o_i} называется *последней строкой* $S(L'_\varphi)$.

Утверждение следующей леммы получается из определений и из того факта, что любая строка, которая отображается в φ , может быть отображена так, что ее начальная позиция будет лежать между 1 и $|\varphi|$.

Лемма 16.17.4. *Если S_{o_i} — последняя строка $S(L'_\varphi)$, то длина $S(L'_\varphi)$ равна в точности $|\varphi| + ov(S_{o_i}, S_{o_{i+1}}) \leq |\varphi| + |S_{o_i}|$, где $o_{i+1} = o_1$.*

Ввиду леммы 16.17.4 циклический сдвиг, минимизирующий длину получающейся надстроки $S(L'_\varphi)$, находится выбором индекса o_i , который минимизирует $ov(S_{o_i}, S_{o_{i+1}})$. Этот факт нам понадобится далее в приближении с множителем три.

16.17.5. Приближение с множителем четыре

Опишем алгоритм, находящий надстроку, которая превосходит оптимальную длину не более чем в четыре раза.

Приближенный алгоритм с множителем четыре

1. Найти циклическое покрытие минимальной длины $C^*(\mathcal{P})$ набора \mathcal{P} и ассоциировать каждую строку $S \in \mathcal{P}$ с одним циклом в $C^*(\mathcal{P})$, в который отображается S . {Позднее мы обсудим, как эффективно найти $C^*(\mathcal{P})$.}

2. Для каждой циклической строки φ из $C^*(\mathcal{P})$ сформировать упорядоченный список индексов L_φ и создать строку $S(L_\varphi)$, надстроку набора \mathcal{P}_φ . Пусть \mathcal{P}' — множество надстрок, полученных на этом шаге.
3. Конкатенировать строки \mathcal{P}' (в любом порядке), получив надстроку H набора \mathcal{P} .

Ясно, что H является надстрокой \mathcal{P} , так как каждая $S(L_\varphi)$ является надстрокой для строк, ассоциированных с циклической строкой φ , и $C^*(\mathcal{P})$ — циклическое покрытие \mathcal{P} . Каждая строка $S(L_\varphi) \in \mathcal{P}'$ (найденная на шаге 2) имеет одну последнюю строку. Пусть \mathcal{P}_f — множество этих последних строк. Тогда, используя лемму 16.17.4, получаем следующее утверждение.

Лемма 16.17.5. *Надстрока H имеет длину, не превосходящую $\|C^*\| + \sum_{S \in \mathcal{P}} |S|$.*

Анализ ошибок алгоритма. Подготовка к анализу

Сначала мы установим факт (*лемма о перекрытии*), который будет (*самым*) важным средством анализа всех приближенных алгоритмов поиска надстроки, имеющихся к данному моменту. Лемма о перекрытии в свою очередь использует очень важный факт о периодических строках (теорему об ОНД). Определение периода строки дано на с. 66.

Теорема 16.17.1. (Теорема об ОНД.) *Если строка S имеет два периода длины p и $q \leq p$ и $|S| \geq p + q$, то S имеет период ОНД(p, q)^{*}.*

Доказательство. Сначала покажем, что S имеет период длины $p - q$ (т.е. что $S(i) = S(i + p - q)$ для всех значений i в диапазоне от 1 до $|S| - p + q$). Возможны два случая: либо $i \leq q$, либо $i > q$. В первом случае $i + p \leq |S|$ по предположению, что $|S| \geq p + q$; таким образом, символ $i + p$ существует и равен символу i , так как S имеет период p . Далее, $q \leq p$, так что символ $i + p - q$ также существует и должен равняться символу $i + p$, поскольку S имеет период q . Следовательно, в первом случае $S(i) = S(i + p) = S(i + p - q)$. Во втором случае, когда $i > q$, симметричное рассуждение показывает, что символы $i - q$ и $i - q + p$ существуют, и оба должны равняться символу i . Итак, в обоих случаях $S(i) = S(i + p - q)$ и S имеет период $p - q$. Тривиально верно также, что $|S| \geq q + p - q$.

Теперь каждый шаг алгоритма Евклида (для вычисления ОНД двух произвольных чисел x и y) рекурсивно сводит при $x > y$ задачу вычисления ОНД(x, y) к задаче вычисления ОНД($x, x - y$). Алгоритм Евклида заканчивается получением ОНД(x, y). Следовательно, когда S имеет периоды p и q и $|S| \geq p + q$, выполнение алгоритма Евклида с исходными данными (p, q) будет работать на каждом шаге с двумя числами x и y , которые оба являются длинами периода S . В частности, окончательное число $x = y = \text{ОНД}(x, y)$ также является длиной периода для S , что и доказывает утверждение, что S имеет период ОНД(p, q). \square

Лемма 16.17.6. (Лемма о перекрытии.) *Пусть φ и φ' — две любые циклические строки из $C^*(\mathcal{P})$, а α и α' — две строки (не обязательно из \mathcal{P}), которые отображаются в φ и φ' соответственно. Тогда самое длинное суффиксно-префиксное совпадение α и α' имеет длину меньше $|\varphi| + |\varphi'|$.*

^{*}) Предположение $|S| \geq p + q$ можно было бы не делать. Если p кратно q , теорема очевидно верна, а в противоположном случае $|S|$ не меньше общего наименьшего кратного p и q , что больше $2p$. — Прим. перев.

Прежде чем доказывать лемму, напомним, что длина α может быть произвольно больше длины φ , и то же верно для α' и φ' . Таким образом, лемма не тривиальна. Когда α больше, чем циклическая строка φ , в которую она отображается, α оборачивается не менее одного раза вокруг φ и имеет, таким образом, период, равный $|\varphi|$.

Доказательство леммы 16.17.6. Предположим, что лемма не верна, так что длина перекрытия α и α' не меньше, чем $|\varphi| + |\varphi'|$. Пусть α'' обозначает суффиксно-префиксное совпадение α и α' . Покажем прежде всего, что $|\varphi| \neq |\varphi'|$.

Ясно, что α'' отображается и в φ и в φ' , так как α отображается в φ и α' отображается в φ' , а α'' является подстрокой обеих этих строк. По предположению $|\alpha''| \geq |\varphi| + |\varphi'|$, так что α'' должна полностью оборачиваться вокруг φ и вокруг φ' . Следовательно, если $|\varphi| = |\varphi'|$, то и φ и φ' состоят точно из первых $|\varphi|$ символов α'' и, следовательно, φ является циклическим вращением φ' (см. упражнение 1 на с. 33). Но тогда все строки, которые отображаются в φ , отображаются также в φ' , так что φ из покрытия можно убрать. Это невозможно сделать в циклическом покрытии минимальной суммарной длины, так что $|\varphi| \neq |\varphi'|$.

Так как α'' отображается и в φ и в φ' и оборачивается вокруг каждой из них, α'' должна быть полуperiодической строкой с периодами $|\varphi|$ и $|\varphi'|$. Далее, так как длина α'' предполагается не меньше $|\varphi| + |\varphi'|$, можно применить теорему об ОНД (теорему 16.17.1), которая утверждает, что α'' имеет период, равный $\text{ОНД}(|\varphi|, |\varphi'|)$. Так как $|\varphi| \neq |\varphi'|$, то $\text{ОНД}(|\varphi|, |\varphi'|) < \max\{|\varphi|, |\varphi'|\}$. Предположим, что максимум равен $|\varphi|$. Однако $\text{ОНД}(|\varphi|, |\varphi'|)$ делит нацело $|\varphi|$, так что φ состоит из повторяющихся полных копий строки длины $\text{ОНД}(|\varphi|, |\varphi'|)$. Так как $\text{ОНД}(|\varphi|, |\varphi'|) < |\varphi|$, то φ должна содержать не менее двух таких полных копий. Теперь рассмотрим циклическую строку γ , полученную из φ удалением одной из этих полных копий. Ясно, что все строки, которые отображаются в φ , отображаются также в меньшую циклическую строку γ . Но это противоречит предположению о том, что $C^*(\mathcal{P})$ — циклическое покрытие минимальной длины \mathcal{P} ; следовательно, $|\alpha''| < |\varphi| + |\varphi'|$, и лемма доказана. \square

Теорема 16.17.2. Надстрока H имеет полную длину, превосходящую длину кратчайшей надстроки $S^*(\mathcal{P})$ менее чем в четыре раза.

Доказательство. Напомним, что $\mathcal{P}_f \subseteq \mathcal{P}$ — это набор последних строк, полученных на шаге 2 нашего алгоритма. По определению любые две строки в \mathcal{P}_f отображаются в две различные циклические строки из $C^*(\mathcal{P})$. Ясно, что кратчайшая надстрока для \mathcal{P} не меньше кратчайшей надстроки \mathcal{P}_f . По лемме о перекрытии (16.17.6) любые две строки, которые отображаются в две различные циклические строки φ и φ' из $C^*(\mathcal{P})$, могут перекрыться не более чем на $|\varphi| + |\varphi'|$ символов. Поэтому если упорядоченный список $L = o_1, \dots, o_r$ определяет кратчайшую надстроку набора строк \mathcal{P}_f , то $\sum_{i=1}^{r-1} ov(S_{o_i}, S_{o_{i+1}}) < 2||C^*||$. Отсюда следует, что кратчайшая надстрока \mathcal{P}_f (и следовательно, \mathcal{P}) имеет длину больше, чем $\sum_{S \in \mathcal{P}_f} |S| - 2||C^*||$. Значит, $|S^*(\mathcal{P})| > \sum_{S \in \mathcal{P}_f} |S| - 2||C^*||$, так что $\sum_{S \in \mathcal{P}_f} |S| < 3|S^*(\mathcal{P})|$ (так как $||C^*|| \leq |S^*(\mathcal{P})|$). Однако по лемме 16.17.5 $|H| \leq \sum_{S \in \mathcal{P}_f} |S| + |S^*(\mathcal{P})|$ и $|H| \leq 4|S^*(\mathcal{P})|$. \square

Надеюсь, что читателю понравится центральная роль леммы о перекрытии в анализе алгоритма с множителем четыре. Эта лемма играет центральную роль и в алгоритме с множителем три, к которому мы сейчас перейдем. Следовательно,

интуитивно должно чувствоваться, что улучшения леммы о перекрытии или ее применения приведут к уменьшению гарантированных границ ошибки. Такой подход принят в [246, 77], где показано (в основном, без деталей), что перекрытие α и α' можно ограничить значением $(2/3)(|\varphi| + |\varphi'|)$ вместо $|\varphi| + |\varphi'|$. Родственному подходу следуют в [29]. Подробности слишком сложны для воспроизведения здесь. Улучшения до множителя три, которые мы обсуждаем далее, используют другую идею, но улучшения в лемме о перекрытии применялись и для уменьшения этой оценки.

16.17.6. Доведение до множителя три

Идея, побуждающая к улучшению, лежит в вопросе: почему алгоритм с множителем четыре просто конкатенирует строки в \mathcal{P}' ? Почему бы вместо этого не использовать строки из \mathcal{P}' как исходные данные для еще одного исполнения алгоритма (шагов 1 и 2)? Если это исполнение найдет перекрытия строк из \mathcal{P}' , то оно создаст более короткую подстроку, чем строка, полученная простой конкатенацией строк из \mathcal{P}' .

К сожалению, мы ничего не выиграем выполнением шагов 1 и 2 для \mathcal{P}' . Проблема в том, что кратчайшее циклическое покрытие \mathcal{P}' будет тем же, что и кратчайшее циклическое покрытие \mathcal{P} , потому что каждая строка в \mathcal{P}' будет только обертываться снова и образовывать циклическую строку из $C^*(\mathcal{P})$, из которой она выведена. Мы оставим формальное доказательство этого утверждения на упражнения. Итак, никакого прогресса от повторного исполнения шагов 1 и 2 с \mathcal{P}' не будет. Однако установим, что если каждая циклическая строка в покрытии \mathcal{P}' имеет не меньше двух строк, которые с ней ассоциированы, то (с привлечением еще одной идеи) найдутся дополнительные перекрытия между строками \mathcal{P}' , что приведет к гарантированному сокращению получающейся надстроки.

Определение. Определим *нетривиальное* циклическое покрытие \mathcal{P}' как циклическое покрытие, с каждой циклической строкой которого ассоциированы по меньшей мере две строки из \mathcal{P}' .

Мы обсудим позднее, как найти для набора строк нетривиальное циклическое покрытие минимальной длины. А сейчас представим улучшенный приближенный алгоритм, предполагая, что такое покрытие можно найти. При несколько отличном описании первые два шага будут такими же, как в предыдущем алгоритме (с множителем четыре).

Приближенный алгоритм с множителем три

- Найти циклическое покрытие минимальной длины $C^*(\mathcal{P})$ для \mathcal{P} и ассоциировать каждую строку $S \in \mathcal{P}$ строго с одним циклом в $C^*(\mathcal{P})$, а именно с тем, в который отображается S .
- Для каждой циклической строки φ из $C^*(\mathcal{P})$ сформировать список индексов L_φ и создать строку $S(L_\varphi)$ — надстроку строк, ассоциированных с φ . Пусть $\mathcal{P}' = \{S'_1, S'_2, S'_3, \dots, S'_r\}$ обозначает множество полученных надстрок.
- Найти *нетривиальное* циклическое покрытие минимальной длины $C^*(\mathcal{P}')$ для множества строк \mathcal{P}' .

4. Для каждой циклической строки $\bar{\varphi}$ из $C^*(\mathcal{P}')$ пусть $L_{\bar{\varphi}} = o_1, o_2, \dots, o_r$ — индексы строк из \mathcal{P}' (не строк из \mathcal{P}), ассоциированных с $\bar{\varphi}$, в порядке их начальных позиций в $\bar{\varphi}$. По определению нетривиального циклического покрытия $t \geq 2$. Каждый циклический сдвиг $L_{\bar{\varphi}}$ определяет надстроку строк, которые отображаются в $\bar{\varphi}$. Найти циклический сдвиг, который дает кратчайшую из этих надстрок. Это значит выбрать $o_i \in L_{\varphi'}$, минимизирующий $ov(S'_{o_i}, S'_{o_{i+1}})$, и сдвинуть $L_{\bar{\varphi}}$ так, чтобы строка S'_{o_i} стала последней. Обозначить получившееся упорядочение строк через $L'_{\bar{\varphi}}$ и сформировать строку $S(L'_{\bar{\varphi}})$, которая является надстрокой строк из \mathcal{P}' , ассоциированных с $\bar{\varphi}$.
5. Конкатенировать надстроки, созданные на шаге 4, в строку $H'(\mathcal{P})$.

Ясно, что $H'(\mathcal{P})$ будет надстрокой для \mathcal{P}' и, следовательно, для \mathcal{P} .

Анализ ошибки

Определение. Пусть Q — набор строк и q — надстрока для Q . Назовем величину $\|Q\| - |q|$ *сжатием* (compression) набора Q , достигнутым надстрокой q , и обозначим ее через $comp(q)$.

Лемма 16.17.7. *Если набор Q содержит k строк $\{q_1, q_2, \dots, q_k\}$ и Π — перестановка чисел от 1 до k , то сжатие Q , достигнутое надстрокой $S(\Pi)$, равно $\sum_{i=1}^{k-1} ov(S_{o_i}, S_{o_{i+1}})$. Далее, кратчайшей надстрокой Q является надстрока, достигающая максимального сжатия Q .*

Доказательство леммы 16.17.7 в основном содержалось в рассмотрении целевых функций в п. 16.17.2 (см. с. 517). Формальное доказательство оставляется на упражнения.

Определение. Пусть $S^*(\mathcal{P}')$ обозначает кратчайшую надстроку \mathcal{P}' , а $comp^* = \|\mathcal{P}'\| - |S^*(\mathcal{P}')|$ — размер сжатия \mathcal{P}' , достигнутый $S^*(\mathcal{P}')$.

Следующая лемма ограничивает $comp^*$ сверху. Хотя это на первый взгляд может показаться отвлечением внимания, такая граница на самом деле будет применима.

Лемма 16.17.8. $comp^* \leq 2\|C^*(\mathcal{P})\|$. Значит, сжатие, достигнутое кратчайшей надстрокой \mathcal{P}' , не превосходит $2\|C^*(\mathcal{P})\|$, где $C^*(\mathcal{P})$ — циклическое покрытие минимальной длины для \mathcal{P} , вычисленное на шаге I алгоритма.

Доказательство. Пусть \mathcal{P}' содержит r строк. Кратчайшая надстрока \mathcal{P}' определяется некоторой перестановкой o_1, o_2, \dots, o_r чисел от 1 до r , и $comp^* = \sum_{i=1}^{r-1} ov(S'_{o_i}, S'_{o_{i+1}})$. Каждая строка в \mathcal{P}' является строкой $S(L_\varphi)$, полученной из отдельной циклической строки $\varphi \in C^*(\mathcal{P})$, и каждая такая $S(L_\varphi)$ отображается в циклическую строку φ , из которой она была получена. Следовательно, по лемме о перекрытиях (лемме 16.17.6) максимально возможное перекрытие двух строк в \mathcal{P}' , получаемое, скажем, из циклических строк φ и φ' , меньше, чем $|\varphi| + |\varphi'|$. Отсюда следует, что $comp^* = \sum_{i=1}^{r-1} ov(S'_{o_i}, S'_{o_{i+1}}) < 2\|C^*(\mathcal{P})\|$. \square

Определение. Сжатие набора \mathcal{P}' , достигаемое на его циклическом покрытии C , определяется как $\|\mathcal{P}'\| - \|C\|$ и обозначается через $comp(C)$.

Пусть φ — циклическая строка в C и Q — множество строк, ассоциированных с φ . Тогда сжатие Q , достигаемое φ , определяется как $\|Q\| - |\varphi|$.

Лемма 16.17.9. $|H'(\mathcal{P})| \leq |S^*(\mathcal{P}')| + comp^*/2$.

Доказательство. Напомним, что $C^*(\mathcal{P}')$ — это нетривиальное циклическое покрытие \mathcal{P}' минимальной длины. По определению сжатия и из того факта, что $||C^*(\mathcal{P}')|| \leq |S^*(\mathcal{P})|$, следует, что $||\mathcal{P}'|| - comp(C^*(\mathcal{P}')) = ||C^*(\mathcal{P}')|| \leq |S^*(\mathcal{P}')| = ||\mathcal{P}'|| - comp^*$. Следовательно, $comp(C^*(\mathcal{P}')) \geq comp^*$.

Напомним, что в шаге 3 алгоритма S'_{o_i} обозначает последнюю строку $S(L'_{\bar{\varphi}})$. $C^*(\mathcal{P}')$ — нетривиальное покрытие, так что по меньшей мере две строки из \mathcal{P}' ассоциированы с $\bar{\varphi}$. Далее, o_i было выбрано так, чтобы минимизировать $ov(S'_{o_i}, S'_{o_{i+1}})$. Отсюда следует, что $ov(S'_{o_i}, S'_{o_{i+1}})$ не превосходит половины сжатия, достигнутого $\bar{\varphi}$. Но $ov(S'_{o_i}, S'_{o_{i+1}})$ равно в точности дополнительной длине, добавляемой к $|\bar{\varphi}|$, чтобы получить строку $S(L'_{\bar{\varphi}})$ из $\bar{\varphi}$. Теперь сжатие \mathcal{P}' , достигаемое $H'(\mathcal{P})$, равно сжатию \mathcal{P}' , достигаемому $C^*(\mathcal{P}')$, за вычетом тех длин, которые нужно добавить для получения $H'(\mathcal{P})$ из $C^*(\mathcal{P}')$. Следовательно, $comp(H'(\mathcal{P})) \geq comp(C^*(\mathcal{P}')) - comp(C^*(\mathcal{P}'))/2 = comp(C^*(\mathcal{P}'))/2 \geq comp^*/2$. В итоге $comp(H'(\mathcal{P})) \geq comp^*/2$.

С учетом приведенных фактов получим $|H'(\mathcal{P})| = ||\mathcal{P}'|| - comp(H'(\mathcal{P})) \leq ||\mathcal{P}'|| - comp^*/2 = ||\mathcal{P}'|| - comp^* + comp^*/2 = |S^*(\mathcal{P}')| + comp^*/2$. \square

Лемма 16.17.10. $|S^*(\mathcal{P}')| \leq |S^*(\mathcal{P})| + ||C^*(\mathcal{P})|| \leq 2|S^*(\mathcal{P})|$.

Доказательство. Чтобы доказать лемму, мы предъявим надстроку \mathcal{P}' , длина которой не более чем вдвое превышает длину $S^*(\mathcal{P})$ — кратчайшей надстроки \mathcal{P} .

Напомним, что \mathcal{P}_f — это множество последних строк, полученных из циклического покрытия $C^*(\mathcal{P})$. Пусть $S^*(\mathcal{P}_f)$ — кратчайшая надстрока \mathcal{P}_f . Так как $\mathcal{P}_f \subseteq \mathcal{P}$, то длина $S^*(\mathcal{P}_f)$ не превосходит $|S^*(\mathcal{P})|$. Образуем надстроку для \mathcal{P}' следующим образом. Пусть S — какая-либо строка из \mathcal{P}_f и φ — цикл из $C^*(\mathcal{P})$, с которым ассоциирована S . Тогда S является последней строкой в $S(L_\varphi)$, и $S(L_\varphi)$ входит в \mathcal{P}' . Мы “расширим” копию S в $S^*(\mathcal{P}_f)$, чтобы она содержала $S(L_\varphi)$, но еще сохраняла перекрытия, найденные в $S^*(\mathcal{P}_f)$. Когда это будет сделано для каждой строки S из \mathcal{P}_f , получающаяся строка будет надстрокой для \mathcal{P}' . Опишем теперь, как расширять строки.

Пусть $S_b \in \mathcal{P}_f$ — последняя строка в $S^*(\mathcal{P}_f)$ и $S_a \in \mathcal{P}_f$ — находящаяся непосредственно слева от нее в $S^*(\mathcal{P}_f)$. Пусть $\varphi \in C^*(\mathcal{P})$ — циклическая строка, с которой ассоциирована S_b , и $S(L_\varphi) \in \mathcal{P}'$ — строка, созданная на шаге 2 из φ с последней строкой S_b . Пусть S_l обозначает первую строку $S(L_\varphi)$. Образуем строку $pref(S_b, S_l)S(L_\varphi)$. Так как φ — циклическая строка и S_b — последняя строка $S(L_\varphi)$, строка S_b является и префиксом и суффиксом строки $pref(S_b, S_l)S(L_\varphi)$ (см. пример на рис. 16.20). Следовательно, S_a перекрывает $pref(S_b, S_l)S(L_\varphi)$ не меньше, чем S_b . Далее, длина $pref(S_b, S_l)S(L_\varphi)$ в точности равна $|\varphi| + |S_b|$. Таким образом, мы можем заменить S_b на $pref(S_b, S_l)S(L_\varphi)$ в $S^*(\mathcal{P}_f)$ и перекрыть S_a не меньше, чем раньше. Чистое приращение длины получающейся строки будет в точности $|\varphi|$. Применяя это рассуждение к каждой строке из \mathcal{P}_f и сдвигаясь с каждой итерацией влево в $S^*(\mathcal{P}_f)$, мы получим надстроку для \mathcal{P}' , которая имеет длину не более $|S^*(\mathcal{P}_f)| + ||C^*(\mathcal{P})|| \leq |S^*(\mathcal{P})| + ||C^*(\mathcal{P})|| \leq 2|S^*(\mathcal{P})|$.

Эта надстрока имеет длину не меньшую, чем кратчайшая надстрока \mathcal{P}' , и лемма доказана. \square

$$\begin{array}{l}
 a) \quad S_a \quad \begin{array}{c} p \quad q \quad e \quad e \quad a \quad z \\ \hline S_3 \end{array} \\
 \qquad\qquad\qquad S_1 \quad \hline
 \\[10pt]
 b) \quad e \quad e \quad a \quad z \quad w \quad a \quad d \quad c \quad s \quad t \quad e \quad e \quad a \quad z \\
 \hline S_3 \qquad\qquad\qquad S_1 \\
 \qquad\qquad\qquad S_2 \quad \hline \\
 \qquad\qquad\qquad S_4 \quad \hline
 \\[10pt]
 c) \quad \begin{array}{c} e \quad e \quad a \quad z \quad w \quad a \quad d \quad c \quad s \quad t \quad e \quad e \quad a \quad z \\ \hline p \quad q \quad e \quad e \quad a \quad z \end{array}
 \end{array}$$

Рис. 16.20. Расширение $S^*(\mathcal{P}_f)$. Здесь строка $S_1 = eeaz$ является последней строкой, а $S_3 = azwadis$ — первой строкой, S_1 в $S(L_\varphi)$ показана на рис. 16.19. Если $b = 1$, так что $eeaz$ — последняя строка в $S^*(\mathcal{P}_f)$, и, скажем, $S_a = pqeea$, то S_a перекрывает $\text{pref}(S_1, S_3)S(L_\varphi)$ в точности на столько же, насколько она перекрывает S_3 . Длина расширения в точности равна $|\varphi|$

Теорема 16.17.3. $|H'(\mathcal{P})| < 3|S^*(\mathcal{P})|$.

Доказательство. Получается немедленно комбинацией трех предыдущих лемм. \square

16.17.7. Эффективная реализация

Шаг 1 в обоих приближенных алгоритмах — и с множителем четыре, и с множителем три — находит циклическое покрытие \mathcal{P} минимальной полной длины. Такое покрытие можно найти, сводя эту задачу к “задаче о назначениях”. Если \mathcal{P} содержит k строк, то исходными данными для задачи о назначениях будет некоторая матрица A размера $k \times k$. Полное назначение в A представляет собой набор M из k клеток A , который содержит по одному элементу из каждой строчки и каждого столбца A . Вес назначения M равен сумме чисел в клетках M .

Чтобы увидеть соотношение между полным назначением и циклическим покрытием \mathcal{P} , определим цикл в M как набор различных индексов $\{i_1, i_2, \dots, i_t\}$, такой что элементы $A(i_1, i_2), A(i_2, i_3), \dots, A(i_t, i_1)$ содержатся в M . Тогда полное назначение определит набор циклов M , включающих все индексы от 1 до k ровно по одному разу. Действительно, предположим, что $A(i_1, i_2)$ содержится в M ; если $A(i_2, i_1)$ также в M , то определяется цикл из двух индексов (i_1, i_2) . В противном случае в M содержится $A(i_2, i_3)$ для $i_3 \neq i_2$ и т. д. до тех пор, пока для некоторого индекса i_t не окажется, наконец, что $A(i_t, i_1)$ содержится в M . Все индексы из этой последовательности образуют цикл из M . Так как каждый индекс появляется ровно один раз как индекс столбца и ровно один раз как индекс строчки в клетках полного назначения, никакой индекс не может появиться больше чем в одном цикле из полного назначения. Но поскольку каждый индекс должен появиться в каком-то цикле, то циклы, определенные M , учитывают каждый индекс ровно один раз.

В задаче назначения при сопоставлении каждого индекса i строке S_i в \mathcal{P} , каждый цикл в M определяет циклическую строку φ . Полагая $A(i, j)$ равным $ov(S_i, S_j)$ для каждой пары (i, j) , мы увидим, что циклическое покрытие \mathcal{P} с минимальной полной длиной получается из полного назначения максимального веса. Доказательство оставляется в качестве упражнения. Отметим, что при таком определении назначения клетка $A(i_1, i_1)$ может принадлежать назначению i , следовательно, образовывать отдельный, *тривиальный* цикл в M .

Есть несколько хорошо известных алгоритмов, которые находят полное назначение максимального веса за время $O(k^3)$ при произвольных весах [294]. Однако, когда каждое $A(i, j)$ равно $ov(S_i, S_j)$, есть жадный метод, который работает за время $O(k^2 \log k)$ и также находит полное назначение максимального веса. Следовательно, жадный метод получает циклическое покрытие минимальной длины быстрее, чем метод для общей задачи о назначениях. Эти утверждения обсуждаются в упражнениях 13, 14 и 15.

Нетривиальное циклическое покрытие. Шаг 3 в приближении с множителем три приводит к нахождению нетривиального циклического покрытия минимальной длины. Его можно выполнить за время $O(k^3)$, также полагая $A(i, i) = -\infty$ и затем находя результирующее полное назначение максимального веса. Такие диагональные значения заставляют выбрать полное назначение, в котором никакой индекс не сопоставляется сам себе.

Как эффективно формировать матрицу. Для определения матрицы A нужно вычислить значение $ov(S_i, S_j)$ по каждой упорядоченной паре строк S_i, S_j из \mathcal{P} . Это можно сделать эффективно. В п. 7.10 мы показали, как вычислить все k^2 необходимых суффиксно-префиксных совпадений за время $O(|\mathcal{P}| + k^2)$. Главным побудительным мотивом для этого результата было его использование в задаче о надстроке.

16.18. Расшифровка гибридизацией

В задаче о надстроке есть интересный и важный частный случай, который (в идеализированной форме) проще, чем общая задача. Этот частный случай встречается в предлагаемом методе расшифровки ДНК, называемом *расшифровкой гибридизацией* (*sequencing by hybridization* — SBH). Варианты исходной идеи сейчас интенсивно изучаются экспериментально и частично развиваются коммерчески [368, 134, 135, 98]. Предлагаемое здесь описание экспериментальных аспектов SBH будет сильно упрощено, но поможет показать общий дух метода и ввести некоторые связанные с ним (идеализированные) вычислительные задачи.

Идеализированная расшифровка гибридизацией

В методе расшифровки гибридизацией сначала создают физическую матрицу (называемую *чипом*) из 4^k клеток, для фиксированного числа k . Каждая клетка содержит отдельную известную копию одной из 4^k строк ДНК длины k . (Как создать такую матрицу — задача, интересная сама по себе [15, 163].) Затем делают много копий изучаемой строки ДНК, и к каждой копии прикрепляют радиоактивную или флюоресцентную метку (ярлык). Этот набор копий изучаемой ДНК вступает в контакт с матрицей, и в результате копия ДНК будет гибридизоваться с конкретной строкой χ длины k в том и только том случае, если χ является подстрокой изучаемой

строки. Каждая строка χ длины k прикреплена к отдельной клетке, и следовательно, каждая копия ДНК, гибридизовавшаяся с χ , также прикреплена к этой клетке. Когда негибридизовавшиеся копии ДНК вымываются из матрицы, клетки матрицы, содержащие строки ДНК, определяют множество подстрок длины k изучаемой ДНК. Расположение этих клеток легко определяется машиной, и так как каждая клетка соответствует известной строке длины k , результатом эксперимента будет полный список \mathcal{L} всех подстрок длины k , входящих в изучаемую ДНК. Отметим, однако, что никакая подстрока не включается в \mathcal{L} больше одного раза, независимо от того, как часто она входит в изучаемую строку. Этот список \mathcal{L} рассматривается как исходные данные для следующей вычислительной задачи.

Определение. Задача *SBH* заключается в вычислении, насколько возможно, строки ДНК S по списку \mathcal{L} всех возможных подстрок длины k , имеющихся в S . В частности, если возможно, в однозначном вычислении исходной строки S по списку \mathcal{L} .

Число k выбирается достаточно большим (в большинстве предложений оно около десяти), так что при *случайных* условиях неправдоподобно, чтобы какая-нибудь подстрока появилась в изучаемой строке S больше одного раза. Таким образом, в *идеализированной* версии задачи SBH мы предполагаем, что никакая подстрока длины k не появляется в S больше одного раза. Ясно, что изучаемая строка S является обычной надстрокой всех подстрок из \mathcal{L} , и при предположении, что никакая подстрока из \mathcal{L} не появляется больше одного раза, S является кратчайшей надстрокой набора \mathcal{L} . Однако набор \mathcal{L} больше структурирован, чем произвольная задача о надстроке, так как любые две последовательные подстроки длины k из S перекрываются по $k - 1$ нуклеотиду. Этот высокий уровень перекрытия можно использовать для получения очень эффективного и элегантного способа моделирования и штурма идеализированной задачи. Действительно, задача SBH сводится к вопросам нахождения эйлеровских путей в направленном графе. Это сведение было проведено Певзнером [366].

16.18.1. Сведение к эйлеровским путям

Пусть задан список \mathcal{L} всех подстрок длины k в искомой строке S . Построим ориентированный граф $G(\mathcal{L})$ следующим образом. Создадим 4^{k-1} вершин, каждую из них пометим отдельной строкой ДНК длины $k - 1$. Для каждой строки χ из \mathcal{L} проведем дугу из вершины, помеченной левыми $k - 1$ символами χ , в вершину, помеченную правыми $k - 1$ символами χ . Пометим эту дугу последним символом χ . Обратите внимание, что некоторые вершины $G(\mathcal{L})$ могут не соприкасаться с дугами; эти вершины можно удалить (см. пример на рис. 16.21).

Определение. Эйлеровский путь в ориентированном графе G — это путь, который проходит по каждой дуге G ровно по одному разу. Эйлеровский обход — это эйлеровский путь, у которого начало и конец совпадают.

Путь в $G(\mathcal{L})$ определяет строку S следующим образом. Стока S начинается с метки первой вершины пути и идет по нему, присоединяя по порядку метки проходимых дуг. Например, эйлеровский путь $\{AC, CA, AC, CG, GC, CA, AA, AC, CT, TT, TA, AA, AA\}$ определяет строку $S = ACACGCAACTTAA$. Отметим, что все

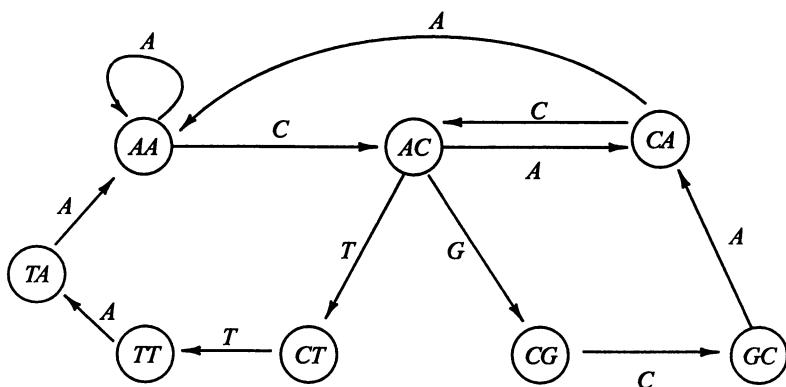


Рис. 16.21. Ориентированный граф $G(\mathcal{L})$, полученный из списка $\mathcal{L} = AAA, AAC, ACA, CAC, CAA, ACG, CGC, GCA, ACT, CTT, TTA, TAA$

тройки, перечисленные на рис. 16.21, входят в S , а другие тройки не входят. Теперь можно формализовать эти наблюдения.

Определение. Стока S называется *совместимой* с \mathcal{L} в том и только том случае, если S содержит каждую подстроку \mathcal{L} и (предполагая, что \mathcal{L} содержит подстроки длины k) S не содержит других подстрок длины k .

Теорема 16.18.1. Стока S совместима с \mathcal{L} в том и только том случае, если S определена эйлеровским путем в $G(\mathcal{L})$.

Доказательство теоремы прямо следует из определений \mathcal{L} и $G(\mathcal{L})$ (и оставляется читателю).

Следствие 16.18.1. Предположим, что каждая подстрока \mathcal{L} встречается в изучаемой строке ДНК строго один раз. Тогда набор данных \mathcal{L} однозначно определяет эту строку в том и только том случае, если $G(\mathcal{L})$ имеет единственный эйлеровский путь.

Следствие 16.18.2. Существует одно-однозначное соответствие между эйлеровскими путями в $G(\mathcal{L})$ и строками, совместимыми с \mathcal{L} . Значит, каждая совместимая строка соответствуетциальному эйлеровскому пути, а каждый эйлеровский путь определяет единственную совместимую строку.

Если данные были собраны правильно и никакая подстрока длины k не входит в искомую строку больше одного раза (как предполагалось), то граф $G(\mathcal{L})$ должен иметь по меньшей мере один эйлеровский путь, определяющий искомую строку ДНК. Когда эйлеровских путей в $G(\mathcal{L})$ больше одного, то возникает трудность, но даже в этом случае из набора возможных путей можно извлечь некоторую информацию относительно искомой строки. Поэтому (идеально) реконструирование искомой строки по набору данных \mathcal{L} сводится к задаче об эйлеровских путях в ориентированных графах.*)

*.) Действительно, каждый граф $G(\mathcal{L})$ является подграфом известных графов де Брёйна, так что вопросы об эйлеровских путях в $G(\mathcal{L})$ родственны вопросам об эйлеровских путях в графах де Брёйна (интересное обсуждение этого вопроса см. в [419]).

Как можно было ожидать, этот идеализированный вариант задачи SBH редко встречается на практике. Как всегда, есть ошибки в данных, и когда вероятны повторы в ДНК, то нерезонно исключать повторы подстрок из \mathcal{L} . Полное обсуждение расшифровки гибридизацией и возникающих вычислительных задач см. в [368, 135].

16.18.2. Непрерывность совместимых строк

В случае когда через вершины графа проходит больше одного эйлеровского пути, способов построения совместимой строки по данным гибридизации \mathcal{L} будет больше одного. Эта неопределенность затрудняет задачу нахождения исходной строки ДНК, но может еще оставаться возможность извлечения полезной информации о строке или о множестве всех совместимых строк. Например, может случиться, что некоторые подстроки содержатся во всех совместимых строках. Поэтому такие подстроки должны быть в искомой строке, а неясности — в оставшихся частях строки. Следовательно, можно заинтересоваться структурой *множества* всех совместимых строк или, что эквивалентно, множества всех эйлеровских путей $G(\mathcal{L})$.

Для получения множества всех эйлеровских путей графа можно применить два подхода. Один основан на соотношении между множеством всех остовных деревьев и множеством всех эйлеровских путей графа (этот подход описан в [144]) и том факте, что множество всех остовных деревьев можно удобно перенумеровать. Мы не будем здесь следовать этому предложению. Другой подход основан на предположении, сделанном Укконеном [440] и доказанном (в более широком контексте) Певзнером [372]. В этом подходе преобразуется любая совместимая строка в любую другую совместимую строку последовательностью простых строковых операций, так что каждая промежуточная строка также совместима. Это обеспечивает своего рода непрерывность множества строк, совместимых с данными о гибридизации.

Определение. Пусть S — строка $\alpha\beta\gamma\delta\epsilon$, где каждая греческая буква обозначает подстроку S , и подстроки β и δ непусты. *Строковой ротацией* на S называется операция на S , которая производит строку $\alpha\delta\gamma\beta\epsilon$. Значит, непустые подстроки β и δ поворачиваются вокруг (возможно, пустой) подстроки γ .

Например, возьмем совместимую строку $S = ACACGCAACTTAA$ из предыдущего примера. Рассмотрим конкретный выбор $S = AC A C GCA ACTTAA$, где пробелы разделяют выбранные подстроки, так что $\alpha = AC$, $\beta = A$, $\gamma = C$, $\delta = GCA$ и $\epsilon = ACTTAA$. При таком выборе строковая ротация S создает строку $S' = AC GCA C A ACTTAA$. Как видно на рис. 16.21, S' также задается эйлеровским обходом $G(\mathcal{L})$, и следовательно, строка S' также совместима. Ниже мы докажем, что это не случайно.

Отметим, что строковая ротация является обобщением *циклической ротации* (см. упражнение 1 на с. 33). В частности, циклическая ротация строки S является строковой ротацией, в которой α , γ и ϵ все пусты, так что $S = \beta\delta$. Таким образом, строка $\delta\beta$ получается циклической ротацией S . Можно теперь сформулировать следующую теорему непрерывности для совместимых строк.

Теорема 16.18.2. (Теорема непрерывности SBH.) Пусть S и S' — любые две различные строки, совместимые с данными о гибридизации. S можно преобразовать в S' такой последовательностью строковых ротаций, что каждая промежуточная строка в этой серии сама будет совместима с данными о гибридизации.

Доказательство. Пусть P и P' — эйлеровские пути в $G(\mathcal{L})$, которые определяют, соответственно, строки S и S' . Будем преобразовывать P в P' по последовательности промежуточных эйлеровских путей, используя операции, которые отвечают строковым ротациям.

Для начала, если первые вершины P и P' различны, будем делать циклические ротации S так, чтобы получающаяся строка S'' задавалась эйлеровским путем P'' , имеющим ту же первую вершину, что и P' . Чтобы доказать возможность этого, отметим, что если $G(\mathcal{L})$ не содержит эйлеровского обхода, но содержит эйлеровский путь, то $G(\mathcal{L})$ имеет только одну вершину, в которую больше дуг входит, чем выходит (и единственную вершину, из которой больше дуг выходит, чем входит). Следовательно, с этой единственной вершиной должен начинаться любой эйлеровский путь в $G(\mathcal{L})$. Таким образом, если P и P' действительно начинаются с различных вершин, в графе должен быть эйлеровский обход. Но тогда в любой вершине число входящих дуг равно числу выходящих, и любой эйлеровский путь должен на самом деле быть эйлеровским обходом. В частности, и P должен быть эйлеровским обходом.

Определим P'' как эйлеровский обход (путь), который начинается в той же вершине v , что и P' , и затем обходит дуги $G(\mathcal{L})$ в том же (циклическом) порядке, что и P . Стока S'' , определяемая P'' , является результатом циклической ротации строки S . Более подробно, пусть α — префикс S , заданный путем P вплоть до точки, где P впервые входит в v , и γ — остаток S (указанный остатком P). Путь P'' задает строку $S'' = \gamma\alpha$. Поэтому одна начальная циклическая ротация S создает строку с соответствующим эйлеровским путем, которая стартует в той же вершине, что и P' .

Используя приведенный аргумент, будем считать сейчас, что P и P' начинают с одной и той же вершины. Пусть A обозначает начальный подпуть (возможно, не содержащий дуг), на котором P и P' совпадают, и пусть x — конечная точка A . Значит, P и P' совпадают до точки x , но продолжаются дальше уже двумя разными дугами, e и e' соответственно. (Существование такой точки установлено в предыдущем абзаце.) Так как дуга e' не входит в подпуть A , но должна быть частью всего пути P , путь P должен будет еще вернуться в вершину x после прохождения дуги e . Обозначим через B подпуть P , начинающийся дугой e и кончающийся как раз перед прохождением дуги e' . Пусть C обозначает оставшийся подпуть P (рис. 16.22).

Мы утверждаем, что подпути B и C встречаются по крайней мере один раз, скажем, в точке y (рис. 16.23). Чтобы увидеть это, отметим, что эйлеровский путь P' уходит с подпути A вдоль дуги e' , которая лежит на подпуть C , но P' должен дальше

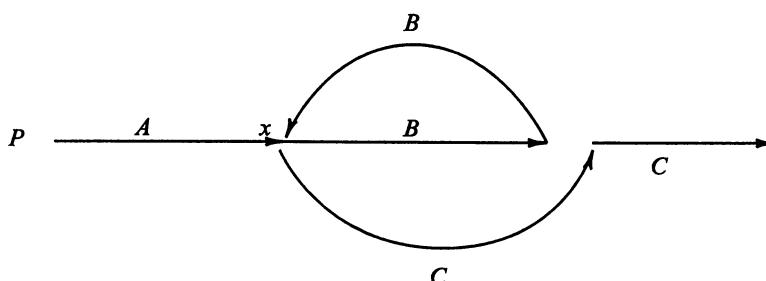


Рис. 16.22. Схема эйлеровского пути P , соответствующего строке S .
Подпути P и P' совпадают до точки x

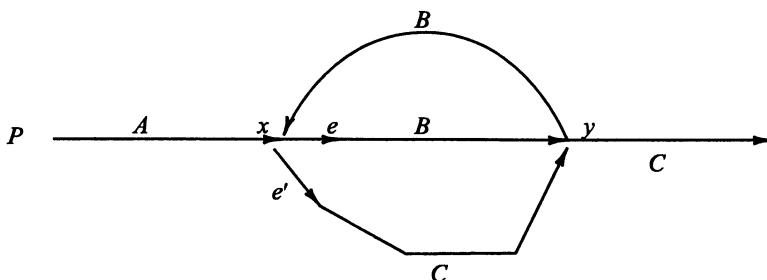


Рис. 16.23. Подпути B и C должны встречаться в некоторой точке y

пройти дугу e , которая лежит на подпуть B . Следовательно, должна быть первая точка на C , где C входит в точку из подпути B . Обозначим эту точку через y .

Из существования y следует, что можно создать эйлеровский путь P'' , который совпадает с P' на подпуть A и на дуге e' . Путь P'' начинается с подпути A , проходит первую часть C до достижения точки y , идет по второй части B от точки y до точки x , затем продолжает первую часть B до y и завершается второй частью C . Путь P'' является эйлеровским путем, содержащим начальный подпуть, который согласуется с P' по крайней мере на одну дугу больше, чем P . Повторяя этот аргумент столько раз, сколько потребуется, мы создадим последовательность эйлеровских путей, начальные подпути которых согласуются с начальными путями P во все большей степени.

Преобразование P в P'' носит общий характер, так как сделано лишь предположение, что P и P' — эйлеровские пути. Назовем это преобразование *ротацией пути*. Можно сказать, что любой эйлеровский путь P в G можно преобразовать в любой другой эйлеровский путь P' в G последовательностью ротаций пути, причем каждый промежуточный путь P'' сам является эйлеровским путем. Чтобы завершить доказательство теоремы, осталось связать ротацию пути со строковой ротацией.

Как соотносятся между собой строки S и S'' , полученные из P и P'' ? Установив существование точки y , мы разбили S на пять подстрок: $\alpha\beta\gamma\delta\varepsilon$, где α — подстрока вдоль подпути A , β — подстрока вдоль подпути B , до достижения точки y , γ — подстрока вдоль оставшейся части подстроки B , δ — подстрока вдоль C до точки y и ε — оставшаяся часть подпути C (рис. 16.24). При этих обозначениях эйлеровский

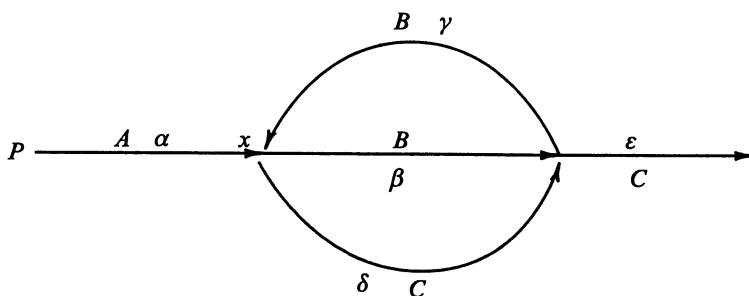


Рис. 16.24. Соотношение эйлеровских путей P и P'' со строками S и S'' . Ротация пути соответствует строковой ротации

путь P'' образует строку $S'' = \alpha\delta\gamma\beta\epsilon$. Следовательно, S'' получена из S одной строковой ротацией. Каждая ротация пути порождает строковую ротацию, и теорема доказана. \square

16.18.3. Последние замечания по поводу SBH

Теорема 16.18.2 очень элегантна и может использоваться дальше для получения информации относительно искомой строки ДНК (результаты такого типа см. в [204, 210]). Однако снова подчеркнем, что теорема относится к идеализированной версии задачи SBH, лишь отдаленно напоминающей реальность.

Первоначально предложенные для полной расшифровки строк ДНК методы матричной гибридизации и гибридизационных чипов в результате недавних исследований и коммерческого развития*) оказались направленными на более скромные цели. Наиболее обещающими кажутся три приложения: *проверка и коррекция последовательностей*, основанная на расшифровке *генетическая диагностика* и *мониторинг экспрессии генов*.

Идея, стоящая за первым приложением, заключается в том, что можно расшифровывать ДНК “тяп-ляп” (quick-and-dirty), используя либо направленные, либо дробовые методы, и затем проводить независимый прогон SBH, пытаясь отметить в последовательности ошибки. “Тяп-ляп” может иметь три смысла: использование для чтения индивидуальных порций длиной значительно выше обычных 300–600 оснований, от чего увеличивается доля ошибок; выполнение расшифровки без дублирования; или расшифровка без внутренних проверок состоятельности и выявления ошибок. Расшифровку “тяп-ляп” следует проверять на согласованность с данными SBH. Если согласованности нет, то данные SBH и получившийся набор решений могут быть использованы для предсказания мест, в которых первую расшифровку нужно дополнительно исследовать.

Второе приложение, имеющее отношение к генетической диагностике, основано на каталогизации известных мутаций, вызывающих генетические заболевания. Зная для данного гена “нормальную” последовательность и зная обычные мутации в нем, вызывающие заболевания, можно создать диагностический чип, который распознавал бы, не содержит ли данный экземпляр ДНК отклонений от нормальной последовательности, и если да, то с какой именно мутацией (см., например, [98]). Получается дешевая альтернатива полной расшифровке индивидуального гена, которую следовало бы предпринять для выяснения того, какие из известных мутаций может нести в своем гене данный индивидуум. В том же ключе сформулирована идея каталогизации в чипе ДНК патогенов человека (или любого множества организмов), чтобы с помощью SBH можно было бы быстро идентифицировать конкретный патогенный организм. В случае бактериальных или вирусных заболеваний этот подход может быть значительно быстрее и дешевле, чем существующие методы идентификации патогенных культур.

Третье приложение обеспечивает средство для изучения центрального вопроса активации и регуляции генов: когда (при каких обстоятельствах) и где (в каких клетках) активируются конкретные гены? Этот критический для биологии вопрос остается без ответа. Матрицы гибридизации можно использовать для того, чтобы

*) Есть две компании, специализирующиеся главным образом на технологии SBH: Affymetrix и Hyseq; обе расположены в Силиконовой долине.

с умеренными затратами определять, какие гены транскрибируются, следя за иРНК, которая производится в данной клетке или органе в любой конкретный момент. Идея в том, что каждый нужный ген будет иметь опознаваемую “сигнатуру” или образец гибридизации, делающий его иРНК или кДНК легко распознаваемой. Одна из трудностей этого подхода связана с тем, что производимое количество иРНК не всегда пропорционально количеству синтезируемого белка — некоторые иРНК высоко стабильны и могут использоваться повторно, для синтеза больше одной молекулы белка, тогда как другие иРНК легко деградируют.

16.19. Упражнения

1. Докажите теорему 16.13.1 на с. 507.
2. Обсуждение дробовой сборки последовательности было упрощением реальной ситуации, и многие детали были опущены. Однако одну сложность нужно упомянуть. Расшифровываемая ДНК состоит из двух скрученных нитей, но обычными методами можно расшифровывать только одну из них. Эти две нити разделяются, и обе разрезаются случайно. Полученный набор фрагментов содержит куски из обеих нитей ДНК, но имеющиеся методы не могут определить, из какой нити происходит конкретный фрагмент. Каждый фрагмент расшифровывается от конца 5' к концу 3', но на одной нити эта расшифровка идет “слева направо”, а на другой — “справа налево”. Итак, задача сборки последовательности на самом деле занимается сборкой двух комплементарных последовательностей для двух нитей по заданному набору фрагментов, покрывающих обе нити. Опять-таки, усложнение в том, что неизвестно, из какой нити произошел каждый конкретный фрагмент. Очевидно, вам не захочется перечислять за экспоненциальное время выборы нитей для каждого фрагмента. Объясните, как развить технику сборки последовательности, описанную в книге, чтобы справиться с этой задачей. Обсудите связь этой задачи с *задачей о надстроке без ориентации*, упомянутой на с. 516.
3. Недавние усилия по расшифровке и сборке фрагментов сосредоточились на получении и использовании дополнительной побочной информации, которая упрощает решение задачи сборки. Один из типов такой информации — изучение для некоторых пар фрагментов, какой фрагмент из пары находится справа от другого. Другой тип побочной информации определяет для некоторых пар фрагментов грубое расстояние между ними. Информация может быть получена использованием частей этих двух фрагментов как праймеров PCR на исходной полной строке с последующим измерением длины расширяющейся подстроки. Третья идея, называемая *двусторонним дробовиком*, заключается в расшифровке или снятии дактилограмм с обоих концов каждого фрагмента. Выигрыш в том, что для каждого фрагмента получается по два отпечатка, и их относительные позиции зафиксированы и известны. Обсудите, как эта информация может быть включена в методы сборки фрагментов и насколько она полезна.
4. **Добавленная кратность покрытия расшифровки.** Для повышения надежности проекты расшифровки иногда находят и расшифровывают больше одного набора фрагментов ДНК, покрывающего изучаемую строку ДНК. Каждый нуклеотид в изучаемой ДНК должен содержаться в нескольких покрытиях, но никакой фрагмент ДНК не должен содержаться больше чем в одном покрытии. Значит, покрытия должны быть фрагментно дизъюнктивны. Предлагаем такую формализацию. Пусть задан параметр k (желаемая кратность покрытия). Выбрать подмножество фрагментов с минимальной полной длиной так, чтобы из этого набора можно было сконструировать k фрагментно дизъюнктивных покрытий. Этую

задачу можно эффективно решить как задачу о *потоке минимальной стоимости* на ориентированном графе. Восполните детали и докажите корректность указанного подхода. Есть много способов и готовых программ для решения задачи о потоке минимальной стоимости. Мы не знаем, используются ли они на практике для этой задачи выбора фрагментов.

5. Докажите теорему 16.8.4 на с. 496.
6. При нахождении самой плотной раскладки клонов двум клонам разрешалось размещаться произвольно близко друг к другу, пока между ними оставалось некоторое ненулевое расстояние. Обсудите, как изменятся задача и решение, если от раскладки потребуется, чтобы расстояние между левыми концами любых двух клонов было не меньше заданного.
7. Останется ли верна лемма 16.17.6, если $|\varphi| + |\varphi'|$ заменить на $\max\{|\varphi|, |\varphi'|\}$? Предложите доказательство или опровергающий пример. Чем бы это помогло, если бы было верно?
8. В алгоритме с множителем четыре для задачи о надстроке и ее анализа мы предположили (не формулируя этого), что набор \mathcal{P}' не содержит вложенных подстрок. Докажите, что это так.
9. Предположим, что мы могли бы вычислить циклическое покрытие минимальной длины, в котором в каждый цикл отображается не меньше трех строк. Уменьшится ли от этого граница оценки с множителя три? Если да, то какая была бы граница?
10. Объясните, почему циклическое покрытие минимальной длины для набора \mathcal{P}' (без требования, что покрытие нетривиальное) просто заново получило бы циклическое покрытие минимальной длины для \mathcal{P} .
11. Докажите лемму 16.17.7 на с. 524.
12. Полностью обоснуйте утверждение, что циклическое покрытие минимальной длины для множества \mathcal{P} можно найти, вычисляя полное назначение максимального веса, когда $A(i, j) = \text{ov}(S_i, S_j)$.
13. **Жадное назначение.** Шаг с доминантным временем в двух приближенных алгоритмах для надстроки, описанных в тексте, — это время на вычисление полного назначения максимального веса. В общей ситуации его можно найти за время $O(k^3)$. Однако, когда веса берутся из суффиксно-префиксных перекрытий, назначение максимального веса и циклическое покрытие можно найти быстрее и проще *жадным методом*.

Жадное назначение

Исходные данные: матрица A размером $k \times k$. (Результат: полное назначение M .)

1. Положить $M = \emptyset$ и объявить все клетки A доступными.
2. while в A есть доступные клетки do begin
 - 2.1. Среди доступных клеток A выбрать клетку (i, j) наибольшего веса.
 - 2.2. Поместить клетку (i, j) в M и сделать клетки в строке i и столбце j недоступными.
- end;
3. Выдать полное назначение M .

Ясно, что этот алгоритм работает со временем $O(k^2 \log k)$ (необходимым для одной только сортировки весов клеток), что много лучше, чем в общем алгоритме для назначений. Докажите, что алгоритм вырабатывает полное назначение.

- 14. Неравенство Монжа.** Пусть u , u' , v и v' — четыре вершины в полном взвешенном двудольном графе $G = (N_1, N_2, E)$, такие что u и u' находятся в N_1 , а v и v' — в N_2 . (Полный двудольный граф — это такой, у которого для любых вершин $u \in N_1$ и $v \in N_2$ найдется дуга из u в v .) Не умаляя общности, предположим, что $A(u, v) \geq \max\{A(u, v'), A(u', v), A(u', v')\}$.

Если $A(u, v) + A(u', v') \geq A(u, v') + A(u', v)$, то говорят, что эти четыре вершины удовлетворяют *неравенству Монжа*. Полный взвешенный двудольный граф называется удовлетворяющим неравенствам Монжа, если неравенство Монжа выполняется для любой четверки вершин — двух из N_1 и двух из N_2 .

Докажите, что, когда значения $A(u, v) = ov(S_u, S_v)$ для всех пар (u, v) используются как веса дуг полного двудольного графа, этот граф удовлетворяет неравенствам Монжа.

- 15.** В общем случае жадное назначение не найдет полного назначения максимального веса. Однако если веса полного двудольного графа удовлетворяют неравенствам Монжа и все веса неотрицательны, то жадное назначение является назначением максимального веса. Докажите это утверждение.

- 16.** Покажите, что жадное назначение не всегда находит нетривиальное полное назначение максимального веса.

- 17.** Для произвольных весов дуг жадный алгоритм найдет полное назначение, вес которого не меньше половины максимального веса. Докажите это.

- 18.** Предположим, что мы модифицировали приближенный алгоритм с множителем три так, что для нахождения нетривиального циклического покрытия используется жадный алгоритм. Это ускорит алгоритм, но, так как жадный алгоритм не гарантирует нахождения нетривиального циклического покрытия максимального веса, граница ошибки три не гарантирована.

Используя границу ошибки $1/2$, полученную в предыдущей задаче, определите, какая (простая) приближенная оценка длины строки будет гарантирована при использовании жадного алгоритма для нахождения нетривиального циклического покрытия.

- 19.** Мы объясняли оба алгоритма, с множителем три и четыре, в терминах задачи назначения, хотя для алгоритма с множителем четыре можно было использовать метод жадного назначения. В оригинальной литературе этот алгоритм излагается без ссылки на задачу назначения. Вместо нее используется алгоритм жадного слияния строк. *Слияние* двух строк α и β определено как строка $\text{pref}(\alpha, \beta)\beta$. Алгоритм *жадного слияния* находит надстроку \mathcal{P} следующим образом:

Скопировать множество \mathcal{P} в множество \mathcal{S} .

repeat

Найти в \mathcal{S} пару строк α, β с наибольшим перекрытием (т. е. наибольшим суффиксно-префиксным совпадением).

Заменить α и β в \mathcal{S} их слиянием.

until \mathcal{S} содержит только одну строку.

Покажите, что нужные суффиксно-префиксные совпадения пар строк в \mathcal{S} можно определить суффиксно-префиксными совпадениями пар строк в исходном множестве \mathcal{P} .

- 20.** Покажите, что алгоритм жадного слияния может быть реализован за время $(k^2 \log k + |\mathcal{P}|)$.

21. Покажите, что алгоритм жадного слияния создает надстроку \mathcal{P} , достигая сжатия \mathcal{P} , которое не меньше половины сжатия, достигаемого кратчайшей надстрокой \mathcal{P} . Этот факт был впервые установлен в [431, 436].
22. Сопоставив алгоритм жадного слияния с алгоритмом жадного назначения, докажите, что алгоритм жадного слияния создает надстроку, длина которой никогда не превосходит длины кратчайшей надстроеки больше чем в четыре раза.
23. Теорема об ОНД (с. 521) может быть усиlena следующим образом.

Теорема 16.19.1. *Если строка S имеет два периода длин p и q и выполняется неравенство $|S| \geq p + q - \text{ОНД}(p, q)$, то S имеет период длины $\text{ОНД}(p, q)$.*

Докажите это усиление теоремы об ОНД.

24. Вспомним определение *полупериодической* строки, данное в п. 3.2.1. Докажите следующее утверждение.

Если никакая строка в \mathcal{P} не полупериодична, то приближенный алгоритм с множителем четыре со с. 520 создает надстроку \mathcal{P} , длина которой не больше чем вдвое превосходит длину кратчайшей надстроеки.

25. Большинство статей по задаче о кратчайшей надстроке причисляет сжатие текстов к приложениям задачи о кратчайшей надстроке. Идея, очевидно, в том, что можно заменить и сохранить набор из n строк \mathcal{S} надстрокой \mathcal{S} , вместе с $2n$ указателями на эту надстроку, которые укажут начало и конец каждой строки в \mathcal{S} . Конечно, так как неизвестно никаких практических алгоритмов вычисления кратчайшей надстроеки, сжатие, достигнутое на практике, будет меньше оптимального.

В качестве альтернативы вычислению надстрок опишите простой метод сжатия n строк из \mathcal{S} с использованием циклического покрытия \mathcal{S} минимальной длины. Объясните, как восстановить исходные строки из сжатых строк.

Напомним, что циклическое покрытие \mathcal{S} минимальной длины можно вычислить эффективно и что оно не больше (а обычно меньше), чем кратчайшая надстрока \mathcal{S} (которая на практике не может быть вычислена). В свете этих фактов будет ли сжатие текстов интересным или убедительным приложением задачи о надстроке? (Задача предложена Полом Стеллингом.)

26. Вывод из предыдущей задачи таков, что сжатие текстов — это неподходящее (ложное) приложение задачи о кратчайшей общей надстроке. Но предположим, что циклическое покрытие \mathcal{S} минимальной длины используется для получения сжатого представления набора строк. Каждая циклическая строка в покрытии сама будет представлена как *линейная строка*, полученная из циклической строки разрезом в некоторой точке. Так как выбор точек разреза произволен, может быть, их удастся выбрать так, чтобы получающееся множество линейных строк имело короткую надстроку. Это позволило бы дальше сжать циклические строки и, следовательно, исходный набор строк.

По существу описанная идея приводит к следующей задаче. Задан набор линейных строк. Можно вращать любую из этих строк на любой шаг. Как повернуть их, чтобы получающаяся общая надстрока была кратчайшей? Или как их повернуть, чтобы получить наилучшую, эффективно вычисляемую гарантию длины кратчайшей надстроеки?

Изучите и обсудите эту задачу. Рассмотрите сначала случай двух линейных строк. Этот частный случай имеет простое решение. Можете ли вы доказать в общем случае предел размера возможного сжатия в сравнении с длиной минимального циклического покрытия, когда линейные строки можно вращать?

27. Докажите теорему 16.18.1 на с. 529.

28. Легко определить, существует ли эйлеровский путь в связном ориентированном графе G . В контексте задачи SBH это позволяет нам эффективно определить, совместимы ли данные с некоторой строкой. Но для наибольшей эффективности мы хотели бы знать, определяют ли эти данные *единственную* строку. Дайте эффективный алгоритм (с линейным временем) для определения того, существует ли в ориентированном графе G *единственный* эйлеровский путь.
29. Предположим, что эксперименты с гибридизацией могут определить, сколько раз каждая строка из \mathcal{L} входит в исследуемую строку. Как эту информацию включить в метод решения?

Строки и эволюционные деревья

Доминирующий взгляд на эволюцию жизни состоит в том, что все существующие организмы произошли от некоторого общего предка и что все новые виды появляются не от смешения двух популяций в одну, а в результате разделения одной популяции на две (или больше), которые дальше не скрещиваются. Поэтому на высшем уровне история жизни идеально организована и представлена как *ориентированное дерево с корнем*. Нынешние виды (и некоторые из ушедших видов) изображаются листьями этого дерева, каждая внутренняя вершина обозначает точку, в которой два множества видов разошлись (общего предшественника этих видов), длина и направление каждой дуги — переход во времени или события эволюции, произошедшие за это время, так что путь от корня дерева до каждого листа символизирует историю эволюции представленных в нем организмов. Цитируем Дарвина:

“…великое Дерево Жизни заполняет земную кору своими мертвыми и сломанными ветвями и покрывает поверхность вечно ветвящимися и прекрасными отростками” [119].

Такой взгляд на историю жизни как на дерево зачастую корректируется при обсуждении эволюции вирусов, или даже бактерий, или отдельных генов, но он остается доминирующим взглядом на эволюцию организмов высшего уровня в современной биологии. Ежегодно публикуются сотни (а может быть, и тысячи) статей, приводящих варианты найденных эволюционных деревьев. Несколько биологических журналов специализируются в основном на таких деревьях и методах их получения.

Все больше методов создания эволюционных деревьев записывается в компьютерные программы, а полученные из этих программ данные основываются прямо или косвенно на данных о молекулярных последовательностях. В этой главе мы

обсудим некоторые математические и алгоритмические вопросы, возникающие при построении эволюционного дерева, и взаимоотношения методов построения деревьев с алгоритмами, работающими со строками и последовательностями.

Биологические противоречия

В мире биологической систематики (классификации) и эволюционной биологии имеются три основные конкурирующие друг с другом теории о том, что должны означать классификационные деревья и как их следует конструировать: *эволюционная систематика, фенетика, или численная систематика, и кладистика*. Споры между сторонниками разных теорий весьма интенсивны и зачастую озадачивают тех, кто не связан с эволюционной биологией непосредственно. Хотя техническая сторона построения деревьев может показаться предметом чистой теории графов и комбинаторной оптимизации, фундаментальные вопросы, которые определяют обоснованность этих методов, иногда обсуждаются в терминах, больше подходящих для религии. Проницательную и (совершенно) непредвзятую оценку некоторых из этих древесных баталий можно найти в [380]; математическую трактовку отдельных вопросов — в [148, 149, 151]. И конечно, более подробное обсуждение того, как реконструировать историю эволюции, проходит через построение деревьев, с тем аргументом, что некоторые важные аспекты молекулярной эволюции не древовидны вообще (например, см. [430]).

Алгоритмы построения деревьев

Несмотря на все дебаты о биологическом основании построения деревьев, большинство алгоритмов их создания можно классифицировать по двум главным категориям: методы, основанные на расстояниях, и методы максимальной бережливости.*¹) В методах, *основанных на расстояниях*, исходные данные состоят из эволюционных расстояний (таких, как редакционное расстояние от последовательностей, температура плавления гибридных ДНК, сила перекрестных реакций антител и др.), их цель — воссоздать взвешенное дерево, в котором попарные расстояния “согласованы” с данными эволюционными расстояниями. Когда данные о расстояниях являются *ультраметрическими*, задача имеет элегантное решение, подробно описанное в п. 17.1. Когда данные не ультраметрические, но *аддитивные*, задача имеет эффективное решение, описанное в п. 17.4, оно основано на алгоритме для ультраметрических расстояний. Встречаются реальные задачи, в которых данные не аддитивны. В таком случае нужно найти дерево, расстояния в котором “лучше всего приближают” исходные данные. В литературе по вычислительной биологии предложены разные определения “степени приближения” и придумано много (обычно эвристических) алгоритмов, которые строят деревья, исходя из этих определений. Но в целом проблема остается открытой, так как нет подхода, который бы и приводил к гарантированно эффективному алгоритму, и следовал полностью приемлемому определению “хорошего приближения”.

*¹) Фельсенштейн [150] пишет: “В реконструкции молекулярных филогенетических деревьев доминируют два вычислительных метода: бережливость и расстояние. Метод бережливости находит эволюционное дерево, которое требует наименьшего числа изменений нуклеотидов для объяснения эволюции наблюдаемых последовательностей. Методы расстояний вычисляют попарные величины различий между последовательностями и пытаются согласовать их с ожидаемыми попарными расстояниями, вычисленными по дереву”.

Методы *максимальной бережливости* используют другой подход и не сводят биологических данных к расстояниям. Методы бережливости являются *символно-ориентированными* и работают прямо с *символьными данными*, очень часто используя выровненные последовательности.*) Нужно построить дерево с исходными таксонами в листьях и предположительными таксонами во внутренних вершинах, которое минимизировало бы полное количество или цену “мутаций”, подразумеваемых этой историей эволюции. (*Минимизация мутаций* часто описывается в литературе как *максимизация бережливости*.) Когда каждый исходный таксон представлен молекулярной последовательностью, задача о максимальной бережливости обобщает задачу о филогенетическом выравнивании, рассмотренную в главе 14. Задача о бережливости ищет дерево с листовыми пометками, в котором филогенетическое выравнивание имеет минимальную цену среди всех возможных деревьев.

Алгоритмические подходы к построению филогенетических деревьев зародились в статье Фитча и Марголиаша [160]. Читатель, интересующийся философией и техникой реального построения деревьев, практикуемой в современной молекулярной и эволюционной биологии, может обратиться к [380] за философией, к [223, 429] за техникой, к [164] за их комбинацией и к [127] за проницательными комментариями. Приложения этих методов к конкретным исследованиям по филогенетике и систематике повсеместны и часто отражаются популярной прессой (см. недавний пример в [78]). Более подробные статьи в популярной прессе (относительно происхождения большой панды и красного волка) см. в [351] и [470]. Иллюстрации из представительных статей исследовательского уровня см. в [207, 314, 131].

Алгоритмы для строк и деревьев

Даже из этого краткого описания методов расстояния и бережливости должно быть ясно, что строковые алгоритмы типа рассмотренных в нашей книге играют важную роль в окончательном успехе или провале методов эволюционного вывода. Этот вывод прямо получается в случае методов максимальной бережливости, которые оперируют с сырьими последовательностями или с символами, полученными из последовательностей (возможно, множественно выровненными). Он менее непосредственен в случае методов расстояния, так как последние оперируют с числами. Однако зачастую исходные данные для метода, основанного на расстояниях, берутся из какого-либо конкретного строкового алгоритма, который находит парные расстояния или вычисляет множественные выравнивания.**) Таким образом, и модель расстояния, или множественного выравнивания, заложенная в алгоритм (глобальная, локальная, типа пропусков, взвешенная, невзвешенная, с суммами пар, согласительная, с переменной установкой параметров и др.), и качество алгоритмического решения могут значительно влиять на надежность окончательного дерева эволюции.

В следующих параграфах мы представим некоторые идеализированные варианты задач построения деревьев, основанные на расстоянии и на максимальной бережливости. Предпочтение отдается “элегантному и доказанному”, а не “реалистичному и практичному”, и результаты могут показаться слишком абстрагированными

*) Слово “символ” здесь обозначает наблюдаемую черту или характеристику, которая иногда может быть символом алфавита ДНК или аминокислот, но это не обязательно.

**) В некоторых исследованиях по реконструированию используются парные расстояния индуцированных парных выравниваний, взятых из множественного выравнивания всех последовательностей.

от реальности, чтобы иметь прямое практическое применение. Более того, мы сосредоточимся на комбинаторных аспектах построения деревьев, даже несмотря на ту большую теоретическую и практическую роль, которую имели в этой области статистические вопросы [148, 149, 151], и на все более широкое использование методов максимального правдоподобия. Задачи и результаты, которые мы рассматриваем, служат введением в эту область и ясно показывают модели идеального мира, которые обосновывают наиболее практические методы построения деревьев и приводят к ним.

17.1. Ультраметрические деревья и ультраметрические расстояния

17.1.1. Введение

Начнем с обсуждения *ультраметрических деревьев и расстояний* — конструкций, которые можно использовать в эволюционном восстановлении, когда данные “идеально соответствуют” некоторым строгим предположениям. Даже если соответствие не идеально, ультраметрические деревья неявно возникают во многих методах реконструкции дерева, основанных на числах. Ультраметрические деревья или их приближения можно использовать и для вывода шаблонов ветвления в истории эволюции и в какой-то степени для измерения времени прохождения вдоль каждой ветви.

Определение. Пусть D — симметричная матрица вещественных чисел размером $n \times n$. *Ультраметрическим деревом* для D называется дерево с корнем T , обладающее следующими свойствами:

1. T имеет n листьев, каждый из которых помечен единственной строчкой из D .
2. Каждая внутренняя вершина T помечена одним элементом D и имеет не менее двух детей.
3. Вдоль каждого пути от корня до листа числа, помечающие внутренние вершины, *строго убывают*.
4. Для любых двух листьев i, j из T значение $D(i, j)$ равно метке наименьшего общего предка i и j в T .

Таким образом, ультраметрическое дерево для D (если такое существует) является компактным представлением матрицы D (см. пример на рис. 17.1).

a)

	a	b	c	d	e
a	0	8	8	5	3
b		0	3	8	8
c			0	8	8
d				0	5
e					0

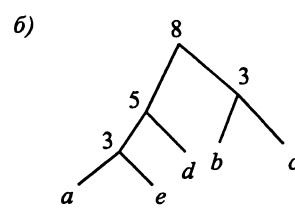


Рис. 17.1. a — симметричная матрица D ; б — ультраметрическое дерево для матрицы D

Определение. Дерево с корнем T называется *min-ультраметрическим деревом* для D , если оно обладает всеми свойствами ультраметрического дерева, но свойство 3 заменено следующим: вдоль любого пути от корня до листа метки внутренних вершин должны *строго возрастать*.

Для того, что мы определили термином “*min-ультраметрическое*”, общепринятое термина нет, и это обстоятельство иногда вызывает путаницу. Отметим, что не каждая матрица D обязательно имеет представляющее ее ультраметрическое или *min-ультраметрическое* дерево. Ультраметрическое или *min-ультраметрическое* дерево включает в себя не больше $n - 1$ внутренних вершин, так что если D имеет больше чем $n - 1$ различных значений, то ультраметрического или *min-ультраметрического* дерева для D существовать не может.

Основные математические и алгоритмические задачи относятся к характеристике условий, при которых матрица D может быть представлена ультраметрическим или *min-ультраметрическим* деревом и к развитию эффективного алгоритма для построения таких деревьев, когда это возможно. Мы детально изучим обе задачи после установления связи ультраметрических деревьев с эволюционными деревьями.

17.1.2. Эволюционные деревья как ультраметрические деревья

Если истинная история эволюции n таксонов образует ориентированное дерево с корнем T , листьями которого являются нынешивущие таксоны, то каждая внутренняя вершина v из T представляет *событие дивергенции*, или историческое ветвление. Событие дивергенции — это момент времени, когда истории эволюции (не менее) двух таксонов, скажем, i и j , разошлись. Для простоты мы будем говорить, что v — это момент, когда “ i и j разошлись”. Было бы поспешно утверждать, что один из них является предком другого или оба — потомки некоторого другого (возможно, недожившего) таксона. Верно только, что до точки v история эволюции i и j была общей.

Предположим, что кроме топологии дерева (или *порядка ветвления* на языке *кладистики*) известно время (абсолютное или относительное) наступления каждого события дивергенции. Если их записать в вершинах дерева, то вдоль каждого пути от корня эти времена должны строго возрастать. Более того, если вершина v является наименьшим общим предком листьев i и j в T , то меткой вершины v будет время дивергенции i и j . Таким образом, T является *min-ультраметрическим* деревом для матрицы D размера $n \times n$, в которой для каждой пары листьев i и j значением $D(i, j)$ будет время дивергенции i и j . Таким образом истинная история эволюции (ориентированное дерево плюс времена дивергенции) образует *min-ультраметрическое* дерево для данных о временах попарных дивергенций.

Равным образом, если известна истинная история эволюции, можно пометить каждую вершину v из T временем, протекшим от события дивергенции, представленного вершиной v . При таком способе пометки числа строго убывают вдоль пути от корня. Таким образом, если $D(i, j)$ теперь представляет время, прошедшее от дивергенции i и j , то этот второй способ пометки делает T ультраметрическим деревом для D . Как мы увидим ниже, реальные биологические данные обычно приближают время, прошедшее от дивергенции, чем время дивергенции, так что сосредоточиться на ультраметрических деревьях будет более естественно.

Теперь настоящая проблема заключается в том, что мы не знаем истинной истории эволюции ни деревьев, ни времен дивергенции. Скорее, мы хотим воссоздать правдоподобную историю из данных, отражающих время от дивергенции. Зная, что истинная история эволюции должна образовывать ультраметрическое дерево, мы ставим задачу сформировать ультраметрические деревья из данных о времени до дивергенции. Немедленно порождается вопрос: даже если есть ультраметрическое дерево T для матрицы данных о дивергенциях D , насколько можно верить в то, что T действительно отражает истинную историю эволюции, которую мы ищем, а не что-то иное? Мы займемся этим вопросом, а также основным алгоритмическим вопросом, как строить ультраметрические деревья, в следующем пункте. Затем мы вернемся к нахождению биологических данных; этот вопрос при построении ультраметрических деревьев наиболее интересен.

17.1.3. Как тестировать ультраметрические деревья

В этом пункте мы сформулируем и докажем основную теорему относительно ультраметрических деревьев и расстояний и предложим эффективный алгоритм, который строит ультраметрические деревья, когда они существуют.

Определение. Вещественная симметричная матрица D определяет *ультраметрическое расстояние* в том и только том случае, когда для любых трех индексов i , j и k имеется равенство (или ничья — tie) максимумов из $D(i, j)$, $D(i, k)$ и $D(j, k)$. Это значит, что максимум из этих трех чисел *не единственен*. Аналогично, D определяет *min-ультраметрическое расстояние* в том и только том случае, если для любой тройки i , j и k есть ничья минимумов $D(i, j)$, $D(i, k)$ и $D(j, k)$.

Когда D определяет ультраметрическое расстояние, мы будем часто называть *матрицу D ультраметрической*. Сходным образом используется далее и термин “*min-ультраметрическая матрица*”.

Легко видеть, что если D имеет ультраметрическое (или *min-ультраметрическое*) дерево, то матрица D — ультраметрическая (или *min-ультраметрическая*). Дальнейшие объяснения см. на рис. 17.2. Обратное утверждение не так очевидно, но мы сейчас его докажем.

Теорема 17.1.1. *Чтобы симметричная матрица D имела ультраметрическое дерево (или *min-ультраметрическое дерево*), необходимо и достаточно, чтобы она была ультраметрической (или *min-ультраметрической*).*

Доказательство. Необходимость уже показана на рис. 17.2, так что остается доказать, что из ультраметричности матрицы D следует существование ультраметрического дерева для D . Это доказательство конструктивно и ведет к эффективному алгоритму построения ультраметрических деревьев, когда они существуют. Случай *min-ультраметрических* матриц и деревьев аналогичен и оставляется читателю.

Чтобы построить ультраметрическое дерево T по ультраметрической матрице D , сначала возьмем отдельный лист, скажем, лист i . Для простоты предположим, что $D(i, i) \neq D(i, j)$ для всех $j \neq i$.*) Если в строке i матрицы D есть d различных

*) Определение “ультраметричности” обычно требует, чтобы $D(i, i) = 0$, но для общности мы от этого условия в определении освободились.

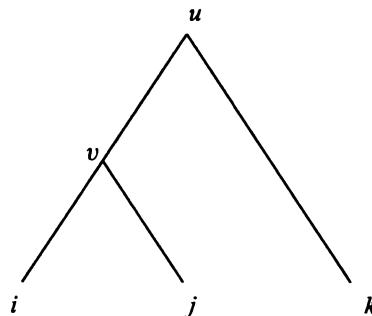


Рис. 17.2. Предположим, что D имеет ультраметрическое дерево. Здесь показана принципиальная схема поддерева, содержащего листья i, j, k . Поддерево может иметь и другие вершины, но они не показаны. Вершина v является наименьшим общим предком i и j , а вершина u — наименьшим общим предком трех листьев i, j и k . Так как это ультраметрическое дерево для D , число в вершине u должно быть строго больше v . По определению число v равно $D(i, j)$, а число в u равно $D(i, k) = D(j, k)$. Следовательно, эти три числа удовлетворяют условию ничьей: максимум из $D(i, j)$, $D(i, k)$ и $D(j, k)$ не единственен. Индексы i, j, k произвольны, так что картина носит общий характер и показывает, что если D имеет ультраметрическое дерево, то матрица D — ультраметрическая

элементов, то любое ультраметрическое дерево T для D должно содержать путь от корня до листа i с ровно d вершинами. Более того, каждая вершина должна быть помечена одним из d различных чисел из строчки i , и метки должны стоять на пути в порядке убывания. Итак, вершины и метки на пути к листу i определяются только элементами строчки i матрицы D . Кроме того, любая внутренняя вершина v на пути с пометкой $D(i, j)$ должна быть наименьшим общим предком листьев i и j . Это определяет место, где лист j должен появиться в T относительно пути до листа i (рис. 17.3). Таким образом, путь к листу i разбивает оставшиеся $n - 1$ листьев (вершины, отличные от i) на $d - 1$ классов. Назовем это разбиение \mathcal{D} . Листья j и k находятся в одном классе \mathcal{D} в том и только том случае, когда $D(i, j) = D(i, k)$. Отсюда следует, что каждый класс в \mathcal{D} задается отдельной вершиной на пути до i .

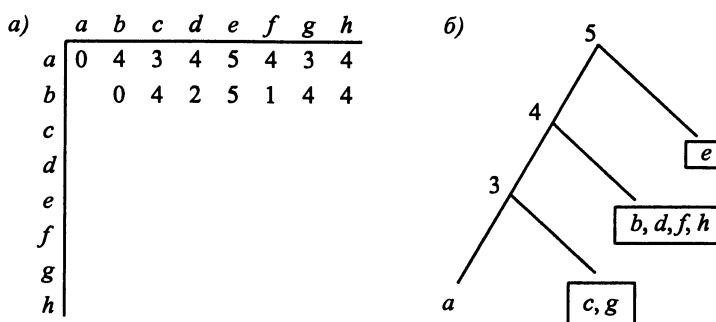


Рис. 17.3. *a* — две строчки симметричной матрицы D . Строчка для таксона a используется для получения пути к листу a , показанному на рисунке. *б* — номера вершин на этом пути разбивают остальные таксоны, как показано

В частности, вершина, которая определяет класс, содержащий j , — это вершина с пометкой $D(i, j)$.

При заданном разбиении \mathcal{D} , определенном путем до i , мы бы хотели найти ультраметрическое дерево для каждого из этих $d - 1$ классов в \mathcal{D} и затем присоединить каждое полученное поддерево к той вершине на пути до i , которая этот класс определила. Значит, мы хотели бы решить задачу об ультраметрическом дереве рекурсивно для каждого класса, а затем соединить эти деревья и получить ультраметрическое дерево для полной матрицы D . Покажем, что если D — ультраметрическая матрица, то этот подход работает правильно.

Рассмотрим класс, определенный внутренней вершиной v . Пусть в этом классе содержится лист j , а l — какой-то другой лист. Для положения l в разбиении возможны три случая: в том же классе, что и j , в классе, расположенному между листом i и вершиной v , или в классе между вершиной v и корнем дерева T . Например, на рис. 17.3, если i — это a и j — это b , то $l = d$ соответствует первому случаю, $l = g$ — это второй случай, а $l = e$ — третий случай. В первом случае $D(i, j) = D(i, l)$, поэтому $D(j, l) \leq D(i, j)$, так как матрица D ультраметрическая. Это означает, что если ультраметрическое поддерево, содержащее листья j и i , прикреплено в v , то $D(j, l)$ представлено в новом дереве корректно, и дерево обладает тем требуемым свойством, что номера вершин строго убывают вдоль любого пути от корня. (Если $D(j, l) = D(i, j)$, то степень вершины v будет больше двух, как это показано на рис. 17.3 при $l = h$.) Во втором случае $D(i, l) > D(i, j)$, так что $D(j, l) = D(i, l)$. Поэтому если ультраметрическое дерево для класса, содержащего j , присоединено в v , то $D(j, l)$ будет правильно приписано к общему наименьшему предку листьев j и l . В третьем случае $D(i, l) < D(i, j)$, так что $D(j, l) = D(i, j)$, и v должна быть общим наименьшим предком j и l . Следовательно, во всех трех случаях ультраметрическое дерево класса, определенного v , может быть правильно прикреплено к v . Поэтому, когда матрица D ультраметрическая, этот рекурсивный подход корректно конструирует ультраметрическое дерево для D . \square

Дополнительные следствия

Из доказательства теоремы 17.1.1 непосредственно получается несколько важных фактов.

Теорема 17.1.2. *Если D — ультраметрическая матрица, то ультраметрическое дерево для нее единственно.*

Чтобы убедиться в этом, отметим, что в доказательстве корректности рекурсивного подхода мы видели, что классы в \mathcal{D} , определенные путем до вершины i , были вынужденными. Значит, вершины на пути до i и их метки должны появляться в любом ультраметрическом дереве для D . Более того, разбиение, определенное этим путем, и позиция каждого конкретного класса также вынужденные и должны располагаться в одном и том же порядке в любом ультраметрическом дереве для D . Единственность прямо следует из этих фактов, что будет существенно в следующем пункте. Единственность для min-ультраметрического дерева получается точно так же.

Следует предупредить о возможном источнике недоразумений, связанном с единственностью. Ультраметрические деревья иногда определяются без требования, чтобы номера вершин строго убывали вдоль любого пути от корня. Если эта “строгость” отсутствует (вопреки используемым нами здесь определениям) и какая-то

вершина ультраметрического дерева имеет больше двух детей, то можно получить другое ультраметрическое дерево, заменяя эту вершину цепью. Тогда теорема 17.1.2 не выполняется. Однако, даже если “строгости” не требовать, все ультраметрические деревья для D очень тесно связаны друг с другом. Читателю оставляется возможность разработать этот вопрос.

Тот факт, что ультраметрическое дерево для ультраметрической матрицы единственно, отвечает на вопрос, поставленный ранее в п. 17.1.2: насколько информативно ультраметрическое дерево, полученное из D ? То есть мы пытаемся образовать неизвестное эволюционное дерево T , которое считается причиной появления наблюдаемых данных в D . Как соотносится дерево, полученное из D , с этим неизвестным T ? Ответ такой: если данные в D действительно пропорциональны числам, помечающим вершины неизвестного эволюционного дерева T , то ультраметрическим деревом, полученным из D , должно быть T . Это очень сильное и желательное свойство.

Из теоремы 17.1.1 следует, что ультраметрическое дерево для ультраметрической матрицы D можно построить за время $O(n^2 \log n)$. Первоначально мы формулировали в качестве вывода из теоремы 17.1.1 следующую теорему.

Теорема 17.1.3. *Если D — ультраметрическая матрица, то ультраметрическое дерево для D можно построить за время $O(n^2)$.*

Ссылка на теорему 17.1.1 оказалась неверной, но сам факт верен. (Ниже предлагается другое доказательство, взятое из интернетовского сайта автора^{*}) и вставленное в основной текст при переводе.)

По матрице D построим граф G , вершины которого соответствуют строчкам D , и каждой дуге (i, j) сопоставляется значение $D(i, j)$. За время $O(n^2)$ следующий алгоритм найдет очень специфичный путь в графе G :

```

Положить  $N$  равным множеству всех чисел от 1 до  $n$ .
Положить  $L$  равным пустому пути.
Произвольно выбрать вершину в качестве  $i$ .
repeat  $n - 1$  times begin
    удалить  $i$  из  $N$ ;
    найти такой индекс  $j \in N$ , чтобы
         $D(i, j) \leq D(i, k)$  для всех  $k \in N$ 
    поместить дугу  $(i, k)$  в путь  $L$ ;
    положить  $i$  равным  $k$ ;
end;
```

В результате получается путь L из ровно n дуг, и алгоритм исполняется за время $O(n^2)$. Оказывается, что L является минимальным остовным деревом графа G , но этот факт не требуется.

Используем теперь L для рекурсивного построения ультраметрического дерева.

Возьмем на пути L с наибольшим весом дугу (p, q) . Пусть P — множество вершин слева от p на этом пути (включая p), а Q — множество дуг справа от q (включая q). Из ультраметричности матрицы D следует, что $D(i, j) = D(p, q)$ для любой пары вершин (i, j) , где $i \in P$ и $j \in Q$. Это можно доказать индукцией по числу дуг между i и j в L (применяя условие ничьей из определения ультраметричности). Отсюда следует, что в ультраметрическом дереве, которое мы строим

^{*}) См. www.cs.ucdavis.edu/~gusfield/ultraerrat/ultraerrat.html

(и в любом ультраметрическом дереве для D), общий наименьший предок любых двух листьев (i, j) , где $i \in P$ и $j \in Q$, должен быть корнем ультраметрического дерева и пометка корня должна равняться $D(p, q)$.

Если при определении дуги с максимальным весом обнаруживается $k > 1$ нильевых, то удаление этих k дуг создает $k + 1$ путей из вершин, и применение того же рассуждения к любым вершинам i и j из разных подпутей показывает, что их общий наименьший предок должен лежать в корне дерева, который опять-таки должен иметь пометку $D(p, q)$. Следовательно, любое ультраметрическое дерево для D имеет ровно $k + 1$ дуг, выходящих из корня, и множество листьев ниже каждой такой дуги в точности совпадает с множеством вершин в одном из $k + 1$ подпутей.

Независимо от значения k удаление k дуг максимального веса из L и разбиение N занимают время только $O(n)$. Продолжая описание алгоритма, предположим для удобства, что $k = 1$. Пусть LP и LQ обозначают два подпути, получившиеся после удаления из L дуги максимального веса. Теперь мы хотим найти ультраметрическое дерево для P и отдельно для Q ; затем эти два ультраметрических дерева будут прикреплены к корню, чтобы получилось полное ультраметрическое дерево для D . Но заметим, что мы уже располагаем нужными частями LP и LQ , которые получились бы, если бы мы применили рекурсивно тот же метод (ясно, что LP получилось бы, если бы мы применили алгоритм отдельно к P , и аналогично для LQ и Q). Таким образом, нам нужно только найти дугу (дуги) максимального веса в LP и отдельно в LQ . Эти дуги можно найти за полное время $O(n)$. Снова, так как множество дуг было на первом шаге разделено, эта временная оценка верна и для $k > 1$.

Продолжая, мы построим ультраметрическое дерево за полное время $O(n^2)$. \square

Легко также установить, что проверить, является ли D ультраметрической матрицей, можно также за время $O(n^2)$. Мы оставляем это как упражнение.

17.1.4. Откуда берутся ультраметрические данные?

Из теоремы 17.1.2 следует, что ультраметрические данные по “времени-от-дивергенции” — это очень мощное средство воссоздания истории эволюции как топологии дерева, так и относительных времен дивергенции. Но есть ли реальная надежда добиться ультраметрические данные?

Теория молекулярных часов

Теория молекулярных часов, предложенная в начале 1960-х годов Эмилем Цюкеркандлом и Лайнусом Полингом [489] (см. также [279]), утверждает, что для любого данного белка *приемлемые мутации* в его аминокислотной последовательности (и в лежащей в основе ДНК) возникают с постоянной интенсивностью. Принятый здесь термин означает, что мутации сохраняют способность белка к нормальному функционированию (т. е. они для белка не летальны; см. п. 15.7.2). Отсюда заключение, что число приемлемых мутаций, возникающих в любом временном интервале, пропорционально длине интервала. Следовательно, после калибровки этих часов длину неизвестного интервала можно измерять количеством приемлемых мутаций, в нем произошедших. Конечно, на эту теорию несколько влияет “зернистость”, и она приемлема только для “достаточно больших” интервалов времени. Вдобавок, интенсивность приемлемых мутаций различна для различных белков (т. е. у них разные часы). Например, гемоглобин мутирует быстрее, чем цитохром c , но оба они

относительно устойчивы (и очень похожи у всех млекопитающих) по сравнению с другими белками, например *фибринопептидами* [11]. На самом деле, разные части белка могут эволюционировать с разной интенсивностью [127], так что такими часами нужно пользоваться с осторожностью. За теорией молекулярных часов стоит предположение, что мутации (приемлемые или нет) во всех ДНК появляются с некоторой постоянной интенсивностью. Различия в интенсивностях приемлемых мутаций происходят из различий в том, насколько ограничены конкретные белки (естественной селекцией на уровне организма или физической химией на молекулярном уровне).

По поводу молекулярных часов было пролито немало чернил (и крови), и эта теория тесно связана с другим важным пунктом противоречий в эволюционной биологии — с *нейтральной теорией молекулярной эволюции*, развитой Мотоо Кимурой [275] и др. Она касается важности для эволюции естественной селекции по сравнению со случайным генетическим дрейфом. Есть два интересных обсуждения нейтральной теории, одно Кимуры, а другое Стивена Джая Гульда [274, 188].

Очевидно, что молекулярные часы упрощают задачу сбора ультраметрических данных. Пусть *A* и *B* — два таксона, которые синтезируют и применяют “один и тот же” белок (например, гемоглобин). Предположим, что произошло *k* приемлемых мутаций в последовательностях ДНК или аминокислот гемоглобинов таксонов *A* и *B* с того времени, как они дивергировали. (По теории молекулярных часов отсюда следует, что за это время примерно по *k*/2 отдельных приемлемых мутаций произошло в каждой из этих двух историй эволюции.) Если число приемлемых мутаций (или некоторое число, ему пропорциональное) можно сопоставить каждой паре из *n* изучаемых таксонов, то эти C_n^2 чисел пропорциональны соответствующим временем от момента дивергенции таких пар таксонов. Тогда C_n^2 чисел удовлетворяют требованиям, накладываемым на ультраметрическое дерево, и по теореме 17.1.2 дерево восстанавливает истинную историю эволюции этих таксонов. Итак, теория молекулярных часов сводит задачу нахождения данных о “времени-от-дивергенции” к задаче нахождения числа мутаций между двумя таксонами. Это сведение очень привлекательно, но как получить необходимые данные о мутациях?

Методы, основанные на лабораторных данных

Первые методы, использованные для оценки числа приемлемых мутаций между двумя таксонами, обычно применяли физические/химические средства. В одном подходе ДНК двух таксонов денатурировали (нагреванием), так что двойные нити расплелись, смешивали вместе одиночные нити из обоих источников ДНК, чтобы дать им возможность гибридизоваться, и затем проверяли температуру, при которой расплелись гибридные нити. Идея в том, что чем более похожи два источника ДНК, тем сильнее гибридизация и тем выше должна быть температура для того, чтобы разделить две нити. Предполагается, что температура разделения двух таксонов *A* и *B* (при некоторых подгонках) пропорциональна полному числу приемлемых мутаций, произошедших за время после дивергенции *A* и *B*.

Это значительно упрощенное описание экспериментальной процедуры в основном показывает то, что было сделано в классическом солидном исследовании Сибли и Алквиста [406, 407] по эволюции птиц. Более того, данные этих авторов имели

сильное свойство “самопроверки” — данные оказались почти ультраметрическими.* В свою очередь, ультраметрическая природа их данных оказалась внушительную поддержку теории молекулярных часов, так как ультраметричность трудно объяснить, не принимая этой теории и не веря в то, что получающиеся данные корректно отражают истинную историю эволюции.**)

Методы, основанные на секвенировании

Более современные методы оценки числа приемлемых мутаций основаны на прямой расшифровке ДНК или аминокислотных последовательностей. Для двух таксонов *A* и *B* число приемлемых мутаций, произошедших после их дивергенции, оценивается проверкой различий в ДНК или аминокислотных последовательностях, общих для *A* и *B*. Обычно (см. [127]) оценка связана с *редакционным расстоянием*. (Ситуацию не следует рассматривать как неожиданную; это должно быть редакционное расстояние или что-нибудь подобное, чтобы вопрос соответствовал содержанию книги о строковых алгоритмах.)

В действительности редакционное расстояние должно быть подправлено, потому что данная позиция нуклеотида может муттировать в ходе эволюции несколько раз. Кроме того, некоторые мутации ДНК (в частности, в третьем нуклеотиде кодона) не изменяют аминокислоты, определяемой этим участком ДНК. Но после корректировки факторов (см. [127]) редакционное расстояние между существующими последовательностями ДНК или белков для *A* и *B* используется для оценки полного числа приемлемых мутаций, произошедших с момента дивергенции *A* и *B*. Попарные редакционные расстояния образуют матрицу *D*, и если *D* ультраметрическая, то ее дерево используется как гипотетическая история истинной эволюции таксонов.

В качестве короткого резюме сформулируем, что приняв теорию молекулярных часов, можно использовать редакционное расстояние для оценки числа приемлемых мутаций, произошедших с момента дивергенции любых двух таксонов. Это число должно быть пропорционально действительному времени, прошедшему после их дивергенции. Следовательно, при заданном наборе из *n* таксонов и C_n^2 редакционных расстояниях между парами таксонов ультраметрическое дерево (если оно существует) для этих данных должно давать возможную историю эволюции, объясняющую эти данные. Более того, когда существует ультраметрическое дерево для *D*, оно может быть только одно. Отсюда следует, что если попарные редакционные расстояния можно использовать для получения чисел, пропорциональных истинным попарным данным о времени после дивергенции (согласно теории молекулярных часов), то единственное ультраметрическое дерево для этих данных должно быть истинным деревом эволюции с правильной топологией и длинами дуг, пропорциональными правильным. Далее, если для любой пары известно абсолютное время дивергенции

* На языке эволюционной биологии тест, с помощью которого проверяют ультраметричность данных, называется *тестом относительной скорости*. Он был предложен В. Саричем и А. Уилсоном. Алквист и Сибли пишут: “Мы нашли, что часы ДНК, кажется, тикают с одной и той же скоростью во всех родах птиц. Это наблюдение получено из процедуры, известной как тест относительной скорости. Тест относительной скорости сравнивает все возможные тройки видов...” [406].

**) Нам следует отметить, однако, что химические данные, стоящие за работой Сибли и Алквиста, резко оспаривались [395] наряду с некоторыми вопросами протоколирования их исследований, так что надежность их результатов сомнительна. Альтернативный взгляд на эволюцию птиц появился в [147].

(скажем, по окаменелостям), то молекулярные часы можно “настроить”, обеспечив перевод пропорционально правильных отсчетов времени в правильное абсолютное время. Так это выглядит в идеале.

Последние замечания

Чаще всего реальные данные не ультраметричны, и даже ультраметрические данные не обязательно отражают истинное время, прошедшее от дивергенции. Тем не менее быть ультраметрическими — это свойство данных, которое, по-видимому, появилось неслучайно, так что, когда данные ультраметричны или близки к таковым, это служит серьезным аргументом в пользу теории молекулярных часов и сильным аргументом за то, что данные отражают истинную историю эволюции.

Для неультраметрических данных можно рассмотреть задачу варьирования этих данных с “наименьшей” возможной вариацией, при которой они становятся ультраметрическими. Если возмущения невелики, то данные можно все еще считать согласованными с теорией молекулярных часов. Изучалось несколько вариантов задачи о возмущениях. Рассмотрим вариант, в котором возмущенные данные должны стать ультраметрическими, а возмущения могут только уменьшать первоначальные значения. Для этой задачи имеется решение, в котором каждое значение (одновременно) уменьшается так мало, как возможно, и конечный результат становится ультраметрическим. Это будет использовано в упражнении 21. Когда изменения могут быть любого знака, то существует эффективный алгоритм, который создает ультраметрические данные и минимизирует максимальное изменение одного из значений. Если разрешены только увеличения, то задача NP-трудна. Последние два результата принадлежат Фараху, Кэннану и Уорну [145].

17.2. Деревья с аддитивными расстояниями

17.2.1. Введение

Ультраметрические данные — это “Святой Грааль” филогенетической реконструкции: когда данные о времени до дивергенции ультраметричны, есть вера в то, что можно воссоздать истинную историю эволюции. Но это в основном идеализированная абстракция, и настоящие данные редко ультраметричны. Что можно сделать в таком случае? Более слабое требование к данным об эволюционных расстояниях — *аддитивность*.

Определение. Пусть D — симметричная матрица $n \times n$ с нулевой диагональю и строго положительными внедиагональными элементами. Пусть T — дерево с весами на ребрах, в котором не меньше n вершин, причем n различных вершин T помечены строчками из D . Дерево T называется *аддитивным деревом* для матрицы D , если для любой пары помеченных вершин (i, j) путь от i до j имеет суммарный вес (или расстояние), в точности равный $D(i, j)$.

Значит, T так кодирует матрицу D , что каждый элемент $D(i, j)$ равен расстоянию в T от вершины i до вершины j (см. пример на рис. 17.4).

Задача об аддитивном дереве. Пусть задана симметричная матрица D с нулевой диагональю и положительными всеми остальными элементами. Найти для D аддитивное дерево или установить, что его не существует.

Доказательство. Пусть T — компактное аддитивное дерево для D и $e = (x, y)$ — любое ребро, не входящее в T . Так как цепь в T , соединяющая x и y , должна иметь полный вес, равный $D(x, y)$, и все веса ребер в T строго положительны, $D(x, y)$ должно быть строго больше веса любого ребра из цепи. Используем этот факт для того, чтобы показать, что ребро (x, y) не может входить ни в какое минимальное оствовное дерево для G . Отсюда будет следовать, что T должно само быть минимальным оствовным деревом для G и при этом единственным.

Доказывая от противного, предположим что $e = (x, y)$ входит в некоторое минимальное оствовное дерево \bar{T} . Удаление e из \bar{T} разделит вершины G на две компоненты связности, обозначим их S и \bar{S} . Пусть $x \in S$ и $y \in \bar{S}$; в цепи, соединяющей x и y в T , должно найтись ребро e' , соединяющее вершину из S с вершиной из \bar{S} . Ясно, что e' не принадлежит дереву \bar{T} . По построению e — единственное ребро в \bar{T} , соединяющее S и \bar{S} . Добавим e' к \bar{T} и удалим ребро e . Получится снова оствовное дерево для G , обозначим его через T' . Но суммарный вес ребер в T' меньше, чем в T , так как вес ребра e' , как уже говорилось, меньше веса e . Это противоречит предположению о минимальности оствовного дерева \bar{T} . Следовательно, e не может входить в минимальное оствовное дерево для G , и теорема доказана. \square

Согласно теореме 17.2.1, задачу о компактном аддитивном дереве можно решить за время $O(n^2)$: по заданному D (неявно) построить $G(D)$ и затем запустить любой алгоритм построения минимального оствовного дерева со временем $O(n^2)$ (см., например, [294]), который (для простоты изложения) работает с одним растущим деревом. Предположим по индукции, что алгоритм знает веса цепей $d(i, j)$ для каждой пары вершин (i, j) в текущем дереве. Когда к дереву добавляется новое ребро (x, y) и x уже входит в дерево, вычислим $d(i, y) = d(i, x) + D(x, y)$ для каждой вершины i этого дерева. Затем проверим равенства $d(i, y) = D(i, y)$. Это займет время $O(n)$ на одну итерацию и, следовательно, полное время $O(n^2)$. Если алгоритм устанавливает, что $d(i, j) \neq D(i, j)$ для какого-либо i , то компактного аддитивного дерева D не существует.

17.3. Бережливость: символьно-ориентированное эволюционное воссоздание

17.3.1. Введение

В этом пункте мы опишем другой подход к воссозданию истории эволюции, а именно *символьно-ориентированный*. В нем исходной информацией будет набор *атрибутов*, называемых *символами*, которыми могут обладать объекты.*). Если исходные символы хорошо выбраны, то распределение атрибутов по объектам можно использовать для получения частичной истории эволюции в форме дерева эволюции. Дерево обеспечивает порядок ветвлений истории, но не приписывает событиям дивергенции временных оценок. Иными словами, эти атрибуты можно использовать для *таксономии* (систематической классификации) объектов без каких-либо предположений об их исторической важности. После рассмотрения задач воссоздания на основе символов мы свяжем этот подход с ультраметрическими деревьями.

*.) Заметим, что в этом контексте слово “символ” (character) относится не к элементу алфавита, а к атрибуту (или свойству) объекта.

Алгоритмическое изучение символьно-ориентированного воссоздания было очень активно в последние годы, и здесь могли бы обсуждаться многие из полученных результатов. Но в наши намерения входит только введение в такое воссоздание, мы остановимся на некоторых очень идеализированных и упрощенных вариантах задач, символьно-ориентированных (или максимальной бережливости). Мы исследуем подробно лишь задачи с *двоичными символами*, в которых рассматриваются объекты, обладающие или нет конкретным символом; затем упомянем возможные обобщения. Главное внимание будет уделено (двоичной) задаче *совершенной филогении*, которая является частным случаем общей задачи о максимальной бережливости.

Определение. Пусть M — $0-1$ (двоичная) матрица размера $n \times m$, представляющая n объектов в терминах m символов или свойств, описывающих эти объекты. Каждый символ принимает одно из двух возможных *состояний*, 0 или 1, и $M(p, i) = 1$ в том и только том случае, если объект p имеет символ i .

Определение. Пусть M — двоичная матрица размера $n \times m$ для n объектов. *Филогенетическое дерево* для M — это дерево с корнем T , имеющее ровно n листьев и обладающее следующими свойствами:

1. Каждый из n объектов помечает ровно один лист T .
2. Каждый из m символов помечает ровно одну дугу T .
3. Для любого объекта p символы, которые помечают дуги вдоль единственного пути от корня до листа p , определяют все символы p с состоянием единицы.

В примере на рис. 17.6 первая матрица M имеет филогенетическое дерево T , а вторая матрица M' не имеет.

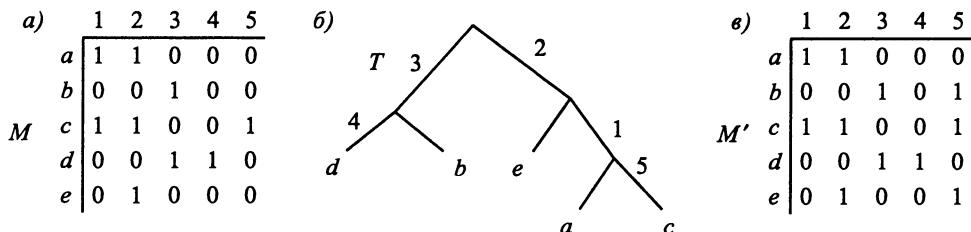


Рис. 17.6. а — матрица M ; б — филогенетическое дерево T матрицы M ; в — матрица M' , не имеющая филогенетического дерева

Интерпретация филогенетического дерева M заключается в том, что оно предлагает оценку истории эволюции объектов (в терминах образца ветвления, но не времени), основанную на следующих биологических предположениях:

1. Корень дерева представляет наследственный объект, не имеющий ни одного из присутствующих m символов. То есть состояние каждого символа в наследственном объекте нулевое.
2. Каждый символ переходит из нулевого состояния в единичное ровно один раз и никогда не переходит из единичного состояния в нулевое.

Основная черта филогенетического дерева (без которой не возникло бы интересной задачи) в том, что каждый символ помечает *ровно одну* дугу дерева. Она соответствует второму из приведенных выше биологических предположений и представляет точку в истории эволюции объектов, когда символ переходит из нулевого состояния в единичное. Следовательно, любые объекты ниже этой дуги данным символом определенно обладают.

Нахождение набора символов, на которых выполняются предположения, — часто трудная (и противоречивая) работа в исследовании эволюции. Поэтому заслуживает обсуждения вопрос о том, откуда приходят символические данные.

17.3.2. Откуда появляются символические данные?

В случае биологических объектов (обычно видов) используемые символы традиционно были морфологическими чертами или свойствами объектов. Эти морфологические символы могут быть крупными свойствами, такими как “наличие позвоночника”, или очень мелкими чертами, понятными только специалистам, изучающим эти организмы. Превосходный пример эволюционного дерева, основанного исключительно на двоичных символах, появился недавно в [231]. Ранняя история распространения птиц была получена из таких символов, как “наличие перьев” или “утрата посторбитальной кости”. Но символы могут основываться также и на последовательностях ДНК или белков, которыми обладают различные виды. Например, содержится конкретная подстрока в расшифровке аминокислоты для данного белка или нет — это идеальный двоичный символ.

Символы, основанные на морфологии, могут быть проблематичны потому, что под давлением селекции сходные морфологические черты могут появляться независимо, какая-нибудь черта может появиться, исчезнуть и появиться снова. Например, считается, что крылья независимо появлялись несколько раз. Поведенческие характеристики столь же проблематичны. Например, символ “ходит на суставах” — свойство, которым обладают шимпанзе и гориллы, но (обычно) не человек.*). Поэтому если эта черта используется для построения филогенетического дерева, то шимпанзе и гориллы окажутся в поддереве, которое не содержит человека. Однако вопреки такому рассуждению сейчас считается, что человек и шимпанзе вместе ответвились от гориллы.

Нефункциональные подстроки в ДНК обычно более удачны как символы для воссоздания эволюции, чем свойства морфологии или поведения. Подстроки ДНК вне областей кодирования белков (генов) и вне областей регуляции мутируют полуслучайно, так что длинную случайно мутировавшую подстроку из ДНК одного вида совершенно неправдоподобно найти в ДНК эволюционно отстоящего вида. Но если интенсивность мутаций достаточно мала для изучаемого исторического диапазона, случайную подстроку (или ееrudименты) можно распознать в более тесно связанных видах. Важность такого воссоздания эволюции по символам из определенных участков ДНК будет возрастать по мере того, как будет получено больше расшифровок геномных ДНК.

Одним из типов символов, который используется при изучении эволюции и значение которого возрастает, является конкретный нуклеотид в конкретной позиции

*.) Спасибо за этот пример Гэри Черчиллю.

расшифровки ДНК. Обычно сначала аналогичные последовательности ДНК множественно выравниваются. После этого каждый столбец выравнивания определяет отдельный символ, который может быть в одном из четырех состояний — *A*, *T*, *C* или *G*. Состояние символа *i* для любого данного вида *p* — это нуклеотид в позиции *p*, *i* множественного выравнивания. В этом случае символы не двоичные, но иногда расшифровки ДНК записываются в двоичном представлении с группировкой *A* с *G* (пурины) и *C* с *T* (пиrimидины). Аналогично, множественно выстроенные последовательности аминокислот используются как символы, имеющие двадцать состояний (или меньше, если символы сгруппированы).

Другой потенциально важный тип *двоичного* символа основан на том, *регулируется ли* экспрессия конкретного белка другим конкретным белком. (Вводные обсуждения регуляции и трех общих структур в регуляторных белках — цинковых пальцев, мотивов спираль–поворот–спираль и лейциновых застежек — см. в [378, 54].) Растет понимание того, что в дифференциации видов регуляция и контроль экспрессии белков (когда и в каких клетках экспрессируется конкретный белок) столь же важны, как различия в последовательностях белков (а, возможно, и еще важнее). Например, мыши и люди синтезируют в основном один и тот же комплект белков, и по многим конкретным белкам почти нет различий в их последовательностях аминокислот.*.) Отличия мыши от человека в большей степени определяются различиями в регуляции белков (которая отражается в последовательностях ДНК вне областей кодирования белков), чем различиями в самих последовательностях белков. Недавняя работа [208] поддерживает этот взгляд, что отражено в обзоре статьи [311]: “Гены могут получить новые функции развития не только изменением их белковых последовательностей, но также приобретением новых регуляторных последовательностей, которые меняют время и место экспрессии этих белков”. Поэтому историю эволюции можно лучше понять, прослеживая изменения в регуляции белков (какие белки способствуют усилению или подавлению экспрессии других белков и каких именно), чем прослеживая изменения аминокислотной последовательности. Более того, символ “белок А способствует усилению (или подавлению) экспрессии белка В” — это *двоичный* символ, и следовательно, такие двоичные символы могут принести дополнительную пользу по мере того, как будет получено больше данных о регуляции.

Наконец, упомянем неявный двоичный символ, использованный в [45], чтобы разобраться с противоречивым взаимоотношением грибов, растений и животных. Вопрос такой: к кому ближе грибы — к растениям или животным? В подходе, описанном в [45], прежде всего было вычислено множественное выравнивание некоторых последовательностей белков грибов, растений и животных. Это множественное выравнивание содержало пропуски, и у многих выстроенных строк пропуски были одни и те же. Ширина, регулярность и частота этих общих пропусков были таковы, что каждый мог рассматриваться как двоичный символ, хотя такой язык в работе и не употреблялся. Относительно множественного выравнивания каждая строка либо содержит конкретный пропуск, либо нет. Эти “двоичные символы” были затем использованы (неявно) для построения эволюционного дерева, в котором грибы были расположены ближе к животным, чем к растениям. Аналогично можно распознать дискретные пропуски в выравнивании последовательностей РНК на рис. 11.6 на с. 294 (и на более полном рисунке в [123]). Эти пропуски можно использовать

*.) Я слышал, как один биолог говорил о мышах как о “пушистых человечках”.

как двоичные символы для построения эволюционного дерева вирусов, содержащих рассматриваемые последовательности.

Обсудив примеры двоичных символов, вернемся к алгоритмическому вопросу, как построить по ним эволюционные деревья.

17.3.3. Совершенная филогенетика

Задача о совершенной филогенетике. Пусть задана $0-1$ матрица M размера $n \times m$. Определить, имеется ли для M филогенетическое дерево, и если да, то построить его.

Мы решим задачу о совершенной филогенетике очень простым алгоритмом со временем $O(nm)$, где каждая операция сравнения и каждая выборка из M требуют одной единицы времени.

Для алгоритма и доказательства его корректности будет удобно сначала переупорядочить столбцы M . Рассматривая каждый столбец M как двоичное число (со старшим значимым битом в строчке 1), отсортируем эти m чисел по убыванию, помещая наибольшее число в столбец 1. Пусть \bar{M} обозначает переупорядоченную матрицу M . В качестве примера см. рис. 17.7. Конечно, M имеет филогенетическое дерево тогда и только тогда, когда его имеет \bar{M} . Начиная с этого момента каждый символ будет называться по его столбцу в \bar{M} . Следовательно, символ по имени j будет справа (в \bar{M}) от любого символа по имени i в том и только том случае, если $i < j$.

	2	1	3	5	4	
	1	2	3	4	5	
a	1	1	0	0	0	
b	0	0	1	0	0	
\bar{M}	c	1	1	0	1	0
	d	0	0	1	0	1
	e	1	0	0	0	0

Рис. 17.7. Матрица \bar{M} , полученная сортировкой столбцов матрицы M , см. рис. 17.6. (Каждый столбец трактуется как двоичное число.) Первая строчка чисел над \bar{M} показывает исходное имя каждого символа в M . Вторая строчка дает новые имена символов

Определение. Для любого столбца k матрицы \bar{M} пусть O_k обозначает множество объектов с единицей в столбце k (т. е. объектов, которые имеют символ k).

Ясно, что если O_k строго включает в себя O_j , то столбец (символ) k должен быть слева от столбца j в матрице \bar{M} (см. рис. 17.7). Кроме того, все повторяющиеся экземпляры столбца должны располагаться в матрице \bar{M} одним сплошным блоком столбцов. Теперь можно сформулировать главную теорему и основу эффективного решения задачи о совершенной филогенетике.

Теорема 17.3.1. *Матрица \bar{M} (или M) имеет филогенетическое дерево в том и только том случае, если для каждой пары столбцов i, j , множества O_i и O_j либо дизъюнкты, либо содержатся одно в другом.*

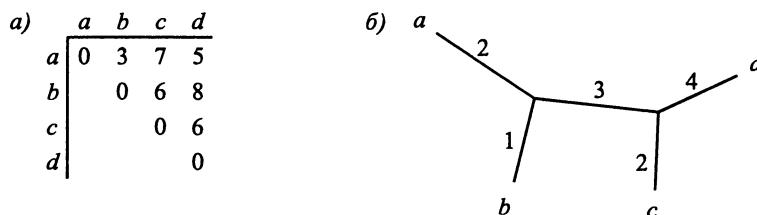


Рис. 17.4. a — симметричная матрица D ; b — аддитивное дерево T для матрицы D . Отметим, что некоторые вершины в T не помечены

Когда данные в D отражают некоторые эволюционные расстояния между парами таксонов, как взвешенные редакционные расстояния или другие оценки времени от дивергенции, аддитивное дерево для D дает одно из возможных воссозданий истории эволюции. Дерево задает образец ветвления и длины дуг (отражающие время), совместные с данными в D . Однако, так как дерево неориентированное, оно не показывает отношений наследования или направления эволюции. Их нужно вывести отдельными средствами.

Легко установить, что если D ультраметрична и $D(i, i) = 0$ для любого i , то D также аддитивна. На самом деле такая ультраметрическая матрица D может быть охарактеризована следующим образом: D ультраметрична, если существует аддитивное дерево T и вершина v в нем, такие что все листья T имеют одно и то же расстояние до v . Однако неверно, что, когда D аддитивна, она должна быть ультраметрической. Таким образом, условие аддитивности расстояний слабее для эволюционной правильности, чем условие ультраметричности. Кроме того, задача об аддитивном дереве, как и ультраметрическая задача, очень идеализирована, так как данные редко аддитивны либо из-за маленьких ошибок в данных, либо из-за больших проблем с эволюционной моделью. Одна из основных проблем состоит в том, что эволюция (возможно, растений, и определенно — бактерий) не всегда расходится (т. е. древовидна). Генетический материал может сливаться посредством *горизонтального переноса* (см., например, [430]), заставляя и ветви эволюции скорее сливаться, чем разделяться.

Когда данные не безупречно аддитивны, но древовидная модель еще верна, то ищется дерево, у которого попарные расстояния отклоняются от расстояний в D “так мало, как возможно”. Это обширная область исследований с различными определениями отклонения и многими предлагаемыми (эвристическими) алгоритмами. Практическую трактовку этой задачи см. в [429]. Теоретический метод с *доказательным анализом* см. в [145, 6]. Результаты об NP-полноте см. в [120].

17.2.2. Алгоритмы для задачи об аддитивном дереве

В литературе представлено несколько алгоритмов со временем $O(n^2)$, которые решают задачу воссоздания, когда это возможно, аддитивного дерева по матрице $n \times n$ (например, [46, 118, 218, 458]). До таких алгоритмов первое решение с полиномиальным временем появилось в [83, 85], где было доказано базисное “условие четырех точек” (см. упражнение 31 этой главы). Из условия получается прямой алгоритм для задачи со временем $O(n^4)$. Интерес к задаче об аддитивном дереве возник в различных областях, и многие результаты несколько раз переоткрывались. История

и библиография большинства этих исследований появились в [49], где утверждается: “Разные авторы поразительно мало осведомлены о работах друг друга. Так, например, условие четырех точек было опубликовано не менее пяти раз!” В действительности сейчас известно о десяти статьях, предлагающих решения с полиномиальным временем для задачи об аддитивном дереве.

Многие из опубликованных алгоритмов для задачи об аддитивном дереве трудоемкости $O(n^2)$ похожи друг на друга и включают в себя решения некоторого числа линейных систем, которые затемняют истинную комбинаторную природу задачи об аддитивном дереве. Поэтому мы отложим алгоритмические вопросы до п. 17.4, где некоторые задачи об эволюционных деревьях будут сведены к задачам об ультраметрических деревьях. Мы покажем, что задача об аддитивном дереве за время $O(n^2)$ сводится к задаче об ультраметрическом дереве, так что данный (комбинаторный) алгоритм для построения ультраметрического дерева можно адаптировать для построения аддитивного дерева. Таким образом, мы решим задачу об аддитивном дереве, используя только *сортованный порядок* некоторых чисел, полученных из матрицы D , не решая никаких уравнений. Но сначала обсудим другой комбинаторный результат для одного частного случая задачи об аддитивном дереве.

Компактные аддитивные деревья

Задача о компактном аддитивном дереве. По заданной симметричной матрице D размера $n \times n$ с нулевой диагональю и строго положительными остальными элементами определить, существует ли для D аддитивное дерево, содержащее ровно n вершин.

Например, дерево, изображенное на рис. 17.4, является аддитивным деревом для D , но не компактным. Как мы вскоре увидим, для этой матрицы компактного аддитивного дерева не существует. Однако матрица, показанная на рис. 17.5, может быть представлена компактным аддитивным деревом. У читателя есть возможность обнаружить его за время, требуемое для переворачивания нескольких страниц.

	a	b	c	d	e	f
a	0	3	7	9	2	5
b		0	6	8	1	4
c			0	6	5	2
d				0	7	4
e					0	3
f						0

Рис. 17.5. Этую матрицу можно представить компактным аддитивным деревом. Каким?

Определение. Пусть задана матрица D . Обозначим через $G(D)$ (или для краткости через G) n -вершинный полный неориентированный граф, вершины которого занумерованы от 1 до n и каждое ребро (i, j) имеет вес $D(i, j)$.

Теорема 17.2.1. Если для матрицы D существует компактное аддитивное дерево T , оно является единственным минимальным остовным деревом графа $G(D)$.

Доказательство. Сначала допустим, что T — филогенетическое дерево для \bar{M} , и рассмотрим два символа i и j . Пусть e_i — дуга T , на которой символ i меняет значение с 0 на 1, а e_j — аналогичная дуга для символа j . Все объекты, которые обладают символом i (или j), находятся в листьях T ниже дуги e_i (или дуги e_j). Может встретиться один из четырех случаев: 1) $e_i = e_j$, 2) e_i лежит на пути из корня T к e_j , 3) e_j лежит на пути из корня T к e_i или 4) пути к этим дугам расходятся до достижения e_i и e_j . В случае 1 объекты, обладающие символами i и j , идентичны (так что $O_i = O_j$); в случае 2 все объекты, обладающие символом j , должны обладать также символом i (так что $O_j \subset O_i$); случай 3 симметричен случаю 2 и поэтому $O_i \subset O_j$, а в случае 4 $O_i \cap O_j = \emptyset$. Во всех случаях либо O_i и O_j дизъюнктивны, либо одно из них содержится в другом. Половина доказательства завершена.

Докажем вторую половину построением. Рассмотрим объекты p и q , и пусть k — самый большой (самый правый в \bar{M}) символ, которым обладают p и q . Мы утверждаем, что если p обладает символом $i < k$, то q должен также содержать символ i . Чтобы убедиться в этом, заметим, что O_i и O_k пересекаются (так как p обладает обоими символами); поэтому по предположению O_i содержит O_k , и следовательно, q также обладает символом i . Так как p и q произвольны, то, если q обладает символом $i < k$, p должен также обладать этим символом. В итоге $\bar{M}(p, i) = \bar{M}(q, i)$ для любого символа $i \leq k$ и $\bar{M}(p, j) = \bar{M}(q, j)$ для $j > k$ в том и только том случае, если $\bar{M}(p, j) = \bar{M}(q, j) = 0$.

Чтобы закончить доказательство, воспользуемся нашими знаниями о деревьях ключей (из п. 3.4). Пометим объект p строкой, состоящей из символов, которыми обладает p , в порядке их появления в \bar{M} . Как обычно, добавим также концевой символ, скажем, $\$$, не входящий в исходный алфавит. Следовательно, ни одна получающаяся строка не будет префиксом любой другой строки. Например, объект A на рис. 17.7 имеет строку 1, 2, $\$$, а объект D — строку 3, 5, $\$$. Согласно утверждению из предыдущего абзаца строки для двух объектов p и q должны совпадать до некоторого символа k , а дальше не иметь общих символов (и в строгом, и в филогенетическом смысле). Отсюда следует, что дерево ключей (без связей неудач) для n строк, полученных для n объектов в \bar{M} , определяет совершенную филогению для матрицы \bar{M} (рис. 17.8). Чтобы получить из дерева ключей совершенную филогению, нужно просто удалить знаки $\$$ с дуг дерева. \square

Неконструктивное доказательство теоремы 17.3.1 появилось в ряде работ (например, [139, 141, 140]). Отметим, что алгоритм, основанный на прямой реализации

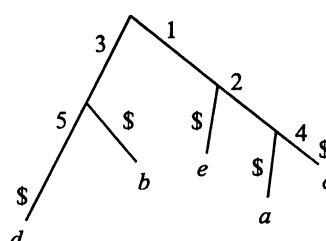


Рис. 17.8. Дерево ключей для строк, определенных матрицей \bar{M} , показано на рис. 17.7. Это дерево определяет совершенную филогению для \bar{M} . Читатель может проверить, что когда номера столбцов \bar{M} переводятся обратно в их номера в M , то дерево превращается в филогенетическое дерево с рис. 17.6

теоремы 17.3.1, занял бы время $\Omega(nm^2)$ только для определения, филогенетична ли матрица M . Первый алгоритм совершенной филогении со временем $O(nm)$ был дан в [200], но доказательство, приведенное выше, предлагает другой алгоритм со временем $O(nm)$.

17.3.4. Алгоритм со временем $O(nm)$ для задачи совершенной филогении

1. Будем рассматривать каждый столбец M как двоичное число. Используя поразрядную сортировку [10], отсортировать эти числа по убыванию, поместив наибольшее число в столбце 1. Назовем новую матрицу \bar{M} и каждый символ — по позиции его столбца в \bar{M} .
2. Для каждой строчки p матрицы \bar{M} образовать строку из символов в возрастающем порядке, которыми p обладает.
3. Построить дерево ключей T для n строк, созданных на шаге 2.
4. Проверить, будет ли T совершенной филогенией для \bar{M} .

Этот алгоритм можно реализовать за время $O(nm)$, используя поразрядную сортировку с указателями, чтобы избежать перестановки столбцов за исключением одного (см. детали этой сортировки в [10]). Все остальные операции алгоритма выполняются за это время тривиально. Отметим, что поразрядная сортировка не обязательно упорядочивает столбцы по *количество* единиц в столбце. Такую сортировку можно использовать для решения задачи совершенной филогении, но за время $O(nm)$ ее не выполнить.

17.3.5. Совместимость деревьев: применение совершенной филогении

Значительное внимание было привлечено одним обобщением задачи совершенной филогении. В нем нужно определить, описывают ли два (или более) различных филогенетических дерева “совместную” историю эволюции, и если да, то как комбинировать их в одно филогенетическое дерево, соединяющее в себе всю известную историю. Задачи такого типа имеют центральное значение при построении деревьев из реальных данных, поскольку разные методы построения деревьев и разные компьютерные пакеты зачастую дают деревья, отличающиеся в деталях. Кроме того, эволюционное дерево обычно строится для набора таксонов, основываясь на сравнении отдельного белка или отдельной позиции выровненной последовательности белков, но очень часто результат будет зависеть от выбора конкретного белка или позиции. Сейчас общепринято строить несколько деревьев, каждый по отдельному белку или позиции, и далее убеждаться, что они “достаточно состоятельны”, прежде чем полученная история эволюции будет признана надежной. Есть несколько подходов к определению и получению “состоятельной” истории. Так как наша цель — только введение в предмет и соотнесение его с совершенной филогенией, то будет рассмотрена лишь формальная сторона.

Определение. Филогенетическое дерево T' называется *очисткой* T , если T можно получить последовательностью сжатий дуг T' .

Если T' очищает T , то T' согласовано со всей историей эволюции, представленной в T , и отображает некоторую дополнительную историю, не содержащуюся в T .

Пусть T_1 и T_2 — два филогенетических дерева для множества из n объектов. Мы будем предполагать, что оба дерева представлены в “приведенной форме”, т. е. это двоичные деревья, и ни одна вершина, кроме корня, не может иметь только одного потомка.

Определение. Деревья T_1 и T_2 называются *совместимыми*, если существует филогенетическое дерево T_3 , очищающее и T_1 и T_2 (рис. 17.9).

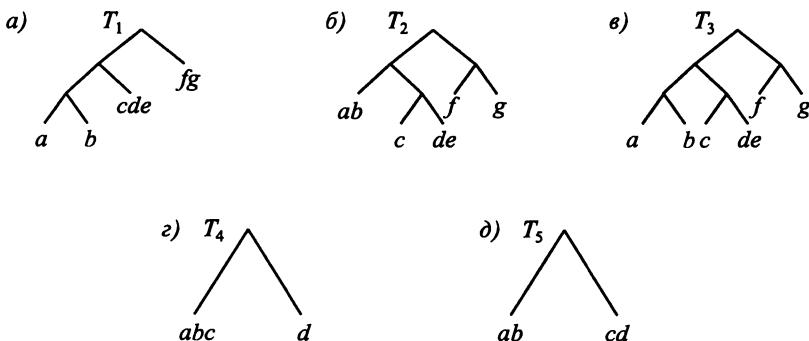


Рис. 17.9. Деревья T_1 и T_2 совместимы; они очищены деревом T_3 .
Деревья T_4 и T_5 не совместимы

Задача совместности деревьев. Пусть заданы деревья T_1 и T_2 . Определить, совместимы ли эти два дерева, и если да, то построить очищающее дерево T_3 .

Пусть M_1 — 0–1 матрица, в которой строчки соответствуют объектам, а столбцы — внутренним вершинам j дерева T_1 . Элемент $M_1(i, j)$ имеет значение 1 в том и только том случае, если лист для объекта i находится ниже вершины j . Значит, в столбце j матрицы M_1 записаны объекты, находящиеся в поддереве T_1 с корнем в j . Матрица M_2 определена аналогично для T_2 , а матрица M_3 образована объединением столбцов M_1 и M_2 . Тогда справедлива следующая теорема.

Теорема 17.3.2. T_1 и T_2 совместимы в том и только том случае, если существует филогенетическое дерево для M_3 . Это филогенетическое дерево T_3 является очисткой для обоих деревьев T_1 и T_2 .

Доказательство этой теоремы оставляется на упражнения. См. подробности в [142, 143].

Теорема 17.3.2 сводит задачу совместности к задаче совершенной филогенетии. Ниже мы увидим, что задача совершенной филогенетии сводится к минимультраметрической задаче, так что задача совместности может рассматриваться как замаскированная ультраметрическая задача. Из теоремы 17.3.2 получается алгоритм для задачи совместности со временем $O(n^2)$, если использовать алгоритм, описанный для задачи совершенной филогенетии. Однако эта задача может быть решена более прямым методом за время $O(n)$. Подробности можно найти в [200], этот вопрос также оставляется как упражнение.

17.3.6. Обобщенная совершенная филогенетическая задача

Мы подробно рассмотрели задачу совершенной филогенетии в случае, когда каждый символ может быть в двух состояниях. *Обобщенная задача о филогенетии* позволяет символу быть больше чем в двух состояниях. В этом случае совершенная филогенетия для M — это ориентированное дерево T , в котором каждый объект, как и раньше, помечает один лист из T , но дуги помечены *переходами символ–состояние*. Значит, меткой дуги считается *упорядоченная тройка* (c, x, y) , обозначающая, что на этой дуге символ c меняет свое состояние из x в y .

Как и в двоичном случае, определим для каждого символа начальное состояние в корневой вершине и потребуем, чтобы путь от корня до листа с меткой p описывал состояния символов объекта p . Окончательные состояния, определенные изменениями на пути до p , должны правильно задавать состояния символов объекта p . Здесь критично требование, что для любого состояния у любого символа c должно быть не больше одной дуги, где состояние символа c изменяется на y . Значит, может быть не больше одной дуги, которая помечена тройкой (c, x, y) с каким-либо x .

Определение филогенетического дерева в случае двоичных символов, как легко видеть, является частным случаем этого более общего определения. В двоичном случае символьная метка c является просто сокращением для тройки $(c, 0, 1)$. Следовало бы отметить, что данное здесь описание обобщенной совершенной филогенетии нестандартно, но эквивалентно обычным определениям, которые можно найти, например, в [258]. Определение, использованное здесь, делает связь между двоичным и общим случаем более явной.

Обобщенная задача совершенной филогенетии. Пусть задана матрица символов M , где каждый символ может принимать одно из r состояний. Определить, имеется ли для M совершенная филогенетика, и если да, то построить ее.

Детальное обсуждение обобщенной задачи выходит за рамки этой книги. Она была впервые поставлена в работе Бунемана [84] в 1974 г., и вопрос оставался открытым почти двадцать лет. Кэннан и Уорну [258] впервые установили, что задача имеет решение с временем, полиномиальным по n и m , если r фиксировано на значении 3 или 4. Независимо от них Дресс и Стил [133] предложили решение с полиномиальным временем для $r = 3$. Но если значение r переменное (определенное начальными данными), то обобщенная задача совершенной филогенетии является NP-полной [72, 418]. Агарвала и Фернандес-Бака [7] впоследствии установили, что если r — любое фиксированное число, то обобщенная задача совершенной филогенетии может быть решена за полиномиальное время в терминах n, m . Конечно, r появляется в экспоненте временной оценки наихудшего случая. Недавно Кэннан и Уорну [259] упростили решение и рассмотрели способ представления всех решений любого фиксированного экземпляра задачи. Этот подход может доказать свою ценность для вычисления статистических оценок “значимости” любого заданного дерева или любой дуги этого дерева.

17.4. Центральное место ультраметрической задачи

Несмотря на то что рассмотренные нами подробно четыре задачи — о деревьях ультраметрических, аддитивных, (двоичных) совершенной филогенетии и о совместности деревьев — на первый взгляд совершенно различные, на самом деле они тесно

связаны. В действительности все они решаются как задачи об ультраметрических деревьях. Мы уже видели, что задача совместимости деревьев сводится к (двоичной) задаче совершенной филогении. Цель этого параграфа — показать, как свести задачу об аддитивном дереве и (двоичную) задачу совершенной филогении к задачам об ультраметрических деревьях. В случае аддитивного дерева это сведение даст также эффективный алгоритм для ее решения.

17.4.1. Задача об аддитивном дереве как ультраметрическая задача

Мы покажем, как свести задачу об аддитивном дереве к ультраметрической задаче за время $O(n^2)$, построив матрицу D' , которая будет ультраметрической в том и только том случае, если матрица D аддитивна. Для знакомства с идеей этого сведения предположим, что D аддитивна и аддитивное дерево T для D известно. Также не умаляя общности, предположим, что каждый из n таксонов в D помечает лист T .*) Пометим вершины T конкретными числами, чтобы создать ультраметрическое дерево, и оно покажет идею желаемого сведения.

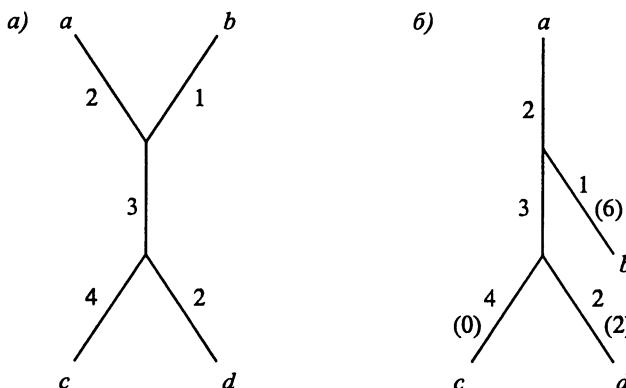


Рис. 17.10. *a* — дерево с весами на ребрах и с четырьмя помеченными листьями. Вершина v — это вершина a , и $m_v = 9$. *б* — ориентированное дерево с корнем в вершине a ; числа в скобках — значения $m_a - D(a, i)$, которые у каждого листа добавляются к весу его листовой дуги. После этого каждый лист окажется от корня на расстоянии 9. Каждая внутренняя вершина отстоит от всех листьев ее поддерева на одинаковое расстояние

Пусть v — строкка D , содержащая наибольший элемент матрицы D , и m_v — наибольшее значение. Следовательно, из всех вершин T вершина v имеет наибольшее расстояние до какого-либо листа из T . Сделаем v корнем T , создав ориентированное дерево. Мы хотим “растянуть” листовые дуги (дуги, кончающиеся в листах) таким образом, чтобы в получающемся дереве расстояние от v до каждого листа было одним и тем же. Для этого в каждом листе i просто добавим $m_v - D(v, i)$ к длине дуги, ведущей в лист i . Получится дерево T' с корнем и весами дуг, причем расстояние от v до любого листа будет равно в точности m_v и расстояние от каждой внутренней вершины до всех листьев ее поддерева будет одним и тем же (рис. 17.10).

*) Для этого требуется, чтобы мы ослабили условие *строгости* в определении ультраметричности, разрешив двум смежным вершинам на пути из корня иметь равные метки вершин.

Отметим, что изменены только длины листовых дуг. Теперь пометим каждую вершину T' (этим единственным) расстоянием от нее до листьев в ее поддереве. Эти метки не возрастают, и поэтому их можно использовать для определения ультраметрической матрицы D' , где $D'(i, j)$ — метка общего наименьшего предка листьев i и j дерева T' .

Как уменьшить работу с матрицей

В представленном изложении матрица D' обусловлена T' , зависящим от T , которое получается по D , и это определяет сведение аддитивной матрицы D к ультраметрической матрице D' . Но мы хотим прямо сводить D к D' без промежуточного построения T и T' . Чтобы увидеть идею прямого сведения, рассмотрим два листа i и j дерева T , и пусть w будет их общим наименьшим предком. Мы можем найти их (одинаковое) расстояние до w в T' , не имея явно заданного T' .

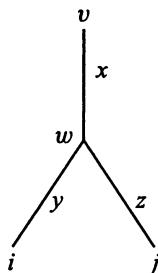


Рис. 17.11. Схема расстояний в дереве T . Расстояние от корня v до w равно x , а расстояние от w до листа i (или j) равно y (или z). Поэтому $D(v, i) = x + y$, $D(v, j) = x + z$ и $D(i, j) = y + z$. Отсюда следует, что $2x = (x+y)+(x+z)-(y+z) = D(v, i)+D(v, j)-D(i, j)$

Пусть x — расстояние в T от корня v до вершины w , а y — от вершины w до листа i . Расстояние от w до i (или j) в T' в точности равно $y + m_v - D(v, i)$. Что такое y ? Расстояние y в точности равно $D(v, i) - x$, а $2x$, как легко видеть из рис. 17.11, равно $D(v, i) + D(v, j) - D(i, j)$. Аналогичные рассуждения можно использовать для расчета расстояния от w до j . Следовательно, верна лемма:

Лемма 17.4.1. *Не зная явно T и T' , можно установить, что*

$$D'(i, j) = m_v + (D(i, j) - D(v, i) - D(v, j))/2.$$

Используя лемму 17.4.1, получаем следующее утверждение.

Теорема 17.4.1. *Если матрица D аддитивна, то матрица D' ультраметрическая, где $D'(i, j) = m_v + (D(i, j) - D(v, i) - D(v, j))/2$.*

Доказательство. Для доказательства нужно собрать полученные утверждения. $D'(i, j) = y + m_v - D(v, i)$, $y = D(v, i) - x$, $x = D(v, i) + D(v, j) - D(i, j)$. Подставляя и сокращая одинаковые члены, получим утверждение теоремы. \square

Следовательно, если мы имеем матрицу D и хотим проверить, аддитивна ли она, можно построить матрицу D' и проверить ее ультраметричность. Если нет, то D не аддитивна. А что можно сказать в противном случае?

Теорема 17.4.2. Если матрица D' ультраметрическая, то матрица D аддитивна.

Доказательство. Пусть T'' — ультраметрическое дерево для матрицы D' . (Мы не используем здесь T' , так как оно определено по T , которое неизвестно.) Сначала назначим дугам T'' такие веса, чтобы путь до любого листа i от предка w имел длину, равную числу, помечающему вершину w . (Метка каждого листа равна нулю.) Для этого назначим каждой дуге (p, q) абсолютную разность между числом в вершине p и числом в q . Получающаяся длина пути между парой листьев (i, j) равна удвоенному числу, записанному в общем наименьшем предке i и j . Однако, так как T'' — ультраметрическое дерево для матрицы D' , это расстояние равно $2 \times D'(i, j) = 2 \times m_v + D(i, j) - D(v, i) - D(y, j)$. Поэтому если для любого листа i теперь “сократим” листовую дугу листа i на $m_v - D(v, i)$, то длина получающегося пути между листьями i и j будет равна в точности $D(i, j)$. Значит, строится аддитивное дерево для D по ультраметрическому дереву для D' . \square

В итоге матрица D аддитивна в том и только том случае, если D' ультраметрическая. Более того, если D аддитивна, то аддитивное дерево T для D можно построить следующим образом:

Алгоритм построения аддитивного дерева.

Построить матрицу D' по D и ультраметрическое дерево T'' по D' . Затем сопоставить каждому ребру длину, равную абсолютной разности меток вершин в ее концах. Для каждого листа i вычесть $m_v - D(v, i)$ из длины ребра, ведущего в лист i . Получающееся дерево будет аддитивным деревом для матрицы D .

См. пример на рис. 17.12.

a)	<table border="1"> <tr> <td></td><td>b</td><td>c</td><td>d</td></tr> <tr> <td>a</td><td>3</td><td>9</td><td>7</td></tr> <tr> <td>D</td><td>b</td><td>8</td><td>6</td></tr> <tr> <td>c</td><td></td><td>6</td><td></td></tr> </table>		b	c	d	a	3	9	7	D	b	8	6	c		6	
	b	c	d														
a	3	9	7														
D	b	8	6														
c		6															

б)	<table border="1"> <tr> <td></td><td>b</td><td>c</td><td>d</td></tr> <tr> <td>D'</td><td>a</td><td>9</td><td>9</td><td>9</td></tr> <tr> <td>b</td><td></td><td>7</td><td>7</td><td></td></tr> <tr> <td>c</td><td></td><td></td><td>4</td><td></td></tr> </table>		b	c	d	D'	a	9	9	9	b		7	7		c			4	
	b	c	d																	
D'	a	9	9	9																
b		7	7																	
c			4																	

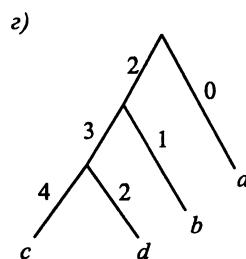
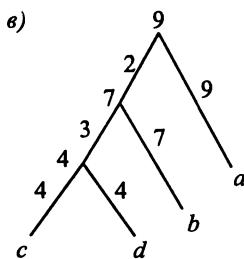


Рис. 17.12. а — матрица расстояний D , полученная из дерева на рис. 17.10, а. Наибольшее значение равно 9 и находится в строке a ; б — полученная ультраметрическая матрица D' ; в — ультраметрическое дерево T'' вместе с полученными весами дуг; г — итоговое дерево после вычитания $m_a - D(a, i)$ из весов листовых дуг. Исходное дерево получается после стягивания дуги нулевого веса к листу a .

Все шаги алгоритма легко реализуются за время $O(n^2)$, следовательно:

Теорема 17.4.3. *Аддитивное дерево для аддитивной матрицы можно построить за время $O(n^2)$.*

В сущности этот алгоритм основывается на алгоритме построения ультраметрического дерева (п. 17.1.3), а в основе этого алгоритма лежит только сортировка и разбиение чисел. Следовательно, при полной реализации алгоритма в соответствии с этими идеями, аддитивное дерево для D можно построить комбинаторным алгоритмом (а не численным), который только сортирует и разбивает числа, содержащиеся в матрице D' .

17.4.2. Задача совершенной филогении как ультраметрическая задача

Показав, как задача об аддитивном дереве решается с помощью задачи об ультраметрическом дереве, теперь покажем, как к той же задаче сводится решение (двоичной) задачи совершенной филогении.

Определение. Пусть задана матрица символов M размера $n \times m$. Определим следующую $n \times n$ матрицу D_M : для каждой пары объектов p, q положим $D_M(p, q)$ равным числу символов, которыми обладают оба объекта p и q . Значит, $D_M(p, q)$ равно числу столбцов i , для которых $M(p, i) = M(q, i) = 1$.

Первое соотношение между M и D_M дается следующей леммой.

Лемма 17.4.2. *Если M имеет совершенную филогению, то D_M — min-ультраметрическая матрица.*

Доказательство. Пусть T — совершенная филогения для M . Будем переводить T в min-ультраметрическое дерево для матрицы D_M , устанавливая, что D_M min-ультраметрическая. Сначала запишем нуль в корневую вершину T . Затем обработаем вершины T в порядке сверху вниз, последовательно записывая в каждой вершине v число, записанное в родительской вершине, плюс число символов в метке дуги, идущей в v из родительской вершины. В результате если p и q — объекты, помечающие листья ниже v , то в v записано число символов, общее для p и q (т. е. $D_M(p, q)$). Более того, эти числа строго возрастают вдоль любого пути из корня. Следовательно, дерево T вместе с этими числами в вершинах составляет min-ультраметрическое дерево для матрицы D_M . \square

Лемма 17.4.2 обеспечивает необходимое условие для того, чтобы M имела совершенную филогению. Она не дает, однако, достаточного условия. Действительно, D_M может быть ультраметрической, даже когда M и не имеет совершенной филогении. Мы предоставим читателю найти соответствующий пример. Но задачу совершенной филогении можно решить, используя ультраметрическую технику. Напомним, что если матрица min-ультраметрическая, то min-ультраметрическое дерево для нее единственno. Таким образом, если M имеет совершенную филогению, то min-ультраметрическое дерево, полученное из матрицы D_M , должно иметь ту же топологию, как совершенная филогения T , полученная прямо из M алгоритмом совершенной филогении. Если топология T известна без знания меток дуг (символов M), правильное назначение символов дугам T , обеспечивающее полную совершенную

филогению для M , становится простым. Получается следующий метод определения, имеет ли M совершенную филогению:

Совершенная филогенетическая реконструкция через ультраметрию

1. Построить матрицу D_M по M , как указано выше.
2. Попытаться построить min-ультраметрическое дерево T' по D_M . Если D_M не имеет min-ультраметрического дерева, то M не имеет совершенной филогении.
3. Если D_M имеет min-ультраметрическое дерево T' , то попытаться пометить дуги T' m символами из M , преобразуя T' в совершенную филогенетическую реконструкцию для M . Если это не получается, то M не имеет совершенной филогенетической реконструкции. В противном случае преобразованное T' является совершенной филогенетической реконструкцией T .

Таким образом, задача совершенной филогенетической реконструкции может рассматриваться как специальная форма ультраметрической задачи и может решаться этой техникой. Отсюда следует, что задача совместимости деревьев также может рассматриваться как своего рода ультраметрическая задача. Однако такой взгляд на нее интересен только концептуально, а решать задачи совершенной филогенетической реконструкции и совместимости можно более эффективно с помощью специальных алгоритмов.

17.5. Максимальная бережливость, штейнеровы деревья и совершенная филогенетическая реконструкция

Мы уже несколько раз упоминали “задачу о максимальной бережливости” и утверждали, что задача о совершенной филогенетической реконструкции является ее частным случаем, но еще не объяснили этого и не дали четкой формулировки задачи о максимальной бережливости. Для этого введем класс задач, известных как задачи о штейнеровом дереве, и затем рассмотрим задачу о максимальной бережливости над символьными данными как задачу о штейнеровом дереве над графом, который является гиперкубом. Сначала ограничимся двоичными символами.

17.5.1. Основные определения

Определение. Пусть $G = (N, E)$ — неориентированный граф с множеством вершин N и множеством ребер E , с неотрицательными весами $w(i, j)$ ребер (i, j) . Пусть $X \subseteq N$ — заданное подмножество вершин. Штейнерово дерево ST для X — это любое связное поддерево G , которое содержит все вершины X , хотя может содержать также и вершины из $N \setminus X$. Весом штейнерова дерева ST называется сумма весов его ребер и обозначается через $W(ST)$. При заданных G и X задача о взвешенном штейнеровом дереве заключается в нахождении штейнерова дерева минимального полного веса. Когда веса всех дуг равны единице, задача называется задачей о невзвешенном штейнеровом дереве.

Определение. Гиперкубом размерности d называется неориентированный граф, у которого 2^d вершин и вершины помечены числами от 0 до $2^d - 1$. Две вершины гиперкуба смежны в том и только том случае, если двоичные представления их меток отличаются ровно в одном бите.

Определение. Задача о взвешенном штейнеровом дереве на гиперкубе — это задача о взвешенном штейнеровом дереве, в которой граф является гиперкубом. Задача о невзвешенном штейнеровом дереве — это частный случай, когда вес каждого ребра равен 1.

До сих пор мы определяли задачу о максимальной бережливости легко и неопределенно как задачу воссоздания истории эволюции с наименьшим числом мутаций. Вооруженные языком гиперкубов и штейнеровых деревьев, мы можем теперь дать точную постановку “задачи о максимальной бережливости”. Сделаем это для случая двоичных символов.

Определение. Для множества исходных таксонов X , описанных в терминах d двоичных символов, задача о максимальной бережливости — это задача о невзвешенном штейнеровом дереве на d -мерном гиперкубе. Каждый элемент набора X описан двоичным вектором длины d и, следовательно, соответствует одной вершине этого гиперкуба.

Этот d -мерный гиперкуб имеет по одной вершине на каждый из 2^d возможных объектов, которые можно описать d -мерным двоичным вектором. Два возможных объекта, отличающихся на одно мутационное событие (изменение состояния отдельного символа), соответствуют двум смежным вершинам гиперкуба. Поэтому имеется штейнерово дерево, соединяющее данный набор вершин X и использующее l ребер в том и только том случае, если есть соответствующее филогенетическое дерево, содержащее l указанных мутаций.

Обобщение этого подхода на случай, когда символы могут быть больше чем в двух состояниях, оставляется как упражнение.

Штейнерова интерпретация совершенной филогении

Теперь можно установить точную связь задачи совершенной филогении с задачей о максимальной бережливости. Если исходные данные задачи о максимальной бережливости содержат d двоичных символов и каждый символ нетривиален (в том смысле, что он содержится в некоторых, но не во всех таксонах), то дерево максимальной бережливости должно иметь не менее d мутаций (т. е. мест, где изменяется состояние одного символа). Таким образом, задачу совершенной филогении можно рассматривать как вопрос, имеет ли задача о максимальной бережливости решение, стоимость которого равна d — очевидной нижней границе. Обобщенная задача совершенной филогении может рассматриваться по тому же принципу, а нижняя граница основывается на том, что, если символ имеет r состояний, должно быть не меньше $r - 1$ мест в дереве максимальной бережливости, где этот символ меняет состояние.

Итак, с точки зрения задачи о штейнеровом дереве и максимальной бережливости задача о совершенной филогении спрашивает, должно ли оптимальное штейнерово дерево иметь стоимость, большую чем очевидная нижняя граница. Для задачи о штейнеровом дереве нет известного эффективного решения на произвольных невзвешенных графах и даже, в частности, на гиперкубах. Однако решения за полиномиальное время обобщенной задачи совершенной филогении показывают, что если r фиксировано, то на этот конкретный вопрос о штейнеровых деревьях можно ответить за полиномиальное время.

17.5.2. Приближения к максимальной бережливости

Было показано, что задача о невзвешенном штейнеровом дереве на гиперкубах NP-трудна [165]. Обычно NP-трудность невзвешенной задачи немедленно влечет NP-трудность взвешенного варианта задачи. В конце концов, невзвешенный случай — это просто взвешенный случай, когда все веса приняты равными 1. Однако случай задачи о штейнеровом дереве на гиперкубах отличен, и результат в [165] не влечет NP-трудности задачи о взвешенном штейнеровом дереве на гиперкубах. Мы оставим читателю разгадать, почему бы это могло быть. Ответ, вместе с доказательством того, что взвешенный вариант действительно NP-труден, содержится в [198].

Хотя задача о взвешенном штейнеровом дереве на гиперкубах NP-трудна, существует эффективный алгоритм ее приближенного решения с множителем, меньшим чем два. Это следует из возможности эффективно аппроксимировать задачу о взвешенном штейнеровом дереве на любом графе с границей ошибки 11/6 [62, 484]. До этого результата было известно, что можно с помощью минимального оствовного дерева получить штейнерово дерево, полный вес которого меньше чем вдвое превосходит вес оптимального штейнерова дерева [284]. Приспособливая метод к случаю гиперкуба, его вкратце можно описать так:

1. Вычислить в гиперкубе расстояние $d(i, j)$ между любыми объектами i и j множества X (в случае двоичных символов это просто расстояние Хэмминга).
2. Образовать полный неориентированный граф K_X с одной вершиной для каждого объекта из X и одним ребром между каждыми двумя вершинами и назначить вес $d(i, j)$ каждому ребру (i, j) из K_X .
3. Вычислить минимальное оствовное дерево T для K_X (где каждая дуга в T соответствует пути в гиперкубе).
4. Превратить каждое ребро в T в соответствующий ему путь в гиперкубе.
5. Наложить найденные в шаге 4 пути друг на друга и получить граф G' , а затем найти в G' любое оствовное дерево.

Получится штейнерово дерево для X , полный вес которого не более чем вдвое превосходит оптимальный.

Приведенное описание имеет только принципиальное значение, а в случае двоичных можно проводить вычисления без явного погружения задачи в гиперкуб. Это очень важно для эффективности. Более совершенные приближения, предложенные в [62, 484], можно также приспособить для задачи о максимальной бережливости без явного использования гиперкуба. Мы оставляем это читателю как упражнение.

17.6. Снова филогенетическое выравнивание

Предыдущие параграфы фокусировались на методах получения истории эволюции по данным о последовательностях, и это направление изучения эволюции привлекло наибольшее внимание популярной прессы (например, эпизод

с “Евой” [481, 86, 175, 31] *). Но выводы в противоположном направлении также важны. Значит, если история эволюции множества таксонов хорошо устанавливается по окаменелостям или морфологическим данным, можно использовать ее, чтобы пролить свет на *молекулярную* эволюцию гомологичных последовательностей, по одной из каждого таксона в множестве.

Есть несколько способов формализации этого подхода к изучению молекулярной эволюции. Один из них — это *задача о филогенетическом выравнивании*, обсуждавшаяся уже в п. 14.8. Задано дерево с известной последовательностью в каждом листе, и требуется определить последовательности во внутренних вершинах так, чтобы минимизировать полное эволюционное (редакционное) расстояние на дугах. Именно, когда последовательности во внутренних вершинах назначены, длиной дуги является редакционное расстояние между последовательностями, помечающими концы дуги. Длина дерева — это сумма длин его дуг. Требуется определить последовательности во внутренних вершинах так, чтобы минимизировать длину дерева.

В более раннем обсуждении филогенетического выравнивания (п. 14.8) полученное дерево предполагалось использовать для руководства при множественном выравнивании доживших до наших дней последовательностей. Множественное выравнивание было совместно с помеченным деревом, и, таким образом, дерево неявно определяло, какие пары последовательностей должны быть выделены в выравнивании. Однако после получения выравнивания доживших последовательностей последовательности из внутренних вершин должны были быть удалены. Здесь взгляд изменился. Самая ценная информация из филогенетического выравнивания — это как раз полученные внутренние последовательности, поскольку они представляют гипотетические последовательности предков, давшие жизнь известным современным последовательностям. В этом смысле задача о филогенетическом выравнивании является частным случаем задачи о бережливости. Если редакционное расстояние реалистично моделирует эволюционное расстояние между последовательностями (т. е. цену мутаций и др.) и если глобально оптимальное филогенетическое выравнивание правильно выделяет процесс, который на самом деле происходит в серии локальных изменений, то оптимальное филогенетическое выравнивание может служить разумной гипотезой истории эволюции.

Основные технические аспекты задачи о филогенетическом выравнивании подробно рассмотрены в п. 14.8 и не требуют повторения здесь. Однако есть особенно простой, широко используемый вариант этой задачи, который имеет прямой смысл упомянуть в контексте выводов об эволюции, а не в контексте множественного выравнивания. Следовательно, его нужно обсуждать здесь, а не в главе 14. Этот вариант называется *задачей о минимальной мутации*. Она имеет простое эффективное решение, предложенное У. Фитчем и Дж. Хартиганом [159, 216, 217]. Это решение было включено в методы, с помощью которых удалось справиться с задачей о полном филогенетическом выравнивании (см. п. 17.7).

*) Теория Евы заключается в том, что все живущие люди имеют общего предка по женской линии, жившего около 200 тыс. лет назад. Все другие человеческие роды (не происходящие от этой Евы) вымерли. Теория основана прежде всего на сравнении последовательностей митохондриальных ДНК. Митохондриальная ДНК используется отчасти потому, что она наследуется только от матери (облегчая прослеживание наследуемости), отчасти потому, что она быстро мутирует у людей (что делает полезной проверку коротких промежутков времени), а отчасти потому, что эти мутации обычно нейтральны (нет мутаций ни летальных, ни особенно благоприятных для организма). Хотя теория Евы получила большое внимание популярной прессы и имеет очень сильных сторонников, часть ранних работ была отозвана (отражая вычислительные трудности исследования), и ее основной тезис остается спорным.

17.6.1. Задача Фитча–Хартигана о минимальной мутации

Определение. *Задача о минимальной мутации* — вариант задачи о филогенетическом выравнивании, когда кроме дерева и строковых меток листьев исходные данные содержат *множественное выравнивание* этих строк.

Этот вариант задачи может быть удобен, когда исходные данные в задаче о филогенетическом выравнивании состоят из белковых строк, так тесно связанных, что их множественное выравнивание вопроса не вызывает. Например, в классических случаях *цитохрома с* или *гемоглобина* аминокислотные последовательности для разных млекопитающих имеют очень близкие длины, и биологически осмысленное множественное выравнивание легко получается использованием небольшого числа пробелов. По этому выравниванию каждая аминокислота в каждой строке получает конкретную позицию относительно выбранной справочной строки. История эволюции этих строк рассматривается как комбинированная, но независимая история эволюции каждой позиции аминокислот. Такова модель, лежащая в основе построения РАМ единиц и РАМ матриц (см. пп. 15.7 и 17.6.2).

Приписывая каждую аминокислоту конкретной позиции и предполагая историю эволюции каждой позиции независимой от других, мы сильно упрощаем задачу о филогенетическом выравнивании. Каждый столбец множественного выравнивания может изучаться отдельно. Существенно, что строки, назначенные во внутренние вершины, должны согласовываться с данным множественным выравниванием. Следовательно, общая задача о филогенетическом выравнивании сводится к отдельным экземплярам задачи, где каждая исходная строка состоит всего из *одного символа*. Разрешенные исходные символы — это символы исходного алфавита, дополненно-го специальным символом пробела. Сейчас мы рассмотрим этот простой вариант задачи, но ограничимся случаем, когда цена несовпадения равна 1, а цена совпадения — 0. Обобщение на произвольные цены не представляет трудности и оставляется на упражнения.

Задача о минимальной мутации для отдельной позиции. Пусть задано дерево с корнем, имеющее n вершин (не обязательно двоичное), и каждый его лист помечен отдельным символом. Задача о *минимальной мутации* заключается в пометке каждой внутренней вершины дерева отдельным символом так, чтобы минимизировать число дуг, у которых метки начала и конца различны.

Эта задача легко решается с помощью динамического программирования. Для вершины v обозначим через T_v поддерево с корнем v , а через $C(v)$ — цену оптимального решения задачи о мутации на поддереве T_v . Для каждого символа x пусть $C(v, x)$ обозначает цену наилучшей пометки поддерева T_v , когда вершина v должна быть помечена символом x . Пусть v_i обозначает i -го потомка вершины v . Границные условия определяют значения $C(v)$ и $C(v, x)$ для каждого листа v и каждого символа алфавита x : $C(v) = 0$ и $C(v, x) = 0$, если x — символ, помечающий лист v , и $C(v, x) = \infty$ — в противном случае. Когда вершина v внутренняя, то

$$C(v) = \min_x C(v, x)$$

и

$$C(v, x) = \sum_i \min\{C(v_i) + 1, C(v_i, x)\}.$$

Счет по рекуррентным соотношениям ведется от граничных условий. Это соответствует проходу по дереву снизу вверх, начиная от листьев, так как соотношения в каждой вершине v могут быть вычислены после и только после вычислений для всех потомков v . Как обычно в ДП, полный вид оптимального решения определяется на фазе обратного хода после расчета всех значений $C(v, x)$. В этой фазе алгоритм находит сначала символ в корне r , выбирая x , для которого $C(r) = C(r, x)$. Затем он обходит дерево сверху вниз. Если символ в вершине v установлен в y , то символ его потомка v_i полагается равным y , если $C(v_i) + 1 > C(v_i, y)$; в противном случае — равным символу x , для которого $C(v_i) = C(v_i, x)$.

Прямо получается, что соотношения корректны и что их можно вычислить снизу вверх за время $O(n\sigma)$, где σ — размер алфавита. Когда используются произвольные цены совпадений и несовпадений, решение может быть обобщено и потребует времени $O(n\sigma^2)$ (см. упражнение 36). Упражнение 37 рассматривает исходный метод Фитча–Хартигана, которое отличается от представленного здесь.

17.6.2. Использование филогенетического выравнивания для вычисления матриц РАМ

Конструирование предложенных М. Дейхоф матриц РАМ обсуждалось в п. 15.7.4, но одна подробность метода была опущена. В п. 15.7.4 мы говорили, что подстановки аминокислот высчитываются [122] по выровненным парам белковых последовательностей с известными малыми РАМ-расстояниями. В действительности эти подсчеты идут от пар известных последовательностей, дополненных предполагаемыми последовательностями, полученными при решении задачи *филогенетического выравнивания* на 71-м эволюционном дереве [122]. Листья каждого дерева помечены существующими белковыми последовательностями, и прародительские последовательности реконструированы и сопоставлены внутренним вершинам каждого дерева. Данные о подстановках аминокислот созданы по парам последовательностей, которые помечают соседние вершины дерева. Таким образом получены дополнительные пары последовательностей, что расширило данные; эти пары оказались даже больше похожи друг на друга, чем пары из существующих последовательностей в листьях. Дейхоф и др. утверждают: “За счет сравнения наблюдаемых последовательностей с вычисляемыми прародительскими, а не друг с другом, получена более четкая картина приемлемой точковой мутации” [122].

17.7. Связи между множественным выравниванием и построением деревьев

Несмотря на то что мы рассматривали построение деревьев и множественное выравнивание в основном по отдельности, многие подходы в биологической литературе интегрируют или итерируют эти две задачи. В п. 14.10.2 мы отмечали, что история присоединений, определенная методом множественного выравнивания на основе кластеров, используется также для определения эволюционных деревьев. Методы прогрессивного выравнивания формализуют эту двойную задачу построения эволюционных деревьев и множественных выравниваний.

Другие методы используют множественное выравнивание как шаг в вычислении эволюционных деревьев, или, напротив, используют эти деревья как шаг в построении множественных выравниваний, или переключаются от одного к другому. Например, алгоритм Фитча–Хартигана можно итерировать с алгоритмами для общей задачи о филогенетическом выравнивании (где никакого выравнивания исходных данных не задано). После того как филогенетическое выравнивание для дерева T вычислено (любым методом), его можно улучшить следующим образом. Сначала вычислить множественное выравнивание $\mathcal{M}(T)$, которое совместно с T . По теореме 14.6.1 в $\mathcal{M}(T)$ сохраняются оптимальные попарные выравнивания, соответствующие дугам T . Далее удалим из $\mathcal{M}(T)$ выведенные последовательности, отвечающие внутренним вершинам T . Затем используем результат множественного выравнивания исходных последовательностей как начальные данные в методе Фитча–Хартигана. Получится филогенетическое выравнивание T , которое не хуже, а обычно лучше, чем предыдущее. Это филогенетическое выравнивание можно использовать для создания окончательного множественного выравнивания или оставить его в виде дерева как гипотетическую историю эволюции.

Описанный подход можно также адаптировать, чтобы попробовать построить филогенетическое выравнивание, в котором строка в любой внутренней вершине v была бы примерно на одном расстоянии от всех строк в листьях ее поддерева. Такое филогенетическое выравнивание лучше согласовывалось бы с теорией молекулярных часов, чем просто поднятое выравнивание.

Другой общий способ, с помощью которого построение эволюционного дерева итерируется с множественным выравниванием, заключается в том, что сначала множественно выравнивается исходный набор строк, далее вычисляются наведенные парные расстояния, заданные множественным выравниванием, а затем расстояния используются для построения эволюционных деревьев методами, основанными на расстояниях. Противоположный подход предлагает сначала строить дерево T для попарных редакционных расстояний, вычисленных прямо по строкам, затем решить задачу о филогенетическом выравнивании для T , потом вычислить множественное выравнивание, совместное с помеченным деревом T и наконец удалить строки, соответствующие внутренним вершинам. Конечно, эти два подхода можно итерировать. Более того, в тех пунктах метода, где есть и множественное выравнивание, и дерево, можно решать задачу о филогенетическом выравнивании, используя метод Фитча–Хартигана, как описано выше.

17.8. Упражнения

- Предложите простой алгоритм, который за время $O(n^2)$ может определить, является ли матрица размером $n \times n$ ультраметрической. Вам не потребуется использовать предложенный в главе 8 метод с константным временем для определения общего наименьшего предка.
- Предлагается вариант определения ультраметрической матрицы. Вспомним определение графа $G(D)$ из п. 17.2.2, и пусть MD будет минимальным оствовым деревом для $G(D)$. D — ультраметрическая в том и только том случае, если для каждой пары вершин i, j дуга максимального веса на пути, соединяющем в MD вершины i и j , имеет вес $D(i, j)$. Докажите это.

3. Докажите, что если матрица D ультраметрическая, то существует минимальное оствовное дерево $G(D)$, состоящее из одного пути из n вершин.
 4. Когда условие “строгости” убирается из определения ультраметрического дерева, то для данной матрицы D может существовать больше одного ультраметрического дерева. Тем не менее вариация в разрешенных деревьях невелика. Опишите точно класс ультраметрических деревьев, возможных для матрицы D .
 5. Завершите подробности доказательства единственности min-ультраметрического дерева, используя определение min-ультраметричности в тексте (т. е. с включенным условием “строгости”).
 6. Верно ли, что если есть аддитивное дерево для матрицы D , то оно единственno?
 7. Не используя сведение задачи об аддитивном дереве к задаче об ультраметрическом дереве, постройте алгоритм со временем $O(n^2)$, который получал бы $n \times n$ матрицу D и определял бы, есть ли для нее аддитивное дерево. Конечно, алгоритм должен строить дерево, когда оно есть.
 8. Покажите, что если матрица D ультраметрическая и $D(i, i) = 0$ для каждого i , то D также аддитивна. Покажите, что обратное неверно. Как обстоит дело в случае min-ультраметрических матриц?
 9. Предположим, что матрица D аддитивна, но, возможно, не представима компактным аддитивным деревом. Пусть T — минимальное оствовное дерево данных из D . Рассмотрим произвольную пару строчек i и j из D . Возможно ли, чтобы расстояние в T от i до j было меньше, чем $D(i, j)$? Можно ли использовать эти наблюдения для алгоритма построения аддитивных деревьев со временем $O(n^2)$?
 10. Пусть M — матрица, содержащая C_n^2 попарных редакционных расстояний n последовательностей. Мы знаем, что если матрица M аддитивна, то редакционные расстояния можно представить междулистовыми расстояниями в аддитивном дереве со взвешенными ребрами и с n листьями. А как насчет обратной ситуации? Предположим, что задано дерево со взвешенными ребрами и с n листьями. Всегда ли можно создать n последовательностей и взаимно однозначно отобразить последовательности и листья так, чтобы для любой пары листьев (i, j) и соответствующих им последовательностей S_i, S_j расстояние между i и j равнялось редакционному расстоянию между S_i и S_j ? Теперь адресуйте такой же вопрос ультраметрическим деревьям.
 11. Предположим, что M представлена структурой данных для редкозаполненных матриц. То есть для каждого объекта p символы, которыми он обладает, представлены цепным списком. Пусть l — суммарная длина этих n списков. Предложите алгоритм со временем $O(l)$, проверяющий, имеет ли M совершенную филогению, и строящий ее, если это возможно.
 12. Вспомним определение свойства сплошных единиц для строчек матрицы из п. 16.5.1. Аналогичное свойство сплошных единиц для столбцов требует, чтобы все единицы каждого столбца составляли сплошной блок. Известно [217], что если M имеет филогенетическое дерево, то можно так переставить строчки M , чтобы каждый столбец обладал свойством сплошных единиц. Докажите это.
- Свойство сплошных единиц позволяет улучшить визуальное представление матрицы. Алгоритм переупорядочения строчек со временем $O(n^2m)$ был предложен в [217]. Известны алгоритмы с линейным временем, но очень сложные [74], а для матриц с филогенетическим деревом есть метод с линейным временем, значительно более простой. Найдите его.

13. Верно ли, что если матрица M обладает свойством сплошных единиц, то для M должна существовать совершенная филогенетия?
14. Используя теорему 17.3.1, установите, что при проверке того, имеет ли M совершенную филогенетию, в худшем случае должен быть проверен каждый элемент M .
15. В рассмотренной в тексте задаче совершенной филогенетии с двоичными символами предполагалось, что в корне дерева каждый символ находится в нулевом состоянии. Часто это предположение оказывается слишком сильным. Однако, когда выбор корневых состояний уже сделан, эти состояния можно переобозначить так, чтобы все корневые состояния стали нулевыми, и получающуюся матрицу проверить на наличие у нее филогенетического дерева. Конечно, если M имеет совершенную филогенетию, то она имеет и такую, где корень (вершина предка) помечен любой выбранной строкой из M . Однако может показаться неестественным помещать в корень дерева один из исходных объектов. Более естественный выбор для корневого состояния символа j — это состояние *большинства*: назначить ему 1 в том и только том случае, если $|O_j| \geq n/2$. В [139] показано, что если какой-нибудь выбор корневых состояний приводит к филогенетическому дереву, то к нему приводят и выбор для каждого символа состояния большинства.
16. Предложите простой алгоритм с линейным временем для определения того, изоморфны ли две филогенетии. Полезен тот факт, что метки на листьях различны.
17. Эта задача обсуждалась на интернет-конференции `bionet.molbio.evolution`, занимающейся вопросами эволюции. Каким способом можно выяснить, не представляют ли два двоичных дерева, заданные в нью-хэмпширском формате, одно и то же дерево? Дискуссия в сети не разрешила этот вопрос полностью (в терминах наихудшего случая), хотя был предложен один метод с кубическим (или более) временем и один очень практический, но в худшем случае экспоненциальный. Возможен простой метод с линейным временем.

Пусть T — двоичное дерево, каждый лист которого имеет уникальную метку. Нью-хэмпширский код T — это строка, которая описывает T . Ее можно получить обходом T следующим образом. Стока накапливается слева направо добавлением символов в правый конец; при первом посещении нелистовой вершины к растущей строке добавляется открывающая скобка; при посещении листа добавляется его метка; при первом возврате обхода к вершине добавляется запятая, а при втором возврате — закрывающая скобка. Например, дерево, изображенное на рис. 17.8, имеет нью-хэмпширский код $((D, B), (E, (C, A)))$. Заметим, что, хотя код однозначно идентифицирует дерево, код не единственен из-за вариантов выбора при обходе в глубину. Сформулируем иначе: даже если T содержится в структуре данных, которая определяет конкретного левого и конкретного правого потомка для каждой нелистовой вершины, выбор “кто левый – кто правый” может быть различен в двух разных экземплярах T . Поэтому, даже если потребовать, чтобы обход был фронтальным (левый потомок посещается до правого потомка), можно создать две различные строки для двух различных экземпляров T . Это и приводит к обсуждавшейся в сети задаче об определении того, задают ли два несовпадающих нью-хэмпширских кода одно и то же дерево.

Покажите, как использовать решение предыдущей задачи об изоморфизме деревьев для решения данной задачи за линейное время.

Альтернативный подход заключается в выборе “канонического” нью-хэмпширского кода с тем свойством, что любое дерево кодируется в нем однозначно. Если произвольный нью-хэмпширский код для дерева может быть эффективно переведен в канонический, то проблема сравнения двух кодов легко решается: нужно каждый из двух

кодов перевести в каноническую форму и сравнить результаты. Очевидный канонический код — “лексикографически наименьший”. Например, записанный выше код должен иметь вид $((A, C), E), (B, D))$.

Дайте точное определение “лексикографически наименьшего” так, чтобы каждое дерево имело единственный лексикографически наименьший нью-хэмпширский код. Затем укажите простой алгоритм с линейным временем, который переводил бы любой нью-хэмпширский код в лексикографически наименьший. Полезен тот факт, что метки различны.

18. Докажите теорему 17.3.2 на с. 561.
19. Предложите алгоритм со временем $O(n)$ для проверки совместимости двух деревьев (каждое размером $O(n)$).
20. Расстояние Хэмминга между двумя двоичными векторами — это число позиций, в которых два вектора различаются. Для матрицы двоичных символов M , имеющей n строчек, пусть D будет $n \times n$ матрицей попарных расстояний Хэмминга. Покажите, что если M имеет совершенную филогению, то D аддитивна. Что можно сказать об обратном утверждении? Можно ли использовать это соотношение для эффективного сведения задачи совершенной филогении к задаче об аддитивном дереве?
21. Пусть D — числовая матрица $n \times n$, которая не ультраметрична. Мы хотим уменьшить значения в D так, чтобы она стала ультраметрична. Рассмотрим следующую процедуру: построить полный помеченный граф G с n вершинами, в котором ребро (i, j) имеет вес $D(i, j)$; образовать минимальное оствовное дерево T графа G ; пусть $D'(i, j)$ — наибольший вес ребра на пути от i до j в T .
Докажите, что $D'(i, j) \leq D(i, j)$ и что матрица D' ультраметрическая.
Пусть D'' — любая ультраметрическая матрица и $D''(i, j) \leq D(i, j)$ для каждой пары (i, j) . Докажите, что $D''(i, j) \leq D'(i, j)$.
Объясните, как это решает задачу уменьшения значений на “наименьшее количество” чтобы получить ультраметрическую матрицу для всех интерпретаций “наименьшего количества”.
22. Рассмотрите задачу увеличения значений матрицы на “наименьшее количество” для получения min-ультраметрической матрицы. Что можно сделать?
23. Можно ли эффективно свести ультраметрическую задачу к задаче совершенной филогении?
24. При сведении задачи об аддитивном дереве к задаче об ультраметрическом дереве выбиралась вершина v , максимальное расстояние от которой до любой другой вершины было бы максимально. Покажите, что это сведение можно делать, используя любую вершину v . Детали сведения будут другими, но большая часть логики сохранится и оценка времени счета останется $O(n^2)$.
25. Следуя сведению, данному на с. 565, опишите отдельный алгоритм, который проверяет аддитивность матрицы и при положительном ответе строит аддитивное дерево.
26. Покажите, как эффективно реализовать шаг 3 алгоритма *совершенная филогения через ультраметрию*, описанный в п. 17.4.2 (с. 567).
27. Обобщите взгляд на максимальную бережливость и совершенную филогению с помощью штейнерова дерева (п. 17.5) на случай, когда число состояний символов больше двух.
28. Докажите, что метод минимального оствовного дерева из [284], кратко изложенный в п. 17.5.2, дает штейнерово дерево, вес которого не более чем вдвое превосходит

оптимальный для любого графа. Объясните, как это можно использовать в задаче бережливости без явного построения гиперкуба.

29. В гиперкубе есть очень простой метод построения оптимального штейнерова дерева для любых трех вершин. Найдите этот метод. Попробуйте распространить его на четыре вершины и объясните, почему он не проходит (предполагая, что он не проходит).

30. Рассмотрим неориентированное дерево T с n листьями, помеченными различными числами. Для заданного множества из четырех листьев рассмотрим наименьшее соединяющее их поддерево T и удалим (сожмем) все внутренние вершины, степень которых равна двум. Получающееся дерево называется *квартетным деревом* для этих четырех листьев. Квартетное дерево — это либо дерево с четырьмя листьями, четырьмя ребрами и одной внутренней вершиной степени четыре, либо дерево с четырьмя листьями, пятью ребрами и двумя внутренними вершинами, каждая степени три.

Рассматривая метки листьев, покажите, что есть три неизоморфных квартетных дерева второго рода.

Покажите, как однозначно восстановить T по его множеству квартетных деревьев. Попробуйте создать эффективный алгоритм для этого восстановления.

31. Симметричная неотрицательная матрица D с нулевыми значениями на диагонали удовлетворяет *условию четырех точек* в том и только том случае, если для любого упорядоченного выбора четырех строчек x, y, z и t выполняется $D(x, y) + D(z, t) \leq \max\{D(x, z) + D(y, t), D(x, t) + D(y, z)\}$.

Докажите, что матрица D аддитивна в том и только том случае, если она удовлетворяет условию четырех точек.

32. Рассмотрим двоичное дерево с корнем T с n листьями, помеченными различными числами. Для любого набора из трех листьев рассмотрим наименьшее поддерево T , соединяющее эти три листа, и удалим (сожмем) внутренние вершины, которые имеют только одного потомка. Назовем получившееся дерево *тройником* этих трех листьев. Тройник имеет три листа и две внутренние вершины, каждая с двумя потомками. Важно, что тройник определяет, какие два из трех листьев имеют общего предка ниже общего наименьшего предка всех трех листьев.

Рассматривая метки листьев, покажите, что есть три неизоморфных тройника.

Покажите, что можно однозначно восстановить дерево с корнем T по его множеству тройников.

33. Если вместо того, чтобы брать все тройники сразу, каждый раз решать, какой тройник запросить, то T можно будет восстановить всего по $O(n^2)$ тройникам. Докажите это. Затем попробуйте создать адаптивный алгоритм со временем $O(n \log n)$. Решение дано в [256].

34. Предположим, что задан набор тройников, которые могут быть или не быть совместны с двоичным деревом с корнем. Рассмотрите сложность проверки совместности этого набора с каким-либо двоичным деревом с корнем.

35. Прочтите статьи о штейнеровом дереве [62, 484], которые предлагают приближенный метод с границей оценки $11/6$. Затем покажите, как этот метод можно применить к задаче о бережливости без явного создания гиперкуба.

36. Докажите корректность решения задачи о минимальной мутации с помощью динамического программирования и временного анализа. Покажите, как решить эту задачу за время $O(n\sigma^2)$ при произвольных оценках совпадений и несовпадений.

37. В тексте главы задача о минимальной мутации для отдельной позиции решена с помощью динамического программирования. Однако известный исходный метод Фитча–Хартигана не использует ДП. Их алгоритм таков:

Исходный алгоритм Фитча–Хартигана для отдельной позиции

Сделаем два прохода по дереву, сначала снизу вверх, а затем сверху вниз. Во время прохода снизу вверх рекурсивно построим множество *равенства* S_v для каждой внутренней вершины v . Символ x принадлежит S_v в том и только том случае, если x появляется в множествах равенства потомков v наибольшее число раз по сравнению с другими символами. Например, если v — вершина, все потомки которой листья, то S_v содержит символы, появляющиеся в этих листьях наибольшее число раз. Отметим, что множество равенства может состоять из одного символа. Отметим также, что при конструировании S_v алгоритм смотрит лишь, сколько раз символ встречается в множествах равенства потомков v , а не вершин ниже этих потомков.

Проход снизу вверх назначает каждой вершине символ. Для начала если множество равенства корня состоит из единственного символа x , то корню сопоставляется символ x . В противном случае корню сопоставляется любой символ из множества равенства. Теперь, двигаясь сверху вниз, назначим каждой внутренней вершине v символ по следующему правилу:

Если множество равенства в вершине v содержит символ x , сопоставленный родительской вершине, то сопоставить v тот же символ x . В противном случае произвольно назначить v любой символ из S_v .

Отметим, что если S_v состоит только из одного символа, то наше правило назначит его вершине v .

Доказать корректность исходного алгоритма Фитча–Хартигана и исследовать его время работы. Можно ли его обобщить так, чтобы учесть случай произвольной платы за несовпадение, как это делается в случае ДП? Какие преимущества дает исходный алгоритм Фитча–Хартигана над решением ДП? Как можно развить этот метод, чтобы получить все оптимальные решения задачи о минимальной мутации?

38. Когда имеются множественные оптимальные решения задачи о минимальной мутации, то, возможно, захочется получить решение, которое назначает любой внутренней вершине v строку, которая примерно равнодалена от всех строк в листьях ее поддерева. Почему желательно такое решение? Исследуйте задачу получения такого оптимального решения.
39. В связи с предыдущей задачей найти эффективное решение следующей. Пусть заданы две строки, S_1 и S_2 . Построить строку S' , минимизирующую $|D(S', S_1) - D(S', S_2)|$ при условии $D(S', S_1) + D(S', S_2) = D(S_1, S_2)$.

Три короткие темы

18.1. Сравнение ДНК с белком при ошибках смещения рамки считывания

В п. 15.11.3 мы обсуждали канонический совет транслировать вновь расшифрованный ген в выведенную аминокислотную последовательность, чтобы искать для нее проявления сходства в базах данных белков. Этот подход противопоставляется поиску оригинальной строки ДНК в базах данных ДНК. При использовании выведенных аминокислотных последовательностей имеется, однако, одна техническая трудность. Если в записи ДНК пропустить отдельный нуклеотид, изменится окно чтения оставшейся части ДНК (рис. 18.1). Аналогичная ситуация возникает и когда нуклеотид неправильно вставляется в запись. Пока правильная рамка считывания не переустановится (благодаря дополнительным ошибкам), многие из транслируемых аминокислот будут неправильными и нарушают большинство сравнений для получаемых аминокислотных последовательностей.

Ошибки включения и удаления во время расшифровки ДНК вполне обычны, так что ошибки смещения рамки могут в последующем анализе оказаться очень серьезными. Эти ошибки добавляются ко всем ошибкам подстановки, которые оставляют рамку считывания неизменной. Более того, информативные выстраивания часто содержат относительно малое число точно совпадающих символов, и большие области хуже выровненных подстрок (см. п. 11.7 о локальном выстраивании). Поэтому две подстроки, которые хорошо выровнялись бы без ошибки смещения рамки, но плохо выровняются с такой ошибкой, легко спутать с областями, которые плохо выравниваются только из-за ошибок подстановки. Следовательно, без некоторых дополнительных приемов ошибка смещения рамки легко пропустить и трудно исправить.

$$a) \quad G \ C \ A | G \ A \ G | C \ A \ G | C \ T \ G$$

$$\qquad \qquad \qquad A \qquad \qquad G \qquad \qquad G \qquad \qquad L$$

$$6) \quad G \ C \ A | A \ G \ C | A \ G \ C | T \ G \\ \quad \quad \quad A \ \ \ \ S \ \ \ \ S$$

Рис. 18.1. *a* — последовательность ДНК с транслированной аминокислотной последовательностью, изображенной ниже; *б* — та же последовательность после удаления четвертого нуклеотида (G) и новая транслированная аминокислотная последовательность с ошибкой смещения рамки

Можно ли продуктивно сравнивать последовательности ДНК с аминокислотными последовательностями при наличии ошибок и смещения окна и подстановки? Мы дадим и теоретический, и практический ответ на вопрос. Основной подход базируется на технике *неточного сравнения строки с строкой*. Эта техника будет использована также в обсуждении поиска генов в п. 18.2.

18.1.1. Сравнение строки с сетью

В пп. 11.4 и 11.6.3 задача о взвешенном редакционном расстоянии (для строк S_1 и S_2) представлялась как задача о кратчайшем пути на ориентированной ациклической решетке, полученной из двух исходных строк, S_1 и S_2 . На этой решетке каждая диагональная дуга обозначала либо совпадение, либо подстановку, каждая горизонтальная дуга — включение символа в S_1 , а каждая вертикальная дуга — исключение символа из S_1 . Поэтому задача о взвешенном редакционном расстоянии рассматривалась как задача нахождения пути минимальной стоимости из вершины $(0, 0)$ в вершину (n, m) , которая определяет (“произносимую”) строку S_1 , когда разрешены ошибки с установленной платой за них.

Редакционное расстояние — только одна из многих строковых задач, которые можно продуктивно представлять как задачи неточного сравнения строки с путем в ориентированном ациклическом графе. Для постановки задачи в общем виде рассмотрим ориентированный ациклический граф G , называемый *сетью*, с заданной начальной вершиной s и вершиной остановки t и меткой из одного символа на каждой дуге. Таким образом, каждый путь от s до t определяет (или произносит) некоторую строку.

Определение. Для сети G , определенной, как выше, мы скажем, что сеть G содержит строку S' , если S' задана некоторым путем из s в t в графе G . Тогда *сетевая задача сравнения* заключается в следующем: пусть указана строка S , найти строку S' , содержащуюся в G , (взвешенное) редакционное расстояние которой до S минимально по всем строкам, содержащимся в G .

Сетевая задача сравнения эффективно решается с помощью набора прямых рекуррентных соотношений ДП. Пусть $D(i, v)$ обозначает минимальное (взвешенное) редакционное расстояние между префиксом $S[1..i]$ и любой строкой, задаваемой путем из вершины s в вершину v в графе G . Для вершины v обозначим через $E(v)$ множество дуг G , направленных в вершину v , и для дуги $e = (u, v) \in E(v)$ пусть $T(i, e)$ обозначает стоимость преобразования символа $S(i)$ в символ, помечающий дугу e .

Вообще, $T(i, e)$ будет нулем, если эти два символа совпадают, и положительно, если они не совпадают. Пусть $I(e)$ — цена вставки символа на дуге e в S , а $DL(i)$ — стоимость удаления символа $S(i)$ из S .

Для $i > 0$ и $v \neq s$ $D(i, v)$ равно минимуму из трех возможных значений:

$$\begin{aligned} \min_{e=(u,v) \in E(v)} \{D(i-1, u) + T(i, e)\}, \\ \min_{e=(u,v) \in E(v)} \{D(i, u) + I(e)\}, \\ D(i-1, v) + DL(i). \end{aligned}$$

Границные условия состоят из $D(i, s)$ для $i \geq 0$ и $D(0, v)$ для $v \neq s$. Ясно, что $D(0, s) = 0$, и $D(i, s)$ для $i > 0$ — это просто стоимость удаления первых i символов строки S . Редакционное расстояние $D(0, v)$ получается повторяющимся применением второго из приведенных соотношений к вершинам G . Мы оставим подробности на упражнения. На упражнения будут оставлены также доказательство корректности этих соотношений и нижеследующая теорема. Если читатель добрался в книге до этого места, такие упражнения должны считаться простыми.

Теорема 18.1.1. *Если (ациклическая) сеть G имеет множество дуг E , то сетевая задача сравнения может быть решена с помощью ДП за время $O(|S||E|)$.*

Обратимся теперь к обоснованию задачи сравнения ДНК с белком при ошибках смещения рамки.

18.1.2. Сравнение ДНК и белка как сетевое сравнение

Пусть S — строка вновь расшифрованной ДНК и P — аминокислотная последовательность, которую нужно выровнять с S . Пусть \mathcal{P} — множество всех обратных переводов P в строки ДНК, кодирующие P . Вот один из способов формализации задачи сравнения строки ДНК S с белковой строкой P при возможности ошибки смещения рамки.

Сравнение ДНК с белком. Найти строку ДНК $S' \in \mathcal{P}$, (взвешенное) редакционное расстояние которой до S минимально среди всех строк из \mathcal{P} . Редакционное расстояние между S и S' будет затем принято как мера сходства S и P .

Кажется, что эта формализация корректно отражает проблему, но так как размер \mathcal{P} экспоненциально зависит от длины $|P|$, неясно, можно ли этот концептуальный подход реализовать эффективно. Реализовать его неэффективно — нетрудно. Действительно, есть компьютерные пакеты, выполняющие сравнение ДНК с белком, используя обратные переводы, явно генерируя множество \mathcal{P} и затем выравнивая S с каждой строкой из \mathcal{P} . Но мы ищем другое решение.

Чтобы эффективно реализовать метод обратного перевода, построим сеть G , соответствующую P , и затем решим сетевую задачу сравнения S и G . Для каждой аминокислоты из P построим подсеть, которая определяет все кодоны ДНК этой аминокислоты. Затем соединим эти подсети в ряды, создав сеть G . Множество строк, содержащихся в G , в точности равно множеству \mathcal{P} . В качестве примера на рис. 18.2 изображена подсеть для пары аминокислот VL . После этого найдем строку из G с минимальным редакционным расстоянием до строки ДНК S .

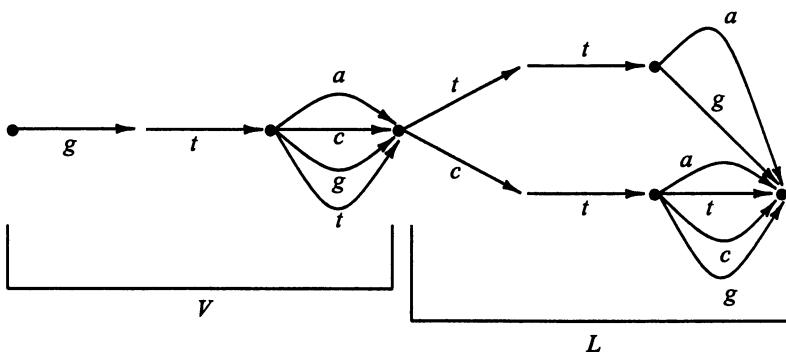


Рис. 18.2. Подсеть для пары аминокислот VL

Так как для любой аминокислоты есть не более шести кодонов и каждый из них состоит из трех нуклеотидов, сеть G имеет $O(|P|)$ дуг. Следовательно, задача нахождения строки $S' \in \mathcal{P}$ с минимальным редакционным расстоянием до S решается за время $O(|S| |P|)$. Такое же время требовалось бы для выравнивания S и P , если бы они были строками над одним и тем же алфавитом.

Описанный подход к сравнению ДНК с белком, основанный на сетевом сравнении, был в основном разработан в [364]. Обоснование сублинейности ожидаемого времени для него было описано в [94]. Другой, практический, подход к сравнению ДНК с белком описан в упражнении 5, а две формализации этого подхода — в упражнениях 6 и 7 в конце этой главы. Наконец, Клавери [103] применил к этой задаче уже совсем другой подход — эвристический. Он основан на использовании матрицы подстановки аминокислот, которая отражает возможные ошибки смещения рамки в изучаемой ДНК. Другую попытку выравнивания ДНК с белком, для нескольких иных целей, см. в [219]. Глубокую проверку нескольких методов см. в [485].

18.2. Предсказание гена

При возникшей недавно возможности расшифровывать большие количества ДНК и ее реализации задача идентификации генов, содержащихся во вновь расшифрованных ДНК, приобрела решающее значение. Эта задача возникает двумя способами. Во-первых, поток новых, относительно мало изученных последовательностей ДНК создается в различных геномных проектах. Распознание генов в этих последовательностях — один из первых шагов в их аннотировании и характеристике. Во-вторых, при позиционном клонировании можно расшифровать большой отрезок ДНК, в котором ожидается ген, вызвавший заболевание (см. с. 481). Тогда смотрят эту последовательность, чтобы найти места, которые позволяют “увидеть и понять” конкретный ген. В случае эукариот задача усложняется инtronами, которые прерывают кодирующие области генов, так что в предсказанном гене нужно различать интроны и экзоны.

Задача нахождения генов у эукариот сложнее, чем у прокариот, и обычно делится на две части: нахождение *кандидатов в экзоны* в длинной последовательности ДНК (подозреваемой на наличие гена) и выбор подмножества этих кандидатов для формирования предсказанного гена.

Поиск кандидатов в экзоны выходит за область интересов этой книги, и мы лишь упомянем некоторые из используемых идей. Самое важное — распознание в последовательности всех максимальных *открытых рамок считывания* (open reading frame — ORF). Открытая рамка считывания — это подстрока, которая не содержит стоп-кодонов при чтении в отдельных рамках считывания (см. в словаре еще одно определение ORF). Если правильная рамка считывания неизвестна, то проверяются все три (или шесть) возможные рамки. Для предсказания того, содержит ли ORF экзон, можно использовать дополнительную информацию. Например, ключевым индикатором является длина ORF. Кодирующие области существенно длиннее, чем средняя максимальная ORF. Кроме того, частоты, с которыми различные синонимические кодоны используются в экзонах, часто отличны от частот в некодирующих областях. Это называется *кодоновым смещением*. Распознание кодирующих и некодирующих областей на основе кодоновского смещения делается более эффективным от того факта, что различные организмы обладают различным кодоновым смещением. Другие индикаторы основаны на конкретных мотивах, которые ожидается найти в кодирующих областях. Методы идентификации кандидатов в экзоны обычно создают большое число перекрывающихся кандидатов, но дают также и оценки, отражающие “правдоподобие” того, что распознанный кандидат является истинным экзоном. В [412] опубликованы оценки для каждой подстроки длинной строки. Методы нахождения экзонов часто встречают самые большие трудности с идентификацией коротких экзонов и в точном определении экзоно-интронных границ (мест склеек). Обзор некоторых индикаторов и методов содержится в [412]. См. также [172, 174, 156, 192].

Вторая часть работы с выбором “хорошего” подмножества неперекрывающихся кандидатов в экзоны для формирования предсказываемого гена называется *экзонной сборкой*. В более сложном варианте могут также распознаваться кандидаты в интроны. В этом случае задача экзонной сборки заключается в нахождении неперекрывающихся чередующихся кандидатов в экзоны и кандидатов в интроны, покрывающих последовательность.

18.2.1. Экзонная сборка

Рассмотрим простой вариант экзонной сборки, а именно выбор подмножества неперекрывающихся кандидатов в экзоны, игнорируя кандидатов в интроны.

Так как каждому распознанному кандидату в экзоны сопоставлена оценка его качества, может показаться, что задача экзонной сборки решается нахождением набора неперекрывающихся кандидатов с наибольшей суммарной оценкой. Это в точности задача об одномерном сцеплении, рассмотренная в п. 13.3. Однако в [174] обосновано утверждается, что такой подход работает недостаточно хорошо. Каждый кандидат в экзоны в оптимальной цепи должен выглядеть как экзон (согласно статистике из многих экзонов в различных организмах), а метод сцепления не отражает информации о коррелированном включении экзонов в гены. Соответствующий, более хитроумный подход к задачам типа экзонной сборки появился в [277].

Альтернативный подход был предложен Гельфандом, Мироновым и Певзнером [173, 174]. Они предложили использовать мощь сравнения последовательностей и поиска в базах данных (т.е. *первый факт* сравнения последовательностей) для решения задачи, которую они называли *задачей склеенного выравнивания*. В статье кандидат в экзоны называется *блоком*, и мы будем использовать здесь этот термин.

Метод заключается в нахождении подмножества неперекрывающихся блоков, конкатенация которых сильно похожа на последовательность из известного гена в базе данных. Поскольку мы рассматриваем эукариотические организмы, последовательности в базах данных — это последовательности кДНК, и следовательно, их интронные последовательности удалены (см. п. 11.8.3).

Определение. Пусть S обозначает последовательность геномной ДНК, $\mathcal{B} = \{B_1, B_2, \dots, B_k\}$ — упорядоченный набор блоков, идентифицированных в S , и T — последовательность кДНК. Пусть S' обозначает строку, полученную конкатенацией блоков из \mathcal{B} в соответствии с их порядком. Стока S' называется *цепью*. Задача склеенного выравнивания заключается в нахождении цепи S^* с наибольшим сходством с T . Здесь “сходство” понимается так же, как в предыдущих главах, что допускает применение специальных схем оценки для наилучшего модельного распознания генов.

При заданной базе данных генных последовательностей задача склеенного выравнивания решается для S и каждой последовательности T из базы. Цепь, соответствующая наилучшему склеенному выравниванию, используется затем для экзонной сборки S .

Нахождение склеенного выравнивания сетевым выравниванием

Задача склеенного выравнивания S и T легко решается выравниванием строки T с сетью G , полученной по набору блоков \mathcal{B} . Однако используемая целевая функция — это максимизируемое сходство, а не минимизируемое редакционное расстояние (см. упражнение 4). Для каждого блока B_i в \mathcal{B} создается подпуть в G с $|B_i|$ дугами. Каждая из этих дуг помечается отдельным символом, так что путь производит подстроку B_i . Для любой пары блоков B_i, B_j последняя вершина пути для B_i соединяется непомеченной дугой с первой вершиной пути для B_j в том и только том случае, если B_i кончается строго до начала B_j (рис. 18.3). Тогда оптимальное выстраивание строки T с этой сетью решает задачу склеенного выравнивания.

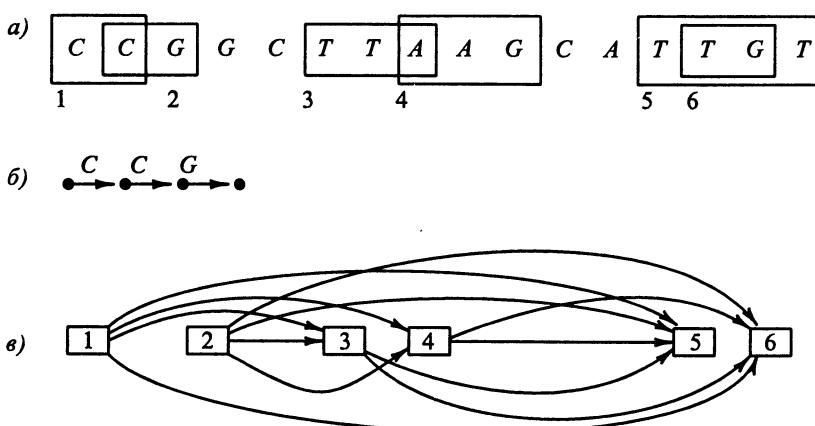


Рис. 18.3. *a* — последовательность S и шесть блоков; *б* — подпуть для блока B_1 ; *в* — представление сети G для S . Каждый прямоугольник обозначает подпуть соответствующего блока, но настоящий путь не показан. Изображены только дуги, соединяющие эти подпути

Если \mathcal{B} содержит k блоков, состоящих из b символов, то сеть G имеет $\Theta(b)$ дуг в путях, полученных из отдельных блоков, плюс $O(k^2)$ дуг, соединяющих эти пути. Следовательно, по теореме 18.1.1 задача склеенного выравнивания может быть решена за время $O(|T|b + |T|k^2)$. Однако сеть G , созданная для склеенного выравнивания, имеет дополнительную структуру, которая позволяет уменьшить время счета до $O(|T|b + |T|k)$. См. упражнение 11 этой главы.

Другой подход к поиску мест экзон/инtronной склейки, предложенный Лаубом и Смитом [293], также использует мощь работы с базами данных, но избегает динамического программирования. Вместо этого смотрят на высокую плотность хороших совпадений между короткими подстроками в целевой ДНК и последовательностями в базе данных экзонов.

18.3. Молекулярные вычисления: при помощи (а не ради) строк ДНК

Идея использования биомолекул в компьютерах и вычислительных устройствах широко эксплуатировалась в последнее десятилетие [67, 68]. Были построены прототипы гибридных компьютеров, основанных на свойствах конкретных белков, а вычислительные устройства и компоненты, сделанные из органо-электронных материалов, получили коммерческое развитие. Некоторые из этих усилий представляли собой настоящую смену ролей. Вместо вычислений *ради* строк шло вычисление *с помощью* строк (полимеров, таких как белки и ДНК). Одно усилие вызвало большое внимание и, выйдя за границы мира теоретиков информатики, было особенно привлекательно для многих теоретиков.

Осенью 1994 г. Леонард Эдлман [4] показал, как решить маленьющую конкретную задачу о гамильтоновом пути (восемь вершин), выполнив лабораторный опыт, который использовал технику рекомбинированных ДНК. Этот опыт привел к мысли, что такие вычисления, основанные на ДНК и опирающиеся на присущий биохимии ДНК параллелизм, могут оказаться способными решать более крупномасштабные трудные задачи, чем это возможно сейчас с помощью традиционных компьютеров. Его работа вызвала пристальное внимание в научной и популярной прессе, включая большую статью в *New York Times* [281]. Исходную статью Эдлмана дополнили другие исследователи, предложившие иные способы использования ДНК. Наиболее широко цитируемая из этих работ принадлежит Ричарду Липтону [305], который описал, как применять ДНК для решения конкретных случаев классической задачи выполнимости. Мы следуем здесь статье Липтона, а не Эдлмана, так как она проще и содержит более общее описание подхода.

Задача выполнимости. Пусть задана формула в *конъюнктивной нормальной форме* (КНФ). Определить, существует ли набор значений ее переменных (“истина” или “ложь”), при котором значением всей формулы будет “истина”.

Формула в КНФ состоит из конъюнктов, которые сформированы из n переменных, обозначенных через x_1, x_2, \dots, x_n . Вхождение переменной x_i , или ее отрицания \bar{x}_i , называется *литералом*. Каждый конъюнкт состоит из литералов, связанных операторами OR (пишется \vee), и заключен в согласованную пару скобок. Конъюнкты

связаны операторами AND (пишется \wedge), и вся строка называется *формулой*. Например, $(x_1 \vee \bar{x}_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3)$ — это формула в КНФ с тремя конъюнктами, составленными из трех переменных и двух литералов в каждом конъюнкте.

В задаче выполнимости требуется установить значения “истина”/“ложь” всем переменным таким образом, чтобы сделать значение всей формулы истиной при следующих правилах: для любой переменной x_i все вхождения x_i должны иметь одно и то же значение, которое противоположно значению \bar{x}_i . OR двух значений истинен в том и только том случае, если истинно по крайней мере одно из этих значений; AND двух значений истинен в том и только том случае, если истинны оба значения. Поэтому формула становится истинной в том и только том случае, если истинен по крайней мере один литерал в каждом конъюнкте.

Задачу выполнимости формул с только двумя литералами на конъюнкт можно решить эффективно, но для формул с большим числом литералов она NP-полна [171]. Поэтому эффективный (по наихудшему случаю) алгоритм неизвестен для задачи выполнимости, и есть убеждение, что такого алгоритма не существует.*). NP-полнота не только превращает задачу выполнимости (или задачу о гамильтоновом пути) в интересный тест для ДНК-вычислений, но и делает очень значимым практическое решение этой задачи. Причина в том, что любой экземпляр любой задачи из класса NP, очень широкого класса задач, можно эффективно свести (для обычного компьютера) к экземпляру задачи выполнимости, решение которой дает решение исходной задачи. Таким образом, метод практического решения задачи выполнимости для больших формул в КНФ был бы важен не только для самой задачи выполнимости.

18.3.1. Подход Липтона к задаче выполнимости

Концептуально подход Липтона к задаче выполнимости — это в явном виде метод грубой силы. Переберем все 2^n способов установки значений n переменных и проверим на каждой установке, не делают ли ее значения формулу истинной. Новизна его в том, что в технологии ДНК перебор и вычисления можно производить в высшей степени параллельно. Обратимся к деталям.

Метод состоит из стадии перебора и стадии вычисления. Стадия перебора может выполняться без знания какой-либо конкретной формулы в КНФ. Для выбранного значения параметра m стадия перебора синтезирует 2^m коротких строк ДНК, каждая из которых описывает отдельный способ установки m переменных. Единожды синтезированные, эти строки могут использоваться при решении задачи выполнимости для любой конкретной формулы в КНФ с $n \leq m$ переменными.

Стадия перебора

Полезно представить все 2^m возможных наборов значений m переменных (x_1, \dots, x_m) как ориентированный ациклический граф с $2m + 2$ вершинами. Пример показан на рис. 18.4. Граф состоит из m столбцов по две вершины в каждом плюс левая начальная вершина s и правая конечная вершина t . Верхняя вершина каждого столбца i представляет выбор для переменной x_i : значения “истина”, а нижняя вершина — выбор значения “ложь”. Имеются дуги из вершины s в каждую из

*.) Однако следует помнить, что NP-полнота — это теория об асимптотических феноменах наихудшего случая, а важные экземпляры NP-полной задачи могут быть на практике вполне решаемыми.

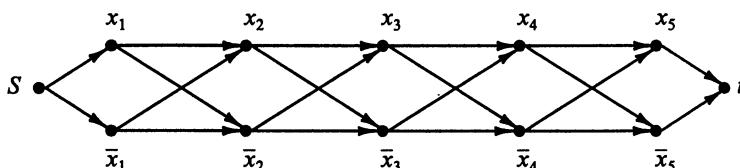


Рис. 18.4. Ациклический ориентированный граф, представляющий все способы выбора значений “истина”/“ложь” переменных x_1, x_2, x_3, x_4 и x_5 . Каждый путь из s в t описывает один способ установки переменных



Рис. 18.5. Олига связи (u, v) гибридизуется с олигами вершин u и v ; $\alpha_1, \alpha_2, \beta_1$ и β_2 представляют олиги длины десять каждой; $\bar{\alpha}_2$ и $\bar{\beta}_1$ представляют олиги, комплементарные, соответственно, к α_2 и β_1

вершин первого столбца, из каждой вершины столбца m в вершину t и из каждой вершины столбца i в каждую вершину столбца $i + 1$ для i от 1 до $m - 1$. Ясно, что имеется взаимно-однозначное соответствие между 2^m способами выбора m переменных и 2^m путями в этом графе из s в t . Если путь идет через верхнюю (нижнюю) вершину столбца i , то x_i имеет значение “истина” (“ложь”).

Метод Липтона синтезирует по одной строке ДНК для каждого пути из s в t следующим образом. Сначала создадим случайную строку ДНК длиной, например, двадцать для каждого пути в этом графе. Такая маленькая строка ДНК называется *) олигонуклеотидом или олигой (oligo). Строки для верхних m вершин называются T -олигами, для нижних m вершин — F -олигами, а для вершин s и t — старт- и стоп-олигами соответственно. Коллективно эти олиги называются олигами вершин.

Затем создадим по строке ДНК для каждой дуги графа. Эти строки, называемые олигами связей, будут также длины двадцать, но уже не случайными. Для произвольной дуги из вершины u в вершину v ее олига связи создается следующим образом. **) Первые десять нуклеотидов комплементарны ($A \leftrightarrow T, C \leftrightarrow G$) к последним десяти нуклеотидам олиги вершины u ; последние десять нуклеотидов комплементарны первым десяти нуклеотидам олиги вершины v . Смысл в том, что олига связи для (u, v) может гибридизоваться с олигами вершин u и v , соединяя две олиги вместе для создания олиги длины сорок (рис. 18.5).

*) Так у автора. Греческое слово *ολίγος* (маленький) в английском языке, как и в русском, в чистом виде не используется. — Прим. перев.

**) Есть еще технический момент, касающийся ориентации ДНК. В этом описании он опущен.

Теперь создадим “большое число” копий всех этих олиг. Точное количество зависит от технологии, но должно быть как минимум 2^m копий старт-олиги и стоп-олиги, по 2^{m-1} копий олиг вершин и по 2^{m-2} копий олиг связей.

Есть одно существенное ограничение на выбор олиг. Олига вершины i должна быть способна гибридизоваться на своем левом конце с двумя олигами связей двух дуг, входящих в i , а на правом конце с двумя олигами связей двух дуг, выходящих из i . Аналогично, олига связи для (i, v) должна быть способна гибридизоваться только с олигами вершин i и v , и эти гибридизации должны происходить на соответствующих концах. Таким образом, никакой суффикс длины десять олиги одной вершины не должен быть префиксом олиги другой вершины. Ожидается, что случайные олиги будут удовлетворять этим требованиям, если не так, то экспериментируя можно создать или отобрать набор из $2m + 2$ олиг, в которых никакая пара не может гибридизоваться. Используя экспериментально полученные олиги вершин и затем создавая олиги связей, как описано выше, мы получим все множество олиг, подчиняющихся ограничениям гибридизации.

На следующем шаге стадии перебора все копии олиг смешиваются, и им разрешается гибридизоваться при описанных выше условиях. Это приводит к строкам ДНК длиной $20m + 20$ нуклеотидов, где в каждой строке первые и последние десять нуклеотидов однонитевые, а остальная часть строки двунитевая. В каждой двунитевой строке одна нить состоит только из олиг вершин, а вторая — только из (сдвинутых) олиг связей. Мы назовем последнюю нить *строкой значения*. Каждая строка значения соответствует пути в графе от s до t и, следовательно, выбору набора из m переменных x_1, x_2, \dots, x_m . В этот момент эти две нити не могут быть разделены, так как олиги вершин удерживаются вместе смещенными олигами связей другой нити. Теперь в качестве последнего шага для каждой двунитевой молекулы соседние олиги одной и той же нити сшиваются (химически соединяются), так что две нити можно разделить. Каждая строка значения теперь становится стабильной однонитевой молекулой ДНК, которая будет использована в стадии вычислений. В общем метод потребует много копий каждой строки значения, но действительное число снова зависит от технологии.

Отметим, что строки значения можно создавать, не зная какой-либо конкретной формулы в КНФ. Они могут быть использованы для любой формулы в КНФ с $n \leq m$ переменными.

Стадия вычисления

Когда задается конкретная КНФ с $n \leq m$ переменными, задача выполнимости для нее решается в k шагов, где k — число конъюнктов формулы. Предположим, что переменные в формуле обозначены через x_1, x_2, \dots, x_n .

Стадия вычисления начинает работать с полной коллекцией (soup — компот^{*}) строк значений, созданных на стадии перебора. (Нити, сделанные из олиг связи, также находятся в компоте, но их можно удалить или игнорировать.) На каждом следующем шаге какое-то число строк (а возможно, что и ни одной) будет удаляться из компота, и “выжившие” строки перейдут на следующий шаг. То есть на каждом шаге обрабатывается один из конъюнктов формулы, и все строки, у которых значения переменных делают этот конъюнкт ложным, из компота удаляются. (При стандартной

^{*}) Слово “компот” обычно для биологической литературы. — Прим. перев.

технологии некоторые строки, на которых этот конъюнкт истинен, т. е. которые “удовлетворяют” этому конъюнкту, также удаляются, и по этой причине требуется много копий строк.) Выжившие строки, которые остались в компоте, используются затем на следующем шаге. Например, если первый обрабатываемый конъюнкт имеет вид $(x_2 \vee \bar{x}_3)$, то любая строка в компоте, содержащая T -олигу для x_2 или F -олигу для x_3 , переживает этот шаг. Все другие строки удаляются. Следовательно, на каждом шаге выжившие строки представляют наборы переменных, которые удовлетворяют всем обработанным к данному моменту конъюнктам. Когда будут обработаны все k конъюнктов, легко определить, остались ли в компоте выжившие строки. Если да, то можно выбрать значения переменных так, чтобы данная формула была истинной; в противном случае это невозможно.

Биотехнические детали

Исходная статья Липтона не уточняет биотехнических деталей реализации шага вычислений. Подумаем по поводу возможной реализации метода.*)

До того как станет известна конкретная КНФ-формула, создается $2m$ дополнительных олиг длины двадцать, по одной для каждого возможного значения каждой из m переменных. Для переменной x_i одна новая олига делается комплементарной к ее T -олиге и одна — комплементарной к F -олиге. Каждая из новых $2m$ олиг реплицируется и помещается в свой собственный нейлоновый фильтр. Так получается $2m$ нейлоновых фильтров, каждый заполнен копиями одной из олиг, которые могут гибридизоваться с одной конкретной олигой вершин.

Эти $2m$ нейлоновых фильтров можно использовать для обработки любого конъюнкта, содержащего до m литералов. Мы опишем метод на примере $(x_2 \vee \bar{x}_3)$. Сначала все строки значений, которые пережили предыдущую обработку (до обработки $(x_2 \vee \bar{x}_3)$), пропускаются через фильтр, заполненный олигой, которая гибридизирует с T -олигой для x_2 . Рассмотрим строку значений, которая содержит T -олигу для x_2 . Эта T -олига может гибридизоваться и приклеиться к олигне в первом фильтре. Однако при правильном понимании ситуации все строки, которые не содержат полной T -олиги для x_2 , должны протечь через фильтр. (В процессе некоторые строки, содержащие T -олигу для x_2 , также теряются.) Строки, которые не приклеились в первом фильтре, пропускаются через фильтр, заполненный олигой, которая может гибридизоваться с F -олигой для x_3 . Как и раньше, многие строки, содержащие F -олигу для x_3 , приклеятся в нем, но ни одна строка, не содержащая эту F -олигу, не приклеится. Строки, которые не приклеились ни в одном из фильтров, больше не нужны, а строки, приклеившиеся в этих двух фильтрах, вымываются и смешиваются. Они-то и являются выжившими строками значений, — те наборы переменных, при которых $(x_2 \vee \bar{x}_3)$ истинно. Следовательно, после этого шага вычислений каждая выжившая строка представляет набор значений, удовлетворяющий всем обработанным конъюнктам.

Когда все конъюнкты будут обработаны, определяется, есть ли в растворе выжившие строки или он пуст. Один из способов сделать это заключается в использовании старт- и стоп-олиг как праймеров PCR для генерирования большого количества копий всех выживших строк. PCR быстро сделают достаточно копий, чтобы продукт

*) Детали, конечно, нереальны, но поймите нашу позицию. Если можно легко придумать кажущиеся правдоподобными лабораторные приемы, реализующие этот метод (по крайней мере, для малых размеров задачи), то профессионалы-биотехники будут способны найти реализации, которые реально работают.

(если он будет) можно было разделить с помощью электрофореза в геле или обнаружить каким-либо другим методом. Если выжившие строки значений имеются, то можно эффективно определить (используя технику вычислений, называемую *саморедукцией*) выбор набора значений, который делает формулу истинной. Как делать это с помощью фильтров и PCR, оставляется на упражнение.

КНФ-формула с k конъюнктами, содержащая l различных литералов, может быть обработана всего за k шагов, используя l пропусков через фильтры. Следовательно, даже хотя внутренняя логика метода Липтона включает в себя утомительный перебор и вычисление всех возможных установок переменных, применение ДНК вводит в процесс высокий уровень параллельности, рождая надежду, что это можно сделать практическим.

18.3.2. Критика

Идея использования ДНК для решения вычислительных задач, конечно, интригующа и элегантна, и ДНК обеспечивает высокий параллелизм много больше того, который доступен на существующих компьютерах на силиконе. Однако есть много технологических барьеров, которые нужно преодолеть. Одна фундаментальная проблема с конкретными методами рассмотрена здесь.

Фундаментальная проблема заключается в том, что функция 2^n экспоненциальна, независимо от того, что считать — время или молекулы. Хуже того, 2^n — это *минимальное* число строк ДНК, нужное для того, чтобы метод Липтона работал. Аналогично было подсчитано [302], что метод Эдлмана для нахождения гамильтонова пути в применении к задаче с 70 городами с умеренной плотностью дуг ($\Theta(n \log n)$) потребовал бы 10^{25} килограммов ДНК только на одну копию каждой требуемой строки! На клочке бумаги я прикинул, что ДНК потребуется много больше. При анализе нескольких новых достижений в нестандартных методах вычислений информационная заметка в *Science* констатировала, что через год после его исходной работы

“Эдлман сейчас считает, что использование того же алгоритма для решения «даже задачи скромного размера» — включающей, скажем, 50 или 100 городов — потребовало бы «тонны ДНК». Позднее Бивер из Penn State обнаружил, что явно многообещающий алгоритм разложения на множители 300-значного числа на самом деле требует «океана [ДНК] размером со Вселенную»” [180].

Минимальное количество ДНК, требуемое для метода Липтона, более скромно (для формулы с 70 переменными оцениваемое минимальное количество ДНК лежит между несколькими граммами иическими тоннами, в зависимости от разных технических предположений). Но даже эта зона относительной практичности быстро исчерпывается. Если для формулы с 70 переменными нужно только несколько граммов, то формула всего со 100 переменными будет требовать как минимум тысячи тонн. А 100 — это не такое большое число, особенно для задач, которые получаются сведением других NP-полных задач к задаче выполнимости.

Итак, основная, сырья идея грубой силы и полного перебора не будет работать на размерах, превышающих скромные, даже при высокой параллельности, свойственной ДНК. Даже игнорируя технические вопросы, связанные с экономикой экспериментов и химией ДНК (а это сами по себе гигантские проблемы), фундаментальная проблема экспоненциального роста означает, что перечислительные методы, предложенные к настоящему времени, будут важны разве лишь для очень маленьких задач. Возникает вопрос: даже если химию ДНК можно заставить работать на уровне грамма или

килограмма, есть ли важные случаи задачи выводимости, которые слишком велики для силиконового компьютера и достаточно малы, чтобы вычисление с помощью ДНК было практическим?

Не пытаясь обеспечить практическое решение NP-полных задач, вычисление с помощью ДНК, основанное на чистом переборе, может оказаться более значимым в ускорении решения задач, которые уже рассматриваются как решаемые. Или можно было бы найти приложения, в которых нужно перебрать лишь малую долю в пространстве решений. Алгоритмы такого типа нашлись в мире *рандомизированных алгоритмов*, где часто основное наблюдение заключается в том, что конкретное событие происходит с достаточно большой вероятностью, так что требуется случайная выборка только “малого” размера. Лишь время (и деньги) скажет, где исходные идеи вычислений с помощью ДНК возьмут верх. Со временем первых работ Эдлмана и Липтона предлагались многие дополнительные идеи (см., например, [53]). Несмотря на непрактичность большинства имеющихся предложений, некоторые методы молекулярных вычислений будущего смогут проследить свое происхождение от этих ранних идей. Остается надеяться, что будущие достижения будут более полно оцениваться до того, как популярная пресса объявит, что компьютерщики “изумили революционными достижениями”.

18.4. Упражнения

1. Докажите корректность и временные оценки метода динамического программирования для сетевой задачи сравнения в п. 18.1.1 (с. 580).
2. Объясните правильность моделирования задачи сравнения ДНК с белком как сетевой задачи сравнения (выполненного в п. 18.1.2).
3. Решение задачи сравнения ДНК с белком, предложенное в п. 18.1, существенно использует глобальное выравнивание получаемой аминокислотной последовательности с белковой последовательностью. Так как локальное выравнивание имеет большее биологическое значение, переработайте этот метод для решения варианта задачи с локальным выравниванием.
4. Переформулируйте сетевую задачу сравнения для максимизации сходства вместо минимизации редакционного расстояния. Постройте рекуррентные соотношения ДП.
5. Один практический подход к сравнению ДНК с белком основан на прямом использовании локального выравнивания. Пусть S — последовательность ДНК, которую нужно сравнить с белковой последовательностью P . Предположим, что правильная ориентация S известна, а рамка считывания нет. Переведем S в три последовательности аминокислот, S_1 , S_2 и S_3 , по одной для каждого выбора рамки считывания. Затем локально выровняем каждую из этих трех последовательностей с P и найдем все локальные выравнивания, у которых сходство выше порога (см. пп. 11.7.3 и 13.2). Получатся три множества хороших локальных выравниваний. Если ошибка смещения рамки произошла в области ДНК, которая имеет высокое сходство с областью P (при переводе в правильной рамке), то это локальное выравнивание будет частью одного из трех множеств. Ошибка смещения рамки в этой области проявляется как два смежных хороших локальных выравнивания, находящихся в двух множествах. Находя такие изменения, можно вручную соединить куски в одно большее выравнивание S и P . Обсудите правильность и практичность этого подхода по сравнению с сетевым подходом из п. 18.1.

6. Подход, предложенный в предыдущем упражнении, может быть автоматизирован. Когда три множества хороших локальных выравниваний получены, задача состоит в нахождении в них подмножества неперекрывающихся локальных выравниваний с высокой оценкой. Это в точности задача о двумерной сцепке, рассмотренная в п. 13.3. Объясните ее подробно и обсудите достоинства и недостатки решения задачи сравнения ДНК с белком таким способом.
7. Гуан и Юбербахер [195] следовали общему подходу к сравнению ДНК с белком, предложеному в упражнении 5, но они автоматизировали подход за счет использования ДП. Мы опишем их метод нахождения отдельного наилучшего локального выравнивания между последовательностью ДНК и белковой последовательностью.

Концептуально решим (параллельно) три задачи локального выравнивания префиксов S_1 , S_2 и S_3 с префиксами P , применяя нормальные соотношения ДП для локального выравнивания. В каждой таблице значение любой клетки получается с учетом значений из трех соседних клеток этой таблицы. Вдобавок здесь используются еще подходящие значения из двух других таблиц, со штрафом, отражающим ошибку смещения рамки считывания. Сделайте эту идею точной. Выпишите подробно соотношения и определите время счета алгоритма. Затем рассмотрите задачу нахождения многих локальных выравниваний S и P .

8. Обобщая задачу выравнивания ДНК с белком, рассмотрим задачу выравнивания двух транслированных аминокислотных последовательностей P_1 и P_2 . Теперь требуется найти обратные трансляции, скажем, P_1 в S_1 и P_2 в S_2 , при которых редакционное расстояние между получающимися строками минимально по всем парам последовательностей обратного перевода. Предложите наиболее эффективное решение этой задачи.
9. Пусть заданы две строки ДНК. Найти минимальное число изменений этих строк (или минимальную цену этих изменений), при которых две строки переводятся в одну и ту же строку аминокислот. Имеет ли задача эффективное решение? Могла бы эта задача встретиться на практике?
10. Модифицируйте сетевую модель, использованную для экзонной сборки (п. 18.2.1), чтобы включить наравне с блоками кандидатов в интроны. Значит, собираемый ген должен состоять из неперекрывающихся, чередующихся блоков и кандидатов в интроны, которые покрывают предполагаемую кодирующую область. Качество решения оценивается по тому, насколько конкатенированные блоки соответствуют какой-либо последовательности в базе данных и насколько выбранные кандидаты в интроны похожи на интроны. Объясните, какие потребуются изменения в решении стандартной сетевой задачи сравнения с помощью ДП.
11. Время счета для задачи склеенного выравнивания можно уменьшить с $O(|T|b + |T|k^2)$ до $O(|T|b + |T|k)$. Этот улучшенный метод по своему характеру похож на задачу одномерной сцепки, разобранную в п. 13.3. В нем идет сканирование S слева направо, но он требует дополнительных усилий в схеме ДП, когда обрабатывается левый конец блока. Вместо того чтобы просто взять отдельное значение для блока, метод должен вычислить соотношения ДП, чтобы получить значение использования этого блока в выравнивании со строкой $T[1..l]$ для каждой позиции l . Для любого блока B_i пусть $C(B_i, l)$ будет значением наилучшего выравнивания $T[1..l]$ и любой цепи, содержащей подмножество блоков из множества от B_1 до B_i . Если B_i — последний блок, кончающийся до начала блока B_{i+1} , то для любой позиции l в T , $C(B_i; l)$ равно максимуму из $C(B_i, l)$ и является значением выравнивания, полученного при использовании всех блоков B_i в выравнивании, заканчивающемся в позиции l в T . Это последнее значение, которое вычисляется с помощью ДП, когда достигается левый край B_i .

Восполните детали метода, докажите его корректность и оцените время счета.

12. Подробно объясните, почему в методе Липтона нити, содержащие только олиги связи, могут игнорироваться. Объясните, как они автоматически удаляются при вычислении первого конъюнкта.
13. Как описано в п. 18.3.1, метод Липтона для задачи выполнимости определяет только, есть ли значения переменных, удовлетворяющие формуле. Даже когда формула может быть удовлетворена, метод не описывает явно, какие значения присвоить переменным. Конечно, можно расшифровать любую строку ДНК, полученную в конце метода, но это крайне нежелательно.

Предложите простое расширение метода Липтона (использующее описанные в тексте биотехнические операции) для эффективного определения набора значений переменных, при котором удовлетворяется формула, когда она выполнима. Один из подходов применяет общую технику *саморедукции*.

Модели мутаций на геномном уровне

19.1. Введение

Средства поиска, редактирования и выравнивания строк широко применялись при изучении эволюции молекул. Однако их использование ориентировалось прежде всего на сравнение строк, представляющих отдельные гены или белки. Например, при изучении эволюции обычно выбирали отдельный белок и проверяли, чем его аминокислотная последовательность различается у разных видов. Соответственно, алгоритмы редактирования и выравнивания строк руководствовались целевыми функциями, моделирующими наиболее общие типы мутаций на уровне отдельного гена или белка, это точковые мутации или замены аминокислот, вставки или удаления отдельных символов и блоковые вставки и удаления (пропуски).

Недавно внимание было привлечено к мутациям, которые встречаются в масштабе, существенно большем, чем у отдельного гена. Эти мутации встречаются на уровне хромосомы или генома и играют центральную роль в эволюции генома в целом. Эти крупномасштабные мутации имеют черты, которые совершенно отличают их от мутаций генного и белкового уровня. Чем больше становится доступно молекулярных данных на уровне генома, тем больше можно сделать крупномасштабных строковых сравнений, проливающих свет на эволюцию, сравнений, которые не видны на уровне отдельного гена или белка.

Движущей силой эволюции генома является “дупликация с модификацией” [126, 128, 301, 468]. В нем части генома дублируются, возможно, очень далеко от исходного места и затем модифицируются. Другие важные мутации геномного уровня включают *инверсии*, когда сегмент ДНК обращается; *транслокации*, когда обмениваются

концевыми участками две хромосомы (тэломеры); и *транспозиции*, когда меняются местами два смежных сегмента ДНК. Эти типы крупномасштабных мутаций в совокупности называются *перестройками генома*.

19.1.1. Перестройки генома проливают новый свет на эволюцию

Изучение геномных перестроек может привести к некоторым выводам об эволюции, которые нельзя было сделать по сравнениям генного уровня. Во многих организмах мутации на уровне генома происходят медленнее, чем на генном уровне. Это позволяет сравнивать молекулярные данные видов, которые дивергированы очень давно. Например, для установления порядка дивергенции трех видов традиционно вычисляют три попарных сходства, используя белок, общий для этих трех видов. Затем, исходя из того, что значения сходства “состоительны”, делают вывод, что пара с наибольшим сходством дивергировала от общего предка, который ранее дивергировал с третьим видом. Но если время дивергенции было достаточно давно в прошлом, накопленные мутации генов могли быть столь велики, что все три попарных значения сходства стали нераспознаваемыми и неприменимыми для надежного определения тонких деталей истории дивергенций. Напротив, более медленно происходящие перестройки генома могут позволять подробные исторические выводы. Хотя ген мутирует на уровне ДНК, его все еще можно распознать как один и тот же ген в двух различных видах по его белковому продукту и/или функции. Это позволяет сравнить порядок, в котором гены появляются в хромосоме в различных видах, и по различиям порядка генов делать выводы об истории эволюции в далеком прошлом.

На другом конце континуума находится ДНК, которая предельно устойчива и не накапливает многих мутаций генного уровня, но показывает перестройки генома. Это верно, например, для митохондриальных ДНК растений*):

“... митохондриальный геном *Brassica* может рассматриваться как коллекция неизменных последовательностей, относительное расположение которых предельно подвижно” [355].

Близкий пример, где перестройка генома может быть информативна, дает X-хромосома млекопитающих:

“Гены X-хромосомы были менее мобильны в ходе эволюции, чем аутосомные гены. Ген, найденный в X-хромосоме в одном из видов млекопитающих, с высокой вероятностью найдется в X-хромосоме фактически всех других млекопитающих” [317].

Это позволяет изучать *порядок генов* в X-хромосомах, сравнивая всех млекопитающих.

Работа над алгоритмами для анализа перестроек генома началась всего несколько лет назад, но уже появилась впечатляющая литература (см. [272, 270, 271, 40, 39, 209, 211, 212, 213, 154, 61, 261, 38, 390]). Мы детально изложим здесь один из первых результатов, иллюстрирующий дух большинства выполненных работ.

*) Этот момент может ввести в заблуждение, так как последовательность митохондриальной ДНК у животных видов мутирует гораздо быстрее, чем у ядерных ДНК, и поэтому обычно используется в точных молекулярных часах для очень коротких временных интервалов [468].

19.2. Инверсионные перестройки генома

В этом параграфе мы опишем часть исходной работы Кесесиоглу и Санкова [271, 272] по перестройкам генома, вызванным только *инверсиями*, другие типы перестроек будут исключены. Такое ограничение предпринято отчасти ради концептуальной простоты, чтобы быстрее дойти до вычислительных вопросов, но есть ДНК (например, митохондриальные ДНК растений [355]), где инверсии в чистом виде являются домinantной формой мутаций.

Для начала предположим, что гены хромосом различимы, так что им можно составить разные числовые метки. Это вполне разумное начальное предположение, так как большинство генов встречается в одном экземпляре, и даже гены, которые мутировали на уровне ДНК, могут быть распознаны (по оставшемуся сходству последовательностей) и получат одинаковый номер. Поэтому гены пронумерованы от 1 до n в исходной хромосоме (это упорядочение называется *тождественной перестановкой*) до того, как имели место инверсии. Хромосомы после инверсий представляются *перестановкой* целых чисел от 1 до n (рис. 19.1).

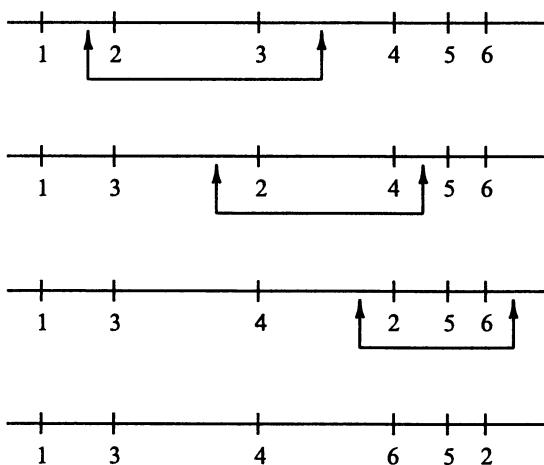


Рис. 19.1. Каждая линия представляет “хромосому” с шестью генами. Три инверсии преобразуют верхнюю хромосому в нижнюю. Две стрелки ограничивают инвертируемый сегмент. Первая хромосома представлена тождественной перестановкой 1, 2, 3, 4, 5, 6; вторая — перестановкой 1, 3, 2, 4, 5, 6; третья — 1, 3, 4, 2, 5, 6; и последняя — 1, 3, 4, 6, 5, 2

Пусть теперь даны две перестановки, представляющие порядок генов в хромосомах двух видов. Мы хотели бы определить *инверсионное расстояние* между двумя видами. Это расстояние должно отражать инверсии молекулярного уровня, которые могут преобразовать один порядок генов в другой. Значит, нам хотелось бы знать, как вторая перестановка может отличаться (или отличается) от первой перестановки рядом естественных инверсий молекулярного уровня. К сожалению, полная модель инверсий в ДНК неизвестна, и никто не знает, как следует делать историческую реконструкцию или каково должно быть наиболее осмысленное определение *инверсионного расстояния*. Однако, так как инверсии относительно редки в хромосомах

и кажутся случайными в сегментах, выбранных для инверсии, обычно прибегают к критерию бережливости.

Определение. По критерию бережливости инверсионное расстояние между двумя перестановками определяется как *минимальное* число инверсий, которые могут преобразовать одну перестановку в другую.

Редукция

Общая задача вычисления инверсионного расстояния между двумя произвольными перестановками эквивалентна задаче преобразования *одной* перестановки в *тождественную*. Эквивалентность легко установить, если принять одну из перестановок за основную, считать ее тождественной и соответственно переопределить другую перестановку. При таком сведении общая задача вычисления инверсионного расстояния превращается в задачу вычисления минимального числа инверсий, переводящих перестановку в тождественную. Об этой задаче известно, что она NP-трудная [87].

Мы представим простой метод [271, 272], который преобразует любую исходную перестановку Π в тождественную, используя число инверсий, превосходящее инверсионное расстояние не более чем вдвое.

19.2.1. Определения и начальные факты

Так как порядок генов в хромосоме представляется перестановкой, нам сначала понадобится терминология для обсуждения перестановок.

Определение. Пусть $\Pi = \pi_1, \pi_2, \pi_3, \dots, \pi_n$ представляет перестановку целых чисел от 1 до n , где π_i — номер в i -й позиции.

Например, если $\Pi = 3, 2, 4, 1$, то $\pi_1 = 3, \pi_2 = 2, \pi_3 = 4$ и $\pi_4 = 1$. Отметим, что π_{i+1} — это число справа от π_i в Π , тогда как $\pi_i + 1$ — это число, на единицу большее π_i .

Определение. *Точка разрыва* в Π возникает между двумя числами π_i и π_{i+1} для $1 \leq i \leq n - 1$ в том и только том случае, если $|\pi_i - \pi_{i+1}| \neq 1$. Будем считать, что имеется точка разрыва в начале Π , если $\pi_1 \neq 1$, и в конце Π , если $\pi_n \neq n$.

Например, в перестановке $\Pi = 3, 2, 4, 5, 1$ есть точки разрыва и в начале, и в конце, а также между 2 и 4 и между 5 и 1. Но между 3 и 2 точки разрыва нет, даром что они стоят не по порядку.

Определение. Пусть $\phi(\Pi)$ обозначает число точек разрыва в перестановке Π .

Лемма 19.2.1. *Инверсионное расстояние любой перестановки Π не меньше $\phi(\Pi)/2$.*

Доказательство. Отдельная операция инверсии может уменьшить число точек разрыва не более чем на два, удаляя по одному разрыву на каждом конце инвертируемого сегмента. Так как тождественная перестановка не содержит точек разрыва, потребуется не менее $\phi(\Pi)/2$ операций инверсии для преобразования перестановки с $\phi(\Pi)$ точками разрыва в тождественную. \square

Определение. Полоса в Π — это максимальный подинтервал в Π , не содержащий точек разрыва.

Например, $\Pi = 1, 5, 6, 7, 4, 3, 2$ содержит три полосы: просто 1, затем 5, 6, 7 и, наконец, 4, 3, 2.

Определение. Полоса называется *возрастающей*, если числа в ней возрастают, и *убывающей*, если убывают. Полоса, состоящая из одного числа, считается *убывающей*.

Например, первая и последняя полосы предыдущего примера убывающие, а вторая полоса — возрастающая. Легко установить, что любая полоса либо возрастаёт, либо убывает. Убывающие полосы будут играть основную роль в предлагаемой эвристике.

19.2.2. Эвристика

Сначала предложим алгоритм, который никогда не использует больше чем $2\varphi(\Pi)$ операций инверсии при любой перестановке Π . Следовательно, число инверсий, которое он делает, не более чем в четыре раза превосходит число инверсий для оптимального алгоритма.

Лемма 19.2.2. Предположим, что Π — нетождественная перестановка, не содержащая убывающих полос. Тогда существует инверсия Π , которая не увеличивает числа точек разрыва, при этом перестановка, которая после нее получается, содержит убывающую полосу длины не менее двух.

Доказательство. Так как убывающих полос нет, то каждая полоса возрастает, и следовательно, каждая полоса содержит не менее двух чисел. Далее, если Π не является еще тождественной перестановкой, то должна найтись полоса, имеющая точку разрыва на каждом конце. Инверсия этой полосы порождает убывающую полосу без увеличения числа точек разрыва. \square

Лемма 19.2.3. Если Π содержит убывающую полосу, то есть инверсия, которая уменьшает число точек разрыва по крайней мере на 1.

Доказательство. Выберем среди убывающих полос ту, у которой значение в правом конце наименьшее. Обозначим это значение через π_i . Ясно, что π_{i+1} не может равняться $\pi_i - 1$ (иначе $\pi_i - 1$ входило бы в убывающую полосу) и не может равняться $\pi_i + 1$ (тогда π_i входило бы в возрастающую, а не в убывающую полосу). Следовательно, должна быть точка разрыва между π_i и π_{i+1} . По аналогичным соображениям, должна быть точка разрыва между числом $\pi_i - 1$ и его соседом справа. Далее, $\pi_i - 1$ расположено справа или слева от π_i . В первом случае мы можем инвертировать интервал, начинающийся числом π_{i+1} и кончающийся числом $\pi_i - 1$ (включительно), чтобы соединить π_i с $\pi_i - 1$, удалив этим точку разрыва и продолжив убывающую полосу, содержащую π_i (рис. 19.2). Эта инверсия может удалить или создать точку разрыва на другом конце, но так как там была точка разрыва до инверсии, то случиться может только уменьшение. Следовательно, число точек разрыва уменьшается не меньше чем на 1 и, возможно, на 2.

Второй случай, когда $\pi_i - 1$ находится слева от π_i , симметричен, и его доказательство оставляется читателю. \square

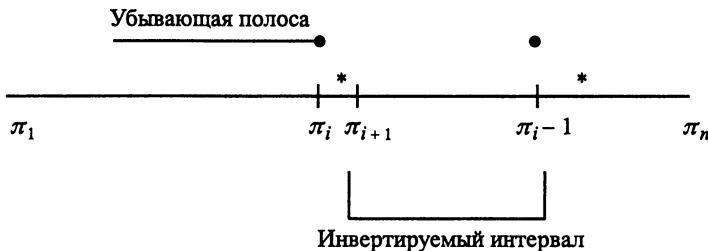


Рис. 19.2. Элемент π_i является наименьшим числом, содержащимся в какой-либо убывающей полосе. Имеются точки разрыва (помеченные звездочками) после π_i и после π_{i-1} . Инвертирование интервала между π_{i+1} и π_{i-1} уменьшает число точек разрыва не меньше чем на 1

Из лемм 19.2.2 и 19.2.3 сразу получается алгоритм с числом инверсий, превосходящим инверсионное расстояние не более чем в четыре раза.

Первая эвристика

```

while в перестановке есть точка разрыва do begin
    if есть убывающие полосы then
        найти и инвертировать ту, которая уменьшает число точек разрыва.
        (Обосновано леммой 19.2.3.)
    else (если убывающих полос нет)
        найти и инвертировать возрастающую полосу (создав убывающую полосу)
        без увеличения числа точек разрыва. (Обосновано леммой 19.2.2.)
end;

```

Теорема 19.2.1. *Первая эвристика использует для преобразования Π в тождественную перестановку не более $2\varphi(\Pi)$ операций инверсии. Следовательно, используемое ею число инверсий не более чем в четыре раза превосходит инверсионное расстояние.*

Доказательство. При использовании первой эвристики число точек разрыва убывает не меньше чем на 1 после каждой второй инверсии. Следовательно, число инверсий не больше удвоенного числа первоначальных точек разрыва, или $2\varphi(\Pi)$. Так как инверсионное расстояние не меньше, чем $\varphi(\Pi)/2$, отсюда следует утверждение теоремы. \square

Улучшение оценки

Если бы каждая инверсия, которая уменьшает число точек разрыва, также оставляла убывающую полосу, то число инверсий, используемых первой эвристикой, не превосходило бы $\varphi(\Pi)$ и, значит, удвоенного инверсионного расстояния. К сожалению, иногда таких удачных инверсий нет. Однако ситуация от этого усложняется не сильно.

Лемма 19.2.4. *Пусть Π — перестановка с убывающей полосой. Предположим, что нет инверсий, которые бы и уменьшали число точек разрыва, и оставляли убывающую полосу. Тогда найдется инверсия Π , которая уменьшает число точек разрыва на 2.*

Доказательство. Рассмотрим убывающую полосу с наименьшим числом π_i , содержащимся в убывающих полосах. В доказательстве леммы 19.2.3 мы показали, что если $\pi_i - 1$ лежит справа от π_i , то инвертирование интервала, начинающегося в π_{i+1} и кончивающегося в $\pi_i - 1$ (включительно), соединит π_i с $\pi_i - 1$, удалив точку разрыва. Более того, эта инверсия сохранит убывающую полосу, в которую входит число π_i , нарушая предположение леммы. Следовательно, число $\pi_i - 1$ должно быть слева от π_i .

Теперь выберем среди убывающих полос ту, которая содержит *наибольшее* число π_j . По аргументам, симметричным случаю π_i , число $\pi_j + 1$ должно лежать справа от π_j . Далее, π_j должно быть слева от π_i , так как в противном случае инверсия интервала, начинающегося в $\pi_i - 1$ и кончивающегося сразу перед π_i , уменьшила бы число точек разрыва на 1 и при этом сохранила бы существующую убывающую полосу, содержащую π_j , вопреки предположению леммы. Так же и π_j должно быть справа от $\pi_i - 1$, так как, если бы оно располагалось слева и от π_i , и от $\pi_i - 1$, то инвертирование все того же интервала уменьшило бы число точек разрыва и оставило убывающую полосу, опять-таки вопреки предположению. Аналогично, $\pi_j + 1$ должно быть справа от π_i . Рис. 19.3 иллюстрирует эту ситуацию.

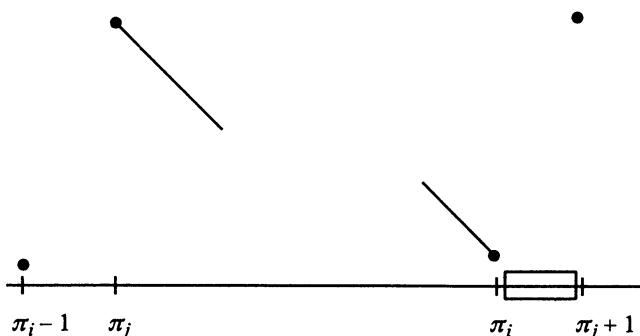


Рис. 19.3. Элемент $\pi_i - 1$ должен быть слева от элемента π_j , π_j — слева от π_i , а π_i — слева от $\pi_j + 1$

Теперь мы утверждаем, что число $\pi_j + 1$ должно стоять сразу после π_i , т.е. $\pi_j + 1 = \pi_{i+1}$, и прямоугольник на рис. 19.3 пуст. Доказывая от обратного, предположим, что в нем есть числа. В нем не может быть убывающих полос, так как если бы они были, то инверсия уже знакомого интервала, начинающегося с $\pi_i - 1$ и кончивающегося в π_i , уменьшила бы число точек разрыва на единицу и оставила бы в прямоугольнике убывающую полосу, вопреки предположению леммы. Так что если бы прямоугольник что-нибудь содержал, то он содержал бы возрастающую полосу. Однако тогда инверсия интервала, начинающегося с π_j и кончивающегося сразу перед $\pi_j + 1$, уменьшила бы число точек разрыва на 1 и перевела бы возрастающую полосу из прямоугольника в убывающую полосу, что снова привело бы к противоречию. Следовательно, прямоугольник должен быть пуст.

Симметричное рассуждение показывает, что число $\pi_i - 1$ должно непосредственно предшествовать π_j . Отсюда следует, что инверсия интервала, начинающегося в π_j и кончивающегося в π_i , удалит точки разрыва в обоих концах и, следовательно, уменьшит число точек разрыва на 2. \square

Лемма 19.2.4 обеспечивает следующее улучшение эвристического алгоритма:

Вторая эвристика

```

while в перестановке есть точка разрыва do begin
    if есть инверсия, уменьшающая число точек разрыва
    и оставляющая убывающую полосу
        then выполнить ее
    else (если такой инверсии нет) begin
        найти инверсию, которая уменьшает число точек разрыва на 2,
        и выполнить ее.
        Найти и инвертировать возрастающую полосу,
        не увеличивая числа точек разрыва.
    end;
end;

```

Теорема 19.2.2. *Вторая эвристика использует не более $\varphi(\Pi)$ операций инверсии при преобразовании перестановки Π в тождественную. Следовательно, она использует инверсий не более чем вдвое по сравнению с оптимальным алгоритмом.*

Доказательство. Во второй эвристике число точек разрыва уменьшается не меньше чем на 1 при каждом выполнении инверсии, оставляющей убывающую полосу. Когда инверсия не оставляет убывающей полосы, она уменьшает на 2 число точек разрыва, и за ней следует инверсия, которая не увеличивает числа точек разрыва, но создает убывающую полосу. Следовательно, число инверсий не превосходит $\varphi(\Pi)$, числа исходных точек разрыва. Отсюда следует утверждение теоремы, так как любой алгоритм должен выполнить не менее $\varphi(\Pi)/2$ операций инверсии. \square

Выполнив более тщательный анализ и заставив алгоритм “глядеть в будущее” больше чем на одну инверсию, Бафна и Певзнер [40] улучшили вторую эвристику и снизили границу ошибки с 2 до 1.75.

19.3. Знакопеременные инверсии

Кесесиоглу и Санков [272] ввели также знакопеременный вариант задачи инверсии. В задаче знакопеременной инверсии каждое число в перестановке имеет знак (+ или -), который изменяется каждый раз, когда число содержится в инвертируемом интервале. Например, когда инвертируются средние три числа в $+5, -2, -4, +3, -1$, получится $+5, -3, +4, +2, -1$. Задача заключается в том, чтобы минимизировать число операций инверсии, переводящих данную исходную знакопеременную перестановку в тождественную, где знак всех чисел должен быть положительным.

Задача о знакопеременной инверсии лучше моделирует эволюцию ДНК, чем обычный вариант задачи. Знакопеременные перестановки моделируют тот факт, что ДНК состоит из двух нитей и что ген экспрессируется (или работает) только на одной из двух нитей. Когда ДНК инвертируется, она также отражается, так что инвертированный сегмент меняет нить, на которой он находится. Следовательно, преобразование одного порядка генов в другой должно не только создавать правильный

порядок, но и размещать каждый ген на правильной нити. Наличие знаков кодирует это требование. Исходная знакопеременная перестановка создается из двух порядков генов следующим образом. Перестановка без знака Π получается в точности, как описано редукцией, используемой для перестановки без знака. Чтобы назначить знаки, припишем положительный знак какому-либо числу из Π , например 1. Затем каждому другому числу i из Π припишем отрицательный знак в том и только том случае, если ген находится на той же нити, что и ген 1, строго в одном из двух порядков генов. Таким образом, отрицательный знак показывает, что ген i должен находиться в нечетном числе инверсий, чтобы в конце оказаться на правильной нити, тогда как положительный знак показывает, что число инверсий должно быть четным.

Задача о знакопеременной инверсии может показаться труднее, чем задача об инверсии без знака, так как нужно менять и порядок генов, и знаки. Но на самом деле эта задача легче. При соответствующей модификации определений возрастающей и убывающей полос (чтобы учесть существование знаков) в [272] было показано, что вторая эвристика не меняется, и теорема 19.2.2 остается верной и для задачи о знакопеременной инверсии (см. упражнение 1). Содержащаяся в ней граница была затем в [40] уменьшена до 1.5, а Ханненхалли и Певзнер [211] показали, в противоположность случаю без знака, что задачу о знакопеременной инверсии можно решить за полиномиальное время. Временная оценка была затем уменьшена до почти $O(n^2)$ [61], а в дальнейшем — до $O(n^2)$ [261].

Знакопеременная версия задачи будет становиться в дальнейшем более правильной, когда будет доступно гораздо больше молекулярных последовательностей геномного уровня. Однако сейчас большинство известных порядков генов описано без ориентации, и следовательно, версия без знаков лучше всего моделирует существующие данные. В действительности, алгоритмические задачи о перестройках геномов, когда даже порядок генов неизвестен, изучались¹ [154, 390], и они моделируют многие из существующих данных по крупномасштабным перестройкам генома.

19.4. Упражнения

- В знакопеременной перестановке Π *смежность* определяется как пара последовательных чисел вида $+i$, $+(i+1)$ или $-(i+1)$, $-i$. Точка разрыва находится между двумя последовательными числами, которые не образуют смежности. Кроме того, есть точка разрыва в начале Π , если первое число отлично от +1, и в конце Π , если последнее число отлично от +n.

Уточните определения возрастающей и убывающей полос так, чтобы второй эвристический алгоритм был применим к задаче о знакопеременной инверсии и сохранялись все леммы и теорема 19.2.2.

- Предположим, что строка S' получается из строки S одной инверсией. Есть ли стандартная модель выравнивания, которую можно было использовать для выравнивания S и S' таким образом, чтобы выявить инверсию? Если да, то объясните необходимость новых алгоритмов для изучения инверсий.
- Эволюция путем рекомбинирования.** При некоторых обстоятельствах популяция организмов может быстро изменяться в процессе повторяющихся в мейозе рекомбинаций. Алгоритмические вопросы воссоздания истории таких рекомбинаций изучались в [268]. Следующие упражнения выделяют наиболее важный результат.

Пусть S_1 и S_2 — две строки, длины n каждая. *Равное перекрещивание* S_1 и S_2 в позиции i может создать любую из двух новых строк, $S_1[1..i]S_2[i+1..n]$ или $S_2[1..i]S_1[i+1..n]$. Предложите алгоритм со временем $O(n)$, который, получив три строки, S_1 , S_2 и S_3 , определяет, нет ли такой позиции i , что S_3 получается из S_1 и S_2 равным перекрещиванием.

Теперь рассмотрите эту задачу с той модификацией, что S_3 может быть меньше, чем S_1 или S_2 .

- Пусть \mathcal{S} — набор из k строк равной длины с суммарной длиной m . Предложите алгоритм со временем $O(m)$ для предварительной подготовки множества \mathcal{S} , такой чтобы по любым трем строкам из \mathcal{S} за *константное время* можно было бы определить, нельзя ли получить третью строку из первых двух равным перекрещиванием в какой-либо позиции.

Совет. Использовать суффиксное дерево. Предложите более простой алгоритм со временем подготовки $O(k^2 + m)$.

- Теперь мы моделируем эволюцию некоторой популяции путем кроссинговеров. Первоначально популяция состоит из трех строк, по которым равным кроссинговером образуется третья строка. В этот момент какая-то пара из трех строк (может быть, исходная пара) создает равным кроссинговером четвертую строку. На каждом шаге какая-то пара из существующих строк образует новую пару равным кроссинговером. Исходная пара строк называется *протопарой* для получающегося таким образом набора \mathcal{S} из k строк.

Рассмотрим ситуацию, когда заключительная популяция \mathcal{S} известна, а история \mathcal{S} не известна. Вычислительная задача заключается в воссоздании правдоподобной истории \mathcal{S} . Предположим, что была выполнена предварительная подготовка со временем $O(m)$ из предыдущей задачи. Предложите алгоритм получения частичного набора \mathcal{S}' из набора \mathcal{S} и строки S_i из \mathcal{S} . Кроме того, он должен уметь определить за время $O(k)$, есть ли пара в \mathcal{S}' , из которой можно получить S_i . Этот алгоритм будет использован в следующем задании.

Предположим снова, что выполнена предварительная подготовка со временем $O(m)$. Пусть задано множество \mathcal{S} из k строк и назначенная пара S_1, S_2 из \mathcal{S} . Предложите алгоритм со временем $O(k^2)$, определяющий, являются ли S_1, S_2 протопарой для \mathcal{S} .

- Используя результаты предыдущих упражнений, можно за время $O(m + k^4)$ проверить, содержит ли множество \mathcal{S} протопару. Выполните предварительную подготовку, а затем проверьте для каждой пары из \mathcal{S} , может ли она быть протопарой. Это время можно уменьшить до $O(m + k^3)$, так как для каждой строки $S_i \in \mathcal{S}$ есть не больше одной другой строки в \mathcal{S} , которую можно было бы присоединить к S_i , чтобы образовать протопару для \mathcal{S} . Объясните, почему. Объясните также, как эффективно найти $k/2$ кандидатов в протопары.

Эпилог

В этой книге я пытался представить основные идеи, алгоритмы и технику, которые имеют широкий диапазон применений и которые, скорее всего, останутся важными, даже если изменятся сегодняшние интересы. Я также пытался объяснить основные причины, почему вычисления над строками и последовательностями продуктивны в биологии и останутся важными, даже если конкретные приложения изменятся. Но имея только 500 страниц (всего 285 639 слов, составленных из 1 784 996 символов, считая по английскому оригиналу), я не мог охватить некоторых алгоритмических методов и некоторых имеющихся и ожидаемых приложений.

Дополнительная техника

По поводу дополнительных результатов из чистой информатики (*computer science*) о точном сравнении читателю рекомендуется книга *Текстовые алгоритмы* М. Крочемора и У. Риттера [117]. Эта книга гораздо больше углубляется в некоторые вопросы чистой информатики, такие как периодичности в строках и параллельные алгоритмы. Обзор алгоритмов многострокового поиска и методов неточного сравнения см. в книге *Алгоритмы строкового поиска* Г. Стефена [421]. Дополнительное изложение вычислительной молекулярной биологии, особенно вероятностных и статистических вопросов, связанных со строками и последовательностями, см. в книге М. Уотермена *Введение в вычислительную биологию* [461]. Еще одно введение в комбинаторные и строковые задачи в вычислительной молекулярной биологии дает книга Дж. Сетубала и Дж. Майдениса *Введение в вычислительную молекулярную биологию* [402]. Вопросы вычислительной молекулярной биологии, более сосредоточенные на вопросах структуры белков, см. в написанной А. Леском главе *Вычислительная молекулярная биология* в [297]. Конкретные открытые технические вопросы из вычислительной биологии, ориентированной на последовательности, см. в книге П. Певзнера и М. Уотермена [371].

Где появляется вычислительная молекулярная биология?

Она появляется там, где появляется крупномасштабная молекулярная биология и где есть данные. Широкий обзор будущего см. в работе Эрика Ландера [291]. Пример из будущего см. в [474].

Какие есть хорошие задачи в вычислительной молекулярной биологии и какие они будут в будущем? Я точно не знаю, но различаю задачи *технологического* и *биологического* происхождения.

Задача дробовой сборки фрагментов последовательности служит хорошим примером задачи технологического происхождения. Похоже, что задачи технологического происхождения легче всего формализуются, и часто легко наблюдается непосредственная польза от таких формализаций, следовательно, задачи технологического происхождения привлекательны для специалистов по информатике, приобщающихся к вычислительной биологии. Однако биотехнология быстро изменяется, и многие вычислительные задачи очень чувствительны даже к малым изменениям, так что хорошие задачи технологического происхождения могут быстро появляться и исчезать.

Легко представить себе день, когда появится альтернативная технология крупномасштабной расшифровки, которая сделает задачу сборки фрагментов неактуальной. (На самом деле конец дробового секвенирования предсказывался, с надеждой, уже много раз в прошлом.) Задачи технологического происхождения всегда будут частью вычислительной биологии, но для работы над такими задачами мы должны обладать пониманием текущего состояния этой технологии. Знакомиться с такими задачами по старым учебникам небезопасно.

Примерами задач биологического происхождения являются: нахождение функционально значимых мотивов в семействе белковых последовательностей; развитие техники обнаружения альтернативных генетических кодов; развитие техники идентификации расширения горизонтального генного и инtronного трансфера; или развитие техники, помогающей понять роль повторов ДНК в эволюции геномов. Эти постановки задач размыты, и требуются конкретные формальные технические постановки. Например, в задаче о мотивах: каково определение мотива? можно ли описать мотив регулярным выражением? похоже ли, что мотивы можно выделить множественным выстраиванием? будут ли значимые мотивы лучше сжимаемыми, чем случайные подстроки? что такое семейство и чем оно определяется: сходством функций, структуры, истории эволюции, общим сходством последовательностей и т. п.?

Задачи биологического происхождения труднее формализовать, и конкретные формализации (и абстрактные модели, на которых они построены), может быть, труднее защитить или оценить, но удачно формализованные задачи биологического происхождения могут иметь значительно больший срок жизни, чем задачи технологического происхождения. Это происходит потому, что лежащие в их основе биологические вопросы более устойчивы, чем биотехнология. Более того, легче сделать общее предсказание того, какие типы задач биологического происхождения появятся в будущем. Есть предсказуемые тенденции, которые будут создавать конкретные задачи биологического происхождения по мере того, как будет исследоваться больше последовательностей и белковых структур.

Одна важная тенденция заключается в том, что будет появляться больше *коррелированных* данных о последовательностях благодаря растущему вниманию к изучению крупномасштабных, комплексных, на уровне генома взаимодействий между генами, между белками и между генами и белками. Корреляция принимает различные конкретные формы: корреляция между белковыми последовательностями и белковыми структурами (которая приводит к гомологическому моделированию белковой структуры); корреляция между мотивами последовательностей и чертами длинных участков геномной ДНК (которая ставит вопросы долгих взаимодействий); корреляция сходств многих родственных последовательностей (для которых будет оставаться важным множественное сравнение и выравнивание строк); корреляция между родственными генами различных организмов (которая приводит к вопросам о порядке генов и эволюции генома); корреляции между последовательностями из различных организмов (для дальнейшего использования знаний, полученных для модельных организмов или для изучения горизонтального трансфера гена); корреляция между последовательностями и функциями, собранными из конкретных линий клеток и тканей во времени и при меняющихся условиях (которая ставит вопрос о том, какие гены экспрессируются где и когда и как это выражение коррелировано с развитием, заболеванием или внешним стрессом); и корреляции между полиморфными свойствами последовательностей (собранными по всей популяции) и появлением заболеваний

или различий в функции. Уже есть проекты организации мониторинга выражения гена в конкретных клетках во времени со сбором всех произведенных иРНК и подсчетом того, как часто выражен конкретный ген. Такой род исследования дойдет в один прекрасный день и до уровня белка. Исследования, подобные этому, произведут огромное количество информации, которую нужно будет анализировать на компьютере: “Мы собираемся теперь получить карты экспрессии 100 000 различных генов. Счастливо это обсчитать!” [349].

Вследствие этого биологическая область внимания может меняться от анализа отдельного гена или отдельного белка до анализа целого пути либо сети, или до анализа целого генома, или до многоклеточного анализа, или до анализа связи многих локусов либо заболевания, или до анализа целой популяции, или до анализа множества организмов. Но несмотря на изменение в области внимания и увеличение сложности изучаемых систем, данные в форме последовательностей и их анализ останутся центральными в вычислительной биологии на долгое время. Крупномасштабный вопрос вычислительной молекулярной биологии, вопрос, стоящий за наиболее специфичными, формальными, строковыми задачами биологического происхождения, — это как “делать” или как будут получать “настоящую биологию”, исследуя данные о последовательностях. Анализ последовательностей останется с нами надолго после того, как расшифровка станет рутинной. Труженики вычислительной молекулярной биологии! Полнее развивайте и повышайте мощь подхода, основанного на строках! Специалисты по строковым алгоритмам! Формализуйте и решайте вычислительные строковые задачи, порожденные этим подходом!*)

Как нужно специалистам по информатике вести себя в вычислительной биологии? Совет специалистам по информатике будет двойным. Во-первых, изучайте настоящую биологию так много, как это будет возможно, — штудируйте книги по настоящей биологии и журналы (*Science*, *Nature* и *Trends in Genetics* особенно хороши для общего обзора), присутствуйте на биологических беседах и конференциях, беседуйте с биологами и серьезно рассматривайте вычислительные задачи и абстрактные модели, уже определенные биологами. Но, во-вторых, не ограничивайтесь уже formalizованными вычислительными задачами и построенным моделями и не разочаровывайтесь, если вы не можете сразу включить всю сложность биологических проблем в ваши formalизованные задачи.

После погружения в настоящую биологию постарайтесь выделить и изучать ваши собственные вопросы (особенно вопросы биологического происхождения), руководствуясь вашим пониманием биологии, целью безусловно решить “полную задачу” и вашим пониманием того, что возможно, а что нет в вычислительном отношении. Информатика принесет биологии наиболее серьезные достижения вследствие появления большого сообщества людей, которые понимают обе области и знают сильные стороны и нужды их обеих. Я не предлагаю, чтобы вы игнорировали задачи, поставленные биологами, — скорее хотелось бы, чтобы вы расширяли и модифицировали эти задачи вашими собственными вопросами. Цитируем Лероя Худа: “По мере того как информатики понимают биологию все больше и больше, они будут прокладывать свой собственный курс. И когда они пойдут по нему, мы не будем говорить им, что делать и куда идти. Они просто уйдут” [280].

*) Авторский текст был несколько изменен по форме, но не по сути. — Прим. перев.

В далекой перспективе сообщество биологически образованных информатиков окажет наибольшее воздействие, поступая так, как поступили молекулярные биологи — отбирая задачи и методы исследования, которые лучше всего соответствуют доступной и возможной технике; развивая новую технику, которая представляется ценной, даже если она не в совершенстве годится для существующих задач; сосредотачиваясь на модельных организмах (модельных вычислительных задачах); выгораживая управляемые, несколько упрощенные задачи, где можно достичь прогресса; работая как сообщество, чтобы строить на результатах других исследователей, добавляя с нарастанием более реалистические черты в задачу биологического происхождения. Этот непрямой и нарастающий подход вполне принят и в информатике, и в биологии, но он создаст много глупых, переидеализированных вычислительных задач, и они могут кое-кого разочаровать. Однако такой подход реалистичен. У него больше шансов привести к успеху, чем у преждевременных лобовых атак на самые сложные вычислительные задачи, по тем же причинам, по которым фундаментальные исследования на модельных организмах в биологии — это часто более продуктивный, хотя и менее прямой путь для получения практических идей в проблемах, встречающихся в более сложных организмах. Очень полезно представлять себе, что меньше одного процента всех известных микроорганизмов успешно культивировалось в лабораторных условиях, что создает ужасное смещение в сфере внимания лабораторных биологов. И тем не менее изучение этого одного процента привело к глубоким открытиям. Интенсивно глядеть под фонарным столбом, не пытаясь при этом распространить его свет по шире, — это часто и необходимая, и успешная стратегия.

Я предложу еще один совет специалистам по теоретической информатике. Больше сосредотачивайтесь на *биологическом качестве* вычисления, а не исключительно на уменьшении затрат времени и памяти. Так как формализация задачи биологического происхождения может быть трудной, должно проверяться биологическое качество получаемых результатов, с возможным последовательным изменением формальной задачи. Навыки специалистов по информатике в итерировании этого процесса могут быть очень важны (для нахождения практических решений в каждой из последовательных формальных задач). Идти на каждом шаге практическим путем может быть более важно, чем оптимально ускорять каждый конкретный шаг.

В заключение, изучайте настоящую биологию, широко беседуйте с биологами и работайте над задачами, важность которых для биологии известна. Но вдобавок пробуйте стать своим собственным консультантом по биологии. Руководствуясь настоящей биологией, выделяйте ваши собственные, решаемые технические вопросы. Будьте готовы к критике, что многие из этих вопросов слишком идеализированы, чтобы иметь непосредственное практическое применение, зная, что в долгой перспективе именно этим накопительным подходом (при многих его провалах) исследовательское сообщество лучше всего решает трудные задачи. Более же всего — будьте любопытны, получайте удовольствие и радуйтесь замечательным возможностям работать в такой увлекательной и важной области.

Библиография

- [1] *Trends in Genetics*. 1995. Vol. 11 (3). См. весь выпуск.
- [2] *Detecting dinosaur DNA* (четыре технических замечания разных авторов) // *Science*. 1995 (May 26). P. 1191–1193.
- [3] Abrahamson K. *Generalized string matching* // *SIAM J. Comput.* 1987. Vol. 16. P. 1039–1051.
- [4] Adleman L. *Molecules computation of solutions to combinatorial problems* // *Science*. 1994. Vol. 266. P. 1021–1024.
- [5] Agarwal P., States D. *The repeat pattern toolkit (RPT): Analyzing the structure and evolution of the C. elegans genome* // Proc. of the Second International Conference on Intelligent Systems for Molecular Biology. 1994. P. 1–9.
- [6] Agarwala R., Bafna V., Farach M., Narangyan B., Paterson M., Thorup M. *On the approximability of numerical taxonomy: fitting distances with trees* // Proc. 7th ACM-SIAM Symp. on Discrete Algs. 1996. P. 365–372.
- [7] Agarwala R., Fernandez-Baca D. *A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed* // *SIAM J. Comput.* 1994. Vol. 23. P. 1216–1224.
- [8] Aho A. *Algorithms for finding patterns in strings* // Handbook of Theoretical Computer Science / Ed.: J. van Leeuwen. MIT Press / Elsevier. 1990. Vol. A. P. 257–300.
- [9] Aho A., Corasick M. *Efficient string matching: an aid to bibliographic search* // Comm. ACM. 1975. Vol. 18. P. 333–340.
- [10] Aho A., Hopcroft J., Ullman J. *Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974. [Русск. пер.: Ахо А., Хопкрофт Дж., Ульман Дж. *Построение и анализ вычислительных алгоритмов*. М.: Мир, 1979.]
- [11] Alberts B., Bray D., Lewis J., Raff M., Roberts K., Watson J. *Molecular Biology of the Cell*. 3rd ed. New York: Garland Press, 1994.
- [12] Alizadeh F., Karp R., Newberg L., Weisser D. *Physical mapping of chromosomes: a combinatorial problem in molecular biology* // *Algorithmica*. 1995. Vol. 13. P. 52–76.
- [13] Alizadeh F., Karp R., Weisser D., Zweig G. *Physical mapping of chromosomes using unique probes* // *J. Comp. Biol.* 1995. Vol. 2. P. 159–184.
- [14] Allison L., Yee C. N. *Minimum message length encoding and the comparison of macro-molecules* // *Bull. Math. Biology*. 1990. Vol. 52. P. 431–453.
- [15] Alper J. *Putting genes on chips* // *Science*. 1994. Vol. 264. P. 1400.
- [16] Altschul S. *Amino acid substitution matrices from an information theoretic perspective* // *J. Mol. Biol.* 1991. Vol. 219. P. 555–565.
- [17] Altschul S., Boguski M., Gish W., Wooton J. *Issues in searching molecular sequence databases* // *Nature Genetics*. 1994. Vol. 6. P. 119–129.
- [18] Altschul S., Erickson B. W. *Optimal sequence alignment using affine gap costs* // *Bull. Math. Biology*. 1986. Vol. 48. P. 603–616.
- [19] Altschul S., Gish W., Miller W., Myers E. W., Lipman D. *A basic local alignment search tool* // *J. Mol. Biol.* 1990. Vol. 215. P. 403–410.

- [20] Altschul S., Lipman D. *Trees, stars, and multiple sequence alignment* // SIAM J. Appl. Math. 1989. Vol. 49. P. 197–209.
- [21] Altschul S. F., Gish W. *Local alignment statistics* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 460–480.
- [22] Amir A., Benson G., Farach M. *Alphabet-independent two-dimensional matching* // SIAM J. Comput. 1994. Vol. 23. P. 313–323.
- [23] Anderson A., Nilsson S. *Efficient implementation of suffix trees* // Software — Practice and Experience. 1995. Vol. 25. P. 129–141.
- [24] Apostolico A. *The myriad virtues of subword trees* // Combinatorics on Words / Eds.: A. Apostolico, Z. Galil. Springer-Verlag, 1985. Vol. 112. P. 85–96. (NATO ASI series.)
- [25] Apostolico A. *Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings* // Information Processing Lett. 1986. Vol. 23. P. 63–69.
- [26] Apostolico A., Giancarlo R. *The Boyer–Moore–Galil string searching strategies revisited* // SIAM J. Comput. 1986. Vol. 15. P. 98–105.
- [27] Apostolico A., Guerra C. *The longest common subsequence problem revisited* // Algorithmica. 1987. Vol. 2. P. 315–336.
- [28] Арлазоров В. Л., Диниц Е. А., Кронрод М. А., Фараджев И. А. *Об экономном построении транзитивного замыкания ориентированного графа* // Доклады АН СССР. 1966. Т. 194. С. 487–488.
- [29] Armen C., Stein C. *A 2 2/3-approximation algorithm for the shortest superstring problem* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 87–101.
- [30] Armen C., Stein C. *A 2 3/4-approximation algorithm for the shortest superstring problem*. Technical report. Dartmouth College. Computer Science Dept. 1994.
- [31] Ayala F. *The myth of Eve: molecular biology and human origins* // Science. 1995. Vol. 270. P. 1930–1936.
- [32] Baase S. *Computer Algorithms*. 2nd ed. Reading, MA: Addison-Wesley, 1988.
- [33] Bacon D., Anderson W. *Multiple sequence alignment* // J. Mol. Biol. 1986. Vol. 191. P. 153–161.
- [34] Baeza-Yates R., Gonnet G. *All-against-all sequence matching*. Неопубликованная заметка.
- [35] Baeza-Yates R., Gonnet G. *A new approach to text searching* // Comm. ACM. 1992. Vol. 35. P. 74–82.
- [36] Baeza-Yates R. A., Perleberg C. *Fast and practical approximate string matching* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 185–192.
- [37] Bafna V., Lawler E., Pevzner P. *Approximation algorithms for multiple sequence alignment* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 43–53.
- [38] Bafna V., Pevzner P. *Sorting by reversals: genome rearrangements in plant organelles and evolutionary history of X chromosome*. Technical report CSE-94-032. Penn. State. 1994.
- [39] Bafna V., Pevzner P. *Sorting by transpositions* // Proc. 6th ACM-SIAM Symp. on Discrete Algs. 1995. P. 614–621.
- [40] Bafna V., Pevzner P. *Genome rearrangements and sorting by reversals* // SIAM J. Comput. 1996. Vol. 25. P. 272–289.

- [41] Bairoch A. *PROSITE: a dictionary of sites and patterns in proteins* // Nucl. Acids Res. 1992. Vol. 20. P. 2013–2018.
- [42] Bairoch A., Bucher P. *PROSITE: recent developments* // Nucl. Acids Res. 1992. Vol. 22. P. 3583–3589.
- [43] Baker B. *A theory of parameterized pattern matching: algorithms and applications* // Proc. of the 25th ACM Symp. on the Theory of Computing. 1993. P. 71–80.
- [44] Baker T. *A technique for extending rapid exact-match string matching to arrays of more than one dimension* // SIAM J. Comput. 1978. Vol. 7. P. 533–541.
- [45] Baldauf S., Palmer J. *Animals and fungi are each other's closest relatives: congruent evidence from multiple proteins* // Proc. Nat. Acad. Sci. USA. 1993. Vol. 90 (11). P. 558–562.
- [46] Bandelt H. J. *Recognition of tree metrics* // SIAM J. Discrete Math. 1990. Vol. 3. P. 3–6.
- [47] Barinaga M. *Missing Alzheimer's gene found* // Research News in Science. 1995 (August). Vol. 269. P. 917–918.
- [48] Barlow D. *Gametic imprinting in mammals* // Science. 1995. Vol. 270. P. 1610–1613.
- [49] Barthélémy J. P., Guenoche A. *Trees and Proximity Representations*. New York: Wiley, 1991.
- [50] Barton G. J. *Computer speed and sequence comparison* (письмо редактору) // Science. 1992. Vol. 257. P. 1609.
- [51] Barton G. J., Sternberg M. J. *Evaluation and improvements in the automatic alignment of protein sequences* // Protein Eng. 1987. Vol. 1. P. 89–94.
- [52] Barton G. J., Sternberg M. J. *A strategy for rapid multiple alignment of protein sequences: confidence levels from tertiary structure comparisons* // J. Mol. Biol. 1987. Vol. 198. P. 327–337.
- [53] *Second annual meeting on DNA based computers* / Eds.: E. Baum, D. Boneh, P. Kaplan, R. Lipton, J. Reif, N. Seeman. DIMACS Workshop at Princeton University. 1996 (June).
- [54] Beardsley T. *Smart genes* // Scientific American. 1991 (August). P. 86–95.
- [55] Begley S., Rogers A. *It's all in the genes: in computational biology, scientists track elusive DNA stands through databases* // Newsweek. 1994 (September 5). P. 64. [Ссылка появилась благодаря Грею Теннону.]
- [56] Benner S. A., Cohen M. A., Gonnet G. H. *Response to Barton's letter: Computer speed and sequence comparison* // Science. 1992. Vol. 257. P. 1609–1610.
- [57] Benner S. A., Cohen M. A., Gonnet G. H. *Empirical and structural models for insertions and deletions in the divergent evolution of proteins* // J. Mol. Biol. 1993. Vol. 229. P. 1065–1082.
- [58] Benson D. *Digital signal processing methods for biosequence comparison* // Nucl. Acids Res. 1990. Vol. 18. P. 3001–3006.
- [59] Benson D. *Fourier method for biosequence analysis* // Nucl. Acids Res. 1991. Vol. 18. P. 6305–6310.
- [60] Benson G. *A space efficient algorithm for finding the best non-overlapping alignment score* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 1–14.
- [61] Berman P., Hannenhalli S. *Fast sorting by reversal* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 168–185.

- [62] Berman P., Ramaiyer V. *Improved approximations for the Steiner tree problem* // J. Algorithms. 1994. Vol. 17. P. 381–408.
- [63] Bieganski P. *Genetic sequence data retrieval and manipulation based on generalized suffix trees*. PhD thesis. Univ. Minnesota. Dept. Computer Science. 1995.
- [64] Bieganski P., Riedl J., Carlis J. V., Retzel E. R. *Generalized suffix trees for biological sequence data: applications and implementation* // Proc. of the 27th Hawaii Int. Conf. on Systems Sci. IEEE Computer Society Press. 1994. P. 35–44.
- [65] Bieganski P., Riedl J., Carlis J. V., Retzel E. R. *Motif explorer — a tool for interactive exploration of amino acid sequence motifs* // Proc. of the First Pacific Symp. on Biocomputing, IEEE Computer Society Press. 1996. P. 705–706.
- [66] Bird R. *Two dimensional pattern matching* // Information Processing Lett. 1977. Vol. 6. P. 168–170.
- [67] Birge R. *Protein-based optical computers and memories* // IEEE Computer. 1992 (November 25).
- [68] Birge R. *Protein-based computing* // Scientific American. 1995. Vol. 272. P. 90–95.
- [69] Blum A., Jiang T., Li M., Tromp J., Yannakakis M. *Linear approximation of shortest superstrings* // J. ACM. 1994. Vol. 41. P. 634–647.
- [70] Blumer A., Blumer J., Haussler D., Ehrenfeucht D., Chen A., Seiferas M. T. *The smallest automaton recognizing the subwords of a text* // Theor. Comp. Sci. 1985. Vol. 40. P. 31–55.
- [71] Blumer A., Blumer J., Haussler D., McConnell R., Ehrenfeucht D. *Complete inverted files for efficient text retrieval and analysis* // J. ACM. 1987. P. 578–595.
- [72] Bodlaender H., Fellows M., Warnow T. *Two strikes against perfect phylogeny* // Proc. of the 19th Int. Colloq. on Automata, Languages and Programming. 1992. P. 273–283.
- [73] Boguski M. S., Schuler G. D. *Establishing a human transcript map* // Nature Genetics. 1995. Vol. 10. P. 369–371.
- [74] Booth K., Lueker G. *Testing for the consecutive ones property, interval graphs and graph planarity testing using pq-tree algorithms* // J. Comp. Sys. Sci. 1976. Vol. 13. P. 333–379.
- [75] Boyer R. S., Moore J. S. *A fast string searching algorithm* // Comm. ACM. 1977. Vol. 20. P. 762–772.
- [76] Brenner S. E. *Network sequence retrieval* // Trends in Genetics. 1995. Vol. 11 (3). P. 247–248.
- [77] Breslauer D., Jiang T., Jiang Z. *Rotations of periodic strings and short superstrings*. Preprint. 1996 (May 30).
- [78] Broad W. J. *Clues to fiery origins of life sought in hothouse microbes* // New York Times. 1995 (May 9).
- [79] Brody J. *Multiple sclerosis. Will anyone recovered from it please communicate with patient* // New York Times. 1995 (May 3).
- [80] Browne M. *Critics see humbler origin in “dinosaur” DNA* // New York Times. 1995 (June 20).
- [81] Brutlag D. L., Dautricourt J. P., Maulik S., Relph J. *Improved sensitivity of biological sequence database searches* // Comp. Appl. Biosciences. 1990. Vol. 6. P. 237–245.
- [82] Bult C., Woese C. R., Venter J. C. et al. *Complete genome sequence of the methanogenic archaeon, Methanococcus jannaschii* // Science. 1996. Vol. 273. P. 1058–1072.
- [83] Buneman P. *The recovery of trees from measures of dissimilarity* // Mathematics in the Archaeological and Historical Sciences / Eds.: D. G. Kendall and P. Tautu. Edinburgh Univ. Press, 1971. P. 387–385.

- [84] Buneman P. *A characterization of rigid circuit graphs* // Discrete Math. 1974. Vol. 9. P. 205–212.
- [85] Buneman P. *A note on metric properties of trees* // J. Combinatorial Theory (B). 1974. Vol. 17. P. 48–50.
- [86] Cann R., Stoneking M., Wilson A. *Mitochondrial DNA and human evolution* // Nature. 1987. Vol. 325. P. 31–36.
- [87] Caprara A. *Sorting by reversals is difficult* // Proc. of RECOMB 97: The first international conference on computational molecular biology. ACM Press. 1997. P. 75–83.
- [88] Carrillo H., Lipman D. *The multiple sequence alignment problem in biology* // SIAM J. Appl. Math. 1988. Vol. 48. P. 1073–1082.
- [89] Cary R., Storno G. *Graph-theoretic approach to RNA modeling using comparative data* // Proc. of Intelligent Systems in Mol. Biol. 1995. P. 75–80.
- [90] Casey D., Cantor C., Spengler S. *Primer on Molecular Genetics*. US Dept. Energy, Human Genome Program. Washington, DC, 1992.
- [91] Caskey C., Eisenberg R., Lander E., Straus J. *Hugo statement on patenting of DNA* // Genome Digest. 1995. Vol. 2. P. 6–9.
- [92] Chan S., Wong A., Chiu D. *A survey of multiple sequence comparison methods* // Bull. Math. Biology. 1992. Vol. 54. P. 563–598.
- [93] Chang W. I., Lampe J. *Theoretical and empirical comparisons of approximate string matching algorithms* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 644. P. 175–184.
- [94] Chang W. I., Lawler E. L. *Sublinear expected time approximate string matching and biological applications* // Algorithmica. 1994. Vol. 12. P. 327–344.
- [95] Chao K., Pearson W., Miller W. *Aligning two sequences within a specified diagonal band* // Comp. Appl. Biosciences. 1995. Vol. 2. P. 6–9.
- [96] Chao K. M. *Computing all suboptimal alignments in linear space* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 1–14.
- [97] Chao K. M., Hardison R., Miller W. *Recent developments in linear-space alignment methods: a mini survey* // J. Comp. Biol. 1994. Vol. 1. P. 271–291.
- [98] Chee M., Fodor P. A. et al. *Accessing genetic information with high-density DNA arrays* // Science. 1996. Vol. 274. P. 610–614.
- [99] Cheever E., Overton G. Christian, Searls D. *Fast Fourier Transform-based correlation of DNA sequences using complex plane encoding* // Comp. Appl. Biosciences. 1991. Vol. 7. P. 143–154.
- [100] Chen T., Skiena S. *Trie-based data structures for sequence assembly*. Preprint. Department of Computer Science. Stony Brook N.Y. 1996 (July 11).
- [101] Chothia C. *One thousand families for the molecular biologist* // Nature. 1992. Vol. 357. P. 543–544.
- [102] Christofides N. *Worst-case analysis of a new heuristic for the travelling salesman problem*. Report 338. Grad. School Industrial Administration. Carnegie-Mellon. 1976.
- [103] Claverie J. M. *Detecting frame shifts by amino acid sequence comparison* // J. Mol. Biol. 1993. Vol. 234. P. 1140–1157.
- [104] Clift A., Haussler D., McConnell R., Schneider T. D., Storno G. *Sequence landscapes* // Nucl. Acids Res. 1986. Vol. 14. P. 141–158.

- [105] Cobbs A. *Fast approximate matching using suffix trees* // Proc. 6th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1995. Vol. 937. P. 41–54.
- [106] Cohen F. E. *Folding the sheets: using computational methods to predict the structure of proteins* // Calculating the Secrets of Life / Eds.: E. Lander and M. S. Waterman. National Academy Press, 1995. P. 236–271.
- [107] Cole R. Неопубликованная рукопись.
- [108] Cole R. *Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm* // SIAM J. Comput. 1994. Vol. 23. P. 1075–1091.
- [109] Commentz-Walter B. *A string matching algorithm fast on average* // Proc. of the 6th Int. Colloq. on Automata, Languages and Programming. 1979. P. 118–132.
- [110] Cooper G. *Oncogenes*. Boston, MA: Jones and Bartlett, 1990.
- [111] Cooper N. *The Human Genome Project*. Mill Valley, CA: Univ. Science Books, 1994.
- [112] Cormen T., Leiserson C., Rivest R. *Introduction to Algorithms*. Cambridge, MA: MIT Press and McGraw Hill, 1992. [Русск. пер.: Кормен Т., Лейзерзон Ч., Ривест Р. *Алгоритмы: построение и анализ*. М.: МЦНМО, 1999.]
- [113] Cox D., Burmeister M., Price E. R., Kim S., Myers R. *Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes* // Science. 1990. Vol. 250. P. 245–250.
- [114] Crochemore M. *An optimal algorithm for computing repetitions in a word* // Information Processing Lett. 1981. Vol. 12. P. 244–250.
- [115] Crochemore M. *Transducers and repetitions* // Theor. Comp. Sci. 1985. Vol. 45. P. 63–86.
- [116] Crochemore M., Rytter W. *Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays* // Theor. Comp. Sci. 1991. Vol. 88. P. 59–82.
- [117] Crochemore M., Rytter W. *Text Algorithms*. New York: Oxford Univ. Press, 1994.
- [118] Culberson J., Rudnicki P. *A fast algorithm for constructing trees from distance matrices* // Information Processing Lett. 1989. Vol. 30. P. 215–220.
- [119] Darwin C. R. *The Origin of Species*. London: John Murray, 1859. [Русск. пер.: Дарвин Чарлз. *Происхождение видов путем естественного отбора*. СПб.: Наука, 2001.]
- [120] Day W. H. *Computational complexity of inferring phylogenies from dissimilarity matrices* // Bull. Math. Biol. 1987. Vol. 49. P. 461–467.
- [121] Dayhoff M. O. *Computer analysis of protein evolution* // Scientific American. 1969 (July). P. 87–94.
- [122] Dayhoff M. O., Schwartz R. M., Orcutt B. C. *A model of evolutionary change in proteins* // Atlas of Protein Sequence and Structure. 1978 Vol. 5. P. 345–352.
- [123] Deacon N. J., Mills J. et al. *Genomic structure of an attenuated quasi species of HIV-1 from a blood transfusion donor and recipients* // Science. 1995. Vol. 270. P. 988–991.
- [124] Dembo A., Karlin S. *Strong limit theorems of empirical functions for large exceedances of partial sums of i.i.d. variables* // Ann. Prob. 1991. Vol. 19. P. 1737–1755.
- [125] Doggett N. A., Moyzis R. R. et al. *An integrated physical map of human chromosome 16* // Nature. 1995. Vol. 377. P. 335–365.
- [126] Doolittle R. F. *Similar amino acid sequences: Chance or common ancestry?* // Science. 1981. Vol. 214. P. 149–159.
- [127] Doolittle R. F. *Of Urfs and Orfs: A primer on How to Analyze Derived Amino Acid Sequences*. Mill Valley, CA: University Science Books, 1986.

- [128] Doolittle R. F. *Redundancies in protein sequences* // Prediction of Protein Structure and the Principles of Protein Conformation / Ed.: G. Fasman. New York: Plenum, 1989. P. 599–624.
- [129] Doolittle R. F. *Searching through sequence databases* // Methods in Enzymology. Vol. 183. Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences / Ed.: R. F. Doolittle. New York: Academic Press, 1990. P. 99–110.
- [130] Doolittle R. F. *What we have learned and will learn from sequence databases* // Computers and DNA / Eds.: G. Bell, T. Marr. Reading, MA: Addison-Wesley, 1990. P. 21–31.
- [131] Doolittle R. F., Feng R. F., Tsang S., Cho G., Little E. *Determining divergence times of the major kingdoms of living organisms with a protein clock* // Science. 1996. Vol. 271. P. 470–477.
- [132] Doolittle R. F., Hunkapiller M., Hood L. E., Devare S., Robbins K., Aaronson S., Antoniades H. *Simian sarcoma virus one gene v-sis, is derived from the gene (or genes) encoding a platelet-derived growth factor* // Science. 1983. Vol. 221. P. 275–277.
- [133] Dress A., Steel M. *Convex tree realizations of partitions* // Appl. Math. Lett. 1993. Vol. 5. P. 3–6.
- [134] Drmanac R., Hood L., Crkvenjakov R. et al. *DNA sequence determination by hybridization: a strategy for efficient large-scale sequencing* // Science. 1993. Vol. 260. P. 1649–1652.
- [135] Drmanac R., Labat I., Brukner I., Crkvenjakov R. *Sequencing of megabase plus DNA by hybridization: theory of the method* // Genomics. 1989. Vol. 4. P. 114–128.
- [136] Eppstein D. *Sequence comparison with mixed convex and concave costs* // J. Algorithms. 1990. Vol. 11. P. 85–101.
- [137] Eppstein D., Galil Z., Giancarlo R., Italiano G. F. *Sparse dynamic programming I: linear cost functions* // J. ACM. 1992. Vol. 39. P. 519–545.
- [138] Eppstein D., Galil Z., Giancarlo R., Italiano G. F. *Sparse dynamic programming II: convex and concave cost functions* // J. ACM. 1992. Vol. 39. P. 546–567.
- [139] Estabrook G., Johnson C., McMorris F. *An idealized concept of the true cladistic character* // Math. Bioscience. 1975. Vol. 23. P. 263–272.
- [140] Estabrook G., Johnson C., McMorris F. *An algebraic analysis of cladistic characters* // Discrete Math. 1976. Vol. 16. P. 141–147.
- [141] Estabrook G., Johnson C., McMorris F. *A mathematical foundation for the analysis of cladistic character compatibility* // Math. Bioscience. 1976. Vol. 29. P. 181–187.
- [142] Estabrook G., McMorris F. *When are two qualitative taxonomic characters compatible?* // J. Math. Bioscience. 1977. Vol. 4. P. 195–200.
- [143] Estabrook G., McMorris F. *When is one estimate of evolutionary relationships a refinement of another?* // J. Math. Bioscience. 1980. Vol. 10. P. 367–373.
- [144] Even S. *Graph Algorithms*. Mill Valley, CA: Computer Science Press, 1979.
- [145] Farach M., Kannan S., Warnow T. *A robust model for finding evolutionary trees* // Algorithmica. 1995. Vol. 13. P. 511–520.
- [146] Farach M., Noordewier M., Savari S., Shepp L., Wyner A., Ziv J. *On the entropy of DNA: algorithms and measurements based on memory and rapid convergence* // Proc. 6th ACM-SIAM Symp. on Discrete Algs. 1995. P. 48–57.
- [147] Feduccia A. *Explosive evolution in tertiary birds and mammals* // Science. 1995. Vol. 267. P. 637–638.

- [148] Felsenstein J. *Numerical methods for inferring evolutionary trees* // Quart. Rev. Biol. 1982. P. 379–404.
- [149] Felsenstein J. *Parsimony in systematics: biological and statistical issues* // Annual Rev. Ecol. Syst. 1983. Vol. 14. P. 313–333.
- [150] Felsenstein J. *Perils of molecular introspection* // Nature. 1988. Vol. 335. P. 118.
- [151] Felsenstein J. *Phylogenies from molecular sequences: inference and reliability* // Annual Rev. Genetics. 1988. Vol. 22. P. 521–565.
- [152] Felsenstein J., Sawyer S., Kochin R. *An efficient method for matching nucleic acid sequences* // Nucl. Acids Res. 1982. Vol. 10. P. 133–139.
- [153] Feng D., Doolittle R. F. *Progressive sequence alignment as a prerequisite to correct phylogenetic trees* // J. Mol. Evol. 1987. Vol. 25. P. 351–360.
- [154] Ferretti V., Nadeau J., Sankoff D. *Original synteny* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 149–167.
- [155] Fickett J. W. *Fast optimal alignment* // Nucl. Acids Res. 1984. Vol. 12. P. 175–180.
- [156] Fickett J. W. *Orfs and genes: How strong a connection?* // J. Comp. Biol. 1995. Vol. 2. P. 117–123.
- [157] Fischer M., Paterson M. *String-matching and other products* // Complexity of Computation / Ed.: R. M. Karp. SIAM-AMS Proc. 1974. P. 113–125.
- [158] Fischetti V., Landau G., Schmidt J., Sellers P. *Identifying periodic occurrences of a template with applications to protein structure* // Information Processing Lett. 1993. Vol. 45. P. 11–18.
- [159] Fitch W. M. *Toward defining the course of evolution: minimum changes for a specific tree typology* // Syst. Zoology. 1971. Vol. 20. P. 406–416.
- [160] Fitch W. M., Margoliash E. *The construction of phylogenetic trees* // Science. 1967. Vol. 155. P. 279–284.
- [161] Fitch W. M., Smith T. F. *Optimal sequence alignments* // Proc. Nat. Acad. Sci. USA. 1983. Vol. 80. P. 1382–1386.
- [162] Fleischmann R., Smith H., Venter J. C. et al. *Whole-genome random sequencing and assembly of Haemophilus influenzae rd* // Science. 1995. Vol. 269. P. 496–512.
- [163] Fodor S., Read J., Pirrung M., Stryer L., Lu A., Solas D. *Light-directed spatially addressable parallel chemical synthesis* // Science. 1991. Vol. 251. P. 767–773.
- [164] Foley P. L., Humphries C. J., Kitching I. L., Scotland R. W., Siebert D. J., Williams D. M. *Cladistics: A Practical Course in Systematics*. Oxford: Oxford Univ. Press, 1992.
- [165] Foulds L. R., Graham R. L. *The Steiner problem in phylogeny is NP-complete* // Advances Appl. Math. 1982. Vol. 3. P. 43–49.
- [166] Fredman M. L. *Algorithms for computing evolutionary similarity measures with length independent gap penalties* // Bull. Math. Biol. 1984. Vol. 46. P. 553–566.
- [167] Fredman M. L., Komlos J., Szemerédi E. *Storing a sparse table with O(1) worst case access time* // J. ACM. 1984. Vol. 31. P. 61–68.
- [168] Galil Z. *On improving the worst case running time of the Boyer–Moore string searching algorithm* // Comm. ACM. 1979. Vol. 22. P. 505–508.
- [169] Galil Z., Park K. *Alphabet-independent two-dimensional witness computations* // SIAM J. Comput. 1996. Vol. 25. P. 907–935.

- [170] Galil Z., Giancarlo R. *Speeding up dynamic programming with applications to molecular biology* // Theor. Comp. Sci. 1989. Vol. 64. P. 107–118.
- [171] Garey M., Johnson D. *Computers and Intractability*. San Francisco: Freeman, 1979.
- [172] Gelfand M. S. *Prediction of function in DNA sequence analysis* // J. Comp. Biol. 1995. Vol. 2. P. 87–115.
- [173] Gelfand M. S., Mironov A. A., Pevzner P. A. *Gene recognition via spliced alignment* // Proc. Nat. Acad. Sci. USA. 1996. P. 9061–9066.
- [174] Gelfand M. S., Mironov A. A., Pevzner P. A. *Spliced alignment: a new approach to gene recognition* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 141–158.
- [175] Gibbons A. *Mitochondrial Eve: wounded, but not dead yet* // Science. 1992. Vol. 257. P. 873–875.
- [176] Gibbs R. A. *Pressing ahead with human genome sequencing* // Nature Genetics. 1995. Vol. 11. P. 121–125.
- [177] Gibbs R. A., Nguyen P. N., Caskey C. T. *Detection of single DNA base differences by competitive oligonucleotide priming* // Nucl. Acids Res. 1989. Vol. 17. P. 2437–2448.
- [178] Giegerich R., Kurtz S. *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction* // Algorithmica. 1997. Vol. 19. P. 331–353.
- [179] Gilbert W. *Towards a paradigm shift in biology* // Nature. 1991. Vol. 349. P. 99.
- [180] Glanz J. *Computer scientists rethink their discipline's foundations* // Science. 1995. Vol. 269. P. 1363–1364.
- [181] Golumbic M. C., Kaplan H., Shamir R. *On the complexity of DNA physical mapping* // Advances Appl. Math. 1994. Vol. 15. P. 251–261.
- [182] Gonnet G., Baeza-Yates R. *An analysis of the Karp–Rabin string matching algorithm* // Information Processing Lett. 1994. Vol. 34. P. 271–274.
- [183] Gonnet G. H., Cohen M. A., Benner S. A. *Exhaustive matching of the entire protein sequence database* // Science. 1992. Vol. 256. P. 1443–1445.
- [184] Gonnet G. H., Baeza-Yates R. *Handbook of Algorithms and Data Structures*. 2nd ed. Reading, MA: Addison-Wesley, 1991.
- [185] Gorbalenya A., Koonin E. *Helicases: amino acid sequence comparisons and structure-function relationships* // Current Opinion Structural Biol. 1993. Vol. 3. P. 419–429.
- [186] Gotoh O. *An improved algorithm for matching biological sequences* // J. Mol. Biol. 1982. Vol. 162. P. 705–708.
- [187] Gotoh O. *Alignment of three biological sequences with an efficient traceback* // J. Theor. Biol. 1986. Vol. 121. P. 327–337.
- [188] Gould S. J. *Through a lens, darkly* // Natural History. 1989 (September). P. 16–24.
- [189] Green E., Green P. *Sequence-tagged site (STS) content mapping of human chromosomes: theoretical considerations and early experiences* // PCR Methods and Applications. Cold Spring Harbor Lab. Press, 1991. P. 77–90.
- [190] Green P., Lipman D., Hillier D., Waterston R., States D., Claverie J. M. *Ancient conserved regions in new gene sequences and the protein databases* // Science. 1993. Vol. 259. P. 1711–1716.
- [190a] Green P. *Against a whole-genome shotgun* // Genome Res. 1997. Vol. 7. P. 410–417.

- [191] Greenberg D. S., Istrail S. *Physical mapping by STS hybridization: algorithmic strategy and the challenge of software evaluation* // J. Comp. Biol. 1995. Vol. 2. P. 219–273.
- [192] Gribskov M., Devereux J. *Sequence Analysis Primer*. New York: Stockton Press, 1991.
- [193] Gribskov M., Luthy R., Eisenberg D. *Profile analysis* // Methods in Enzymology. Vol. 183. Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences / Ed.: R. F. Doolittle. New York: Academic Press, 1990. P. 146–159.
- [194] Gribskov M., McLachlan A., Eisenberg D. *Profile analysis detection of distantly related proteins* // Proc. Nat. Acad. Sci. USA. 1987. Vol. 88. P. 4355–4358.
- [195] Guan X., Uberbacher E. C. *Alignment of DNA and protein sequences containing frameshift errors* // Comp. Appl. Biosciences. 1996. Vol. 12. P. 31–40.
- [196] Guibas L. J., Odlyzko A. M. *A new proof of the linearity of the Boyer–Moore string searching algorithm* // SIAM J. Comput. 1980. Vol. 9. P. 672–682.
- [197] Gupta S., Kececioglu J., Schaffer A. *Making the shortest-paths approach to sum-of-pairs multiple sequence alignment more space efficient in practice* // Proc. 6th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1995. Vol. 937. P. 128–143.
- [198] Gusfield D. *The Steiner tree problem in phylogeny*. Technical Report No. 334. Yale Univ. Computer Science Dept. 1984.
- [199] Gusfield D. *An increment-by-one approach to suffix arrays and trees*. Technical Report CSE-90-39. UC Davis. Dept. Computer Science. 1990.
- [200] Gusfield D. *Efficient algorithms for inferring evolutionary history* // Networks. 1991. Vol. 21. P. 19–28.
- [201] Gusfield D. *Efficient methods for multiple sequence alignment with guaranteed error bounds* // Bull. Math. Biol. 1993. Vol. 55. P. 141–154.
- [202] Gusfield D. *Simple uniform preprocessing for linear-time string matching*. Technical Report CSE-96-5. UC Davis. Dept. Computer Science. 1996.
- [203] Gusfield D., Balasubramanian K., Naor D. *Parametric optimization of sequence alignment* // Algorithmica. 1994. Vol. 12. P. 312–326.
- [204] Gusfield D., Karp R., Wang L., Stelling P. *Graph traversals, genes and matroids: an efficient special case of the travelling salesman problem* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 304–319.
- [205] Gusfield D., Stelling P. *Parametric and inverse-parametric sequence alignment with XPARAL* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 481–494.
- [206] Gusfield D., Wang L. *New uses for uniform lifted alignments*. Technical Report CSE-96-4. UC Davis. Dept. Computer Science. 1996.
- [207] Halanych K., Bacheller J. D., Aguinaldo A. M., Liva S. M., Hillis D. M., Lake J. A. *Evidence from 18s ribosomal DNA that lophophorates are protostome animals* // Science. 1995. Vol. 267. P. 1641–1643.
- [208] Hanks M., Wurst W., Anson-Cartwright L., Auerback A., Joyner A. *Rescue of the en-1 mutant phenotype by replacement of en-1 with en-2* // Science. 1995. Vol. 269. P. 679–682.
- [209] Hannenhalli S. *Polynomial-time algorithm for computing translocation distance between genomes* // Proc. 6th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1995. Vol. 937. P. 162–176.

- [210] Hannenhalli S., Fellows W., Lewis H., Skiena S., Pevzner P. *Positional sequencing by hybridization* // Comp. Appl. Biosciences. 1996. Vol. 12. P. 19–24.
- [211] Hannenhalli S., Pevzner P. *Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals* // Proc. of the 27th ACM Symp. on the Theory of Computing. 1995. P. 178–189.
- [212] Hannenhalli S., Pevzner P. *Transforming mice into men: polynomial algorithm for genomic distance problem* // Proc. of the 36th IEEE Symp. on Foundations of Comp. Sci. 1995. P. 581–592.
- [213] Hannenhalli S., Pevzner P. *To cut or not to cut: applications of comparative physical maps in molecular evolution* // Proc. 7th ACM-SIAM Symp. on Discrete Algs. 1996. P. 304–313.
- [214] Harel D., Tarjan R. E. *Fast algorithms for finding nearest common ancestors* // SIAM J. Comput. 1984. Vol. 13. P. 338–355.
- [215] Harper R. *World Wide Web resources for the biologist* // Trends in Genetics. 1995. Vol. 11 (3).
- [216] Hartigan J. *Minimum mutation fits to a give tree* // Biometrics. 1972. Vol. 29. P. 53–65.
- [217] Hartigan J. *Clustering Algorithms*. New York: Wiley, 1975.
- [218] Hein J. *An optimal algorithm to reconstruct trees from additive distance data* // Bull. Math. Biol. 1989. Vol. 51. P. 597–603.
- [219] Hein J., Stovlbaek J. *Combined DNA and protein alignment* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 402–427.
- [220] Henikoff J. G., Henikoff S. *BLOCKS database and its applications* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 88–105.
- [221] Henikoff S., Henikoff J. G. *Automated assembly of protein blocks for database searching* // Nucl. Acids Res. 1991. Vol. 19. P. 6565–6572.
- [222] Henikoff S., Henikoff J. G. *Amino acid substitution matrices from protein blocks* // Proc. Nat. Acad. Sci. USA. 1992. Vol. 89. P. 915–919.
- [223] *Molecular Systematics* / Eds.: D. Hillis, C. Moritz. Sunderland, MA: Sinauer Associates, 1990.
- [224] Hirschberg D. S. *Algorithms for the longest common subsequence problem* // J. ACM. 1977. Vol. 24. P. 664–675.
- [225] Hodgkin J., Plasterk R. H., Waterston R. H. *The nemotode Caenorhabditis elegans and its genome* // Science. 1995. Vol. 270. P. 410–414.
- [226] *Mathematics in the Archaeological and Historical Sciences* / Eds.: F. Hodson, O. Kendall, P. Tautu. Edinburgh, Scotland: Edinburgh University Press, 1971.
- [227] Holliday R. *A different kind of inheritance* // Scientific American. 1989 (June). P. 60–73.
- [228] Hopcroft J., Ullman J. *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley, 1979.
- [229] Horspool N. *Practical fast searching in strings* // Software — Practice and Experience. 1980. Vol. 10. P. 501–506.
- [230] Horton P., Lawler E. *An analysis of the efficiency of the A* algorithm for multiple sequence alignment*. Неопубликованная заметка. 1994.
- [231] Hou L., Martin L., Zhou Z., Feduccia A. *Early adaptive radiation of birds: Evidence from fossils from Northeastern China* // Science. 1996. Vol. 274. P. 1164–1167.

- [232] Huang X., Miller W. *A time-efficient, linear-space local similarity algorithm* // Adv. Appl. Math. 1991. Vol. 12. P. 337–357.
- [233] Hubbard T.J.P., Lesk A. M., Tramontane A. *Gathering them into the fold* // Nature Structural Biology. 1996 (April). Vol. 4. P. 313.
- [234] Hudson T. J., Lander E. S. et al. *An STS-based map of the human genome* // Science. 1995. Vol. 270. P. 1945–1954.
- [235] Hughey R. *Parallel sequence comparison and alignment* // Proc. Int. Conf. Application-Specific Array Processors. IEEE Computer Society Press. 1995 (July).
- [236] Hui L. *Color set size problem with applications to string matching* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 227–240.
- [237] Hume A., Sunday D. M. *Fast string searching* // Software — Practice and Experience. 1991. Vol. 21. P. 1221–1248.
- [238] Hunt J. W., Szymanski T. G. *A fast algorithm for computing longest common subsequences* // Comm. ACM. 1977. Vol. 20. P. 350–353.
- [239] Idury R., Schaffer A. *Multiple matching of parameterized patterns* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 226–239.
- [240] Irving R. W., Fraser C. B. *Two algorithms for the longest common subsequence of three (or more) strings* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 214–229.
- [241] Irving R. W., Fraser C. B. *Maximal common subsequences and minimal common supersequences* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 173–183.
- [242] Istrail S. *The chimeric clones problem*. Technical Report. New Mexico: Sandia Natl. Lab. 1993.
- [243] Ito M., Shimiza K., Nakanishi M., Hashimoto A. *Polynomial-time algorithms for computing characteristic strings* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 274–288.
- [244] Jacobson G., Vo K. P. *Heaviest increasing/common subsequence problems* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 52–65.
- [245] Jenson R., King M. C., Holt J. et al. *BRCA1 is secreted and exhibits properties of a granin* // Nature Genetics. 1996. Vol. 12. P. 303–308.
- [246] Jiang T., Jiang Z. *Rotation of periodic strings and short superstrings*. Preprint. 1995.
- [247] Jiang T., Karp R. M. *Mapping clones with a given ordering or interleaving* // Proc. 8th ACM-SIAM Symp. on Discrete Algs. 1997.
- [248] Jiang T., Li M., Du D. *A note on shortest superstrings with flipping* // Information Processing Lett. 1992. Vol. 44. P. 195–199.
- [249] Jiang T., Wang L., Lawler E. L. *Approximation algorithms for tree alignment with a given phylogeny* // Algorithmica. 1996. Vol. 16. P. 302–315.
- [250] Johnson D. S., Paadimitriou C. H. *Performance guarantees for heuristics* // The Travelling Salesman Problem / Eds.: E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys. New York: Wiley-Interscience, 1985. P. 145–180.

- [251] Joseph D., Meidanis J., Tiwari P. *Determining DNA sequence similarity using maximum independent set algorithms for interval graphs* // Proc. of the Third Scand. Workshop on Algorithm Theory. Lecture Notes in Computer Science. Springer. 1992. Vol. 621. P. 326–337.
- [252] Jukes T. H., Cantor C. R. *Evolution of protein molecules* // Mammalian Protein Metabolism / Ed.: H. N. Munro. New York: Academic Press, 1969. P. 21–132.
- [253] Jurka J. *Human repetitive elements* // Molecular Biology and Biotechnology / Ed.: R. A. Meyers. New York: VCH Publishers, 1995. P. 438–441.
- [254] Jurka J. *Origin and evolution of alu repetitive elements* // The Impact of Short Interspersed Elements (SINEs) on the Host Genome / Ed.: R. J. Maraia. New York: R. G. Landes, 1995. P. 25–41.
- [255] Jurka J., Walichiewicz J., Milosavljevic A. *Prototypic sequences for human repetitive DNA* // J. Mol. Evol. 1992. Vol. 35. P. 286–291.
- [256] Kannan S., Lawler E., Warnow T. *Determining the evolutionary tree using experiments* // J. Algorithms. 1996. Vol. 21. P. 26–50.
- [257] Kannan S., Myers E. *An algorithm for locating non-overlapping regions of maximum alignment score* // SIAM J. Comput. 1996. P. 648–662.
- [258] Kannan S., Warnow T. *Inferring evolutionary history from DNA sequences* // SIAM J. Comput. 1994. Vol. 23. P. 713–737.
- [259] Kannan S., Warnow T. *A fast algorithm for the computation and enumeration of perfect phylogenies when the number of character states is fixed* // Proc. 6th ACM-SIAM Symp. on Discrete Algs. 1995. P. 595–603.
- [260] Kaplan H., Shamir R., Tarjan R. E. *Tractability of parameterized completion problems on chordal and interval graphs: minimum fill-in and physical mapping* // Proc. 35th IEEE Symp. Found. Computer Science. 1994. P. 780–791.
- [261] Kaplan H., Shamir R., Tarjan R. E. *Faster and simpler algorithm for sorting signed permutations by reversals* // Proc. 8th ACM-SIAM Symp. on Discrete Algs. 1997.
- [262] Karlin S., Altschul S. F. *Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes* // Proc. Nat. Acad. Sci. USA. 1990. Vol. 87. P. 2264–2268.
- [263] Karlin S., Altschul S. F. *Applications and statistics for multiple high-scoring segments in molecular sequences* // Proc. Nat. Acad. Sci. USA. 1993. Vol. 90. P. 5873–5877.
- [264] Karp R. *Mapping of the genome: Some combinatorial problems arising in molecular biology* // Proc. of the ACM Symp. on the Theory of Computing. 1993. P. 278–285.
- [265] Karp R., Miller R., Rosenberg A. *Rapid identification of repeated patterns in strings, trees and arrays* // Proc. of the ACM Symp. on the Theory of Computing. 1972. P. 125–136.
- [266] Karp R., Rabin M. *Efficient randomized pattern matching algorithms* // IBM J. Res. Development. 1987. Vol. 31. P. 249–260.
- [267] Kececioglu J. *The maximum weight trace problem in multiple sequence alignment* // Proc. 4th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1993. Vol. 684. P. 106–119.
- [268] Kececioglu J. D., Gusfield D. *Reconstructing a history of recombinations from a set of sequences* // Proc. 5th ACM-SIAM Symp. on Discrete Algs. 1994. P. 471–480.
- [269] Kececioglu J. D., Myers E. *Exact and approximation algorithms for the sequence reconstruction problem* // Algorithmica. 1995. Vol. 13. P. 7–51.

- [270] Kececioglu J. D., Ravi R. *Of mice and men: algorithms for evolutionary distances between genomes with translocation* // Proc. 6th ACM-SIAM Symp. on Discrete Algs. 1995. P. 604–613.
- [271] Kececioglu J. D., Sankoff D. *Efficient bounds for oriented chromosome inversion distance* // Proc. 5th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1994. Vol. 807. P. 307–325.
- [272] Kececioglu J. D., Sankoff D. *Exact and approximation algorithms for sorting by reversal* // Algorithmica. 1995. Vol. 13. P. 180–210.
- [273] Kececioglu J. D. *Exact and approximation algorithms for the sequence reconstruction problem in computational biology*. PhD thesis. Univ. Ariz., Dept. Computer Science. 1990.
- [274] Kimura M. *The neutral theory of molecular evolution* // Scientific American. 1979 (November). P. 98–126.
- [275] Kimura M. *The Neutral Theory of Molecular Evolution*. Cambridge: Cambridge University Press, 1983.
- [276] Knight J. R., Myers E. W. *Approximate regular expression pattern matching with concave gap penalties* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 67–78.
- [277] Knight J. R., Myers E. W. *Super-pattern matching* // Algorithmica. 1995. Vol. 13. P. 211–243.
- [278] Knuth D. E., Morris J. H., Pratt V. B. *Fast pattern matching in strings* // SIAM J. Comput. 1977. Vol. 6. P. 323–350.
- [279] Kolakowski L. F., Leunissen J., Smith J. E. *Prosearch: Fast searching of protein sequences with regular expression patterns related to protein structure and function* // Biotechniques. 1992. Vol. 13. P. 919–921.
- [280] Kolata G. *Biology's big project turns into challenge for computer experts* // New York Times. 1996 (June 11).
- [281] Kolata G. *A vat of DNA may become fast computer of the future* // New York Times. 1995 (April 11).
- [282] Koonin E. V., Altschul S. F., Bork P. *BRCA1 protein products: functional motifs* // Nature Genetics. 1996. Vol. 13. P. 266–268.
- [283] Kosaraju R., Park J., Stein C. *Long tours and short superstrings* // Proc. of the 35th IEEE Symp. on Foundations of Comp. Sci. 1994. P. 166–177.
- [284] Kou L., Markowsky G., Berman L. *A fast algorithm for Steiner trees* // Acta Informatica. 1981. Vol. 15.
- [285] Krogh A., Brown M., Mian I., Sjolander K., Haussler D. *Hidden Markov models in computational biology: Applications to protein modelling* // J. Mol. Biol. 1994. Vol. 235. P. 1501–1531.
- [286] Krontiris T. G. *Minisatellites and human disease* // Science. 1995. Vol. 269. P. 1682–1683.
- [286a] Kurtz S. *Reducing the space requirement of suffix trees* // Software — Practice and Experience. 1999. Vol. 29. P. 1149–1171.
- [287] Landau G., Vishkin U. *Introducing efficient parallelism into approximate string matching and a new serial algorithm* // Proc. of the ACM Symp. on the Theory of Computing. 1986. P. 220–230.
- [288] Landau G. M., Schmidt J. P. *An algorithm for approximate tandem repeats* // Proc. 4th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1993. Vol. 684. P. 120–133.

- [289] Landau G. M., Vishkin U. *Efficient string matching with k mismatches* // Theor. Comp. Sci. 1986. Vol. 43. P. 239–249.
- [290] Landau G. M., Vishkin U., Nussinov R. *Locating alignments with k differences for nucleotide and amino acid sequences* // Comp. Appl. Biosciences. 1988. Vol. 4. P. 12–24.
- [291] Lander E. *The new genomics: Global views of biology* // Science. 1996. Vol. 274. P. 536–539.
- [292] Lander E., Waterman M. *Genomic mapping by fingerprinting random clones: a mathematical analysis* // Genomics. 1988. Vol. 2. P. 231–239.
- [293] Laub M. T., Smith D. W. *Finding intron/exon splice junctions using INFO, Interruption Finder and Organizer*. Preprint. U.C. San Diego, Dept. of Biology. 1997 (February).
- [294] Lawler E. L. *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart, and Winston, 1976.
- [295] Lawrence C., Altschul S., Boguski M., Liu J., Neuwald A., Wootton J. *Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment* // Science. 1993. Vol. 262. P. 208–214.
- [296] Lesk A. M., Levitt M., Chothia C. *Alignment of the amino acid sequences of distantly related proteins using variable gap penalties* // Protein Eng. 1986. Vol. 1. P. 77–78.
- [297] Lesk A. M. *Computational molecular biology* // Encyclopedia of Computer Science and Technology / Eds.: A. Kent, J.G. Williams. New York: Marcel Dekker, 1994. Vol. 31. P. 101–165.
- [298] Leung M. Y., Blaisdell B., Burge C., Karlin S. *An efficient algorithm for identifying matches with errors in multiple long molecular sequences* // J. Mol. Biol. 1991. Vol. 221. P. 1367–1378.
- [299] Левенштейн В. И. *Двоичные коды с исправлением выпадений, вставок и замещений символов* // Доклады АН СССР. 1965. Т. 163. С. 707–710.
- [300] Levy-Lahad E. et al. *Candidate gene for the chromosome 1 familial Alzheimer's disease locus* // Science. 1995. Vol. 269. P. 973–977.
- [301] Li W. H., Graur D. *Fundamentals of Molecular Evolution*. Sunderland, MA: Sinauer, 1991.
- [302] Linial M., Linial N. *On the potential of molecular computing* // Science. 1995 (April 28). Vol. 268. P. 481.
- [303] Lipman D., Altshul S., Keccecioglu J. *A tool for multiple sequence alignment* // Proc. Nat. Acad. Sci. USA. 1989. Vol. 86. P. 4412–4415.
- [304] Lipman D. J., Pearson W. R. *Rapid and sensitive protein similarity searches* // Science. 1985. Vol. 111. P. 1435–1441.
- [305] Lipton R. J. *DNA solution of hard computational problems* // Science. 1995. Vol. 268. P. 542–544.
- [306] Livingstone C. D., Barton G. J. *Identification of functional residues and secondary structure from protein multiple alignment* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 479–512.
- [307] Main M., Lorentz R. *An O(n log n) algorithm for finding all repeats in a string* // J. Algorithms. 1984. Vol. 5. P. 422–432.
- [308] Manber U., Myers G. *Suffix arrays: a new method for on-line search* // SIAM J. Comput. 1993. Vol. 22. P. 935–948.
- [309] Marshall E. *Emphasis turns from mapping to large-scale sequencing* // Science. 1995 (June). Vol. 268. P. 1270–1271.

- [310] Martinez H. *An efficient method for finding repeats in molecular sequences* // Nucl. Acids Res. 1983. Vol. 11. P. 4629–4634.
- [311] Marx J. *Developmental biology: knocking genes in instead of out* // Science. 1995. Vol. 269. P. 636.
- [312] Masek W. J., Paterson M. S. *How to compute string-edit distances quickly* // Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison / Eds.: D. Sankoff and J. Kruskal. Reading, MA: Addison Wesley, 1983. P. 337–349.
- [313] Masek W. J., Paterson M. S. *A faster algorithm for computing string edit distances* // J. Comp. Sys. Sci. 1980. Vol. 20. P. 18–31.
- [314] Max M., McKinnon P., Seidenman K., Barret R., Applebury M., Takahashi J., Margolskee R. *Pineal opsin: a nonvisual opsin expressed in chick pineal* // Science. 1995. Vol. 267. P. 1502–1506.
- [315] Maynard Smith J. *Evolutionary Genetics*. Oxford: Oxford Univ. Press, 1989.
- [316] McClure M., Vasi T., Fitch W. *Comparitive analysis of multiple protein-sequence alignment methods* // Mol. Biol. Evolution. 1994. Vol. 11. P. 571–592.
- [317] McConkey E. *Human Genetics: The Molecular Revolution*. Boston, MA: Jones and Bartlett, 1993.
- [318] McCreight E. M. *A space-economical suffix tree construction algorithm* // J. ACM. 1976. Vol. 23. P. 262–272.
- [319] McGinnis W., Kuziora M. *The molecular architects of body design* // Scientific American. 1994 (February). P. 58–66.
- [320] Mewes H. W., Heumann K. *Genome analysis: pattern search in biological macromolecules* // Proc. 6th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1995. Vol. 937 P. 261–285.
- [321] Miller W. *An algorithm for locating a repeated region*. Неопубликованная рукопись.
- [322] Miller W., Myers E. W. *Sequence comparison with concave weighting functions* // Bull. Math. Biol. 1988. Vol. 50. P. 97–120.
- [323] Miller W., Ostell J., Rudd K. *An algorithm for searching restriction maps* // Comp. Appl. Biosciences. 1990. Vol. 6. P. 247–252.
- [324] Milosavljevic A. *Discovering dependencies via algorithmic mutual information: a case study in DNA sequence comparisons* // Machine Learning. 1995. Vol. 21. P. 35–50.
- [325] Milosavljevic A., Jurka J. *Discovering simple DNA sequences by the algorithmic significance method* // Comp. Appl. Biosciences. 1993. Vol. 9. P. 407–411.
- [326] Milosavljevic A., Jurka J. *Discovery by minimal length encoding: a case study in molecular evolution* // Machine Learning. 1993. Vol. 12. P. 69–87.
- [327] Миркин Б. Г., Родин С. Н. *Графы и гены*. М.: Наука, 1977.
- [328] Mirzanian A. *A halving technique for the longest stuttering subsequence problem* // Information Processing Lett. 1987. Vol. 26. P. 71–75.
- [329] Monaco A., Larin Z. *YACs, BACs, PACs and MACs: artificial chromosomes as research tools* // Trends in Genetics. 1994. Vol. 12. P. 280–286.
- [330] Monod J. *Chance and Necessity; An Essay on the Natural Philosophy of Modern Biology*. New York: Knopf, 1971.
- [331] Motwani R., Raghavan P. *Randomized Algorithms*. Cambridge: Cambridge Univ. Press, 1995.

- [332] Moyzis R. K. *The human telomere* // Scientific American. 1991 (April). P. 48–55.
- [333] Mullis K. *The unusual origin of the polymerase chain reaction* // Scientific American. 1990 (April). P. 56–65.
- [334] Murata M., Richardson J., Sussman J. *Simultaneous comparison of three protein sequences* // Proc. Nat. Acad. Sci. USA. 1985. Vol. 82. P. 3073–3077.
- [335] Murray A., Szostak J. *Artificial chromosomes* // Scientific American. 1987 (November). P. 62–68.
- [336] Muthukrishnan S. *Detecting false matches in string matching algorithms* // Proc. 4th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1993. Vol. 684. P. 164–178.
- [337] Myers E. *Algorithmic advances for searching biosequence databases* // Computational Methods in Genome Research / Ed.: S. Suhai. New York: Plenum Press, 1994. P. 121–135.
- [338] Myers E., Huang X. *An $O(n^2 \log n)$ restriction map comparison and search algorithm* // Bull. Math. Biol. 1992. Vol. 54. P. 599–618.
- [339] Myers E. W. *A Four-Russians algorithm for regular expression pattern*. Technical Report. Univ. Ariz. 1988.
- [340] Myers E. W. *A Four-Russians algorithm for regular expression pattern matching* // J. ACM. 1992. Vol. 39. P. 430–448.
- [341] Myers E. W. *An $O(nd)$ difference algorithm and its variations* // Algorithmica. 1986. Vol. 1. P. 251–266.
- [342] Myers E. W. *A sublinear algorithm for approximate keyword searching* // Algorithmica. 1994. Vol. 12. P. 345–374.
- [343] Myers E. W. *Towards simplifying and accurately formulating fragment assembly* // J. Comp. Biol. 1995. Vol. 2. P. 275–290.
- [344] Myers E. W., Miller W. *Optimal alignments in linear space* // Comp. Appl. Biosciences. 1988. Vol. 4. P. 11–17.
- [345] Myers E. W., Miller W. *Row replacement algorithms for screen editors* // ACM TOPLAS. 1989. Vol. 11. P. 33–56.
- [346] Naor D., Brutlag D. *On near-optimal alignments in biological sequences* // J. Comp. Biol. 1994. Vol. 1. P. 349–366.
- [347] Needleman S. B., Wunsch C. D. *A general method applicable to the search for similarities in the amino acid sequence of two proteins* // J. Mol. Biol. 1970. Vol. 48. P. 443–453.
- [348] Nicol S. et al. *Genetic identification of a hantavirus associated with an outbreak of acute respiratory illness* // Science. 1993. Vol. 262. P. 914–917.
- [349] Nowak R. *Entering the postgenome era* // Science. 1995. Vol. 270. P. 368–371.
- [350] Nowak R. *Venter wins sequencing race — twice* // Science. 1995. Vol. 268. P. 1273.
- [351] O'Brien S. *The ancestry of the Giant Panda* // Scientific American. 1987 (November). P. 102–116.
- [352] Olson M. V. *A time to sequence* // Science. 1995. Vol. 270. P. 394–396.
- [353] Ott J. *Analysis of Human Genetic Linkage*. Baltimore, MD: Johns Hopkins Univ. Press, 1991.
- [354] Overduin M., Harvey T., Bagby S., Tong K., Yau P., Takeichi M., Ikura M. *Solution structure of the epithelial cadherin domain responsible for selective cell adhesion* // Science. 1995. Vol. 267. P. 386–389.

- [355] Palmer J., Herbon L. *Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence* // J. Mol. Evol. 1988. Vol. 28. P. 87–89.
- [356] Pearson W. R. *Rapid and sensitive sequence comparisons with FASTP and FASTA* // Methods in Enzymology. Vol. 183. Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences / Ed.: R. F. Doolittle. New York: Academic Press, 1990. P. 63–69.
- [357] Pearson W. R. *Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms* // Genomics. 1991. Vol. 11. P. 635–650.
- [358] Pearson W. R. *Comparison of methods for searching protein sequence databases* // Protein Science. 1995. Vol. 4. P. 1145–1160.
- [359] Pearson W. R. *Effective protein sequence comparison* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 227–258.
- [360] Pearson W. R. *Protein sequence comparison and protein evolution*. Tutorial T6 of Intelligent Systems in Mol. Biol. Cambridge, England. 1995 (July).
- [361] Pearson W. R., Lipman D. J. *Improved tools for biological sequence comparison* // Proc. Nat. Acad. Sci. USA. 1988. Vol. 85. P. 2444–2448.
- [362] Peltola M., Soderlund H., Tarhio J., Ukkonen E. *Algorithms for some string matching problems arising in molecular genetics* // Proc. of the 9th IFIP World Computer Congress. 1983. P. 59–64.
- [363] Peltola M., Soderlund H., Ukkonen E. *Sequaid: a DNA sequence assembly program based on a mathematical model* // Nucl. Acids Res. 1984. Vol. 12. P. 307–321.
- [364] Peltola M., Soderlund H., Ukkonen E. *Algorithms for the search of amino acid patterns in nucleic acid sequences* // Nucl. Acids Res. 1986. Vol. 14. P. 99–107.
- [365] Pennisi E. *Research news: premature aging gene discovered* // Science. 1996. Vol. 272. P. 193–194.
- [366] Pevzner P. *I-Tuple DNA sequencing: computer analysis* // J. Biomol. Structure Dynamics. 1989. Vol. 7. P. 63–73.
- [367] Pevzner P. (ed.). *Combinatorial methods for DNA mapping and sequencing* // J. Comp. Biol. 1995. Vol. 2. N 2. Специальный выпуск.
- [368] Pevzner P., Lipshutz R. *Towards DNA sequencing chips* // Proc. 19th Symp. on Math. Found of Comp. Sci. Lecture Notes in Computer Science. Springer. 1994. Vol. 841. P. 143–158.
- [369] Pevzner P., Vingron M. *Multiple sequence comparison and n-dimensional image reconstruction* // Proc. 4th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1993. Vol. 684. P. 243–253.
- [370] Pevzner P., Waterman M. *Matrix longest common subsequence problem, duality and Hilbert bases* // Proc. 3rd Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1992. Vol. 644. P. 79–89.
- [371] Pevzner P., Waterman M. S. *Open combinatorial problems in computational molecular biology* // Proc. of the 3rd Israel Symposium on Computers and Systems. IEEE Computer Society Press. 1995.
- [372] Pevzner P. A. *DNA physical mapping and alternating Eulerian cycles in colored graphs* // Algorithmica. 1994. Vol. 12. P. 77–105.

- [373] Pevzner P. A., Waterman M. *Multiple nitration and approximate pattern matching* // Algorithmica. 1995. Vol. 13. P. 135–154.
- [374] Posfai J., Bhagwat A., Posfai G., Roberts R. *Predictive motifs derived from cytosine methyltransferases* // Nucl. Acids Res. 1989. Vol. 17. P. 2421–2435.
- [375] Pratt V. Частное сообщение.
- [376] Pratt V. *Applications of the Weiner repetition finder*. Неопубликованная рукопись. 1973.
- [377] Prusiner S. *The prion diseases* // Scientific American. 1995 (January). P. 48–57.
- [378] Rhodes R., Klug A. *Zinc fingers* // Scientific American. 1993 (February). P. 56–65.
- [379] Rich E., Knight K. *Artificial Intelligence*. New York: McGraw Hill, 1991.
- [380] Ridley M. *Evolution and Classification: The Reformation of Cladism*. London: Longman, 1986.
- [381] De Roberts E., Oliver G., Wright C. *Homeobox genes and the vertebrate body plan* // Scientific American. 1990 (July). P. 46–52.
- [382] Rodeh M., Pratt V. R., Even S. *A linear algorithm for data compression via string matching* // J. ACM. 1981. Vol. 28. P. 16–24.
- [383] Rosser J. B., Schoenfeld L. *Approximate formulas for some functions of prime numbers* // Illinois J. Math. 1962. Vol. 6. P. 64–94.
- [384] Rytter W. *A correct preprocessing algorithm for Boyer–Moore string searching* // SIAM J. Comput. 1980. Vol. 9. P. 509–512.
- [385] Sagot M. F., Viari A., Soldano H. *Multiple sequence comparison: a peptide matching approach* // Proc. 6th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1995. Vol. 937. P. 366–385.
- [386] Salamon P., Konopka A. *A maximum entropy principle for distribution of local complexity in naturally occurring nucleotide sequences* // Computers and Chemistry. 1992. Vol. 16. P. 117–124.
- [387] Sankoff D. *Minimal mutation trees of sequences* // SIAM J. Appl. Math. 1975. Vol. 28. P. 35–42.
- [388] Sankoff D., Cedergren R. *Simultaneous comparisons of three or more sequences related by a tree*. Tutorial T6 of Intelligent Systems in Mol. Biol. Cambridge, England. 1995 (July). P. 253–264.
- [389] *Time Warps, String Edits, and Macro-molecules: The Theory and Practice of Sequence Comparison* / Eds.: D. Sankoff, J. Kruskal. Reading, MA: Addison-Wesley, 1983.
- [390] Sankoff D., Ferretti V., Nadeau J. *Conserved segment identification* // Proc. of RECOMB 97: The first international conference on computational molecular biology. ACM Press. 1997. P. 252–256.
- [391] Sapienza C. *Parental imprinting of genes* // Scientific American. 1990 (October). P. 52–60.
- [392] Schatz B. *Information retrieval in digital libraries: Bringing search to the net* // Science. 1997. Vol. 275. P. 327–334.
- [393] Schieber B., Vishkin U. *On finding lowest common ancestors: simplifications and parallelization* // SIAM J. Comput. 1988. Vol. 17. P. 1253–1262.
- [394] Schimke R. T. *Gene amplification and drug resistance* // Scientific American. 1980. Vol. 243. P. 60–69.
- [395] Schmid C., Marks J. *DNA hybridization as a guide to phylogeny: chemical and physical limits* // Mol. Evol. 1990. Vol. 30. P. 237–246.

- [396] Schmidt J. *All highest scoring paths in weighted grid graphs and its application to finding all approximate repeats in strings* // SIAM J. Comput. 1998. Vol. 27. P. 972–992.
- [397] Schuler G., Epstein J., Ohkawa H., Kans J. *Entree: molecular biology database and retrieval system* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 141–162.
- [398] Schuler G. D., Altschul S. F., Lipman D. J. *A workbench for multiple alignment construction and analysis* // Proteins: Structure, Function and Genetics. 1991. Vol. 9. P. 180–190.
- [399] Schuler G. D., Lander E. S., Hudson T. J. et al. *A gene map for the human genome* // Science. 1996. Vol. 274. P. 540–546.
- [400] Schwarz R., Dayhoff M. *Matrices for detecting distant relationships* // Atlas of Protein Sequences / Ed.: M. Dayhoff. Nat. Biomed. Res. Found. 1979. P. 353–358.
- [401] Sedgewick R. *Algorithms*. 2nd ed. Reading, MA: Addison-Wesley, 1988.
- [402] Setubal J., Meidanis J. *Introduction to Computational Molecular Biology*. Boston: PWS, 1997.
- [403] Shih C., Weinberg R. *Isolation of a transforming sequence from a human bladder carcinoma cell line* // Cell. 1982. P. 161–169.
- [404] Shiloach Y. *Fast canonization of circular strings* // J. Algorithms. 1981. Vol. 2. P. 107–121.
- [405] Shyue S. K., Hewett-Emmett D., Sperling H., Hunt D., Bowmaker J., Mollon J., Li W. H. *Adaptive evolution of color vision of genes in higher primates* // Science. 1995. Vol. 269. P. 1265–1267.
- [406] Sibley C. G., Ahlquist J. E. *Reconstructing bird phylogeny by comparing DNA's* // Scientific American. 1986 (February). P. 82–92.
- [407] Sibley C. G., Ahlquist J. E. *Phylogeny and Classification of Birds*. New Haven, CT: Yale Univ. Press, 1990.
- [408] Slonim D., Kruglyak L., Stein L., Lander E. *Building human genome maps with radiation hybrids* // Proc. of RECOMB 97: The first international conference on computational molecular biology. ACM Press. 1997. P. 277–286.
- [409] Smith P. D. *Experiments with a very fast substring search algorithm* // Software — Practice and Experience. 1991. Vol. 21. P. 1065–1074.
- [410] Smith P. D. *On tuning the Boyer—Moore—Horspool string searching algorithm* // Software — Practice and Experience. 1994. Vol. 24. P. 435–436.
- [411] Smith T. F., Waterman M. S. *Identification of common molecular subsequences* // J. Mol. Biol. 1981. Vol. 147. P. 195–197.
- [412] Snyder E. E., Stormo G. D. *Identification of protein coding regions in genomic DNA* // J. Mol. Biol. 1995. Vol. 284. P. 1–18.
- [413] Spouge J. L. *Improving sequence-matching algorithms by working from both ends* // J. Mol. Biol. 1985. Vol. 181. P. 137–138.
- [414] Spouge J. L. *Speeding up dynamic programming algorithms for finding optimal lattice paths* // SIAM J. Appl. Math. 1989. Vol. 49. P. 1552–1566.
- [415] Spouge J. L. *Fast optimal alignment* // Comp. Appl. Biosciences. 1991. Vol. 7. P. 1–7.
- [416] Staden R. *Screening protein and nucleic acid sequences against libraries of patterns* // J. DNA Sequencing Mapping. 1991. Vol. 1. P. 369–374.
- [417] Steeg P. *Granin expectations in breast cancer?* // Nature Genetics. 1996. Vol. 12. P. 223–225.

- [418] Steel M. *The complexity of reconstructing trees from qualitative characters and subtrees* // J. Classification. 1992. Vol. 9. P. 91–116.
- [419] Stein S. *The mathematician as explorer* // Scientific American. 1961 (May). P. 149–163.
- [420] Steitz J. *Snurps* // Scientific American. 1988 (June). P. 56–63.
- [421] Stephen G. A. *String Searching Algorithms*. Singapore: World Scientific, 1994.
- [422] Sternberg M. J. *PROMOT: A fortran program to scan protein sequences against a library of known motifs* // Comp. Appl. Biosciences. 1991. Vol. 7. P. 257–260.
- [423] Storer J. A. *Data Compression: Methods and Theory*. Rockville, MD: Computer Science Press, 1988.
- [424] Strauss B. S. *Book review: DNA repair and mutagenesis* // Science. 1995. Vol. 270. P. 1511–1513.
- [425] Sunday D. M. *A very fast substring search algorithm* // Comm. ACM. 1990. Vol. 33. P. 132–142.
- [426] Suzuki D., Griffiths A., Miller J., Lewontin R. *An Introduction to Genetic Analysis*. 3rd ed. New York: Freeman, 1986.
- [427] Sweedyk E. S. *A 1/2 approximation algorithm for shortest common superstring*. PhD thesis. University of California, Berkeley. Department of Computer Science. 1995.
- [428] Swindells M. B. *Commentary: Finding your fold* // Protein Eng. 1994. Vol. 7. P. 1–3.
- [429] Swofford D. L., Olsen G. L. *Phylogeny reconstruction* // Molecular Systematics / Eds.: D. M. Hillis, C. Moritz. Sunderland, MA: Sinauer, 1990. P. 411–501.
- [430] Syvanen M. *Horizontal gene transfer: evidence and possible consequences* // Annual Rev. Genetics. 1994. Vol. 28. P. 237–261.
- [431] Tarhio J., Ukkonen E. *A greedy approximation algorithm for constructing shortest common superstrings* // Theor. Comp. Sci. 1988. Vol. 57. P. 131–145.
- [432] Taylor W. R. *Hierarchical method to align large numbers of biological sequences* // Methods in Enzymology. Vol. 183. Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences / Ed.: R. F. Doolittle. New York: Academic Press, 1990. P. 456–474.
- [433] Taylor W. R. *Multiple protein sequence alignment: algorithms and gap insertion* // Methods in Enzymology. Vol. 266. Computer Methods for Macromolecular Sequence Analysis / Ed.: R. F. Doolittle. New York: Academic Press, 1996. P. 343–367.
- [434] Teng S. H., Yao F. *Approximating shortest superstrings* // Proc. of the 34th IEEE Symp. on Foundations of Comp. Sci. 1993. P. 158–165.
- [435] Trifonov E., Brendel V. *Gnomic: A Dictionary of Genetic Codes*. Deerfield, FL: VCH Press, 1986.
- [436] Turner J. *Approximation algorithms for the shortest common superstring problem* // Information Comput. 1989. Vol. 83. P. 1–20.
- [437] Ukkonen E. *Approximate string-matching over suffix trees* // Proc. 4th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1993. Vol. 684. P. 228–242.
- [438] Ukkonen E. *On-line construction of suffix-trees* // Algorithmica. 1995. Vol. 14. P. 249–260.
- [439] Ukkonen E. *Algorithms for approximate string matching* // Information Control. 1985. Vol. 64. P. 100–118.
- [440] Ukkonen E. *Approximate string matching with q-grams and maximal matches* // Theor. Comp. Sci. 1992. Vol. 92. P. 191–211.

- [441] Urdang L. *Random House Dictionary (College Edition)*. New York: Random House, 1968.
- [442] Venter J. C., Smith H. O., Hood L. *A new strategy for genome sequencing* // Nature. 1996. Vol. 381. P. 364–366.
- [442a] Venter J. C., Adams M. D., Sutton G. G., Kerlavage A. R., Smith H. O., Hunkapiller M. *Shotgun sequencing of the human genome* // Science. 1998. Vol. 280. P. 1540–1542.
- [443] Vingron M. *Multiple sequence alignment and applications in molecular biology*. Preprint 91-12. Univ. Heidelberg, 1991.
- [444] Vingron M., Argos P. *A fast and sensitive multiple sequence alignment algorithm* // Comp. Appl. Biosciences. 1989. Vol. 5. P. 115–121.
- [445] Vingron M., Argos P. *Determination of reliable regions in protein sequence alignments* // Protein Eng. 1990. Vol. 7. P. 565–569.
- [446] Vingron M., Argos P. *Motif recognition and alignment for many sequences by comparison of dot-matrices* // J. Mol. Biol. 1991. Vol. 218. P. 33–43.
- [447] Vingron M., Waterman M. *Sequence alignment and penalty choice* // J. Mol. Biol. 1994. Vol. 235. P. 1–12.
- [448] Vogt G., Etzold T., Argos P. *An assessment of amino acid exchange matrices in aligning protein sequences* // J. Mol. Biol. 1995. Vol. 294. P. 816–831.
- [449] von Heijne G. *Sequence Analysis in Molecular Biology: Treasure Trove or Trivial Pursuit*. New York: Academic Press, 1987.
- [450] Voyles B. A. *The Biology of Viruses*. St. Louis, MO: Mosby, 1993.
- [451] Wagner G. *E-cadherin: a distant member of the immunoglobulin superfamily* // Science. 1995. Vol. 267. P. 342.
- [452] Waldrop M. M. *On-line archives let biologists interrogate the genome* // Science. 1995. Vol. 269. P. 1356–1358.
- [453] Wang L., Gusfield D. *Improved approximation algorithms for tree alignment* // Proc. 7th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science. Springer. 1996. Vol. 1075. P. 220–233.
- [454] Wang L., Jiang T. *On the complexity of multiple sequence alignment* // J. Comp. Biol. 1994. Vol. 1. P. 337–348.
- [455] Waterman M. *Multiple sequence alignment by consensus* // Nucl. Acids Res. 1986. Vol. 14. P. 9095–9102.
- [456] Waterman M., Arratia R., Galas D. *Pattern recognition in several sequences: consensus and alignment* // Bull. Math. Biol. 1984. Vol. 46. P. 515–527.
- [457] Waterman M., Perlitz M. *Line geometries for sequence comparisons* // Bull. Math. Biol. 1984. Vol. 46. P. 567–577.
- [458] Waterman M., Smith T., Singh M., Beyer W. *Additive evolutionary trees* // J. Theor. Biol. 1977. Vol. 64. P. 199–213.
- [459] Waterman M., Vingron M. *Rapid and accurate estimates of statistical significance for sequence data base searches* // Proc. Nat. Acad. Sci. USA. 1994. Vol. 91. P. 4625–4628.
- [460] Waterman M. S. *Sequence alignments in the neighborhood of the optimum with general application to dynamic programming* // Proc. Nat. Acad. Sci. USA. 1983. Vol. 80. P. 3123–3124.
- [461] Waterman M. S. *Introduction to Computational Biology: Maps, Sequences, Genomes*. London, England: Chapman and Hall, 1995.

- [462] Waterman M. S., Eggert M., Lander E. *A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons* // J. Mol. Biol. 1987. Vol. 197. P. 723–728.
- [463] Waterman M. S., Eggert M., Lander E. *Parametric sequence comparisons* // Proc. Natl. Acad. Sci. USA. 1992. Vol. 89. P. 6090–6093.
- [464] Waterman M. S., Griggs J. R. *Interval graphs and maps of DNA* // Bull. Math. Biol. 1986. Vol. 48. P. 189–195.
- [465] Waterman M. S., Smith T. F., Katcher H. L. *Algorithms for restriction maps* // Nucl. Acids Res. 1984. Vol. 12. P. 237–242.
- [466] Waterman M. S. *Efficient sequence alignment algorithms* // J. Theor. Biol. 1984. Vol. 108. P. 333–337.
- [467] Waterman M. S., Jones R. *Consensus methods for DNA and protein sequence alignment* // Methods in Enzymology. Vol. 183. Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences / Ed.: R. F. Doolittle. New York: Academic Press, 1990. P. 221–237.
- [467a] Математические методы для анализа последовательностей ДНК / Ред.: М. С. Уотермен. М.: Мир, 1999. С. 349.
- [468] Watson J., Oilman M., Witkowski J., Zoller M. *Recombinant DNA*. 2nd ed. San Francisco, CA: Scientific American Books, 1992.
- [469] Watson J., Hopkins N., Roberts J., Steitz J., Weiner A. *Molecular Biology of the Gene*. 4th ed. Menlo Park, CA: Benjamin Cummings, 1987.
- [470] Wayne R., Gittleman J. *The problematic Red Wolf* // Scientific American. 1995 (July). P. 36–39.
- [470a] Weber J. L., Myers E. W. *Human whole-genome shotgun sequencing* // Genome Res. 1997. Vol. 7. P. 401–409.
- [471] Weinberg R. *A molecular basis of cancer* // Scientific American. 1983 (May). P. 126–142.
- [472] Weinberg R. *Finding the anti-oncogene* // Scientific American. 1988 (September). P. 44–51.
- [473] Weiner P. *Linear pattern matching algorithms* // Proc. of the 14th IEEE Symp. on Switching and Automata Theory. 1973. P. 1–11.
- [474] Weinstein J., Paul K. et al. *An information-intensive approach to the molecular pharmacology of cancer* // Science. 1997. Vol. 275. P. 343–349.
- [475] West S. C. *The processing of recombination intermediates: mechanistic insights from studies of bacterial proteins* // Cell. 1994. Vol. 76. P. 9–15.
- [476] White R., Lalouel J. M. *Chromosome mapping with DNA markers* // Scientific American. 1988. Vol. 238. P. 40–48.
- [477] Wilbur W. J. *On the PAM matrix model of protein evolution* // Mol. Biol. Evol. 1985. Vol. 2. P. 434–437.
- [478] Williams N. *Closing in on the complete yeast genome sequence* // Science. 1995. Vol. 268. P. 1560–1561.
- [479] Williams N. *Europe opens institute to deal with gene data deluge* // Science. 1995. Vol. 269. P. 630.
- [480] Williams N. *How to get databases talking the same language* // Science. 1997. Vol. 275. P. 301–302.
- [481] Wilson A., Cann R. *The recent African genesis of humans* // Scientific American. 1992 (April). P. 68–73.

- [482] Wu S., Manber U. *Fast text searching allowing errors* // Comm. ACM. 1992. Vol. 35. P. 83–91.
- [483] Wu S., Manber U., Myers G., Miller W. *An O(np) sequence comparison algorithm* // Information Processing Lett. 1990. Vol. 35. P. 317–323.
- [484] Zeeikovsky A. *An 11/6 approximation algorithm for the Steiner tree problem in graphs* // Information Processing Lett. 1993. Vol. 46. P. 79–83.
- [485] Zhang Z., Pearson W. R., Miller W. *Aligning a DNA sequence with a protein sequence* // Proc. of RECOMB 97: The first international conference on computational molecular biology. ACM Press. 1997. P. 337–343.
- [486] Zimmer R., Lengauer T. *Fast and numerically stable parametric alignment of biosequences* // Proc. of RECOMB 97: The first international conference on computational molecular biology. ACM Press. 1997. P. 344–353.
- [487] Ziv J., Lempel A. *A universal algorithm for sequential data compression* // IEEE Trans. on Info. Theory. 1977. Vol. 23. P. 337–343.
- [488] Ziv J., Lempel A. *Compression of individual sequences via variable length coding* // IEEE Trans. on Info. Theory. 1978. Vol. 24. P. 530–536.
- [489] Zuckerkandl E., Pauling L. *Molecular disease, evolution and genic heterogeneity* // Horizons in Biochemistry / Eds.: M. Kash and B. Pullman. New York: Academic Press, 1962. P. 189–225.
- [490] Zuker M. *Suboptimal sequence alignments in molecular biology, alignment with error analysis* // J. Mol. Biol. 1991. Vol. 221. P. 403–420.

Толковый словарь

Этот словарь определяет термины, которые встречаются в нескольких частях книги и могут быть неизвестны читателю. Уровень их определений приемлем для этой книги, но может оказаться недостаточным по сравнению со словарем для биологического текста. Широкоизвестные термины, так же как и узкоспециальные, используемые лишь единожды, опущены. В случае если какой-то термин обсуждается или определяется в тексте книги более подробно, в словаре указывается номер страницы, на которой такое обсуждение начинается.

алгоритм (algorithm)

Сделанное на высоком уровне описание механистического пути решения задачи или вычисления функции. Это описание должно излагать логику метода, но избегать многих программистских деталей низкого уровня, необходимых при его компьютерной реализации. В биологической литературе часто термины “алгоритм” и “программа” используются как взаимозаменяемые, но это неправильно. Программа содержит все детали реализации, необходимые для того, чтобы метод работал на конкретном языке программирования на конкретных компьютерах. Такой уровень описания обычно затмняет логику и идеи самого алгоритма.

allelель (allele)

Одна из двух или более форм, которые может принимать подстрока ДНК (часто ген).

алфавит аминокислот (amino acid alphabet)

Алфавит из двадцати букв: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y, каждая из которых представляет одну из 20 аминокислот, кодируемых в ДНК.

альфа-спираль (alpha helix)

Сpirальная структура белка; одна из двух основных разновидностей вторичной структуры белка (другая — это бета-структура, или конформация). С. 307 и 438.

аминокислота (amino acid)

Молекула, являющаяся строительным блоком белка. В белках найдено двадцать основных аминокислот. Каждая аминокислота кодируется в ДНК кодоном (см. генетический код и кодон). С. 36.

аналогичный белок (analogous protein)

См. гомологичный белок.

антigen (antigen)

Молекула, которая распознается как посторонняя и вызывает ответную реакцию антител.

антитело (antibody)

Белок, который вступает в реакцию с чужеродной молекулой. Относится к иммуноглобулинам. Антитела в природе идентифицируют и разрушают чужеродные тела. В биохимии они используются как средство для идентификации конкретных белков.

аутосома (autosome)

Любая хромосома, отличная от половых хромосом. У человека это любая хромосома, отличная от X- и Y-хромосом.

белок (protein)

Полипептид, цепь из связанных аминокислот; структурный материал и “рабочая лошадка” клетки (ДНК предполагает, а белок располагает). Многие белки являются ферментами, которые катализируют химические реакции в клетке. До конца 1940-х и начала 1950-х годов повсеместно считалось, что белки окажутся носителями информации о наследственности. Всех очень удивило, что эту роль играет ДНК.

бета-лист (beta sheet)

Одна из двух основных разновидностей вторичной структуры белков (другая — это альфа-спираль).

библиотека ДНК (DNA library)

Физическая коллекция неупорядоченных клонированных фрагментов ДНК, возможно кДНК, полученных из иРНК. Эти фрагменты могут быть получены из целого генома, но часто фрагменты в отдельной библиотеке ДНК берутся из определенной ткани или области хромосомы и представляют лишь подмножество всей ДНК организма.

библиотека клонов (clone library)

См. библиотека ДНК.

биоинформатика (bioinformatics)

Искусственное слово, относящееся к широкому спектру вычислительной деятельности в геномных проектах человека и др. Это понятие объединяет развитие баз данных, коммуникационное программное обеспечение, расшифровку, проектирование лабораторных ноутбуков, создание стандартов и др. Большое преимущество этого термина в том, что он не имел никакого смысла раньше, и теперь ему можно придать любой смысл. Негативная сторона в том, что специалисты в этой области будут называться “биоинформатистами”. Хотя этот термин и многогранен, многие люди, идентифицируемые как биоинформатисты, предпочитают говорить, что они работают в “вычислительной молекулярной биологии”.

гаплоид (haploid)

Клетка, содержащая только одну копию каждой хромосомы.

гемоглобин (hemoglobin)

Основной функциональный белок красных кровяных клеток; участвует в связывании и переносе кислорода.

ген (gene)

Сплошной участок ДНК, содержащий информацию, необходимую для кодирования белка или, реже, некоторой РНК. Гены являются основными единицами наследственности.

генетический код (genetic code)

Соответствие между конкретными триплетами нуклеотидов (кодонами) и аминокислотами. Так как число кодонов равно $4^3 = 64$, а основных аминокислот только 20, то некоторые аминокислоты кодируются больше чем одним кодоном. Поэтому говорят, что генетический код вырожден. Например, аминокислота аланин задается кодонами GCA, GCC, GCG, GCU. Имеются также три кодона, которые не кодируют никакой аминокислоты, а обычно определяют конец полипептида. Они называются терминирующими кодонами. С. 36.

генорегуляторный белок (gene regulatory protein)

Белок, который связывается с ДНК, приводя к экспрессии гена.

геном (genome)

Полная генетическая информация, содержащаяся в организме.

генотип (genotype)

Описание конкретного организма в терминах его генома. Понятие противоположно фенотипу, описывающему организм в терминах его характерных черт.

гибридизация (hybridization)

Попарное связывание азотистых оснований двух комплементарных молекул ДНК или РНК. Гибридизация одннитевого фрагмента с более длинной последовательностью ДНК часто используется для определения того, не входит ли в более длинную последовательность участок, комплементарный короткому фрагменту.

глобулярный белок (globular protein)

Белок сферической формы.

гомеобокс (homeobox)

Последовательность ДНК, к которой присоединяются регуляторные белки, что определяет основные направления развития организма. Эти последовательности высоко консервативны у многих видов. С. 287.

гомологичные хромосомы (homologous chromosomes)

Два экземпляра одной и той же хромосомы в диплоидном организме.

гомологичный белок (homologous protein)

Один из двух белков, связанных общей историей эволюции. Например, цитохром С человека гомологичен цитохрому С утконоса плоскостопого. Однако иногда этот термин используется и в случаях, когда общая история неясна. Два белка в различных организмах, имеющие общие биологические или химические свойства (функцию, структуру, мотивы), часто называются “гомологичными”, хотя здесь больше подходит слово “аналогичные”, отражающее сходство белков без какой-либо причины. К сожалению, в некоторых областях биологии выражением “аналогичный белок” принято обозначать сходство белков благодаря конвергенции (сходимости изначально далеких ветвей эволюционного дерева), т. е. когда известно, что у них нет общей истории. Это отклонение от нормального словаупотребления лишает биологию важного обычного слова. Иногда биологи используют выражение “такой же белок” (в кавычках), чтобы передать нормальный смысл выражения “аналогичный белок”. В этой книге термин “гомологичный белок” употребляется только в случаях, когда у белков есть общая история.

гомология (homology)

Сходство благодаря общей истории эволюции. Термин “гомология последовательностей” иногда применяется как равноправный термину “сходство последовательностей”, хотя в последнем случае не подразумевается общей истории эволюции. Часто используются такие фразы, как “степень гомологии”, но их нельзя сделать осмысленными вне данного выше строгого определения.

денатурация (denaturation)

Серьезное разрушение естественной структуры белка или ДНК путем нагрева или химической обработки. В двунитевых ДНК денатурацией часто называется их разделение на отдельные нити. Для белков — это разрушение их трехмерной структуры.

диплоид (diploid)

Организм называется диплоидным, если он содержит по две копии каждой хромосомы. Эти копии не обязательно идентичны, так как каждая может содержать другие аллели. У человека большинство клеток, отличных от половых (сперматозоидов и яйцеклеток), диплоидные. См. также гаплоид.

ДНК (DNA)

Дезоксирибонуклеиновая кислота; молекула, представляющая собой цепь из нуклеотидов (оснований). Основаниями в ДНК являются аденин (А), тимин (Т), цитозин (С) и гуанин (Г). ДНК — основной носитель генетической информации. Она обычно состоит из двух нитей комплементарных последовательностей нуклеотидов, представляющих собой спаренные основания (модель Уотсона—Крика). Поэтому при описании этих двух нитей обычно определяется одна из нитей и ее ориентация. ДНК человека в основном линейна, но вообще ДНК может быть и циклической молекулой (см. плазмида). С. 36.

дрозофila (*Drosophila melanogaster*)

Плодовая мушка, часто используемая в генетических исследованиях.

инициирующий кодон (start codon)

Кодон, сообщающий о начале последовательности, которую нужно транслировать в белок. Часто это AUG, но в разных организмах он может быть различным.

инtron (intron)

См. экзон.

информационная РНК — иРНК (messenger RNA — mRNA)

Шаблон РНК, который кодирует отдельный белок. Каждый кодон этой иРНК определяет одну аминокислоту в белковой последовательности. иРНК обрабатывается рибосомой, которая с помощью транспортных РНК собирает вместе нужные аминокислоты будущего белка.

искусственная хромосома дрожжей (Yeast Artificial Chromosome — YAC)

Искусственно созданный клонирующий вектор, который используется для включения последовательностей ДНК длиной от одного до двух миллионов оснований. С. 504.

карта сцепления (linkage map)

То же, что генетическая карта.

катализатор (catalyst)

Молекула, которая делает возможной или ускоряет конкретную химическую реакцию. Основными катализаторами в клетке являются белки, но эту роль может играть и РНК.

кДНК (cDNA)

Комплементарная, или копийная, ДНК, полученная из иРНК (посредством обратной транскрипции). Если иРНК принадлежит эукариотическому организму, кДНК теряет интроны, содержащиеся в подстроке ДНК, из которой транскрибировалась эта иРНК. С. 296.

кишечная палочка (*Escherichia coli* — *E. coli*)

Бактерия, обитающая в кишечнике человека. *E. coli* — один из подробно исследованных модельных организмов в генетике и молекулярной биологии.

кладистика (cladistics)

Частный систематический подход к построению деревьев эволюции и интерпретации их смысла. В общем, кладистический подход пытается воссоздать образец ветвления дерева эволюции (порядок), но не оценивает времени прохода по какой-либо конкретной ветви.

клон (clone)

Популяция организмов, генов или молекул, состоящая из многих точных копий оригинала. Глагол “клонировать” означает делать много копий. “Клонировать последовательность” относится также к процессу извлечения конкретной последовательности или гена из ДНК большей протяженности.

клонирующий вектор (cloning vector)

Организм, вирус или плазмида, а с недавнего времени и искусственная хромосома (например, YAC), которые содержат в собственной клетке ДНК определенный фрагмент и могут клонировать его вместе со своей ДНК. С. 504.

кодирующий участок (coding region)

Участок ДНК, содержащий кодоны, которые определяют конкретный белок. У эукариот кодирующий участок разбивается на отдельные экзоны. Однако кодирующие участки для различных белков могут иногда перекрываться; следовательно, участок нужно определять как кодирующий для конкретного белка. С. 296.

кодон (codon)

Строка из трех нуклеотидов в ДНК или РНК, которая определяет конкретную аминокислоту. (См. также **генетический код** и **трансляция**.) Два кодона называются “синонимичными”, если они определяют одну и ту же аминокислоту. С. 36.

комплементарная последовательность нуклеотидов (complementary nucleotide sequence)

Последовательность нуклеотидов S комплементарна последовательности S' , если каждое основание в S комплементарно соответствующему основанию в S' .

комплементарное основание (complementary base)

В ДНК комплементарными парами оснований являются А, Т и С, Г; в РНК — это А, У и С, Г. См. также **пары оснований**.

космода (cosmid)

Искусственно созданный клонирующий вектор, способный удержать включения длиной около 40 000 оснований. С. 503 и 507.

кроссинговер (crossing over)

Процесс, происходящий при мейозе, когда две гомологичные хромосомы выравниваются друг напротив друга, разрываются и обмениваются фрагментами ДНК, образуя две гибридные хромосомы. Каждая гибридная хромосома содержит чередующиеся участки ДНК из двух исходных хромосом. В норме оба обменивающихся фрагмента имеют одну и ту же длину, но при неравном кроссинговере их длины могут не совпадать, и одна из новых хромосом получается длиннее другой. См. **рекомбинация**.

лейциновая застежка (leucine zipper)

Общий мотив для фактора транскрипции. С. 90.

лиганд (ligand)

Молекула, которая связывается со специфическим локусом на другой молекуле.

лигировать (ligate)

Соединять две молекулы вместе.

локализация, помеченная последовательностью (sequence tagged site — STS)

Грубо говоря — короткая последовательность ДНК, которая встречается в геноме только в одном месте. Более точно — пара праймеров PCR, отстоящих на ограниченном расстоянии и обладающих тем свойством, что PCR работает с этими праймерами только в одной локализации генома. STS обеспечивают маркеры на протяжении всего генома, но они не должны локализовываться в генах. См. также экспрессированная маркерная последовательность (EST). С. 89 и 482.

маркер (marker)

Свойство хромосомы, которое можно определить.

матрица подстановок аминокислот (amino acid substitution matrix)

Матрица, определяющая оценки использования совпадений и несовпадений, зависящие от символов. Два наиболее широко используемых класса таких матриц — это PAM и BLOSUM. С. 462 и 468.

мейоз (meiosis)

Процесс, во время которого половые клетки (сперматозоиды или яйцеклетки) образуются из родительских клеток. Из одной диплоидной клетки во время двух циклов клеточного деления образуются четыре гаплоидных клетки. Обычно во время мейоза гомологичные хромосомы каждой пары вместе выстраиваются друг напротив друга, подвергаются кроссинговеру (рекомбинации) и воспроизводятся, образуя четыре гибридные хромосомы.

митохондрия (mitochondria)

Органелла размером с бактерию внутри эукариотических клеток; энергетический центр клетки, который содержит свою собственную ДНК. Как ей удается сосуществовать с эукариотическими клетками, остается неразрешимой загадкой. Митохондриальная ДНК наследуется только по материнской линии, так что последовательности таких ДНК часто используются для установления истории эволюции.

Никто не ждет, что вы это поймете (You aren't expected to absorb this)

Фраза, используемая в выступлениях: 1) биологов при показе 35-миллиметровых слайдов с нечитаемыми последовательностями ДНК или аминокислот; 2) программистов при показе через оверхед неудобочитаемых кодов на С или С++; 3) статистиков и математиков после заполнения всей доски непостижимыми уравнениями.

нуклеиновая кислота (nucleic acid)

Цепь нуклеотидов — молекула РНК или ДНК.

нуклеотид (nucleotide)

Аденин (A), тимин (T), цитозин (C) или гуанин (G) в ДНК; аденин (A), урацил (U), цитозин (C) или гуанин (G) в РНК. См. основание.

нынеживущий организм (extant organism)

Существующий в настоящее время. Этот термин недостаточно известен компьютерным специалистам, и нет ни одного случая, где бы он использовался в компьютерных науках. Для эволюционной биологии “нынеживущий” — это стандартное слово.

О большое (Big-Oh)

Если даны две неотрицательные функции $f(n)$ и $g(n)$, то при $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$ мы говорим, что $f(n) = O(g(n))$. Интуитивно (но не точно) это означает, что $f(n)$ растет не быстрее, чем $g(n)$, если игнорировать малые значения n и любые постоянные множители.

Если $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$, мы говорим, что $f(n) = \Omega(g(n))$. Интуитивно (но не точно) это означает, что $f(n)$ растет не медленнее, чем $g(n)$, если игнорировать малые значения n и любые постоянные множители.

Если $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, то мы говорим, что $f(n) = \Theta(g(n))$. Интуитивно (но не точно) это означает, что $f(n)$ растет с той же скоростью, что и $g(n)$, если игнорировать малые значения n и любые постоянные множители. См. [32, 112] по поводу дальнейших уточнений.

обратная транскриптаза (reverse transcriptase)

Фермент, который создает двухнитевую ДНК, “копирующую” однонитевую молекулу РНК.

обратная транскрипция (reverse transcription)

Процесс создания двухнитевой молекулы ДНК, кодирующей последовательность по однонитевой молекуле РНК.

олигонуклеотид (oligonucleotide) или, для краткости, олига (oligo)

Короткая цепь нукleinовой кислоты (ДНК или РНК).

онкоген (oncogene)

Ген, который в активном состоянии способствует развитию рака. Напротив, антионкоген — это ген, который в норме способствует подавлению рака, но допускает его развитие, не будучи активным.

основание (base)

В контексте молекулярной биологии: отдельный нуклеотид в молекуле ДНК или РНК. Основания ДНК условно обозначаются А, Т, С и Г. Основания РНК обозначаются А, У, С и Г. См. [нуклеотид](#).

остаток (residue)

Отдельная единица полимера, которая используется для обозначения как отдельного нуклеотида в молекуле ДНК, так и отдельной аминокислоты в белке.

открытая рамка считывания (open reading frame — ORF)

Подстрока ДНК, не содержащая терминирующих кодонов (UAA, UAG и VGA), читаемая в рамке считывания. Иногда ORF определяется иначе — как подстрока ДНК между инициирующим кодоном и первым терминирующим кодоном, найденным в той же рамке считывания. Это определение больше подходит для прокариот и приводит к путанице для эукариот, так как эукариотические гены иногда определяются как содержащие несколько ORF [192], т. е. несколько экзонов. Единичный экзон удовлетворяет первому определению ORF, но не альтернативному. Однако, по-видимому, абсолютного стандарта не существует, и в двух различных главах в [192] используются два разных определения, несмотря на то что обе они посвящены эукариотическим генам. Идентификация открытых рамок считывания часто является первым шагом в попытках локализовать гены в исследуемой ДНК.

палиндромная последовательность в ДНК (palindromic sequence in DNA)

Строка ДНК, которая становится палиндромом в нормальном смысле этого слова после того, как половина этой строки заменяется комплементарной последовательностью. С. 179.

пара оснований (base pair)

Два основания (два нуклеотида), образующие комплементарную пару. В ДНК такую пару составляют А и Т, а также С и Г. В РНК это А и У, а также С и Г. Эти пары называются также парами Уотсона–Крика.

первичная структура (primary structure)

Описание молекулярной строки (ДНК, РНК или аминокислоты) как последовательности.

пиридин (pyrimidine)

В контексте этой книги пиридины — это цитозин (С) и тимин (Т).

плазмида (plasmid)

Циклическая (кольцевая) молекула ДНК, находящаяся в клетке помимо основного генома организма. Типична для бактерий. Часто используется в качестве клонирующего вектора. С. 33.

позиционное клонирование (positional cloning)

Совсем недавний подход к идентификации генов, имеющих отношение к заболеванию. Сначала с помощью генетического картирования находится интервал, в котором геном, скорее всего, содержит этот ген; затем расшифровываются части этого интервала у больных и их здоровых родственников, чтобы найти систематические отличия, которые показали бы, что найден нужный ген. С. 481.

поиск в глубину (depth-first search — *dfs*)

Очень общий рекурсивный алгоритм, используемый для анализа графа. В этой книге он применяется только к деревьям. Поиск в глубину заходит сначала в корень дерева (первая “текущая вершина”), а затем последовательно обходит деревья, корнями которых являются потомки данной текущей вершины. По завершении этих рекурсивных вычислений поиск возвращается из текущей вершины в родительскую. Дальнейшие подробности см. в [10, 32, 112, 401].

полимер (polymere)

Длинная линейная молекула, составленная из идентичных или похожих частей.

полимераза (polymerase)

Фермент, который способствует созданию молекулы нуклеиновой кислоты, комплементарной существующей однонитевой молекуле нуклеиновой кислоты (называемой матрицей).

полипептид (polypeptide)

Длинная цепь из соединенных аминокислот. Иногда заменяет термин белок.

половые клетки (germ cells)

У человека сперматозоиды и яйцеклетки. Это гаплоидные клетки; они содержат по одному экземпляру каждой хромосомы.

прокариоты (prokaryote)

Организмы, у которых ДНК не заключена в ядро в отличие от эукариот.

промотор (promoter)

Подстрока в восходящем потоке ДНК гена, где полимераза РНК (помогающая транскрибировать ДНК в РНК в ходе транскрипции) связывается с ДНК.

протеаза (protease)

Фермент, который разрезает протеин (белок).

протеин (protein)

См. белок.

protoонкоген (proto-oncogene)

Нормальный ген, который может быть конвертирован в онкоген по одному из нескольких механизмов.

пурин (purine)

В контексте этой книги пурин — это либо аденин (A), либо гуанин (G). Как это запомнить? Студенты-биохимики в Университете Дейвиса (the AGgies) используют меморику: AGgies are PURe.

рамка считывания (reading frame)

Одно из трех мест для начала чтения при переводе строки из алфавита ДНК в алфавит аминокислот. Если неизвестно также и направление строки, то часто говорят о шести возможных рамках считывания. С. 36.

расшифровка, или секвенирование, ДНК (DNA sequencing)

Процесс определения полной последовательности спаренных оснований изучаемой строки ДНК. С. 501.

реакция полимеризации цепи (polymerase chain reaction — PCR)

Процесс, используемый для копирования (или амплификации) ДНК в интервале, ограниченном двумя последовательностями праймеров. Сама ДНК может быть *in vitro* (т. е. вне живой клетки). Праймеры выбирает экспериментатор, однако эти праймеры должны располагаться на “приемлемом” расстоянии друг от друга. PCR работает циклически, и в каждом цикле число копий этого интервала ДНК примерно удваивается. Возможность быстро и дешево производить практически неограниченные количества фрагментов ДНК между любыми двумя (относительно близкими) и определяемыми пользователем точками молекулы революционизировала практику молекулярной биологии. PCR характеризуют как средство, означающее “для генов тоже, что некогда печатный станок Гутенберга — для написанного слова” [90]. Изобретатель PCR Кейри Муллис был удостоен Нобелевской премии по химии за 1994 год. Очень интересное описание этого изобретения самим Муллисом можно прочитать в [333].

рекомбинация (recombination)

Общий термин для нескольких механизмов, которые приводят к разрыву или разрезанию с последующим соединением интервалов ДНК. Рекомбинация возникает как естественным путем (например, при мейозе посредством кроссинговера), так и в лабораторных условиях.

репрессор (repressor)

Белок, который уменьшает (или прекращает) экспрессию другого белка, влияя на транскрипцию его ДНК. Обычным механизмом действия репрессора является тесное связывание с ДНК, кодирующей подавляемый белок, которое нарушает его транскрипцию.

ретровирус (retrovirus)

Вирус с геномом, представленным РНК, который для репродуцирования должен быть транскрибирован в двунитевую молекулу ДНК.

рестриктаза (restriction enzyme)

Фермент, который разрезает ДНК в местах, содержащих особые короткие (обычно палиндромные) последовательности.

рибосома (ribosome)

Комплекс из РНК и белка, в котором синтезируется белок, определяемый иРНК.

РНК — молекула рибонуклеиновой кислоты (RNA — ribonucleic acid molecule)

См. нуклеиновая кислота.

сателлитные ДНК (satellite DNA)

Короткие строки ДНК, которые многократно повторяются в геноме. Они подразделяются на мини- и микросателлиты по длине повторяющейся подстроки. С. 180.

сцепление (linkage)

Степень, в которой два маркера в ДНК наследуются вместе. Сцепление отражает в некоторой степени физическое расстояние. Чем ближе друг к другу находятся два маркера в одной хромосоме, тем более вероятно, что они вместе наследуются и не разделяются при кроссинговере (рекомбинации).

ТАТА-блок (TATA box)

Название подстроки “ТАТА”, часто встречаемой в промоторной области многих генов.

тэломера (telomere)

ДНК, образующая конец хромосомы. Она содержит многократно повторяемые короткие подстроки. С. 595 и 7.11.1 с. 178.

терминирующий кодон (stop codon)

Кодон, который сообщает о завершении последовательности, которую нужно транслировать в белок; часто это UAA, UAG или UGA.

тихая мутация (silent mutation)

Мутация в кодоне ДНК, не изменяющая определяемую им аминокислоту. Чаще всего тихая мутация происходит в третьем нуклеотиде кодона. Например, TCN кодирует аминокислоту серин при любом из четырех значений N, возможных для нуклеотида ДНК. Таким образом, мутация в третьем нуклеотиде — это тихая мутация.

точковая мутация (point mutation)

Изменение одного нуклеотида (в ДНК или РНК) или одного аминокислотного остатка (в белке).

транскрипция (transcription)

Процесс синтеза молекулы РНК, комплементарной молекуле ДНК в гене. У эукариот и интроны и экзоны транскрибируются в РНК. В дальнейшем интроны сращиваются, образуя иРНК.

трансляция (translation)

Процесс синтеза белка. В нем используется “калька”, задаваемая молекулой информационной РНК.

транспортная РНК — тРНК (transfer RNA — tRNA)

Молекула РНК, которая транспортирует конкретную аминокислоту к растущей цепи аминокислот в соответствии с определенным кодоном в иРНК.

усиливающая последовательность (enhancer sequence)

Последовательность ДНК, связывающаяся с регуляторными белками. Это связывание вызывает изменение скорости, с которой транскрибируются некоторые интервалы ДНК, а следовательно, синтезируются некоторые белки. Усиливающую последовательность часто смешивают с промоторной последовательностью, но, в отличие от последней, усиливающая последовательность может отстоять очень далеко от регулируемой ею ДНК. Противоположностью усилителя является репрессор.

фактор транскрипции (transcription factor)

Общий термин для белков, способствующих инициации и регулированию транскрипции. См. также лейциновая застежка или цинковый палец. С. 90.

фенотип (phenotype)

Полный наблюдаемый набор особенностей организма. См. также генотип как его противоположность.

фермент (enzyme)

Белковый катализатор, ускоряющий специфические химические реакции в клетке.

филогения (phylogeny)

Древовидная история эволюции таксонов. С. 539.

фуши таразу (fushi tarazu)

Одно из самых забавных имен мутантного аллеля дрозофилы — организма, многие мутанты которого имеют необычные имена. По-японски это значит “пропуск полоски”.

хромосома (chromosome)

Большая линейная или кольцевая структура, составленная из ДНК и некоторых белков. Наследственная информация организована в наборе хромосом.

цинковый палец (zinc finger)

Общий мотив для фактора транскрипции. С. 90.

шаг, ограничивающий скорость (rate-limiting step)

Одна из нескольких фраз, охотно используемых в биохимии, а не в программировании. В программировании мы бы назвали это “узким местом”.

экзон (exon)

У эукариот гены обычно разбиваются на чередующиеся участки экзонов и инtronов. Экзоны содержат информацию, переписываемую в информационную РНК (иРНК) и в конечном счете используемую для синтезирования белка (или иногда РНК), в то время как смысл инtronных последовательностей неизвестен. С. 295.

экзоно-инtronовая склейка (exon-intron splice site)

Точка в эукариотическом гене, где инtron соединяется с экзоном. Природа знает, как идентифицировать и отклеивать интраны из транскрибуемой РНК, а у нас полного объяснения для распознания склеек нет. С. 295 и 307.

электрофорез (electrophoresis)

Процесс, при котором сортируются молекулы (ДНК, РНК или белка) с различными свойствами (такими, как заряд, длина или размер). Молекулы помещаются в гель, в котором под действием электрического тока (иногда переменного) молекулы с различными свойствами перемещаются с различной скоростью.

экспрессированная маркерная последовательность (expressed sequence tag — EST)
STS, полученная из молекулы кДНК. Следовательно, EST — это STS, находящаяся в гене, а не в некодирующей области ДНК. С. 89.

экспрессия (expression)

Ген экспрессирован, если белок, который им закодирован, производится. Иногда мы говорим также, что экспрессируется белок.

эукариоты (eukaryote)

Организмы, ДНК которых располагаются в ядре. К эукариотам относится вся “высокоорганизованная” жизнь. В этой книге важной особенностью эукариот является разбиение их генов на чередующиеся экзоны и интроны, что отличает их от прокариот.

PAM

Сокращение для “point accepted mutation” или “percent accepted mutations”. PAM используется как единица расстояния в эволюции, а также как название конкретной матрицы оценок в подстановке аминокислот. С. 462.

Англо-русский словарь терминов

Здесь мы приводим небольшой словарь терминов, которые были приняты в переводе.

alignment — выравнивание
ancient conserved region — древние консервативные участки

base — основание
bind — связывать
block — блок
block-partition — блочное разбиение
bottom-problem — нижняя задача
bottom-up calculation — восходящее вычисление
box — блок

cDNA — кДНК

cell adhesion — адгезия клеток

chimeric — химерный

clique — клика

cloverleaf — клеверный лист

commonality — общность

comparc/shift phase — фаза сравнения/сдвига

complimented palindrom — комплементарный палиндром

contaminant — загрязнение

cooptimal — кооптимальный

cover — покрытие

covered interval — покрытый интервал

crawler — краулер

dictionary problem — задача о словаре

diploid — диплоид

double-barreled shotgun — двусторонний дробовик

edge — дуга (для ориентированных графов); ребро

end-space — концевой пробел

enhancer — усилитель

evolutionary tree — эволюционное дерево

exact set matching problem — множественная задача точного поиска

exon — экзон

exon shuffling — тасование экзонов

expressed sequence tag — экспрессируемая маркерная последовательность

extant — нынешивший

failure function — функция неудач

failure link — связь неудачи

fingerprint — дактилограмма, отпечаток пальца

fingerprint method — дактилоскопический метод

finite state automaton — конечный автомат

fold — кратность

folding problem — задача складывания

forward dynamic programming — впередсмотрящее (прямое) динамическое программирование

frameshift — рамка считывания

gap — пропуск

genetic mapping — генетическое картирование

goodness — “хорошество”

greedy — жадный

hairpin loop — заколка для волос

haploid — гаплоид

hidden Markov model — скрытое марковская модель

homeobox — гомеобокс

homeodomain — гомодомен

indel — индел (от Insertion/Deletion)

inorder numbering — фронтальная нумерация

interleaving — прослойка

jumbling — взбалтывание

jumping gene — прыгающий ген

keyword tree — дерево ключей

Kleen closure — замыкание Клини

layout — раскладка

lca common ancestor — наименьший общий предок

Leucine Zipper — лейциновая застежка

lexically — лексически

lifted alignment — поднятое выравнивание

log-odds — логарифмическое различие; лог-различие

map — карта

mapping — отображение

match-count — счет совпадений

motif — мотив

multiplicity — кратность

multiset — набор

mutational event — мутационное событие

nesting pairing — вложенное паросочтание

node — вершина

objective function — целевая функция

occurrence — вхождение

offset — смещение

offset vector — вектор смещений

oligo — олига

output link — связь выхода

paging — пейджинг; управление страницами

pairing — паросочтание

palindrom — палиндром

parsimony — брехливость

pattern — шаблон

phylogenetic — филогенетическое

point mutation — точковая мутация

pool — пул

pooling — группировка

position tree — дерево позиций

preorder numbering — нисходящая нумерация

preprocessing — предварительная обработка

primer — праймер; затравка

processed pseudogenes — обработанные псевдогены

push you-pull me — тяни-толкай

query — запрос
quick-and-dirty — тяп-ляп

rearrangement — перестройка
recurrence — рекуррентнос соотношенис; иногда
рекуррентия
refinement — измельчение
refiner — измельчитель
regular expression — регулярное выражение
repeat — повтор
represor — подавитель; репрессор
residue — радикал
restricted block function — ограниченная блоковая функция
retrovirus — ретровирус
reverse transcriptase — обратная транскриптаза
rooted tree — дерево с корнем
row — строка (таблицы)
rule-of-thumb — кустарное правило
run — полоса

score — оценка
separated palindrome — разделенный палиндром
scquncc-tagged site — локализация, маркованная
последовательностью
sequencing — секвенирование; расшифровка
shotgun — дробовой
similarity — сходство
site — сайт
skip/count — скачок по счетчику
slippage — проскальзывание
snurp — расшнуровывать
sophomore — второкурсник
soup — компот

spanning tree — оствное дерево
sparse dynamic programming — разреженнос динамическос
программированис
spliced alignment — склесное выравнивание
stem — черенок
string — строка (текста)
strip — полоса
stuttering — заикающаяся
suboptimal — субоптимальный
substitution — подстановка
suffix extension — продолжение суффикса
suffix tree — суффиксное дерево
superfamily — суперсемейство
supersequence — надпослдовательность
superstring — надстрока
systolic array — систолический массив

tandem — tandem
tandem array — tandemный ряд
termination symbol — завершающий символ
tic — ничья
top problem — верхняя задача
top-down calculation — нисходящее вычисление
transcript — предписание (редакционное)
transcript — транскрипция (РНК)
transcription — транскрипция
transcription factor — фактор транскрипции
trick — прием

weighted — взвешенный
wild card — джокер; шаблон

Zink Finger — цинковый палец

Предметный указатель

- agrep* 99, 101, 115, 467
Alu, последовательности 182
- BAC-PAC, сквознорование 322
BLAST 342, 456, 459, 460, 510
BLOCKS 412, 456, 467
BLOSUM 456, 467–469
BRCA1 470
BYP метод 333
- EST 89, 479, 498
- FASTA 342, 456, 457
- GenBank 453, 454, 457
- k* несовпадений, задача 250, 320, 326
k различий, задача истинного совпадения 320, 323, 324, 326
—, повторы tandemных 252, 260
k-общая подстрока, задача 256, 258
k-повтор наименееший 218
k-покрытие 219
- lca*, построение за константное время 239
- P*-против-всех, задача 346
PAM (единица измерения) 463
—, матрица 456, 462, 464, 466, 469, 572
patricia (дерево) 351
PROSITE 93, 98, 116, 412, 456, 466, 470
- q*-грамма 217
- SBH, творчесма о непрерывности 530
Shift-And метод 99, 101, 103, 115, 335, 467
SP(*P*) алгоритм 77
- SP*-выравнивание, задача 418
— множественное 445
—, приближение с ограниченной ошибкой 421
STS 89, 478, 479
—, карта 89
—, порядок 487
—, упорядочение 489
STS-картирование 482, 489
Swiss-Prot 451, 454, 457, 466
- t*-блок 369
- UPGMA 437
- XPARAL 382
- YAC, карта 505
- Z*-алгоритм см. алгоритм *Z*
- А**втомат конечный 94, 167, 170, 214, 215
— для сравнения строк 52
аддитивная матрица 577
аддитивное дерево 551, 563, 574, 576
— компактное 553, 574
— расстояние 540
алгоритм *k* различий 330
— *SP(P)* 77
— *Z* 29, 44, 55
—, использование значений *sp* для вычисления значений *Z* 55
— Ахо–Корасика 46, 78, 81, 86, 92, 96, 153, 156, 170, 213, 216, 334
—, алгоритм поиска 83, 87
—, препроцессинг для функции исходач 84
— Бойера–Мура 45, 55, 59, 114, 119, 155, 170
—, доказательство Коула 64
—, правило Галиса 72
— на основе *Z* 44
— Вайнера 126, 141, 150
— Евклида 521
— жадного слияния 536
— Зива–Лемпеля 221
— Кнута–Морриса–Пратта 38, 46, 50, 52, 81, 96, 119, 155, 159, 170, 200, 216
—, врсмнной анализ 49
—, корректность 48
—, препроцессинг 49
—, функция исходач 51
— на основе *Z* 50
— Мак-Крейга 150
— наибольшей возрастающей подпоследовательности 354
— Ньютона поиска по лучу 383
— редакционного расстояния блоковый 370
—, — “четырех русских” 369, 378
— Укконена 126, 128, 141, 150, 151, 211
—, одна фаза: SPA 140
—, отдельно продолжение 133
— Фитча–Хартигана 578
— Хиршберга 318
алгоритмы построения дерева 540
алфавитная независимость 32, 155
алфавитно-взвешенное редакционное расстояние 280
альтернативная склейка 312
Альгеймса болезнь 497
анализ чувствительности 389
Апостолико–Джанкарло метод 60, 95
арифметика 99
Ахо–Корасика алгоритм см. алгоритм Ахо–Корасика
- Б**аза данных (банк последовательностей ДНК) 323
—, поиск 163, 449, 469, 497
—, — иссязьточная 323
белковые домены 220
блок 36
—, структура 307
—, — вторичная 438
— геликаза 412
брсржливость 554
— максимальная, задача 567

битовые операции 99
 блоковая функция 369
 — ограничнна 369
 блоковый алгоритм редакционного расстояния 370
 Бойера–Мура алгоритм см. алгоритм Бойера–Мура
 — точнос множественное сравнение 202, 203, 207,
 220, 335

Вайнера алгоритм 126, 141, 150
 вектор индикаторный 144
 — клонирующий 504
 — связь 144
 — смещение 373
 вершина, путевая метка 122
 вершинная глубина 135
 — текущая 136, 141, 149
 вес пропуска, выбор 298
 — аффинный 299
 — (и постоянный) 298, 302, 359, 466
 — вогнутый 298, 359, 402
 — постоянный 298
 — произвольный 298, 299, 301
 — стартовый 298
 взбалтывание 438
 взвешенное выравнивание 343, 351
 — редакционное расстояние 279, 378
 Вингрона–Аргос метод множественного
 выравнивания 442
 вирус саркомы Симиана 450
 возрастающая подпоследовательность 352
 — наибольшая 355
 — — — задача 352
 время полиномиальное, приближенные схемы 435
 вс-против-вс-х, задача 343, 347
 второй факт сравнения биологических
 последовательностей 406
 выбор консенсуса 512
 — праймса 224, 331
 выборочный метод Гиббса 444
 выравнивание, граф 284
 —, значениe 281
 —, форма 285, 293
 — *SP*-множественное 445
 — — —, приближение с ограниченной ошибкой 421
 — взвешенное 343, 351
 — глобальное 278, 285, 286, 291, 313, 335, 388, 391,
 417
 — индуцированное 416
 — карт 199, 499, 500
 — консенсусное, приближение с ограниченной
 ошибкой 429
 — кооптимальное 304, 389
 — локальное 286, 291, 298, 319, 342, 402, 591
 — — —, цепление различных локальных выравниваний
 395
 — множественное 416
 — множественное 165, 358, 403, 409, 416, 426, 430,
 436, 447, 467, 511
 — — —, метод Вингрона–Аргос 442
 — — —, повторяющийся мотив, методы 440
 — — —, приближение с ограниченной ошибкой 421
 — глобальное 405, 436
 — — итеративное 436, 438, 439
 — — прогрессивное 440
 — параметрическое 380, 389, 390, 400
 — по сумме пар 421
 — поднятие 431, 434
 — последовательностей 263
 — последовательности и структуры РНК 311

выравнивание прогрессивное 572
 — профиль 410, 412
 — с бесплатными концевыми пробелами 284
 — сетевое 584
 — склесенно 583, 592
 — строк 271
 — субоптимальное 391, 395
 — филогенетическое 426, 430, 434, 541, 569, 572
 — — —, приближение с ограниченной ошибкой 431
 выравнивания, близкое к оптимальным 392
 — — — —, перечисление 393
 — — — —, подсчет 393

Галила метод 114
 — правило в алгоритме Бойера–Мура 72
 гамильтонов путь 586
 геликазные белки 412
 ген 36
 —, сборка 396
 генетическая карта 479
 генетический код 36
 “Геном человека”, проект 89, 479
 геном, перестройка 595
 геномный импринтинг 181
 — проект 479
 Гиббса метод выборочный 444
 гиперкуб 567, 568
 глобальное выравнивание 278, 285, 286, 291, 313,
 335, 388, 391, 417
 — множественное 405, 436
 глубина вершинная 135
 — текущая 136, 141, 149
 — строковая 122
 гомобокс 287
 горизонтальный перенос 552
 Горнера правило 109
 гранины 471
 граф выравнивания 284
 — редакционный 278, 305
 — слова ориентированный ациклический 167
 графы дс Брэйна 529

Дактилоскопические методы Карпа–Рабина 107, 115
 дактилоскопия 99, 109, 491
 двумерного образца поиск, задача 115, 116
 двусторонний дробовик 534
 де Брэйна графы 529
 дерево, алгоритмы построения 540
 — *patricia* 351
 — квартетное 577
 — ключей 79, 80, 122, 153, 155, 160, 201, 213, 215,
 221, 334, 461, 560
 — компактное аддитивное 553, 574
 — основное максимальное 437, 439
 — — минимальное 437, 553, 569, 574, 576
 — очистки, повторяющаяся подстрока 221
 — позиций 120, 154, 199
 — с аддитивными расстояниями 551, 563, 574, 576
 — суффиксное 119, 121, 155, 159, 166, 170, 174, 185,
 188, 190, 199, 201, 215, 218, 221, 320,
 326, 335, 340, 343, 345, 347, 351, 375, 510
 — — —, вопросы реализации 152
 — — —, длина 346
 — — —, сжатие дуговой метки 138
 — — —, сокращение 169
 — — —, явно 127
 — — обобщенное 151, 161, 166, 174, 176, 199, 205,
 224, 246

дерево ультраметрическое 542, 563, 565, 566
 — минимальное 543
 — филогенетическое 465, 555, 558
 — штейнсово 567, 568, 576
 — эволюционное 430, 440, 540, 543
 дерево-трайник 577
 деревья, совместимость 560, 561
 джокер 90, 96, 106, 116, 249
 динамическое программирование 250, 272
 — впередсмотрящее (прямое) 361, 378
 — гибридное 327, 343, 376
 — обратное 420
 — разреженное 357, 358, 360, 398
 длина кода ДНК минимальная, повторяющаяся подстрока 212
 — строки 346
 — суффиксного дерева 346
 ДНК (молекула) 36
 — базы последовательностей (банки ДНК) 323
 — задача о загрязнении 164
 — код минимальной длины, повторяющаяся подстрока 212
 — секвенирование 225
 —, — дробовое 508
 —, факторы транскрипции 90
 — колцовая 33
 — комплементарная 296
 — митохондриальная 161
 — сателлитная, повторяющаяся подстрока 180
 домены белковые 220
 дробовая сборка последовательности 163, 285, 534
 дробовые двусторонние 534
 дробовое секвенирование ДНК 323, 503, 508, 513, 605
 дрожжи, искусственные хромосомы 484, 504

Евклида алгоритм 521
 единицы РАМ 463
 — последовательные, задача 483
 — сплошные 574

Жадная раскладка клонов 494
 жадное назначение 535
 — покрытие 353, 354
 — размещение проб 495
 — слияние, алгоритм 536

Загрязнение ДНК, задача 164
 задача *P*-против-всех 346
 — пороговая 375
 — *SP*-выравнивания 418
 — все-против-всех 343, 347
 — выбора пробы, выбор праймера 331
 — выполнимости 585
 — древовидного выравнивания 430
 — источного совпадения с *k* различиями 320, 323, 324, 326
 — о бородачем торговце 485
 — — —, путь 487
 — о наибольшей возрастающей подследовательности 352
 — — — общая подследовательность 283, 297, 305, 352, 355, 377
 — — — подстроки 158, 161, 162, 165, 174, 214, 510
 — — — множественная 164, 256, 258
 — о подстроке 119, 160, 191
 — — — множественная 447
 — о редакционном расстоянии 270
 — — — операционно-взвешенном 279

задача о *k* несовпадениях 320, 326
 — о *k*-общий подстроке 256, 258
 — о загрязнении ДНК 164, 250
 — о зникающей подследовательности 309
 — о кратчайшей надстроке 175, 514
 — о максимальной бережливости 567
 — о минимуме мутаций 571, 577
 — о надстроке 527, 534
 — о назначениях 526
 — о наибольшем повторяющемся префикссе 158
 — о поиске двумерного образца 115, 249
 — о последовательных единицах 483
 — о праймере с *k* различиями 331
 — о словаре 81, 120
 — о сравнении карт 198
 — о суффиксно-предфиксном совпадении 174, 217, 322, 509, 521
 — поиска по лучу 382
 — складывания ТРНК 309
 — склесенного выравнивания 583
 — совпадения сетей 580
 — сцепления 398, 468, 583, 592
 заколка для волос 503
 звезда центральная 423
 Зива—Лемпеля алгоритм 221

Импринтинг геномный 181
 инверсионное расстояние 596, 597
 инверсия подстрок 312, 594
 — со знаком 295, 601
 индикаторный вектор 144
 индуцированное выравнивание 416
 интроны 37, 180, 212, 220, 295, 452
 исключений методы 332, 338, 342, 457
 искусственные хромосомы дрожжей 484, 504
 исходный препроцессинг Кнута—Морриса—Пратта 74, 76
 итеративное множественное выравнивание 436, 438, 439

Кандидаты, использование списка 363
 Карпа—Рабина методы дактилоскопические 107, 115
 карта, сборка 490
 — STS 89
 — YAC 505
 — генетическая 479
 — рестриктаз 198, 499
 — физическая 479
 картирование, STS 482, 489
 — радиационно-гибридное 485, 489
 — физическое 182, 481
 карты 479
 —, выравнивание 199, 499, 500
 —, задача о сравнении 198
 квартетное дерево 577
 кДНК (комплементарная ДНК) 296
 —, сравнение 295—297
 кладистика 540
 клесверный лист 179
 клон химический 484
 клонированием позиционное 481, 582
 клонирующий вектор 504
 клон, раскладка 492
 —, жадная 494
 Кнута—Морриса—Пратта алгоритм см. алгоритм Кнута—Морриса—Пратта
 — препроцессинг исходный 74, 76
 код генетический 36
 кодон, смысление 583

- кольцевая ДНК 33
 компактное аддитивное дерево 553, 574
 комплементарная ДНК см. кДНК
 комплементарный палиндром 178, 180, 248
 конечный автомат 94, 167, 170, 214, 215
 консенсус, выбор консенсуса 512
 —, представление 426
 консенсусная ошибка 426, 427
 — последовательность 410, 512
 — строка 427, 428
 — штейнерова 426, 428
 — целесообразная функция 426
 консенсусное выравнивание, приближенное с ограниченной ошибкой 429
 консенсусный символ 427
 константное время для построения *lca* 239
 контиг 219, 452, 491
 кооптимальные выравнивания 304, 389
 корректность алгоритма Кнута—Морриса—Пратта 48
 корреляция циклическая 106
 космида 505
 Коула доказательство для алгоритма Бойера—Мура 64–66
 кратчайшая надстрока, задача 175, 514
 — общая надпоследовательность 444
 кратчайший путь 278
 кроссинговер неравный 180
- Л**андau—Вишнина метод 326, 335
 ландшафт 184
 Левенштейна расстояние 270
 лейциновая застежка 90
 лемма о перекрытии 521
 — об ОНД 66
 логарифм относительного различия (лог-различие) 410, 464, 469
 — — —, оценка 472
 локализации, помеченные последовательностью 89, 173
 локальноное выравнивание см. выравнивание локальноное
 — сходство 510
- М**айерса метод с сублингвистическим времнем 338
 Мак-Крейга алгоритм 150
 максимальная близость, задача 567
 — пара повторяющаяся 183
 максимальное оставное дерево 437, 439
 максимальные пары 188, 189
 — — общие 218
 максимальный палиндром 247
 — повтор 184, 186, 218
 маркерная последовательность экспрессируемая 89, 175
 массив суффиксный 191, 199, 340, 499
 — тандемный 34, 180, 222
 — — примитивный 222
 матрица аддитивная 577
 — ультраметрическая 544, 573
 матрицы BLOSUM 456, 469
 — PAM 456, 462, 464, 466, 469, 572
 — оценки 343, 382, 388
 метод вершины путевая 122
 — пути 122
 метод Shift-And 99, 101, 103, 115, 335, 467
 — ВУР 333
 — Апостолико—Джанкарло 60, 95
 — Винграна—Аргос множественного выравнивания 442
 метод Галила 114
 — Гиббса выборочный 444
 — Ландau—Вишнина 326, 335
 — Майерса с сублингвистическим времнем 338
 — Нидлмана—Вунча 291
 — рандомизированной дактилограммы 108, 111
 — реального времени 51
 — Смита—Уотермена 291
 — средней связи 437
 — центральной звезды 423, 429
 — Чанга—Лаулера 336
 — “четырех русских” 369, 378
 — “шотган” см. дробовое секвенирование ДНК
 методы исключения 332, 338, 342, 457
 микросателлиты 182, 481
 минимальное оставное дерево 437, 553, 569, 574, 576
 — ультраметрическое дерево 543
 минимальной облицовки путь 507
 минимум мутаций, задача 571, 577
 множественная задача о наибольшей общей подстроке 164, 256, 258
 — — о подстроке 447
 — фильтрация 338
 множественное выравнивание см. выравнивание множественное
 — сравнение см. сравнение множественное
 моделирование пропуска 311
 модель памяти свободного доступа 108, 112
 молекулярная биология, использование
 — повторяющейся подстроки 181
 — эволюция, нейтральная теория 549
 — эпидемиология 322
 молекулярные вычисления 585
 — последовательности, повторяющаяся подстрока 177
 — часы 573
 мотив 413, 441, 456, 467
 — повторяющийся 442
 мутации, задача о минимуме 571, 577
 — на уровне генома 594
 мутация присмесь 463, 548
- Н**абор образцов 46
 надпоследовательность 377
 — общая 377
 — — кратчайшая 444
 надстрока 29, 515, 527, 534, 537
 — кратчайшая, задача 175, 514
 назначения, задача 526
 наибольшая возрастающая подпоследовательность 352, 355
 — общая подпоследовательность, задача 283, 297, 305, 352, 355, 377
 — — подстрока, задача 158, 161, 162, 165, 174, 214, 510
 — — —, — множественная 164, 256, 258
 наибольшее общее продолжение 245, 326, 327
 — — —, эффективное по памяти 247
 наибольшая возрастающая подпоследовательности алгоритм 354
 наибольший общий префикс, повторяющаяся подстрока 217
 — повторяющийся префикс, задача 158
 наименший *k*-повтор, повторяющаяся подстрока 218
 — общий предшественник 227
 — — —, построение 227
 направление секвенирования 502, 514
 независимость алфавитная 32, 155
 неизбыточная база данных 323

- нейтральная теория молекулярной эволюции 549
 исперекрывающиеся приближенные повторы 401
 непрерывность SBH, тесрома 530
 искривство треугольника 423, 489
 иерархический кроссинговер 180
 неточно повторяющиеся подстроки 306
 неточное совпадение 99, 103, 119, 269, 331, 343
 — с k различиями, задача 320, 323, 324, 326
 Нильмана–Вунча метод 291
 Нью-хэмпширский код 575
- Образца двумерного поиска, задача** 115, 116
образцы, набор 46
общая надпоследовательность 377
 — кратчайшая 444
 — пара максимальная 218
 — подпоследовательность наибольшая, задача 283, 297, 305, 352, 355, 377
 — подстрока 164, 510
 — наибольшая, задача 158, 161, 162, 165, 174, 214, 510
 — — —, множественная 164, 256, 258
 общес продолжение наибольшее 245, 326, 327
 — — —, эффективное по памяти 247
 общий предшественник наименьший 227
 — — —, построение 227
 — префикс наибольший 217
 ограниченная блоковая функция 369
 — ошибка приближения см. приближение
 с ограничнной ошибкой
 ОНД, лемма 66
 —, тесрома 521
 онкогены 182, 450
 операционно-взвешенное редакционное расстояние,
 задача 279
 оптимальное редакционное предписание 270
 основное пропцессинг 27, 38, 42, 52, 78
 — для P 49
 оставное дерево максимальное 437, 439
 — минимальное 437, 553, 569, 574, 576
 открытая рамка считывания 583
 оценка, схемы 292
 — лог-различия 472
 — типа суммы пар 417
 оценок матрица 343, 382, 388
 ошибка консенсусная 426, 427
 — приближения ограничнная см. приближение
 с ограничнной ошибкой
 — сдвига рамки считывания 579
- Палиндром** 178, 247
 — комплементарный 178, 180, 248
 — — —, повторяющаяся подстрока 248
 — максимальный 247
 — приближенный 251
 — разделенный, повторяющаяся подстрока 248
 память 153, 167, 170, 174
 —, использование 313, 331, 377
 —, модель свободного доступа 108, 112
 —, уменьшение 171, 314, 346
 —, эффективное по памяти наибольшее общее
 продолжение 247
 пара максимальная 188, 189
 — — —, повторяющаяся 183
 параметрическое выравнивание 380, 389, 390, 400
 пары максимальные общие, повторяющаяся подстрока
 218
 первый факт анализа биологических
 последовательностей 265, 406
- перекрытие, лемма 521
 —, поиск 509
 — суффиксо-префиксное 176, 535
 — — —, задача 174, 217, 322, 509, 521
 перестройка генома 595
 период строки 66
 периодическая строка 66
 плазмы 33, 259
 пм-тройка 223
 повтор зеркальный 179
 — максимальный 186, 218
 — — —, повторяющаяся подстрока 184, 218
 — почти-супермаксимальный 187, 188
 — префиксный 217
 — приближенный исперекрывающийся 401
 — супермаксимальный 184, 187, 218
 — тандемный 177, 182, 222, 251, 252, 306
 — — — с k различиями 252, 260
 повторяющаяся пара максимальная 183
 — подстрока см. подстрока повторяющаяся
 повторяющиеся структуры 177
 повторяющийся мотив 442
 — префикс наибольший, задача 158
 подпоследовательность 24, 283
 — versus подстрока 24, 263, 283
 — возрастающая 352
 — — — наибольшая 355
 — заикающаяся, задача 309
 — общая наибольшая, задача 283, 297, 305, 352, 355, 377
 подстрока 23, 24
 —, задача 119, 160, 191
 — — —, множественная 447
 —, инверсия 312
 —, раскладка 510
 — общая 164, 510
 — — — наибольшая, задача 158, 161, 162, 165, 174, 214, 510
 — — — —, множественная 164, 256, 258
 — повторяющаяся 34, 178, 218, 226, 297, 306, 514
 — — —, алгоритмические задачи 182
 — — —, гибридное динамическое программование
 343
 — — —, дерзво очистки 221
 — — —, код ДНК минимальной длины 212
 — — —, комплементарный палиндром 178
 — — —, максимальная повторяющаяся пара 183
 — — —, максимальные общие пары 218
 — — —, максимальный палиндром 247
 — — — —, повтор 184, 218
 — — —, наибольшее общее продолжение 245
 — — —, наибольший общий префикс 217
 — — —, наименьший k -повтор 218
 — — —, иссивая 218
 — — —, общие подстроки 164
 — — —, палиндром 178, 247
 — — — —, комплементарный 248
 — — —, разделенный 248
 — — —, перебор по одной 218
 — — —, почти-супермаксимальный повтор 187
 — — —, префиксный повтор 217
 — — —, приближенные палиндромы 251
 — — —, примитивные тандемные массивы 222
 — — —, различные строки 217
 — — —, сателлитная ДНК 180
 — — —, статистика совпадений 171
 — — —, супермаксимальный повтор 184, 187, 218
 — — —, тандемные повторы 177, 182, 252
 — — — — с k различиями 252

- подстрока повторяющаяся, тасование экзонов 220
 — Alu 181
 — в молекулярной биологии 181
 — в молекулярных последовательностях 177
 — собственная 23
 подстрока *k*-общая, задача 256, 258
 подстрока *versus* подпоследовательность 263, 283
 подстроки источно повторяющиеся 306
 позиционное клонированиe 481, 582
 поиск в базе данных 163, 449, 469, 497
 — двумерного образца, задача 115, 116
 — перескрайтий 509
 — по лучу, задача 382
 — по регулярному выражению 93, 343
 покрытие жадное 353, 354
 — циклическое 519, 527, 537
 полиморфная цепная реакция 225
 полиморфизм с ограниченной длиной фрагмента 480
 полиномиальноe время, приближенные схемы 435
 полуperiодическая строка 66
 полуperiодичность 537
 порядок STS 487
 последовательности, выравниваниe 263
 —, множественное сравнение 453
 — биологические, второй факт сравнения 406
 —, первый факт анализа 265, 406
 последовательность 24
 —, сборка 322, 452, 501, 508, 515
 —, — дробовая 163, 285, 534
 — Alu 182
 — консенсусная 410, 512
 — маркерная экспрессируемая 89, 175
 последовательностью помеченные локализации 89, 173
 последовательные единицы, задача 483
 построение *lca* за константное время 239
 — наименшего общего предшественника 227
 поток в сети 535
 почти-супермаксимальный повтор 188
 —, повторяющаяся подстрока 187
 правило Галила в алгоритме Бойера–Мура 72
 — Горнера 109
 — плохого символа 39, 40
 —, —, точное множественное сравнение по Бойеру–Муру 202
 —, —, расширенное 40, 202
 — хорошего суффикса 41, 45, 64, 67, 70
 —, —, пропроцессинг для него 43
 —, —, точное множественное сравнение по Бойеру–Муру 203
 —, —, сильное 38, 42, 56
 —, —, слабое 43
 пример, выбор 224, 331
 — с различиями, задача 331
 представление консенсуса 426
 — симметрическое 408, 409
 — сигнатур 412
 — суперсимметрическое 408, 409
 предшественник общий наименший 227
 —, —, построение 227
 пропроцессинг 27, 119, 123, 170, 201, 206, 215, 245
 — в алгоритме Бойера–Мура 41, 43
 —, — Кнута–Морриса–Пратта 49
 — для *P* основной 27, 49
 — для правила хорошего суффикса 43
 — Кнута–Морриса–Пратта исходный 74, 76
 — основной 29, 38, 42, 52, 78
- префикс 23
 — наибольший общий, повторяющаяся подстрока 217
 — повторяющийся, задача 158
 — собственный 23
 префиксно-полупериодическая строка 66
 префиксный повтор, повторяющаяся подстрока 217
 приближение за полиномиальное время 489
 — с ограниченной ошибкой 434
 —, —, *SP*-выравнивание множественное 421
 —, —, —, выравнивание множественное 421
 —, —, —, — консенсусное 429
 —, —, —, — филогенетическое 431
 приближенное вхождение 285, 333
 — совпадение 171, 263, 499
 приближенные палиндромы, повторяющаяся подстрока 251
 — повторы непрекращающиеся 401
 — схемы с полиномиальным временем 435
 приближенный палиндром 251
 примитивная строка 223
 примитивные tandemные массивы, повторяющаяся подстрока 222
 прогрессивное выравнивание 572
 — множественное 440
 продолжение общее наибольшее 326, 327
 —, —, повторяющаяся подстрока 245
 —, —, эффективное по памяти 247
 продолжение суффикса 129
 проект “Геном человека” 89, 479
 прокариоты 37
 пропуск 292
 —, выбор веса 298
 —, моделирование 311
 —, штраф за него 379
 простые числа, теорема 110
 protoонкогены 451
 профили, выравнивание 410, 412
 профиль 410, 467, 468
 псевдоген 179, 297
 — обработанный 297
 псевдоузел 311
 путевая метка вершины 122
 путь, метка 122
 — бродячего торговца 487
 — гамильтонов 586
 — кратчайший 278
 — минимальной облицовки 507
 — эйлеровский 528, 538
- Разбиение, использование** 333
рамка считывания 579
 —, ошибка свдвига 579
 —, —, свдвиг 36
 — открытая 583
рандомизированная дактилограммы метод 108, 111
раскладка клонов 492
 —, — допустимая 492
 —, — жадная 494
 — подстроки 510
расстояние аддитивное 540
 — в инверсиях 596, 597
 — Левенштейна 270
 — редакционное 269, 270, 303, 324, 351, 369, 378, 430, 499, 550
 —, алгоритм “четырех русских” 369, 378
 — алфавитно-взвешенное 280
 — взвешенное 279, 378
 — операционно-взвешенное, задача 279
 — ультраметрическое 540, 542, 544

- расстояние Хэмминга 487, 569, 576
 расшифровка см. сквенирование
 реальное время, сравнение на основе Z 53
 — — —, строк 51, 53
 регулярное выражение 93, 97, 413, 467
 — — —, поиск 93, 343
 редакционное предписание 270
 — — оптимальное 270
 — — расстояние см. расстояние редакционное
 редакционный граф 278, 305
 рестрикты, карта 179, 198, 491, 499
 ретровирус 450
 РНК выравнивание последовательности и структуры 311
 — информационная (иРНК) 296
 — матричная (мРНК) то же, что информационная
 — транспортная (тРНК), задача складывания 309
 ротация в строке 530
 — циклическая 522, 530
- Саморедукция** 590
- сантиморганица 480
 сборка гена 396
 — карты 490
 — последовательности 322, 452, 501, 508, 515
 — дробовая 163, 285, 534
 — фрагментная 604
 — экзона 583, 592
 связь вектор 144
 связь исхода 83
 — суффиксная 131
 сдвиг рамки считывания 36
 — — —, ошибки 579
 — циклический 65
 сквенирование 182, 501
 — ВАС-РАС 322
 — гибридизацией 513, 527
 — дробовое ДНК 508
 — направленное 502, 514
 семейство, представление 408, 409
 сетевое выравнивание 584
 сети, задача совпадения 580
 сеть, поток 535
 сжатие 524
 — данных 175
 — — по Зиву—Лемпелю 208
 — текста 208, 537
 сигнатура 410
 —, представление 412
 символ консенсусный 427
 — плохой см. правило плохого символа
 скаков по счетчику 134, 136
 скленическое выравнивание 583, 592
 склейка 307
 — альтернативная 312
 скрытое марковские модели 444
 словарь, задача 81, 120
 слово 23
 —, ориентированный ациклический граф 167
 смешение кодона 583
 смешение вектор 373
 Смита—Уотермана метод 291
 совместимость деревьев 560, 561
 совпадение ложное 110, 111
 — источное 99, 103, 119, 269, 331, 343
 — — с k различиями, задача 320, 323, 324, 326
 — приближенное 171, 263, 499
 — стей, задача 580
- совпадение точное 21, 31, 46, 78, 80, 88, 90, 91, 107,
 119, 122, 159, 165, 167, 171, 191, 213,
 216, 245, 249, 264, 331, 333, 343, 447,
 457, 461, 499, 510
 совпадения 103, 104, 106, 250
 —, статистика 171, 174, 216, 246, 336
 сравнение в реальном времени на основе Z 53
 — карт, задача 198
 — кДНК 295–297
 — множественное по Бойеру—Муру точное 207, 220,
 335
 — — — —, правило плохого символа 202
 — — — —, — хорошего суффикса 203
 — последовательностей множественное 453
 — строк, автомат 52
 — в реальном времени 51, 53
 — — множественное 403, 408
 статистика совпадений 174, 216, 246, 336
 — —, повторяющаяся подстрока 171
 строка 23, 24
 —, длина 346
 —, период 66
 —, ротация в исходном 530
 — консенсусная 427, 428
 — — штейнерова 426, 428
 — периодическая 66
 — полупериодическая 66
 — префиксно-полупериодическая 66
 — примитивная 223
 — центральная 423, 427
 — циклическая 33, 189, 259, 518
 строки 271
 —, автомат для сравнения 52
 —, множественное сравнение 403, 408
 —, сравнение в реальном времени 51, 53
 —, сходство 281
 — различных, повторяющаяся подстрока 217
 строковая глубина 122
 структура, ее получение 307
 — белка 307
 — — вторичная 438
 — повторяющаяся 177
 структурные выводы 414
 сублингвисное время, метод Майерса 338
 субоптимальное выравнивание 391, 395
 супермаксимальный повтор, повторяющаяся
 подстрока 184, 187, 218
 суперсемейство, представление 408, 409
 суффикс 23
 —, продолжение 129
 — собственный 23
 — хороший см. правило хорошего суффикса
 суффиксная связь 131
 суффиксо-пресфиксное перекрытие 535
 — — —, задача 174, 176, 217, 322, 509, 521
 суффиксное дерево см. дерево суффиксное
 суффиксный массив 191, 199, 340, 499
 схемы оценки 292
 — приближенные с полиномиальным временем 435
 сходство строк 281
 — — локальное 510
 сцепление, задача 398, 468, 583, 592
 — различных локальных выравниваний 395
 счет совпадений 103, 104, 106
 счетчик, скаков по нему 134, 136
- Тандемный массив** 34, 180, 222, 250
- — примитивный, повторяющаяся подстрока 222
 — повтор 177, 182, 222, 251, 252, 306

тандемный повтор с k различиями 252, 260
 тасование экзонов 220
 теорема о непрерывности SBH 530
 — о простых числах 110
 — об ОНД 521
 теория молекулярной эволюции нейтральная 549
 точносно множественно сравнивание по Бойеру—Муру
 см. сравнивание множественно по
 Бойеру—Муру точно
 — совпадение 21, 31, 46, 78, 80, 88, 90, 91, 107, 119,
 122, 159, 165, 167, 171, 191, 213, 216,
 245, 249, 264, 331, 333, 343, 447, 457,
 461, 499, 510

транслокация 594
 транспозиция 595
 тройник 577

Укконсна алгоритм см. алгоритм Укконсна
 ультраметрическая матрица 544, 573
 ультраметрическое дерево 542, 563, 565, 566
 — — минимальное 543
 — — расстояние 540, 542, 544
 упорядоченные STS 489
 условие четырех точек 577

Факторы транскрипции ДНК 90
 филогенетическое выравнивание 426, 430, 434, 541,
 555, 558, 569, 572
 — —, приближение с ограниченной ошибкой 431
 — дерево 465
 филогения совершенная 555, 558, 560, 562, 566, 568,
 575, 576
 фильтрация множественная 338
 Фитча—Хартигана алгоритм 578
 форма 443
 — выравнивания 285, 293
 фрагментная сборка 604
 функция блоковая 369
 — — ограничивающая 369
 — исходач 47, 81, 96
 — — в алгоритме Кнута—Морриса—Пратта 51
 — целевая консенсусная 426

Фурье-преобразование быстрое (FFT) 99, 103, 104
Химерный клон 484
 Хиршбера алгоритм 318
 хождение по хромосоме 502
 Хэмминга расстояние 487, 569, 576

Целевая функция консенсусная 426
 центральная звезда 423
 — —, метод 423, 429
 — строка 423, 427
 цепная реакция полимеразная 225
 циклическая корреляция 106
 — — ротация 522, 530
 — строка 33, 189, 259, 518
 циклическое покрытие 519, 527, 537
 цинковый палец 90

Чанга—Лаулера метод 336
 “четырех русских” алгоритм редакционного
 расстояния 374
 — — метод 369, 378
 четырех точек условие 577

Шотган см. дробовое секвенирование ДНК
 штейнерова консенсусная строка 426, 428
 штейнерово дерево 567, 568, 576
 штраф за пропуск 379

Эволюционное дерево 430, 440, 540, 543
 эволюция молекулярная, нейтральная теория 549
 эйтсеровский путь 528, 538
 экзоны 212, 220
 — , сборка 583, 592
 — , тасование 220
 экспрессируемая маркерная последовательность 89,
 175
 эпидемиология молекулярная 322
 эффективность по памяти наибольшее общее
 продолжение 247

Научное издание

ГАСФИЛД Дэн

**Строки, деревья и последовательности
в алгоритмах:
Информатика и вычислительная биология**

**Перевод с английского
*И. В. Романовского***

Редактор *О. М. Рощиненко*

**Издательство “Невский Диалект”
195220, Санкт-Петербург, Гражданский пр., 14.
Лицензия ЛР № 065012 от 18.02.1997.**

**Издательство “БХВ-Петербург”
198005, Санкт-Петербург, Измайловский пр., 29.
Лицензия ИД № 02429 от 24.07.2000.**

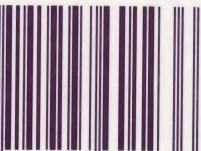
**Подписано в печать 29.01.2003. Формат 70×100 1/16.
Бумага газетная. Печать офсетная.
Гарнитура TimeRoman.
Усл. печ. л. 53,3.
Тираж 2000 экз. Заказ № 728**

**Отпечатано с готовых диапозитивов
в Академической типографии “Наука” РАН
199034, Санкт-Петербург, 9-я линия, 12.**

ИНТЕРНЕТ-МАГАЗИН
www.computerbook.ru



ISBN 5-7940-0103-8



9 795794 001036

БХВ-Петербург

198005, Санкт-Петербург,
Измайловский пр., 29

E-mail: mail@bhv.ru
Internet: www.bhv.ru

тел.: (812) 251-42-44
факс: (812) 251-12-95