

07.04.2025

Динамическое программирование. Жадные алгоритмы.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

N * Новосибирский
государственный
университет
***НАСТОЯЩАЯ НАУКА**

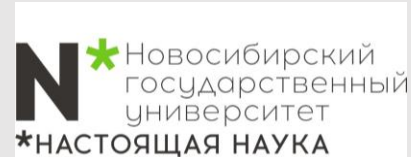


Давайте познакомимся



Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++



Адженда

**Жадные
алгоритмы**

45 минут

**Динамическое
программирова
ние**

45 минут

Адженда

**Жадные
алгоритмы**

45 минут

**Динамическое
программирова
ние**

45 минут

Введение

Дать точное определение жадного алгоритма достаточно трудно, если вообще возможно. Алгоритм называется жадным, если он строит решение за несколько мелких шагов, а решение на каждом шаге принимается без опережающего планирования, с расчетом на оптимизацию некоторого внутреннего критерия. Часто для одной задачи удастся разработать несколько жадных алгоритмов, каждый из которых локально и постепенно оптимизирует некоторую метрику на пути к решению. Когда жадный алгоритм успешно находит оптимальное решение нетривиальной задачи, этот факт обычно дает интересную и полезную информацию о структуре самой задачи; существует локальное правило принятия решений, которое может использоваться для построения оптимальных решений.



Какие алгоритмы из ранее рассмотренных можно отнести к жадным?

Примеры

- Кратчайшие пути в графе
- Задача нахождения минимального остовного
- Реализация алгоритма Крускала
- Коды Хаффмана и сжатие данных

Сегодня рассмотрим еще несколько...

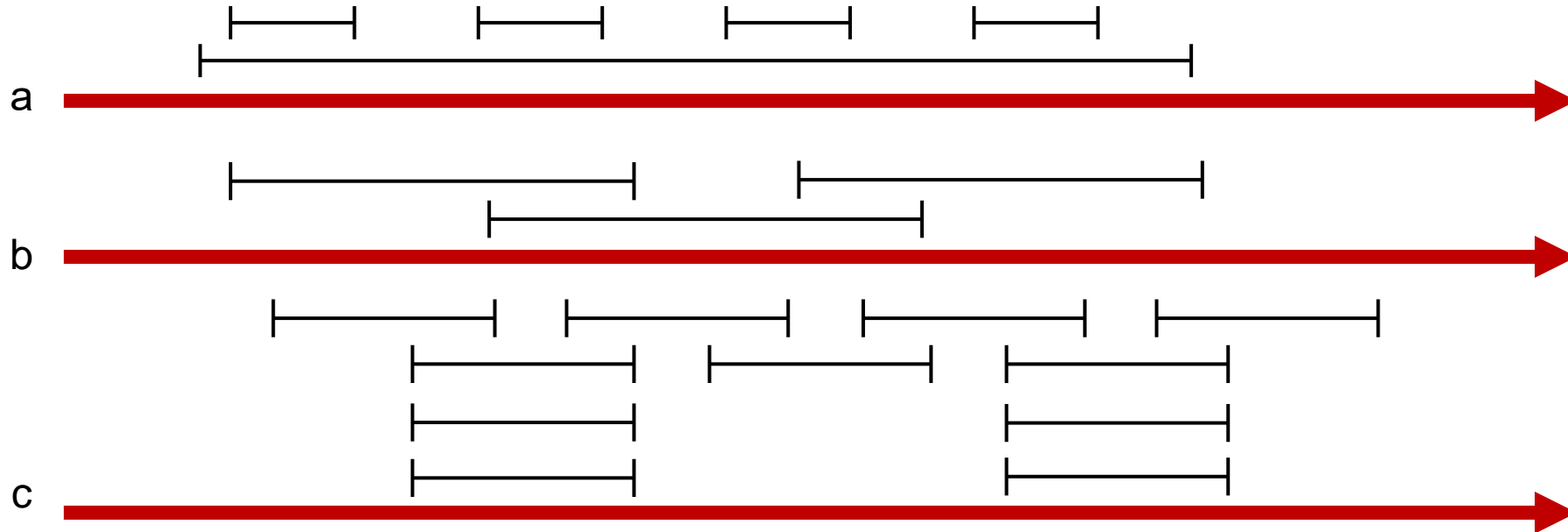


Интервальное планирование: жадный алгоритм опережает

Задача интервального планирования. Имеется множество заявок $\{1, 2, \dots, n\}$; i -я заявка соответствует интервалу времени, который начинается в $s(i)$ и завершается в $f(i)$. Подмножество заявок называется совместимым, если никакие две заявки не перекрываются по времени; наша цель — получить совместимое подмножество как можно большего размера. Совместимые множества максимального размера называются оптимальными.



Проектирование жадного алгоритма



Примеры ситуаций интервального планирования, в которых естественные жадные алгоритмы не приводят к оптимальному решению. В случае a не работает выбор интервала с самым ранним начальным временем; в случае b не работает выбор самого короткого интервала; в случае c не работает выбор интервала с минимальным количеством конфликтов.

Проектирование жадного алгоритма

Определим алгоритм более формально. Пусть R — множество заявок, не принятых и не отклоненных на данный момент, а A — множество принятых заявок.

Инициализировать R множеством всех заявок, A — пустым множеством
 Пока множество R не пусто
 Выбрать заявку $i \in R$ с наименьшим конечным временем
 Добавить заявку i в A
 Удалить из R все заявки, несовместимые с запросом i
 Конец цикла
 Вернуть A как множество принятых заявок

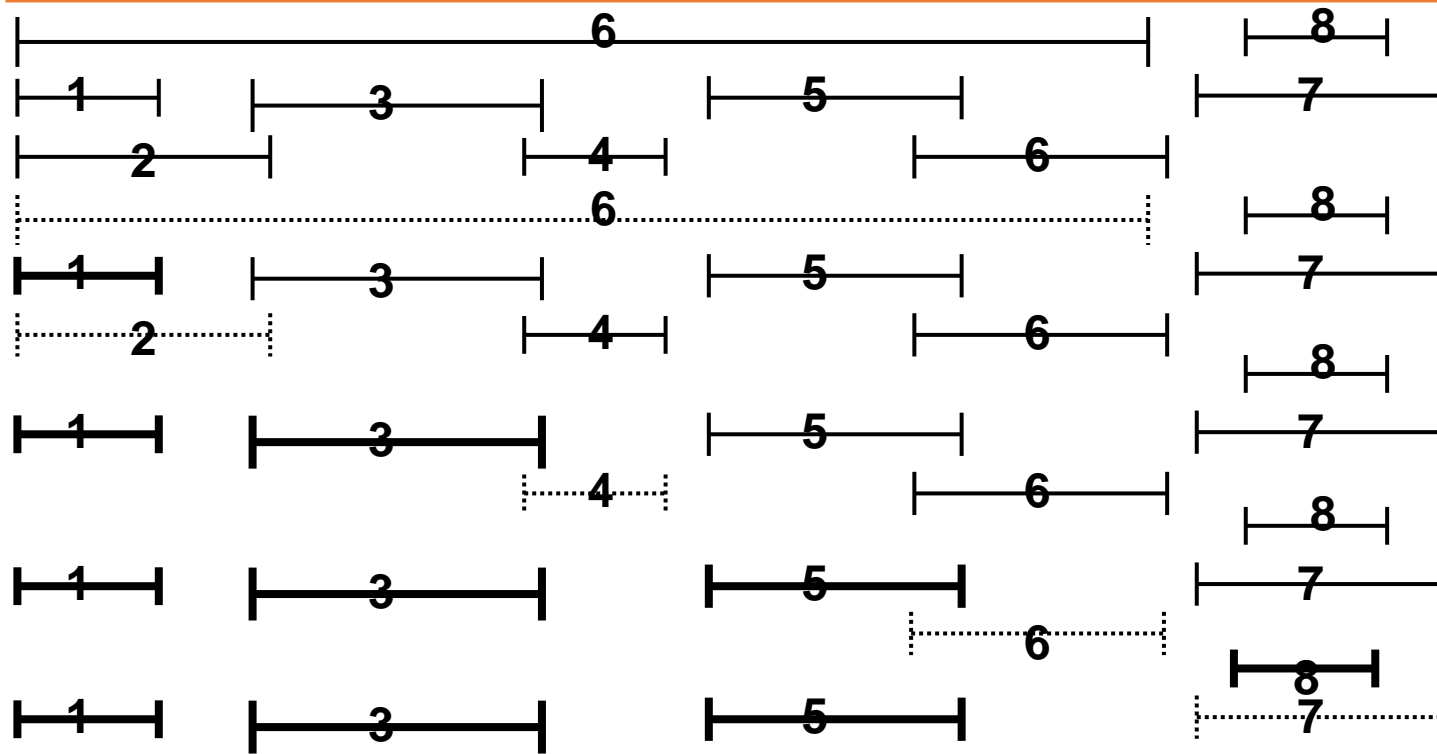
Интервалы, пронумерованные по порядку

Выбор интервала 1

Выбор интервала 3

Выбор интервала 5

Выбор интервала 8



Анализ алгоритма

Утверждение 1.1 Множество A состоит из совместимых заявок.

Продemonстрируем, что это решение оптимально. Итак, пусть O — оптимальное множество интервалов. В идеале хотелось бы показать, что $A = O$, но это уже слишком: оптимальных решений может быть несколько, и в лучшем случае A совпадает с одним из них. Итак, вместо этого мы просто покажем, что $|A| = |O|$, то есть A содержит столько же интервалов, сколько и O , а следовательно, является оптимальным решением.

Нужен критерий, по которому наш жадный алгоритм опережает решение O . Мы будем сравнивать частичные решения, создаваемые жадным алгоритмом, с начальными сегментами решения O и шаг за шагом покажем, что жадный алгоритм работает не хуже.

Пусть i_1, \dots, i_k — множество заявок A в порядке их добавления в A . Заметьте, что $|A| = k$. Аналогичным образом множество заявок O будет обозначаться j_1, \dots, j_m . Мы намерены доказать, что $k = m$. Допустим, заявки в O также упорядочены слева направо по соответствующим интервалам, то есть по начальным и конечным точкам. Не забывайте, что заявки в O совместимы, то есть начальные точки следуют в том же порядке, что и конечные.

Может ли r -ий интервал жадного алгоритма кончаться позднее?



Анализ алгоритма

Выбор жадного метода происходит от желания освободить ресурс как можно раньше после удовлетворения первой заявки.

Утверждение 1.2 Для всех индексов $r \leq k$ выполняется условие $f(i_r) \leq f(j_r)$.

Доказательство. Утверждение будет доказано методом индукции. Для $r = 1$ оно очевидным образом истинно: алгоритм начинается с выбора заявки i_1 с наименьшим временем завершения.

Теперь рассмотрим $r > 1$. Согласно гипотезе индукции, будем считать, что утверждение истинно для $r - 1$, и попробуем доказать его для r . Индукционная гипотеза позволяет считать, что $f(i_{r-1}) \leq f(j_{r-1})$. Если r -й интервал алгоритма не завершается раньше, он должен «отставать». Но это невозможно: вместо того, чтобы выбирать интервал с более поздним завершением, жадный алгоритм всегда имеет возможность выбрать j_r и таким образом реализовать шаг индукции.

Утверждение 1.3 Жадный алгоритм возвращает оптимальное множество A .

Доказательство. Утверждение будет доказано от обратного. Если множество A не оптимально, то оптимальное множество O должно содержать больше заявок, то есть $m > k$. Применяя (1.2) для $r = k$, получаем, что $f(i_k) \leq f(j_k)$. Так как $m > k$, в O должна существовать заявка j_{k+1} . Она начинается после завершения заявки j_k , а следовательно, после завершения i_k . Получается, что после удаления всех заявок, несовместимых с заявками i_1, \dots, i_k , множество возможных заявок R по-прежнему будет содержать j_{k+1} . Но тогда жадный алгоритм останавливается на заявке i_k , а должен останавливаться только на пустом множестве R , — противоречие. ■



Реализация и время выполнения

Алгоритм может выполняться за время $O(n \log n)$. Сначала n заявок следует отсортировать в порядке конечного времени и пометить их в этом порядке; соответственно предполагается, что $f(i) \leq f(j)$, когда $i < j$. Этот шаг выполняется за время $O(n \log n)$. За дополнительное время $O(n)$ строится массив $S[1 \dots n]$, в котором элемент $S[i]$ содержит значение $s(i)$.

Теперь выбор заявок осуществляется обработкой интервалов в порядке возрастания $f(i)$. Сначала всегда выбирается первый интервал; затем интервалы перебираются по порядку до тех пор, пока не будет достигнут первый интервал j , для которого $s(j) \geq f(1)$; он также включается в результат. В более общей формулировке, если последний выбранный интервал заканчивается во время f , перебор последующих интервалов продолжается до достижения первого интервала j , для которого $s(j) \geq f$. Таким образом, описанный выше жадный алгоритм реализуется за один проход по интервалам с постоянными затратами времени на интервал. Следовательно, эта часть алгоритма выполняется за время $O(n)$.

Расширения

Задача интервального планирования, рассмотренная нами, относится к числу относительно простых задач планирования. На практике возникает много дополнительных сложностей. Ниже перечислены проблемы.

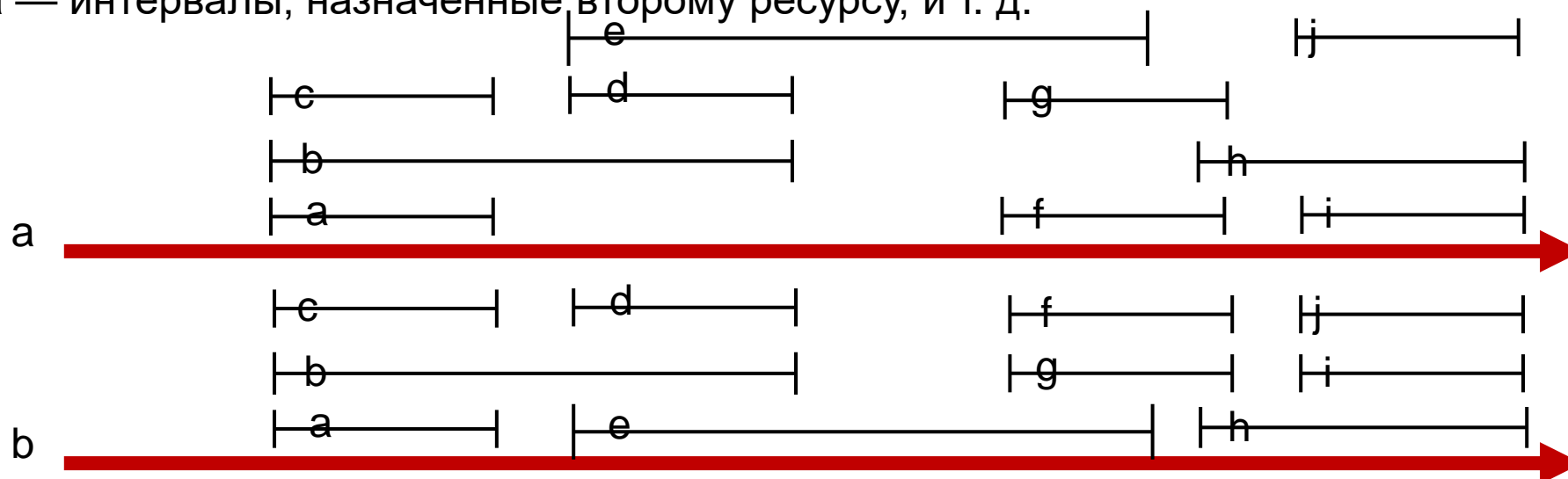
- В определении задачи предполагалось, что все заявки были известны алгоритму планирования при выборе совместимого подмножества. Конечно, также было бы естественно рассмотреть версию задачи, в которой планировщик должен принимать решения по принятию или отклонению заявок до того, как ему будет известен полный набор заявок. Если планировщик будет слишком долго собирать информацию обо всех заявках, клиенты (отправители заявок) могут потерять терпение, отказаться и уйти. Сейчас в области таких оперативных алгоритмов, которые должны принимать решения «на ходу», без полной информации о будущих заявках, ведутся активные исследования.
- Наша постановка задачи стремилась к максимизации удовлетворенных заявок. Однако также можно представить ситуацию, в которой заявки обладают разной ценностью. Например, каждой заявке i может быть присвоен вес v_i (доход от удовлетворения заявки i), и целью становится обеспечение максимального дохода: суммы весов всех удовлетворенных заявок. Так мы подходим к задаче взвешенного интервального планирования — второй из типичных задач.

У задачи интервального планирования существует много других вариаций и разновидностей.

Планирование всех интервалов

В задаче интервального планирования используется один ресурс и много заявок в форме временных интервалов; требуется выбрать, какие заявки следует принять, а какие отклонить. Во взаимосвязанной задаче используется несколько идентичных ресурсов, для которых необходимо распланировать все заявки с использованием минимально возможного количества ресурсов.

Для наглядности рассмотрим пример, а. Все заявки из этого примера могут быть распределены по трем ресурсам, b: заявки размещаются в три строки, каждая из которых содержит набор неперекрывающихся интервалов. В общем случае решение с k ресурсами можно представить как задачу распределения заявок в k строк с неперекрывающимися интервалами; первая строка содержит все интервалы, назначенные первому ресурсу, вторая строка — интервалы, назначенные второму ресурсу, и т. д.



Планирование всех интервалов

Возможно ли обойтись только двумя ресурсами в этом конкретном примере? Разумеется, нет. Необходимо как минимум три ресурса, потому что, например, интервалы a , b и c занимают общую точку временной шкалы, а следовательно, должны быть распределены по разным ресурсам.

Утверждение 1.4. В любой ситуации интервального разбиения количество необходимых ресурсов не меньше глубины множества интервалов.

Доказательство. Допустим, множество интервалов имеет глубину d , а интервалы I_1, \dots, I_d проходят через одну общую точку временной шкалы. Все эти интервалы должны быть распределены по разным ресурсам, поэтому решению в целом необходимы как минимум d ресурсов.

А теперь рассмотрим два вопроса, которые, как выясняется, тесно связаны друг с другом. Во-первых, можно ли спроектировать эффективный алгоритм, который планирует все интервалы с минимально возможным количеством ресурсов? Во-вторых, всегда ли существует расписание с количеством ресурсов, равным глубине? Положительный ответ на второй вопрос будет означать, что препятствия по разбиению интервалов имеют чисто локальную природу — они сводятся к набору интервалов, занимающих одну точку. На первый взгляд непонятно, могут ли существовать другие, «отложенные» препятствия, которые дополнительно увеличивают количество необходимых ресурсов.

Планирование всех интервалов

Проектирование алгоритма

Пусть d — глубина множества интервалов; каждому интервалу будет назначена метка из множества чисел $\{1, 2, \dots, d\}$ так, чтобы перекрывающиеся интервалы помечались разными числами. Так мы получим нужное решение, поскольку каждое число может интерпретироваться как имя ресурса, а метка каждого интервала — как имя ресурса, которому он будет назначен.

Используемый для этой цели алгоритм представляет собой однократную жадную стратегию упорядочения интервалов по начальному времени:

Пусть I_1, I_2, \dots, I_n — обозначения интервалов в указанном порядке

Для $j = 1, 2, 3, \dots, n$

Для каждого интервала I_i , который предшествует I_j в порядке сортировки и перекрывает его
Исключить метку I_i из рассмотрения для I_j

Конец цикла

Если существует метка из множества $\{1, 2, \dots, d\}$, которая еще не исключена

Присвоить неисключенную метку I_j

Иначе

Оставить I_j без метки

Конец Условия

Конец цикла

Анализ алгоритма

Утверждение 1.5. При использовании описанного выше жадного алгоритма каждому интервалу будет назначена метка, и никаким двум перекрывающимся интервалам не будет присвоена одна и та же метка.

Доказательство. Начнем с доказательства того, что ни один интервал не останется не помеченным. Рассмотрим один из интервалов I_j и предположим, что в порядке сортировки существует t интервалов, которые начинаются ранее и перекрывают его. Эти t интервалов в сочетании с I_j образуют множество из $t + 1$ интервалов, которые все проходят через общую точку временной шкалы (а именно начальное время I_j), поэтому $t + 1 \leq d$. Следовательно, $t \leq d - 1$. Из этого следует, что по крайней мере одна из меток d не будет исключена из этого множества интервалов t , поэтому существует метка, которая может быть назначена I_j .

Далее утверждается, что никаким двум перекрывающимся интервалам не будут назначены одинаковые метки. В самом деле, возьмем два перекрывающихся интервала I и I' и предположим, что I предшествует I' в порядке сортировки. Затем, при рассмотрении алгоритмом I' , интервал I принадлежит множеству интервалов, метки которых исключаются из рассмотрения; соответственно, алгоритм не назначит I' метку, которая использовалась для I . ■

Утверждение 1.6. Описанный выше жадный алгоритм связывает каждый интервал с ресурсом, используя количество ресурсов, равное глубине множества интервалов. Это количество ресурсов является минимально необходимым, то есть оптимальным.

Планирование для минимизации задержки: метод замены

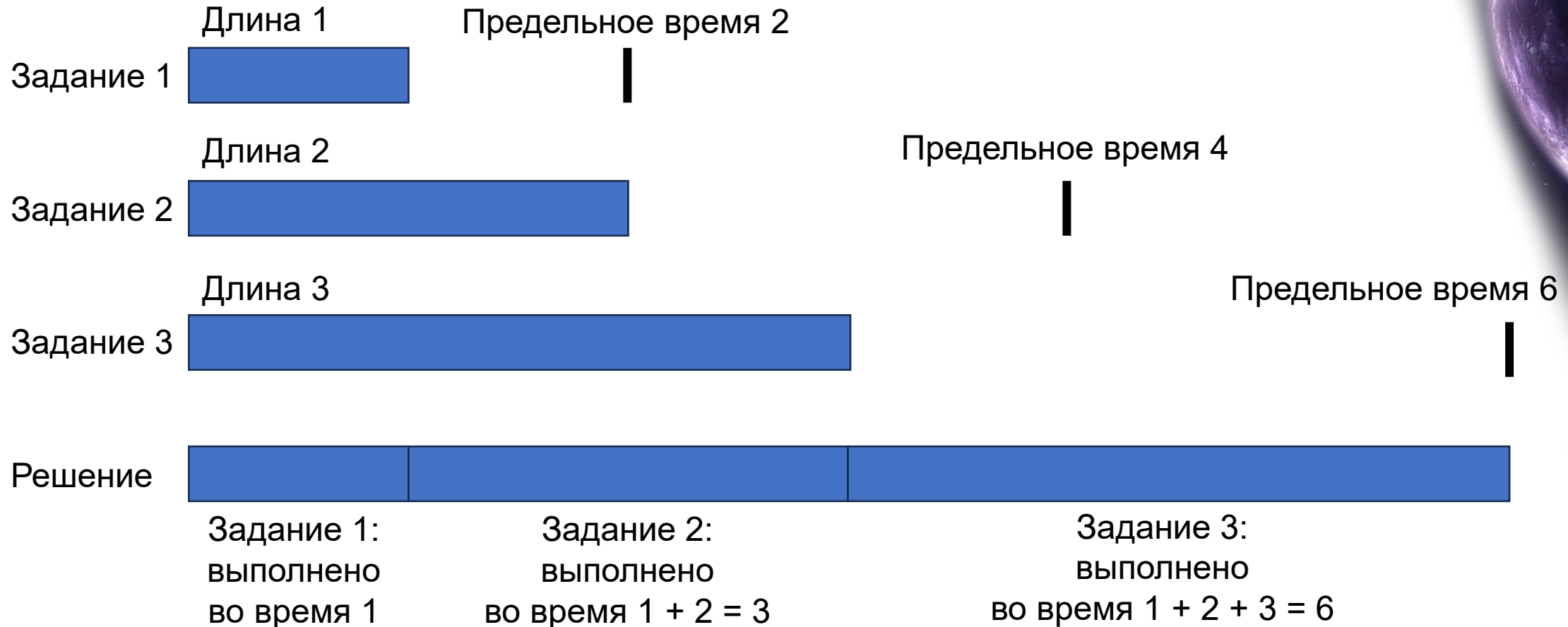
Задача

Вернемся к ситуации с одним ресурсом и набором из n заявок на его использование в течение интервала времени. Будем считать, что ресурс доступен, начиная с времени s . Однако в отличие от предыдущей задачи заявки становятся более гибкими — вместо начального и конечного времени заявка содержит предельное время d_i и непрерывный интервал времени длиной t_i , а начинаться она может в любое время до своего предельного времени. Каждой принятой заявке должен быть выделен интервал времени длиной t_i , и разным заявкам должны назначаться неперекрывающиеся интервалы.

Допустим, мы намерены удовлетворить каждую заявку, но некоторые заявки могут быть отложены на более позднее время. Таким образом, начиная с общего начального времени s , каждой заявке i выделяется интервал времени t_i ; обозначим этот интервал $[s(i), f(i)]$, где $f(i) = s(i) + t_i$. Однако в отличие от предыдущей задачи этот алгоритм должен определить начальное время (а следовательно, и конечное время) для каждого интервала.

Заявка i будет называться задержанной, если она не успевает завершиться к предельному времени, то есть если $f(i) > d_i$. Задержка такой заявки i определяется по формуле $l_i = f(i) - d_i$. Будем считать, что если заявка не является просроченной, то $l_i = 0$. Целью новой задачи оптимизации будет планирование всех заявок с неперекрывающимися интервалами, обеспечивающее минимизацию максимальной задержки $L = \max_i l_i$.

Планирование для минимизации задержки: метод замены



Пример планирования с минимизацией задержки

Проектирование алгоритма

Как же должен выглядеть жадный алгоритм для такой задачи? Есть несколько естественных жадных решений, в которых мы просматриваем данные заданий (t_i, d_i) и используем эту информацию для упорядочения заданий по некоторому простому правилу.

- Одно из решений заключается в планировании заданий по возрастанию длины t_i для того, чтобы поскорее избавиться от коротких заданий. Впрочем, примитивность такого решения сразу же становится очевидной, поскольку оно полностью игнорирует предельное время выполнения заданий. В самом деле, рассмотрим пример с двумя заданиями: у первого $t_1 = 1$ и $d_1 = 10$, а у второго $t_2 = 10$ и $d_2 = 10$. Для достижения задержки $L = 0$ второе задание должно начаться немедленно.
- Предыдущий пример предполагает, что нам следует ориентироваться на задания с очень малым запасом времени $d_i - t_i$, то есть задания, которые должны запускаться с минимальной задержкой. Таким образом, более естественный жадный алгоритм должен сортировать задания в порядке возрастания запаса времени $d_i - t_i$.

К сожалению, это жадное правило тоже не работает. Рассмотрим пример из двух заданий: у первого задания $t_1 = 1$ и $d_1 = 2$, а у второго $t_2 = 10$ и $d_2 = 10$. Сортировка по возрастанию запаса времени поместит второе задание на первое место в расписании, и первое задание создаст задержку 9.



Проектирование алгоритма

Тем не менее существует столь же простой жадный алгоритм, который всегда приводит к оптимальному решению. Задания просто сортируются в порядке возрастания предельного времени d_i и планируются в этом порядке. Это правило основано на интуитивно понятном принципе: задания с более ранним предельным временем завершаются в первую очередь.

Переименовывая задания в случае необходимости, можно принять, что задания помечены в порядке следования их предельного времени, то есть выполняется условие

$$d_1 \leq \dots \leq d_n.$$

Все задания просто планируются в этом порядке. И снова пусть s является начальным временем для всех заданий. Задание 1 начинается во время $s = s(1)$ и заканчивается во время $f(1) = s(1) + t_1$; задание 2 начинается во время $s(2) = f(1)$ и завершается во время $f(2) = s(2) + t_2$ и т. д. Время завершения последнего спланированного задания будет обозначаться f . Ниже приведена формулировка алгоритма на псевдокоде.

Упорядочить задания по предельному времени

Для простоты считается, что $d_1 \leq \dots \leq d_n$

В исходном состоянии $f = s$

Рассмотреть задания $i = 1, \dots, n$ в таком порядке

Назначить задание i временному интервалу от $s(i) = f$ до $f(i) = f + t_i$ Присвоить $f = f + t_i$

Конец

Вернуть множество спланированных интервалов $[s(i), f(i)]$ для $i = 1, \dots, n$

Анализ алгоритма

Утверждение 1.7. Существует оптимальное расписание без времени простоя.

Начнем с рассмотрения оптимального расписания O . Нашей целью будет постепенная модификация O , которая будет сохранять оптимальность на каждом шаге, но в конечном итоге преобразует его в расписание, идентичное расписанию A , найденному жадным алгоритмом. Для начала попробуем охарактеризовать полученные расписания. Будем говорить, что расписание A' содержит инверсию, если задание i с предельным временем d_i предшествует другому заданию j с более ранним предельным временем $d_j < d_i$. Если существует несколько заданий с одинаковыми значениями предельного времени, то это означает, что может существовать несколько разных расписаний без инверсий. Тем не менее можно показать, что все эти расписания обладают одинаковой максимальной задержкой L .

Утверждение 1.8. Все расписания, не содержащие инверсий и простоев, обладают одинаковой максимальной задержкой.

Доказательство. Если два разных расписания не содержат ни инверсий, ни простоев, то задания в них могут следовать в разном порядке, но при этом различаться будет только порядок заданий с одинаковым предельным временем. Рассмотрим такое предельное время d . В обоих расписаниях задания с предельным временем d планируются последовательно (после всех заданий с более ранним предельным временем и до всех заданий с более поздним предельным временем). Среди всех заданий с предельным временем d последнее обладает наибольшей задержкой, причем эта задержка не зависит от порядка заданий. ■

Анализ алгоритма

Утверждение 1.9. Существует оптимальное расписание, не имеющее инверсий и времени простоя.

Доказательство. Согласно (1.7), существует оптимизация расписание O без времени простоя. Доказательство будет состоять из серии утверждений, первое из которых доказывается легко.

(a) Если O содержит инверсию, то существует такая пара заданий i и j , для которых j следует в расписании немедленно после i и $d_j < d_i$. Рассмотрим инверсию, в которой задание a стоит в расписании где-то до задания b и $d_a > d_b$. При перемещении в порядке планирования заданий от a к b где-то встретится точка, в которой предельное время уменьшается в первый раз. Она соответствует паре последовательных заданий, образующих инверсию. Теперь допустим, что O содержит как минимум одну инверсию, и в соответствии с (a) пусть i и j образуют пару инвертированных запросов, занимающих последовательные позиции в расписании. Меняя местами заявки i и j в расписании O , мы уменьшим количество инверсий в O на 1. Пара (i, j) создавала инверсию в O , перестановка эту инверсию устраняет, и новые инверсии при этом не создаются. Таким образом,

(b) После перестановки i и j образуется расписание, содержащее на 1 инверсию меньше. В этом доказательстве труднее всего обосновать тот факт, что расписание после устранения инверсии также является оптимальным.

Анализ алгоритма

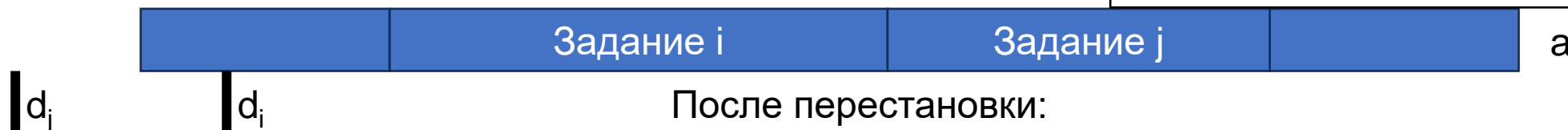
(с) Максимальная задержка в новом расписании, полученном в результате перестановки, не превышает максимальной задержки O . Очевидно, что если удастся доказать (с), то задачу можно считать выполненной.

Исходное расписание O может иметь не более $\binom{n}{2}$ инверсий (если инвертированы все пары), а следовательно, после максимум $\binom{n}{2}$ перестановок мы получим оптимальное расписание без инверсий. ■

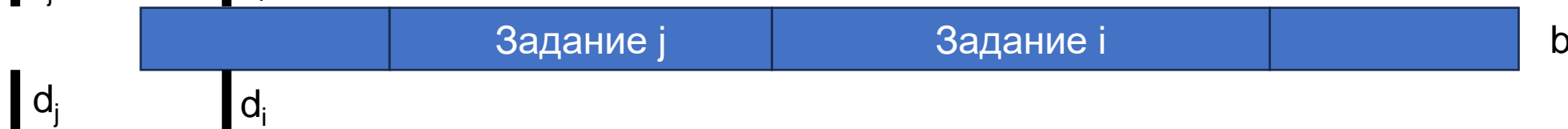
Итак, докажем (с) и продемонстрируем, что перестановка пары последовательных инвертированных заданий не приведет к возрастанию максимальной задержки L расписания. Доказательство (с). Введем вспомогательное обозначение для описания расписания O : предположим, каждая заявка r планируется на интервал времени $[s(r), f(r)]$ и обладает задержкой l'_r . Пусть $L' = \max_r l'_r$ обозначает максимальную задержку этого расписания. Обозначим расписание с перестановкой \bar{O} ; обозначения $\bar{s}(r), \bar{f}(r), \bar{l}_r$ и \bar{L} будут представлять соответствующие характеристики расписания с перестановкой.

Перестановка влияет только на время завершения i и j

До перестановки:



После перестановки:



Анализ алгоритма

Теперь вернемся к двум смежным инвертированным заданиям i и j . Время завершения j до перестановки в точности равно времени завершения i после перестановки. Таким образом, все задания, кроме i и j , в двух расписаниях завершаются одновременно. Кроме того, задание j в новом расписании завершится раньше, а следовательно, перестановка не увеличивает задержку задания j .

Следовательно, остается беспокоиться только о задании i : его задержка могла увеличиться. После перестановки задание i завершается во время $f(j)$, в которое завершалось задание j в расписании O . Если задание i «опаздывает» в новом расписании, его задержка составляет $\bar{l}_i = \bar{f}(i) - d_i = f(j) - d_i$. Но здесь принципиально то, что i не может задерживаться в расписании \bar{O} больше, чем задерживалось задание j в расписании O . А конкретнее, из нашего предположения $d_i > d_j$ следует, что

$$\bar{l}_i = \bar{f}(i) - d_i < f(j) - d_j = l'_j.$$

Так как задержка расписания O была равна $L \geq l'_j \geq \bar{l}_i$, из этого следует, что перестановка не увеличивает максимальную задержку расписания. ■

Утверждение 1.10. Расписание A , построенное жадным алгоритмом, обеспечивает оптимальную максимальную задержку L .

Доказательство. Пункт (4.9) доказывает, что оптимальное расписание без инверсий существует. С другой стороны, согласно (4.8), все расписания без инверсий имеют одинаковую максимальную задержку, так что расписание, построенное жадным алгоритмом, является оптимальным. ■

Оптимальное кэширование: более сложный пример замены

Существует небольшой объем данных, к которым можно обращаться очень быстро, и большой объем данных, для обращения к которым требуется больше времени; вы должны решить, какие данные следует держать «под рукой».

Общим термином **«кэширование»** называется процесс хранения малых объемов данных в быстрой памяти, чтобы уменьшить время, которое тратится на взаимодействия с медленной памятью.

Задача

Рассматривается множество U из n фрагментов данных, хранящихся в основной памяти. Также существует более быстрая память — кэш, способная в любой момент времени хранить $k < n$ фрагментов данных. Будем считать, что кэш изначально содержит множество из k элементов. Мы получаем последовательность элементов данных $D = d_1, d_2, \dots, d_m$ из U — это последовательность обращений к памяти, которую требуется обработать; при обработке следует постоянно принимать решение о том, какие k элементов должны храниться в кэше. Запрашиваемый элемент d_i очень быстро читается, если он уже находится в кэше; в противном случае его приходится копировать из основной памяти в кэш и, если кэш полон, вытеснять из него какой-то фрагмент данных, чтобы освободить место для d_i . Такая ситуация называется кэш-промахом; естественно, мы стремимся к тому, чтобы количество промахов было сведено к минимуму.



Оптимальное кэширование: более сложный пример замены

Итак, для конкретной последовательности обращений к памяти алгоритм управления кэшем определяет план вытеснения (какие элементы в каких точках последовательности должны вытесняться из кэша), а последний определяет содержимое кэша и частоту промахов. Рассмотрим пример этого процесса.

Допустим, имеются три элемента $\{a, b, c\}$, размер кэша $k = 2$ и последовательность a, b, c, b, c, a, b .

В исходном состоянии кэш содержит элементы a и b . Для третьего элемента последовательности можно вытеснить a , чтобы включить c в кэш; на шестом элементе можно вытеснить c , чтобы вернуть a ; таким образом, вся последовательность включает два промаха. Если поразмыслить над этой последовательностью, становится понятно, что любой план вытеснения для этой последовательности должен включать как минимум два промаха. В условиях реальных вычислений алгоритм управления кэшем должен обрабатывать обращения к памяти d_1, d_2, \dots , не располагая информацией о том, какие обращения встретятся в будущем; но для оценки качества алгоритма системные аналитики с первых дней старались понять природу оптимального решения задачи кэширования. Если известна полная последовательность S обращений к памяти, какой план вытеснения обеспечит минимальное количество кэш-промахов?



Разработка и анализ алгоритма

В 1960-х годах Лес Беладди показал, что следующее простое правило всегда приводит к минимальному количеству промахов:

Когда элемент d_i должен быть внесен в кэш, вытеснить элемент, который будет использоваться позднее всех остальных - «алгоритм отдаленного использования».

рассмотрим последовательность вида $a, b, c, d, a, d, e, a, d, b, c$

Все алгоритмы управления кэшем, упоминавшиеся выше, строят планы, которые включают элемент d в кэш на шаге i , если данные d запрашиваются на шаге i и d еще не находится в кэше. Назовем такой план сокращенным — он выполняет минимальный объем работы, необходимый для заданного шага.

Пусть S — план, который может не являться сокращенным. Новый план \bar{S} — сокращение S — определяется следующим образом: на каждом шаге i , на котором S включает в кэш незапрошенный элемент d , наш алгоритм построения \bar{S} «притворяется», что он это делает, но в действительности оставляет d в основной памяти. Фактически d попадает в кэш на следующем шаге j после того, как элемент d был запрошен. В этом случае кэш-промах, инициированный \bar{S} на шаге j , может быть отнесен на счет более ранней операции с кэшем, выполненной S на шаге i .

Утверждение 1.11. Сокращенный план \bar{S} заносит в кэш не больше элементов, чем план S . Заметьте, что для каждого сокращенного плана количество элементов, включаемых в кэш, точно совпадает с количеством промахов.



Доказательство оптимальности алгоритма отдаленного использования

Воспользуемся методом замены для доказательства оптимальности алгоритма отдаленного использования. Рассмотрим произвольную последовательность обращений к памяти D ; пусть S_{FF} обозначает план, построенный алгоритмом отдаленного использования, а S^* — план с минимально возможным количеством промахов.

Утверждение 1.12. Пусть S — сокращенный план, который принимает те же решения по вытеснению, что и S_{FF} , в первых j элементах последовательности для некоторого j . Тогда существует сокращенный план S' , который принимает те же решения, что и S_{FF} , в первых $j + 1$ элементах и создает не больше промахов, чем S .

Доказательство. Рассмотрим $(j + 1)$ -е обращение к элементу $d = d_{j+1}$. Так как S и S_{FF} до этого момента были согласованы, содержимое кэша в них не различается. Если d находится в кэше в обоих планах, то решения по вытеснению не требуются, так что S согласуется с S_{FF} на шаге $j + 1$, и, следовательно, $S' = S$. Аналогичным образом, если элемент d должен быть включен в кэш, но S и S_{FF} вытеснят один и тот же элемент, чтобы освободить место для d , и снова $S' = S$.

Итак, интересная ситуация возникает тогда, когда d необходимо добавить в кэш, причем для этого S вытесняет элемент f , а S_{FF} вытесняет элемент $e \neq f$. Здесь S и S_{FF} уже не согласуются к шагу $j + 1$, потому что у S в кэше находится e , а у S_{FF} в кэше находится f . А это означает, что для построения S' необходимо сделать что-то нетривиальное.

Доказательство оптимальности алгоритма отдаленного использования

Сначала нужно сделать так, чтобы план S' вытеснял e вместо f . Затем нужно убедиться в том, что количество промахов у S' не больше S . Простым решением было бы согласование S' с S в оставшейся части последовательности; однако это невозможно, так как S и S' , начиная с этого момента, имеют разное содержимое кэша. Поэтому мы должны постараться привести кэш S' в такое же состояние, как у S , не создавая ненужных промахов. Когда содержимое кэша будет совпадать, можно будет завершить построение S' , просто повторяя поведение S . А если говорить конкретнее, от запроса $j + 2$ и далее S' ведет себя в точности как S , пока впервые не будет выполнено одно из следующих условий:

- (i) Происходит обращение к элементу $g \neq e, f$, который не находится в кэше S , и S вытесняет e , чтобы освободить для него место. Так как S' и S различаются только в e и f , это означает, что g также не находится в кэше S' ; тогда из S' вытесняется f и кэши S и S' совпадают. Таким образом, в оставшейся части последовательности S' ведет себя точно так же, как S .
- (ii) Происходит обращение к f , и S вытесняет элемент e' . Если $e' = e$, все просто: S' просто обращается к f из кэша, и после этого шага кэши S и S' совпадают. Если $e' \neq e$, то S' также вытесняет e' и добавляет e из основной памяти; в результате S и S' тоже имеют одинаковое содержимое кэша. Однако здесь необходима осторожность, так как S' уже не является сокращенным планом: элемент e переносится в кэш до того, как в нем возникнет прямая необходимость. Итак, чтобы завершить эту часть построения, мы преобразуем S' в сокращенную форму \bar{S}' , используя (1.11); количество элементов, включаемых S' в этом не увеличивается, и S' по-прежнему согласуется с S_{FF} на шаге $j + 1$.

Доказательство оптимальности алгоритма отдаленного использования

Итак, в обоих случаях мы получаем новый сокращенный план S' , который согласуется с S_{FF} F на первых $j + 1$ элементах и обеспечивает не большее количество промахов, чем S . И здесь принципиально (в соответствии с определяющим свойством алгоритма отдаленного использования) то, что один из этих двух случаев возникнет до обращения к e . Дело в том, что на шаге $j + 1$ алгоритм отдаленного использования вытеснил элемент (e), который понадобится в самом отдаленном будущем; следовательно, обращению к e должно предшествовать обращение к f , и тогда возникнет ситуация (ii). ■

Этот результат позволяет легко завершить доказательство оптимальности. Мы начнем с оптимального плана S^* и воспользуемся (1.12) для построения плана S_1 , который согласуется с S_{FF} на первом шаге. Затем мы продолжаем применять (1.12) индуктивно для $j = 1, 2, 3, \dots, m$, строя планы S_j , согласующиеся с S_{FF} на первых j шагах. Каждый план не увеличивает количество промахов по сравнению с предыдущим, а по определению $S_m = S_{FF}$, поскольку планы согласуются на всей последовательности.

Таким образом, получаем: **утверждение 1.13** S_{FF} порождает не больше промахов, чем любой другой план S^* , а следовательно, является оптимальным.

Расширения: кэширование в реальных рабочих условиях

Как упоминалось в предыдущем подразделе, оптимальный алгоритм Беладии предоставляет контрольную точку для оценки эффективности кэширования; однако на практике решения по вытеснению должны приниматься «на ходу», без информации о будущих обращениях.

Эксперименты показали, что лучшие алгоритмы кэширования в описанной ситуации представляют собой вариации на тему принципа LRU (Least-RecentlyUsed), по которому из кэша вытесняется элемент с наибольшим временем, прошедшим с момента последнего обращения.

Если задуматься, речь идет о том же алгоритме Беладии с измененным направлением времени, — только наибольший продолжительный промежуток времени относится к прошлому, а не к будущему. Он эффективно работает, потому что для приложений обычно характерна локальность обращений: выполняемая программа, как правило, продолжает обращаться к тем данным, к которым она уже обращалась ранее. (Легко придумать аномальные исключения из этого принципа, но они относительно редко встречаются на практике.) По этой причине в кэше обычно хранятся данные, которые использовались недавно.

Спустя долгое время после практического принятия алгоритма LRU Слитор и Тарьян показали, что для эффективности LRU можно предоставить теоретический анализ, ограничивающий количество промахов относительно алгоритма отдаленного использования.

Кластеризация

Интересный пример использования минимальных остовных деревьев встречается в области кластеризации.

Задача Под задачей кластеризации обычно понимается ситуация, в которой некую коллекцию объектов (допустим, набор фотографий, документов, микроорганизмов и т. д.) требуется разделить на несколько логически связанных групп. В такой ситуации естественно начать с определения метрик сходства или расхождения каждой пары объектов. Одно из типичных решений основано на определении функции расстояния между объектами; предполагается, что объекты, находящиеся на большем расстоянии друг от друга, в меньшей степени похожи друг на друга. Для точек в реальном мире расстояние может быть связано с географическим расстоянием, но во многих случаях ему приписывается более абстрактный смысл. Например, расстояние между двумя биологическими видами может измеряться временным интервалом между их появлением в ходе эволюции; расстояние между двумя изображениями в видеопотоке может измеряться количеством пикселей, в которых интенсивность цвета превышает некоторый порог.

Для заданной функции расстояния между объектами процесс кластеризации пытается разбить их на группы так, чтобы объекты одной группы интуитивно воспринимались как «близкие», а объекты разных групп — как «далекие». Этот несколько туманный набор целей становится отправной точкой для множества технически различающихся методов, каждый из которых пытается формализовать общее представление о том, как должен выглядеть хороший набор групп.



Кластеризация по максимальному интервалу

Минимальные остовные деревья играют важную роль в одной из базовых формализаций, которая будет описана ниже. Допустим, имеется множество U из n объектов p_1, p_2, \dots, p_n . Для каждой пары p_i и p_j определяется числовое расстояние $d(p_i, p_j)$. К функции расстояния предъявляются следующие требования: $d(p_i, p_i) = 0$; $d(p_i, p_j) > 0$ для разных p_i и p_j , а также $d(p_i, p_j) = d(p_j, p_i)$ (симметричность).

Предположим, объекты из U требуется разделить на k групп для заданного параметра k . Термином « k -кластеризация U » обозначается разбиение U на k непустых множеств C_1, C_2, \dots, C_k . «Интервалом k -кластеризации» называется минимальное расстояние между любой парой точек, находящихся в разных кластерах. С учетом того, что точки разных кластеров должны находиться далеко друг от друга, естественной целью является нахождение k -кластеризации с максимально возможным интервалом.

Мы приходим к следующему вопросу. Количество разных k -кластеризаций множества U вычисляется по экспоненциальной формуле; как эффективно найти кластеризацию с максимальным интервалом?



Разработка алгоритма

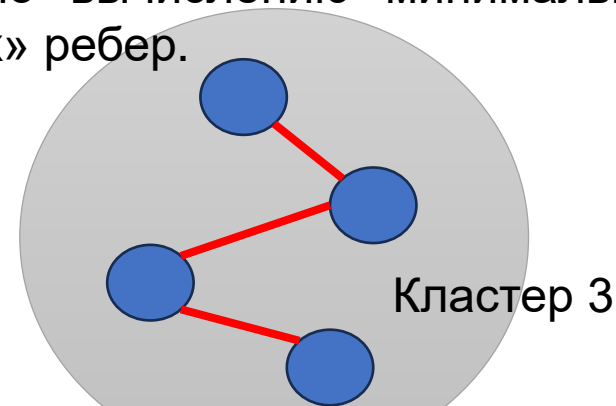
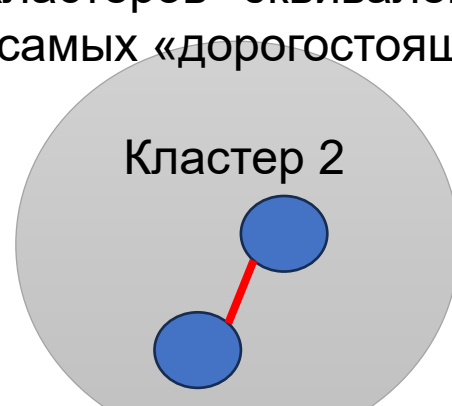
Чтобы найти кластеризацию с максимальным интервалом, мы рассмотрим процедуру расширения графа с множеством вершин U . Компоненты связности соответствуют кластерам, и мы постараемся как можно быстрее объединить близлежащие точки в один кластер (чтобы они не оказались в разных кластерах, находящихся поблизости друг от друга). Начнем с создания ребра между ближайшей парой точек; затем создается ребро между парой точек со следующей ближайшей парой и т. д. Далее мы продолжаем добавлять ребра между парами точек в порядке увеличения расстояния $d(p_i, p_j)$.

Таким образом граф H с вершинами U наращивает ребро за ребром, при этом компоненты связности соответствуют кластерам. Учтите, что нас интересуют только компоненты связности графа H , а не полное множество ребер; если при добавлении ребра (p_i, p_j) выяснится, что p_i и p_j уже принадлежат одному кластеру, ребро добавляться не будет — это не нужно, потому что ребро не изменит множество компонент. При таком подходе в процессе расширения графа никогда не образуется цикл, так что H будет объединением деревьев. Добавление ребра, концы которого принадлежат двум разным компонентам, фактически означает слияние двух соответствующих кластеров. В литературе по кластеризации подобный итеративный процесс слияния кластеров называется методом одиночной связи — частным случаем иерархической агломератной кластеризации. (Под «агломератностью» здесь понимается процесс объединения кластеров, а под «одиночной связью» — то, что для объединения кластеров достаточно одной связи.)



Разработка алгоритма

Какое отношение все это имеет к минимальным остовным деревьям? Очень простое: хотя процедура расширения графа базировалась на идее слияния кластеров, она в точности соответствует алгоритму минимального остовного дерева Крускала. Делается в точности то, что алгоритм Крускала сделал бы для графа G , если бы между каждой парой узлов (p_i, p_j) существовало ребро стоимостью $d(p_i, p_j)$. Единственное отличие заключается в том, что мы стремимся выполнить k -кластеризацию, поэтому процедура останавливается при получении k компонент связности. Другими словами, выполняется алгоритм Крускала, но он останавливается перед добавлением последних $k - 1$ ребер. Происходящее эквивалентно построению полного минимального остовного дерева T (в том виде, в каком оно было бы построено алгоритмом Крускала), удалению $k - 1$ ребер с наибольшей стоимостью (которое наш алгоритм вообще не добавлял) и определению k -кластеризации по полученным компонентам связности C_1, C_2, \dots, C_k . Итак, итеративное слияние кластеров эквивалентно вычислению минимального остовного дерева с удалением самых «дорогостоящих» ребер.



Анализ алгоритма

Утверждение 1.14. Компоненты C_1, C_2, \dots, C_k , образованные удалением $k - 1$ ребер минимального остовного дерева T с максимальной стоимостью, образуют k -кластеризацию с максимальным интервалом.

Доказательство. Пусть C — кластеризация C_1, C_2, \dots, C_k . Интервал C в точности равен длине $d * (k - 1)$ -го ребра с максимальной стоимостью в минимальном остовном дереве; это длина ребра, которое алгоритм Крускала добавил бы на следующем шаге (в тот момент, когда мы его остановили).

Возьмем другую k -кластеризацию C' , которая разбивает U на непустые множества C'_1, C'_2, \dots, C'_k . Требуется показать, что интервал C' не превышает d^* .

Так как две кластеризации C и C' не совпадают, из этого следует, что один из кластеров C_t не является подмножеством ни одного из k множеств C'_s в C' . Следовательно, должны существовать точки $p_i, p_j \in C_t$, принадлежащие разным кластерам в C' , — допустим, $p_i \in C'_s$ и $p_j \in C'_t \neq C'_s$.



Анализ алгоритма

Так как p_i и p_j принадлежат одной компоненте C_r , алгоритм Крускала должен был добавить все ребра p_i - p_j пути P перед его остановкой. В частности, это означает, что длина каждого ребра P не превышает d^* . Мы знаем, что $p_i \in C'_s$, но $p_j \notin C'_s$; пусть p' — первый узел P , который не принадлежит C'_s , а p — узел из P , непосредственно предшествующий p' . Только что было замечено, что $d(p, p') \leq d^*$, так как ребро (p, p') было добавлено алгоритмом Крускала. Но p и p' принадлежат разным множествам кластеризации C' , а значит, интервал C' не превышает $d(p, p') \leq d^*$, что и завершает доказательство. ■

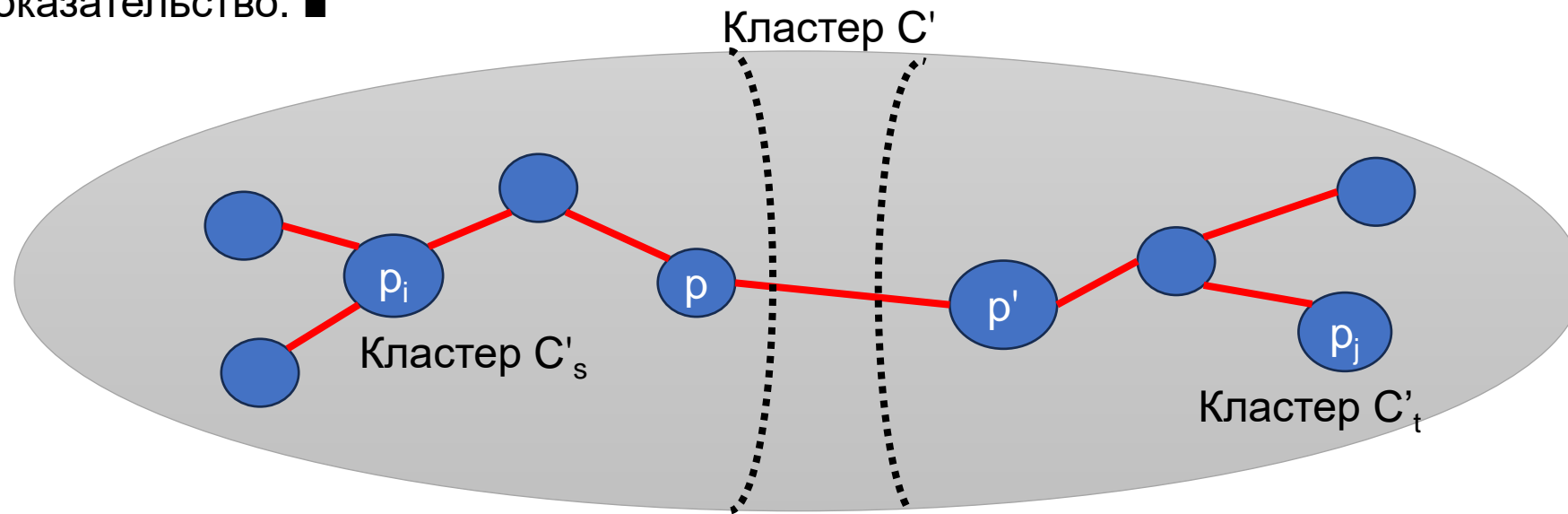


Иллюстрация доказательства (1.14), показывающая, что интервал никакой другой кластеризации не может быть больше интервала кластеризации, найденной алгоритмом одиночной связи



Анализ алгоритма

Утверждение 1.14. Компоненты C_1, C_2, \dots, C_k , образованные удалением $k - 1$ ребер минимального остовного дерева T с максимальной стоимостью, образуют k -кластеризацию с максимальным интервалом.

Доказательство. Пусть C — кластеризация C_1, C_2, \dots, C_k . Интервал C в точности равен длине $d * (k - 1)$ -го ребра с максимальной стоимостью в минимальном остовном дереве; это длина ребра, которое алгоритм Крускала добавил бы на следующем шаге (в тот момент, когда мы его остановили).

Возьмем другую k -кластеризацию C' , которая разбивает U на непустые множества C'_1, C'_2, \dots, C'_k . Требуется показать, что интервал C' не превышает $d*$.

Так как две кластеризации C и C' не совпадают, из этого следует, что один из кластеров C_t не является подмножеством ни одного из k множеств C'_s в C' . Следовательно, должны существовать точки $p_i, p_j \in C_t$, принадлежащие разным кластерам в C' , — допустим, $p_i \in C'_s$ и $p_j \in C'_t \neq C'_s$.



Анализ алгоритма

Утверждение 1.14. Компоненты C_1, C_2, \dots, C_k , образованные удалением $k - 1$ ребер минимального остовного дерева T с максимальной стоимостью, образуют k -кластеризацию с максимальным интервалом.

Доказательство. Пусть C — кластеризация C_1, C_2, \dots, C_k . Интервал C в точности равен длине $d * (k - 1)$ -го ребра с максимальной стоимостью в минимальном остовном дереве; это длина ребра, которое алгоритм Крускала добавил бы на следующем шаге (в тот момент, когда мы его остановили).

Возьмем другую k -кластеризацию C' , которая разбивает U на непустые множества C'_1, C'_2, \dots, C'_k . Требуется показать, что интервал C' не превышает d^* .

Так как две кластеризации C и C' не совпадают, из этого следует, что один из кластеров C_t не является подмножеством ни одного из k множеств C'_s в C' . Следовательно, должны существовать точки $p_i, p_j \in C_t$, принадлежащие разным кластерам в C' , — допустим, $p_i \in C'_s$ и $p_j \in C'_t \neq C'_s$.



Адженда

**Жадные
алгоритмы**

45 минут

**Динамическое
программирова
ние**

45 минут

Введение

Все задачи ранее показанные объединял тот факт, что в конечном итоге действительно находился работающий жадный алгоритм. К сожалению, так бывает далеко не всегда; для большинства задач настоящие трудности возникают не с выбором правильной жадной стратегии из нескольких вариантов, а с тем, что естественного жадного алгоритма для задачи вообще не существует. В таких случаях важно иметь наготове другие методы.

Давайте обратимся к более мощному и нетривиальному методу разработки алгоритмов — динамическому программированию. Охарактеризовать динамическое программирование будет проще после того, как вы увидите его в действии, но основная идея базируется на интуитивных представлениях, лежащих в основе принципа «разделяй и властвуй», и по сути противоположна жадной стратегии: алгоритм неявно исследует пространство всех возможных решений, раскладывает задачу на серии подзадач, а затем строит правильные решения подзадач все большего размера. В некотором смысле динамическое программирование может рассматриваться как метод, работающий в опасной близости от границ перебора «грубой силой»: хотя оно систематически прорабатывает большой набор возможных решений задачи, это делается без явной проверки всех вариантов. Именно из-за этой необходимости тщательного выдерживания баланса динамическое программирование бывает трудно освоить; как правило, вы начинаете чувствовать себя уверенно только после накопления немалого практического опыта.



Взвешенное интервальное планирование: рекурсивная процедура

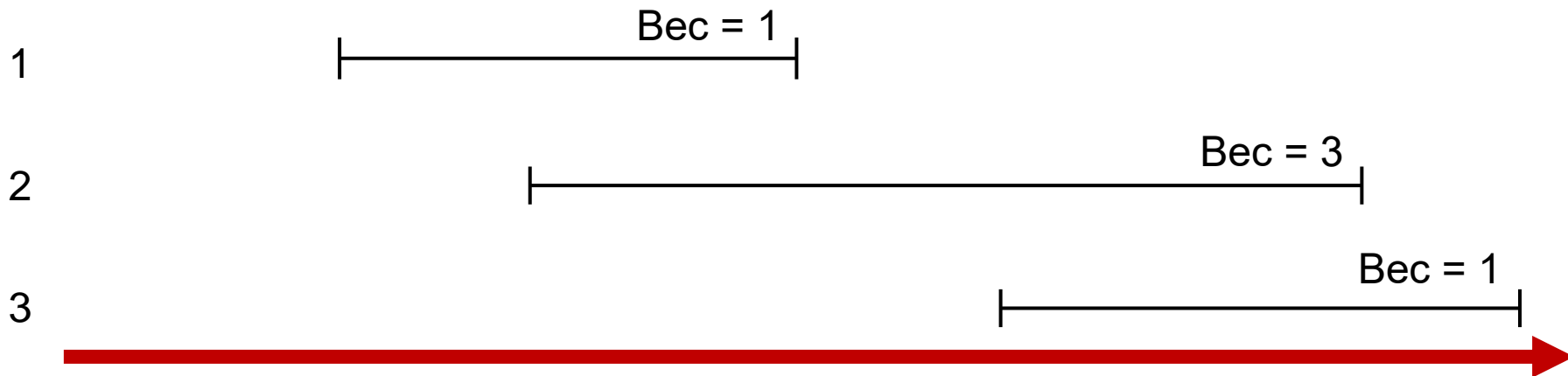
Мы уже видели, что жадный алгоритм обеспечивает оптимальное решение задачи интервального планирования, целью которой является получение множества неперекрывающихся интервалов максимально возможного размера. Задача взвешенного интервального планирования имеет более общий характер: с каждым интервалом связывается определенное значение (вес), а решением задачи считается множество с максимальным суммарным весом.



Разработка рекурсивного алгоритма

Поскольку исходная задача интервального планирования представляет собой частный случай, в котором все веса равны 1, мы уже знаем, что большинство жадных алгоритмов не обеспечивает оптимального решения. Но даже алгоритм, который работал ранее (многократный выбор интервала, завершающегося раньше всех), в этой ситуации уже не является оптимальным.

Индекс Простой пример задачи взвешенного интервального планирования



Действительно, естественный жадный алгоритм для этой задачи неизвестен, что заставляет нас переключиться на динамическое программирование.

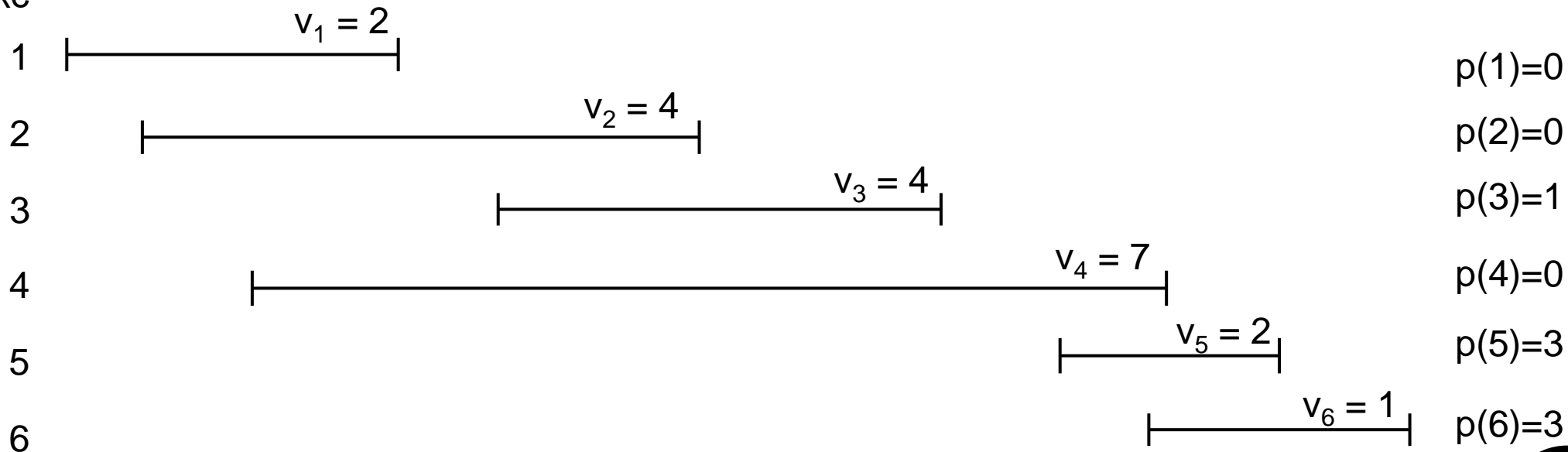
Имеются n заявок с пометками $1, \dots, n$; для каждой заявки i указывается начальное время s_i и конечное время f_i . С каждым интервалом i связывается некоторое значение — вес v_i . Два интервала называются совместимыми, если они не перекрываются. Целью текущей задачи является нахождение подмножества $S \subseteq \{1, \dots, n\}$ взаимно совместимых интервалов, максимизирующего сумму весов выбранных интервалов $\sum_{i \in S} v_i$.

Разработка рекурсивного алгоритма

Предположим, заявки отсортированы в порядке неубывания конечного времени: $f_1 \leq f_2 \leq \dots \leq f_n$. Заявка i называется предшествующей заявке j , если $i < j$. Таким образом определяется естественный порядок «слева направо», в котором будут рассматриваться интервалы. Чтобы было удобнее обсуждать этот порядок, определим $p(j)$ для интервала j как наибольший индекс $i < j$, при котором интервалы i и j не перекрываются. Другими словами, i — крайний левый интервал, который завершается до начала j . Определим $p(j) = 0$, если не существует заявки $i < j$, не перекрывающейся с j .

Задача взвешенного интервального планирования с функциями $p(j)$, определенными для каждого интервала j

Индекс



Разработка рекурсивного алгоритма

Допустим, у нас имеется экземпляр задачи взвешенного интервального планирования; рассмотрим оптимальное решение O , временно забыв о том, что нам о нем ничего не известно. Что можно утверждать о решении O : либо интервал n (последний) принадлежит O , либо нет. Если $n \in O$, то очевидно, ни один интервал, индексируемый строго между $p(n)$ и n , не может принадлежать O , потому что по определению $p(n)$ мы знаем, что интервалы $p(n) + 1$, $p(n) + 2$, ..., $n - 1$ — все они перекрывают интервал n . Более того, если $n \in O$, то решение O должно включать оптимальное решение задачи, состоящей из заявок $\{1, \dots, p(n)\}$, потому что в противном случае выбор заявок O из $\{1, \dots, p(n)\}$ можно было бы заменить лучшим выбором без риска перекрытия заявки n . С другой стороны, если $n \notin O$, то O просто совпадает с оптимальным решением задачи, состоящей из заявок $\{1, \dots, n - 1\}$. Рассуждения полностью аналогичны: предположим, что O не включает заявку n ; если оно не выбирает оптимальное множество заявок из $\{1, \dots, n - 1\}$, то его можно заменить лучшим выбором.

Таким образом, процессе поиска оптимального решения для интервалов $\{1, 2, \dots, n\}$ будет производиться поиск оптимальных решений меньших задач в форме $\{1, 2, \dots, j\}$. Пусть для каждого значения j от 1 до n оптимальное решение задачи, состоящей из заявок $\{1, \dots, j\}$, обозначается O_j , а значение (суммарный вес) этого решения — $OPT(j)$. (Также мы определим $OPT(0) = 0$ как оптимум по пустому множеству интервалов.) Искомое оптимальное решение представляет собой O_n со значением $OPT(n)$. Для оптимального решения O_j по интервалам $\{1, 2, \dots, j\}$ из приведенных выше рассуждений (обобщенных для $j = n$) следует, что либо $j \in O_j$, и тогда $OPT(j) = v_j + OPT(p(j))$, либо $j \notin O_j$, и тогда $OPT(j) = OPT(j - 1)$.

Разработка рекурсивного алгоритма

Так как других вариантов быть не может ($j \in O_j$ или $j \notin O_j$), можно утверждать, что

Утверждение 2.1. $OPT(j) = \max(v_j + OPT(p(j)), OPT(j - 1))$.

Утверждение 2.2. Заявка j принадлежит оптимальному решению для множества $\{1, 2, \dots, j\}$ в том и только в том случае, если $v + OPT(p(j)) \geq OPT(j - 1)$.

Эти факты образуют первый важнейший компонент, на котором базируется решение методом динамического программирования: рекуррентное уравнение, которое выражает оптимальное решение (или его значение) в контексте оптимальных решений меньших подзадач.

Рекурсивный алгоритм для вычисления $OPT(n)$:

Compute-Opt(j)

Если $j = 0$

Вернуть 0

Иначе

Вернуть $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1))$

Конец условия

Правильность алгоритма напрямую доказывается индукцией по j :

Утверждение 2.3. $\text{Compute-Opt}(j)$ правильно вычисляет $OPT(j)$ для всех $j = 1, 2, \dots, n$.

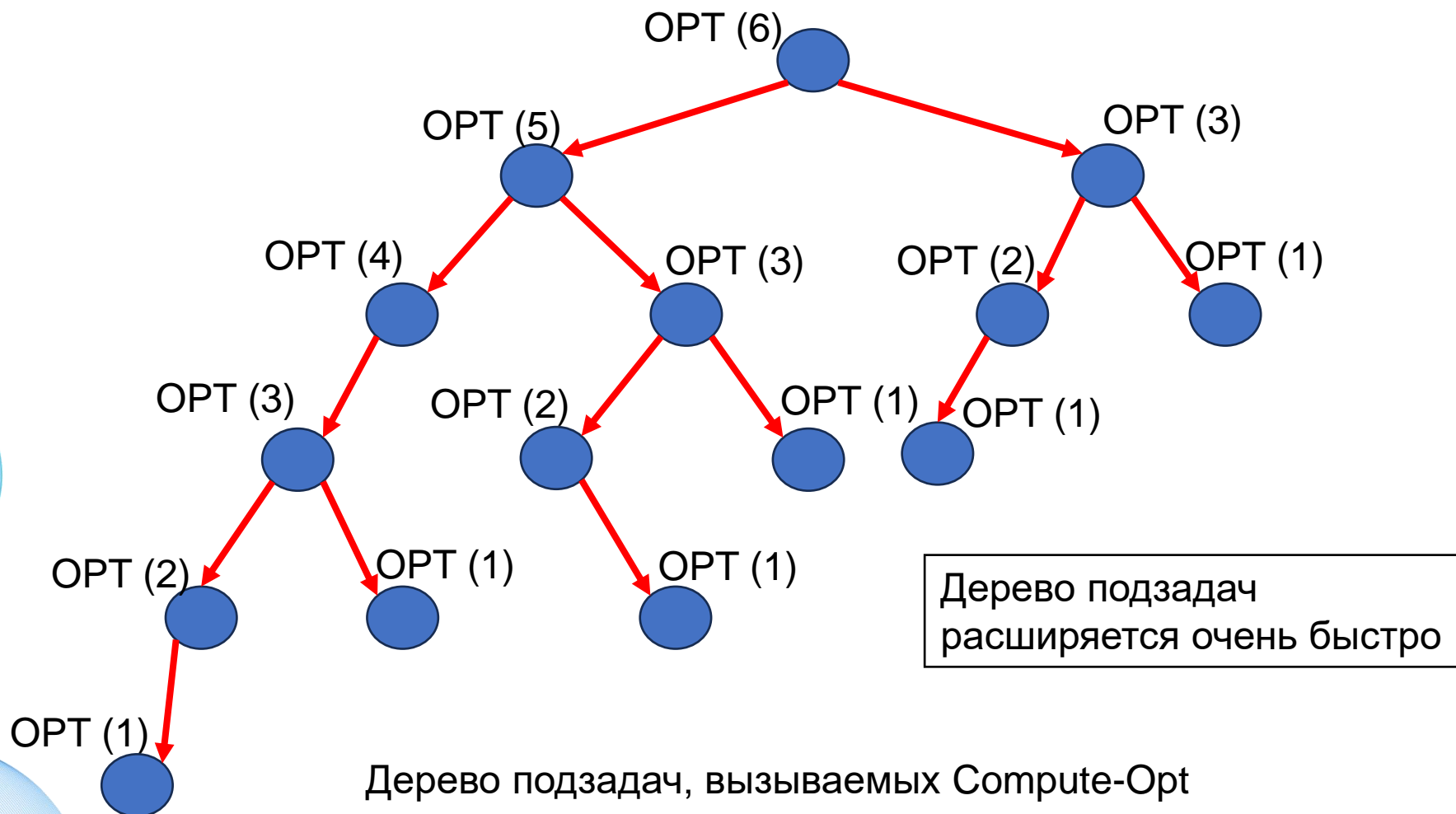
Доказательство. По определению $OPT(0) = 0$. Теперь возьмем некоторое значение $j > 0$ и предположим, что $\text{Compute-Opt}(i)$ правильно вычисляет $OPT(i)$ для всех $i < j$. Из индукционной гипотезы следует, что $\text{Compute-Opt}(p(j)) = OPT(p(j))$ и

$\text{Compute-Opt}(j - 1) = OPT(j - 1)$; следовательно, из (2.1)

$OPT(j) = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j - 1)) = \text{Compute-Opt}(j)$ ■

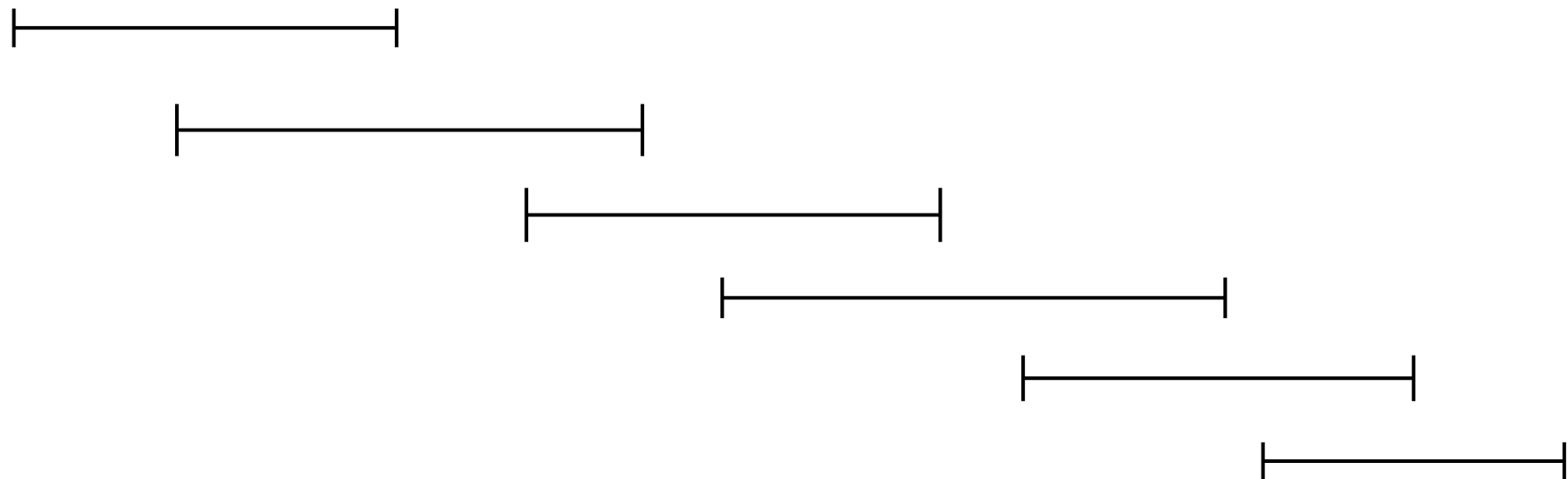
Разработка рекурсивного алгоритма

К сожалению, если реализовать алгоритм Compute-Opt в этом виде, его выполнение в худшем случае займет экспоненциальное время.



Разработка рекурсивного алгоритма

Или более экстремальный пример: в системе с частым наложением, где $p(j) = j - 2$ для всех $j = 2, 3, 4, \dots, n$, мы видим, что `Compute-Opt(j)` генерирует отдельные рекурсивные вызовы для задач с размерами $j - 1$ и $j - 2$. Другими словами, общее количество вызовов `Compute-Opt` в этой задаче будет расти в темпе чисел Фибоначчи, что означает экспоненциальный рост. Таким образом, решение с полиномиальным временем так и остается нереализованным.



Экземпляр задачи взвешенного интервального планирования, для которого простая рекурсия `Compute-Opt` занимает экспоненциальное время. Веса интервалов в этом примере равны 1

Мемоизация рекурсии

Фундаментальный факт, который становится вторым критическим компонентом решения методом динамического программирования, заключается в том, что наш рекурсивный алгоритм `Compute-Opt` в действительности решает $n + 1$ разных подзадач: `Compute-Opt(0)`, `Compute-Opt(1)`, ..., `Compute-Opt(n)`. Тот факт, что он выполняется за экспоненциальное время, приведен просто из-за впечатляющей избыточности количества выдач каждого из этих вызовов.

Как устранить всю эту избыточность? Можно сохранить значение `Compute-Opt` в глобально-доступном месте при первом вычислении и затем просто использовать заранее вычисленное значение вместо всех будущих рекурсивных вызовов. Метод сохранения ранее вычисленных значений называется мемоизацией (memoization).

M-Compute-Opt(j)

Если $j = 0$

Вернуть 0

Иначе Если `M[j]` не пусто

Вернуть `M[j]`

Иначе

Определить $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j - 1))$

Вернуть `M[j]`

Конец условия

Анализ мемоизированной версии

Конечно, этот алгоритм очень похож на предыдущую реализацию; однако мемоизация позволяет сократить время выполнения.

Утверждение 2.4. Время выполнения $M\text{-Compute-Opt}(n)$ равно $O(n)$ (предполагается, что входные интервалы отсортированы по конечному времени).

Доказательство. Время, потраченное на один вызов $M\text{-Compute-Opt}$, равно $O(1)$ без учета времени, потраченного в порожденных им рекурсивных вызовах. Таким образом, время выполнения ограничивается константой, умноженной на количество вызовов $M\text{-Compute-Opt}$. Так как сама реализация не предоставляет явной верхней границы для количества вызовов, для получения границы мы попытаемся найти хорошую метрику «прогресса».

Самой полезной метрикой прогресса является количество «непустых» элементов M . В исходном состоянии оно равно 0; но при каждом переходе на новый уровень с выдачей двух рекурсивных вызовов $M\text{-Compute-Opt}$ заполняется новый элемент, а следовательно, количество заполненных элементов увеличивается на 1. Так как M содержит только $n + 1$ элемент, из этого следует, что количество вызовов $M\text{-Compute-Opt}$ не превышает $O(n)$, а следовательно, время выполнения $M\text{-ComputeOpt}(n)$ равно $O(n)$, как и требовалось. ■

Вычисление решения помимо его значения

Алгоритм `M-ComputeOpt` легко расширяется для отслеживания оптимального решения: можно создать дополнительный массив S , в элементе $S[i]$ которого хранится оптимальное множество интервалов из $\{1, 2, \dots, i\}$. Однако наивное расширение кода для сохранения решений в массиве S увеличит время выполнения с дополнительным множителем $O(n)$: хотя позиция в массиве M может обновляться за время $O(1)$, запись множества в массив S занимает время $O(n)$. Чтобы избежать возрастания $O(n)$, вместо явного хранения S можно восстановить оптимальное решение по значениям, хранящимся в массиве M после вычисления оптимального значения. Из (2.2) известно, что j принадлежит оптимальному решению для множества интервалов $\{1, \dots, j\}$ в том, и только в том случае, если $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j - 1)$.

Find-Solution(j)

Если $j = 0$

Ничего не выводить

Иначе Если $v_j + M[p(j)] \geq M[j - 1]$

Вывести j вместе с результатом `Find-Solution(p(j))`

Иначе

Вывести результат `Find-Solution(j - 1)`

Конец условия

Утверждение 2.5. Для массива M с оптимальными значениями подзадач `Find-Solution` возвращает оптимальное решение за время $O(n)$.

Принципы динамического программирования: мемоизация или итерации с подзадачами

Теперь мы воспользуемся алгоритмом задачи взвешенного интервального планирования для обобщения базовых принципов динамического программирования. Кроме того, он поможет понять принцип, который играет ключевую роль: итерации с подзадачами вместо рекурсивного вычисления решений.

Ранее для построения решения задачи взвешенного интервального планирования с полиномиальным временем мы сначала разработали рекурсивный алгоритм с экспоненциальным временем, а затем преобразовали его (посредством мемоизации) в эффективный рекурсивный алгоритм, который обращался к глобальному массиву M за оптимальными решениями подзадач. Но чтобы понять, что здесь на самом деле происходит, желательно сформулировать практически эквивалентную версию алгоритма. Именно эта новая формулировка наиболее явно отражает суть метода динамического программирования и служит общим шаблоном для алгоритмов.

Разработка алгоритма

Ключом к построению эффективного алгоритма является массив M . В нем закодирована идея о том, что мы используем значение оптимальных решений подзадач для интервалов $\{1, 2, \dots, j\}$ по всем j , и он использует (2.1) для определения значения $M[j]$ на основании значений предшествующих элементов массива. Как только мы получаем массив M , задача решена: $M[n]$ содержит значение оптимального решения для всего экземпляра, а процедура Find-Solution может использоваться для эффективного обратного отслеживания по M и возвращения оптимального решения.

Важно понять, что элементы M можно напрямую вычислять итеративным алгоритмом вместо рекурсии с мемоизацией. Просто начнем с $M[0] = 0$ и будем увеличивать j ; каждый раз, когда потребуется определить значение $M[j]$, ответ предоставляется (2.1). Алгоритм выглядит так:

Iterative-Compute-Opt

$M[0] = 0$

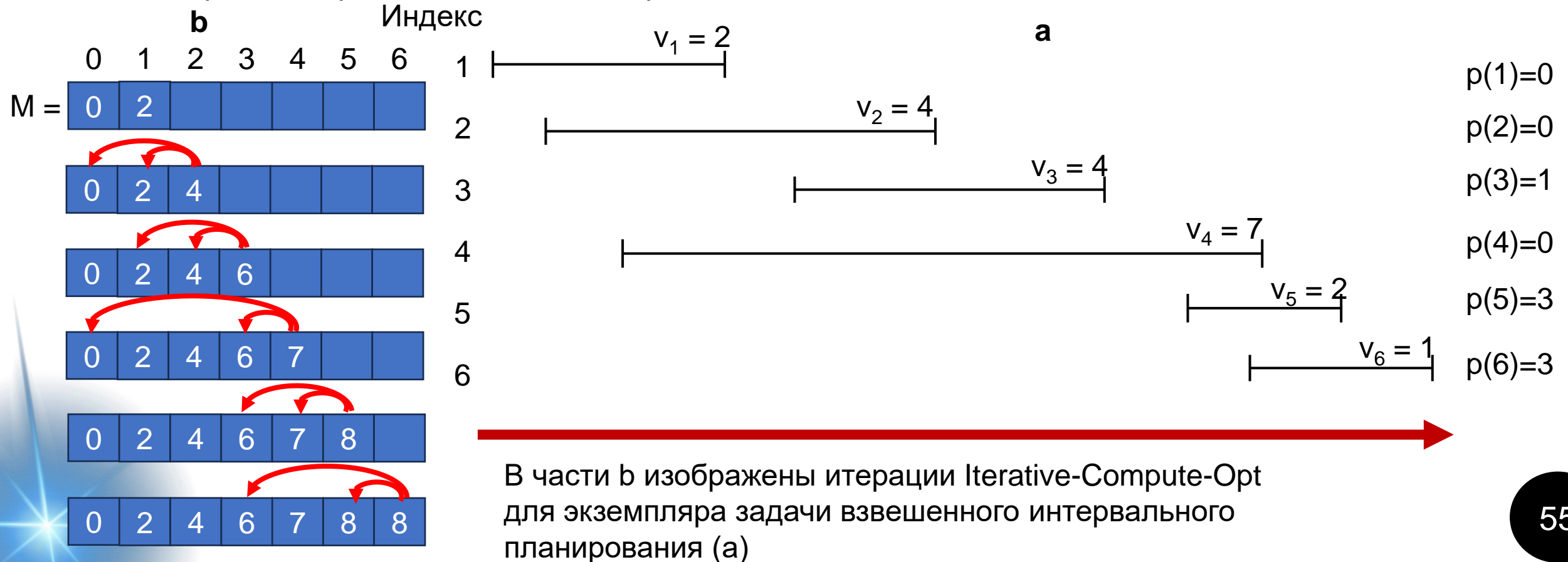
For $j = 1, 2, \dots, n$

$M[j] = \max(v_j + M[p(j)], M[j - 1])$

Конец цикла

Анализ алгоритма

Следуя точной аналогии с доказательством (2.3), можно доказать посредством индукции по j , что этот алгоритм записывает $OPT(j)$ в элемент массива $M[j]$; (2.1) предоставляет шаг индукции. Кроме того, как и ранее, Find-Solution можно передать заполненный массив M для получения оптимального решения помимо значения. Наконец, время выполнения Iterative-Compute-Opt очевидно равно $O(n)$, так как алгоритм явно выполняется в течение n итераций и проводит постоянное время в каждой из них.



Основная схема динамического программирования

Чтобы взяться за разработку алгоритма, основанного на методе динамического программирования, необходимо иметь набор подзадач, производных от исходной задачи, который обладает некоторыми базовыми свойствами.

- (i) Количество подзадач ограничено полиномиальной зависимостью.
- (ii) Решение исходной задачи легко вычисляется по решениям подзадач. (Например, исходная задача может быть одной из подзадач.)
- (iii) Существует естественное упорядочение подзадач от «меньшей» к «большей» в совокупности с легко вычисляемой рекуррентностью (как в (2.1) и (2.2)), которая позволяет определить решение подзадачи по решениям некоторого количества меньших подзадач.

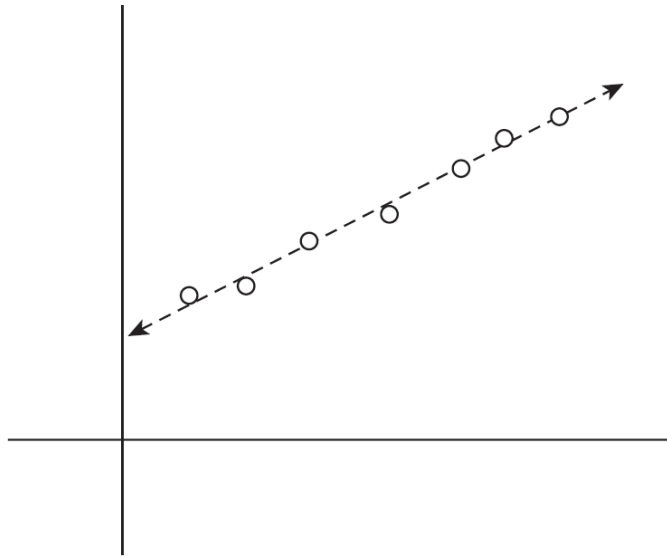
Связь между подзадачами и рекуррентными отношениями, в которой трудно отделить первопричину от следствия, является одной из тонких особенностей, лежащих в основе динамического программирования. Никогда нельзя быть уверенным в том, что набор подзадач будет полезным, пока не будет найдено рекуррентное отношение, связывающее подзадачи воедино; но о рекуррентном отношении трудно думать в отсутствие «меньших» подзадач, с которыми оно работает.

Сегментированные наименьшие квадраты: многовариантный выбор

Перейдем к другой категории задач, которая демонстрирует чуть более сложную разновидность динамического программирования. В задаче, рассматриваемой ниже, в рекуррентности будет задействовано то, что можно назвать «многовариантным выбором»: на каждом шаге существует полиномиальное количество вариантов, которые могут рассматриваться как кандидаты для структуры оптимального решения. Как вы увидите, метод динамического программирования очень естественно адаптируется к этой более общей ситуации. Отдельно стоит сказать о том, что задача этого раздела также хорошо показывает, как четкое алгоритмическое определение может формализовать понятие, изначально казавшееся слишком нечетким и нелогичным для математического представления.

Задача

При работе с научными и статистическими данными, нанесенными на плоскость, аналитики часто пытаются найти «линию наилучшего соответствия» для этих данных



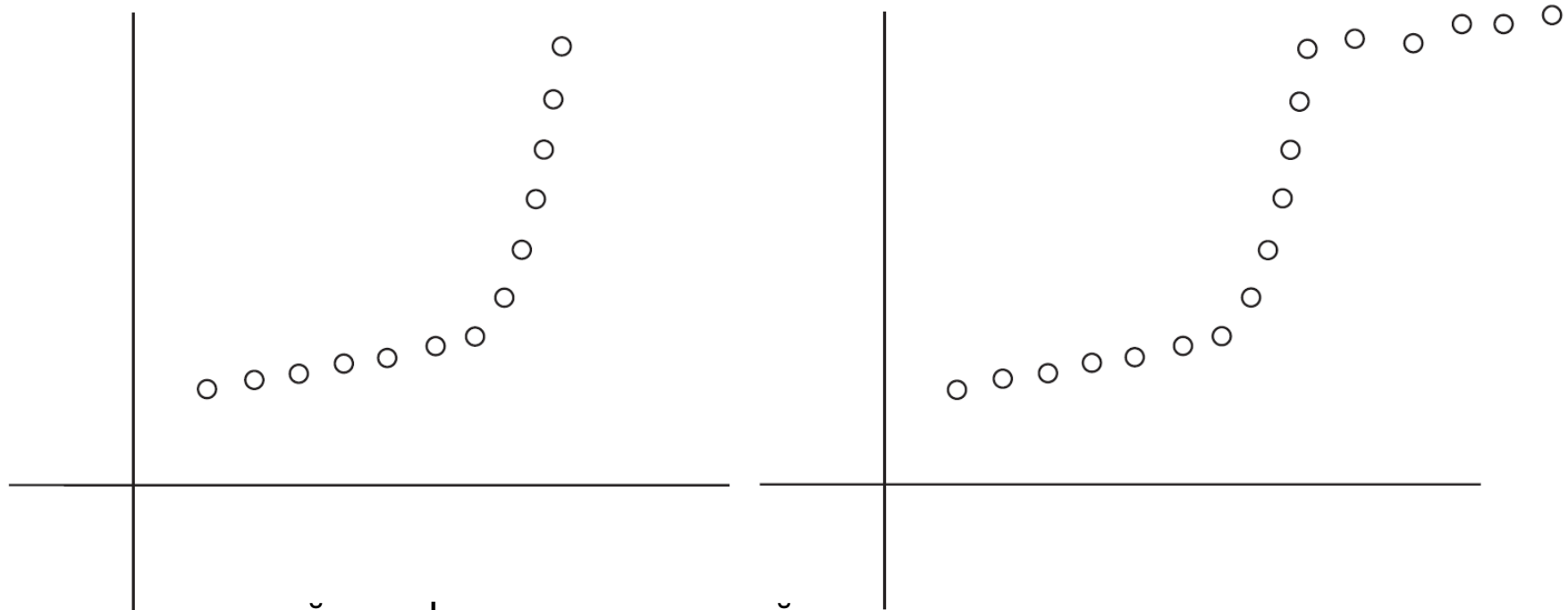
Эта основополагающая задача из области статистики и вычислительной математики формулируется следующим образом. Допустим, данные представляют собой множество P из n точек на плоскости, обозначаемых $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$; предполагается, что $x_1 < x_2 < \dots < x_n$. Для линии L , определяемой уравнением $y = ax + b$, погрешностью L относительно P называется сумма возведенных в квадрат «расстояний» от точек до P :

$$Error(L, P) = \sum_{i=1}^n (y_i - ax_i - b)^2.$$

Естественной целью в такой ситуации будет нахождение линии с минимальной погрешностью. Оказывается, у этой задачи существует компактное решение, которое легко вычисляется методами математического анализа. Опуская промежуточные выкладки, приведем результат: линия минимальной погрешности определяется формулой $y = ax + b$, где

$$a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2} \text{ и } b = \frac{\sum_i y_i - a \sum_i x_i}{n}.$$

Задача



Но существует проблема, для которой эти формулы не подойдут.

В таком случае нам хотелось бы сделать утверждение вида «Точки лежат приблизительно на серии из двух линий». Как формализовать такое понятие? Для любой одиночной линии погрешность у точек на рисунке будет запредельной; но, если использовать две линии, ошибка может быть относительно невелика. Итак, новую задачу можно попытаться сформулировать следующим образом: вместо того, чтобы искать одну линию наилучшего соответствия, мы ищем набор линий, обеспечивающий минимальную погрешность. Но такая формулировка задачи никуда не годится, потому что у нее имеется тривиальное решение: при сколь угодно большом наборе линий можно добиться идеального соответствия, соединив линиями каждую пару последовательных точек в P .

С противоположной стороны можно жестко «запрограммировать» число 2 в задаче — искать лучшее соответствие не более чем с двумя линиями. Но и такая формулировка противоречит здравому смыслу: изначально не было никакой априорной информации о том, что точки лежат приблизительно на двух линиях; мы пришли к такому выводу, взглянув на иллюстрацию.

Формулировка задачи

Как и в приведенном выше описании, имеется множество точек $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, в котором $x_1 < x_2 < \dots < x_n$. Точка (x_i, y_i) будет обозначаться p_i . Сначала необходимо разбить P на некоторое количество сегментов. Каждый сегмент является подмножеством P , представляющим непрерывный набор координат x ; другими словами, это подмножество вида $\{p_i, p_{i+1}, \dots, p_{j-1}, p_j\}$ для некоторых индексов $i \leq j$. Затем для каждого сегмента S в разбиении P вычисляется линия, минимизирующая погрешность в отношении точек S для приведенных выше формул.

Штраф разбиения определяется как сумма следующих слагаемых:

- (i) Количество сегментов, на которые разбивается P , умноженное на фиксированный множитель $C > 0$.
- (ii) Для каждого сегмента — значение погрешности для оптимальной линии через этот сегмент.

Нашей целью в сегментированной задаче наименьших квадратов является нахождение разбиения с минимальным штрафом. Минимизация этой характеристики отражает компромиссы, упоминавшиеся ранее. При поиске решения могут рассматриваться разбиения на любое количество сегментов; с увеличением количества сегментов уменьшаются слагаемые штрафа из части (ii) определения, но увеличивается слагаемое в части (i). (Множитель C предоставляется со входными данными; настраивая C , можно увеличить или уменьшить штраф за использование дополнительных линий.)

Количество возможных разбиений в P растет по экспоненте, и изначально непонятно, возможен ли эффективный поиск оптимального разбиения?

Формулировка задачи

Как и в приведенном выше описании, имеется множество точек $P = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, в котором $x_1 < x_2 < \dots < x_n$. Точка (x_i, y_i) будет обозначаться p_i . Сначала необходимо разбить P на некоторое количество сегментов. Каждый сегмент является подмножеством P , представляющим непрерывный набор координат x ; другими словами, это подмножество вида $\{p_i, p_{i+1}, \dots, p_{j-1}, p_j\}$ для некоторых индексов $i \leq j$. Затем для каждого сегмента S в разбиении P вычисляется линия, минимизирующая погрешность в отношении точек S для приведенных выше формул.

Штраф разбиения определяется как сумма следующих слагаемых:

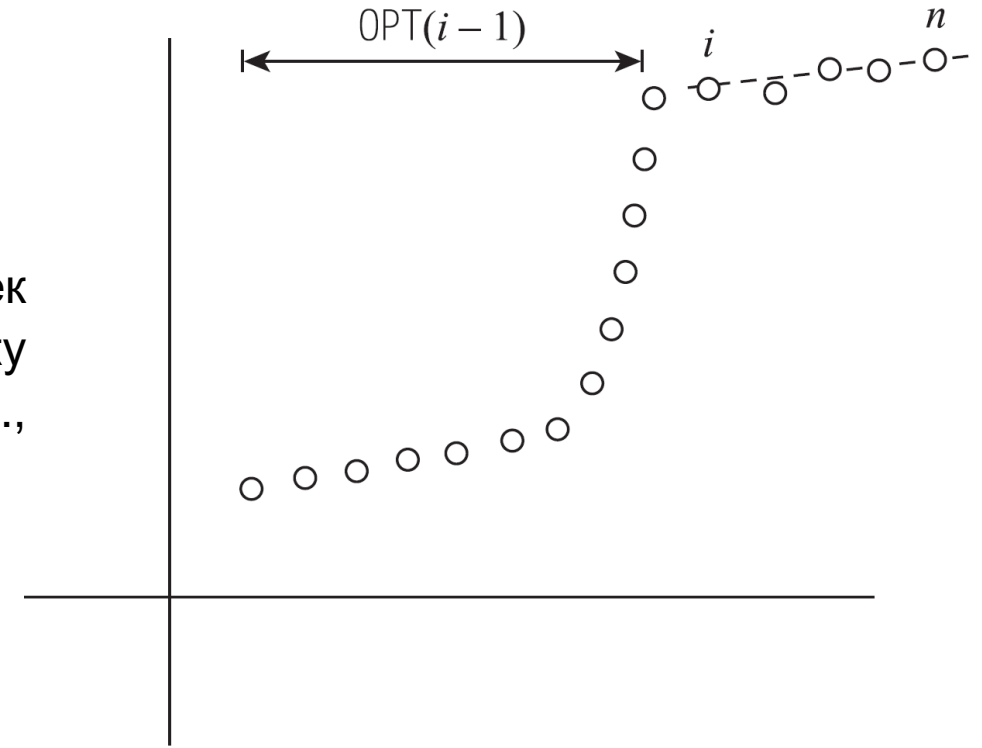
- (i) Количество сегментов, на которые разбивается P , умноженное на фиксированный множитель $C > 0$.
- (ii) Для каждого сегмента — значение погрешности для оптимальной линии через этот сегмент.

Нашей целью в сегментированной задаче наименьших квадратов является нахождение разбиения с минимальным штрафом. Минимизация этой характеристики отражает компромиссы, упоминавшиеся ранее. При поиске решения могут рассматриваться разбиения на любое количество сегментов; с увеличением количества сегментов уменьшаются слагаемые штрафа из части (ii) определения, но увеличивается слагаемое в части (i). (Множитель C предоставляется со входными данными; настраивая C , можно увеличить или уменьшить штраф за использование дополнительных линий.)

Количество возможных разбиений в P растет по экспоненте, и изначально непонятно, возможен ли эффективный поиск оптимального разбиения?

Разработка алгоритма

Возможное решение: подбираем один сегмент для точек p_i, p_{i+1}, \dots, p_n , после чего переходим к поиску оптимального решения для оставшихся точек p_1, p_2, \dots, p_{i-1}



Для алгоритма динамического программирования требуется полиномиальное количество подзадач, решения которых должны дать решение исходной задачи; и решения этих подзадач должны строиться с использованием рекуррентного отношения.

Для сегментированной задачи наименьших квадратов очень полезно следующее наблюдение: последняя точка p_n принадлежит одному сегменту оптимального разбиения, и этот сегмент начинается в некоторой более ранней точке p_i . Подобные наблюдения могут подсказать правильное множество подзадач: зная последний сегмент p_i, \dots, p_n , мы можем исключить эти точки из рассмотрения и рекурсивно решить задачу для оставшихся точек p_1, \dots, p_{i-1} .

Разработка алгоритма

Предположим, $OPT(i)$ обозначает оптимальное решение для точек p_1, \dots, p_i , а $e_{i,j}$ — минимальную погрешность для любой линии в отношении p_i, p_{i+1}, \dots, p_j . (Запись $OPT(0) = 0$ используется как граничный случай.) Тогда приведенное выше наблюдение означает следующее:

Утверждение 2.6. Если последний сегмент оптимального решения состоит из точек p_i, \dots, p_n , то значение оптимального решения равно $OPT(n) = e_{i,n} + C + OPT(i - 1)$.

Используя то же наблюдение для подзадачи, состоящей из точек p_1, \dots, p_j , мы видим, что для получения $OPT(j)$ необходимо найти лучший способ получения завершающего сегмента p_i, \dots, p_j — с оплатой погрешности и слагаемого C для этого сегмента — в сочетании с оптимальным решением $OPT(i - 1)$ для остальных точек. Другими словами, мы обосновали следующее рекуррентное отношение:

Утверждение 2.7. Для подзадачи с точками p_1, \dots, p_j

$$OPT(j) = \min_{1 \leq i \leq j} (e_{i,j} + C + OPT(i - 1)),$$

а сегмент p_i, \dots, p_j используется в оптимальном решении подзадачи в том и только в том случае, если минимум достигается при использовании индекса i .

Разработка алгоритма

Построить решения $OPT(i)$ в порядке возрастания i .

Segmented-Least-Squares(n)

Массив $M[0 \dots n]$

Присвоить $M[0] = 0$

Для всех пар $i \leq j$

 Вычислить наименьшую квадратичную погрешность $e_{i,j}$ для сегмента p_i, \dots, p_j

Конец цикла

Для $j = 1, 2, \dots, n$

 Использовать рекуррентное отношение (2.7) для вычисления $M[j]$

Конец цикла

Вернуть $M[n]$

Как и в алгоритме взвешенного интервального планирования, оптимальное разбиение вычисляется обратным отслеживанием по M .

Find-Segments(j)

Если $j = 0$

 Ничего не выводить

Иначе

 Найти значение i , минимизирующее $e_{i,j} + C + M[i - 1]$

 Вывести сегмент $\{p_i, \dots, p_j\}$ и результат $\text{Find-Segments}(i - 1)$

Конец условия

Анализ алгоритма

Осталось проанализировать время выполнения Segmented-Least-Squares. Сначала необходимо вычислить значения всех погрешностей наименьших квадратов $e_{i,j}$. Чтобы учесть время выполнения, расходуемое на эти вычисления, заметим, что существуют $O(n^2)$ пар (i, j) , для которых эти вычисления необходимы; для каждой пары (i, j) можно использовать формулу, приведенную в начале этого раздела, для вычисления $e_{i,j}$ за время $O(n)$. Следовательно, общее время выполнения для вычисления всех значений $e_{i,j}$ равно $O(n^3)$, на самом деле можно быстрее за $O(n^2)$, но это дз.

Алгоритм состоит из n итераций для значений $j = 1, \dots, n$. Для каждого значения j необходимо определить минимум в рекуррентном отношении (2.7) для заполнения элемента массива $M[j]$; это занимает время $O(n)$ для каждого j , что в сумме дает $O(n^2)$. Итак, после определения всех значений $e_{i,j}$ время выполнения равно $O(n^2)$.

Задача о сумме подмножеств и задача о рюкзаке: добавление переменной

Мы видим все больше примеров того, что область планирования предоставляет богатый источник алгоритмических задач, обладающих практическим смыслом. Ранее мы рассматривали задачи, в которых заявки задаются интервалом времени, а также задачи, в которых для заявки определяется продолжительность и предельное время, но не указывается конкретный интервал времени, в течение которого они должны выполняться.

Далее рассматривается разновидность задач второго типа, с продолжительностями и предельным временем, которую трудно решать напрямую с использованием методов, применявшихся ранее. Для решения задачи будет применен метод динамического программирования, но с небольшими изменениями: «очевидного» множества подзадач оказывается недостаточно, и нам придется создавать расширенный набор подзадач. Как вы вскоре увидите, для этого в рекуррентное отношение, лежащее в основе динамической программы, будет добавлена новая переменная.

Задача

В рассматриваемой задаче планирования имеется одна машина, способная обрабатывать задания, и набор заявок $\{1, 2, \dots, n\}$. Ресурсы могут обрабатываться только в период времени от 0 до W (для некоторого числа W). Каждая заявка представляет собой задание, для обработки которого требуется время w_i .

Если нашей целью является обработка заданий, при которой обеспечивается максимальная занятость машины до «граничного времени» W , какие задания следует выбрать?

В более формальном виде: имеются n элементов $\{1, \dots, n\}$, каждому из которых присвоен неотрицательный вес w_i (для $i = 1, \dots, n$). Также задана граница W . Требуется выбрать подмножество S элементов, для которого $\sum_{i \in S} w_i \leq W$, чтобы этого ограничения значение $\sum_{i \in S} w_i$ было как можно больше. Эта задача называется задачей о сумме подмножеств.

Целью в этой общей задаче является нахождение подмножества с максимальным суммарным значением, при котором общий вес не превышает W .

Поскольку ситуация напоминает другие задачи планирования, уже встречавшиеся ранее, естественно спросить, можно ли найти оптимальное решение при помощи жадного алгоритма. Похоже, ответ на этот вопрос отрицателен — по крайней мере, не известно никакое эффективное жадное правило, которое всегда строит оптимальное решение. Один из естественных жадных методов основан на сортировке элементов по убыванию веса (или по крайней мере для всех элементов с весом, не превышающим W), а затем выборе элементов в этом порядке, пока общий вес остается меньше W . Но если значение W кратно 2, и имеются три элемента с весами $\{W/2 + 1, W/2, W/2\}$, этот жадный алгоритм не обеспечит оптимального решения. Также можно провести сортировку по возрастанию веса, а затем проделать то же самое; но этот способ не работает для входных данных вида $\{1, W/2, W/2\}$.

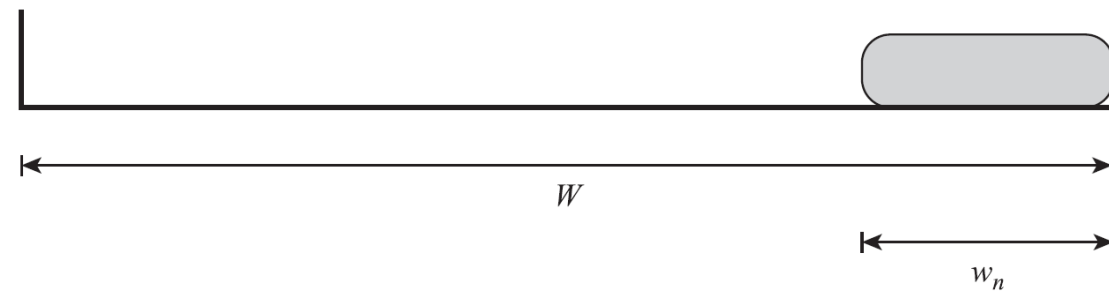
Разработка алгоритма

Неудачная попытка

В случае взвешенного интервального планирования сработала общая стратегия с рассмотрением подзадач, в которых задействованы только первые i заявок. Попробуем применить эту стратегию в данном случае. Лучшее возможное решение, использующее подмножество заявок $\{1, \dots, i\}$, будет обозначаться $\text{OPT}(i)$ (как это делалось ранее). В задаче взвешенного интервального планирования ключевой момент заключался в том, чтобы сосредоточиться на оптимальном решении O нашей задачи и рассмотреть два случая в зависимости от того, принимается или отвергается последняя заявка n этим оптимальным решением. Как и в том случае, одна из частей следует непосредственно из определения $\text{OPT}(i)$.

- Если $n \notin O$, то $\text{OPT}(n) = \text{OPT}(n - 1)$.

Затем необходимо рассмотреть случай, в котором $n \in O$. Хотелось бы иметь простую рекурсию, которая сообщит лучшее возможное значение для решений, содержащих последнюю заявку n . Для взвешенного интервального планирования это было несложно, так как мы могли просто удалить каждую заявку, конфликтующую с n . В текущей задаче дело обстоит сложнее. Из принятия заявки n не следует, что какую-то другую заявку нужно отклонить, — а лишь то, что для подмножества принимаемых заявок $S \subseteq \{1, \dots, n - 1\}$ остается меньше свободного веса: вес w_n используется для принятой заявки n , так что для множества S остальных принимаемых заявок остается только вес $W - w_n$.



Разработка алгоритма

Улучшенное решение

Все это наводит на мысль, что решение потребует больше подзадач: для определения значения $OPT(n)$ необходимо знать не только значение $OPT(n - 1)$, но и лучшее решение, которое может быть получено с использованием подмножества первых $n - 1$ элементов и общего доступного веса $W - w_n$. А следовательно, подзадач будет намного больше: по одной для каждого исходного множества $\{1, \dots, i\}$ элементов и для каждого возможного значения оставшегося доступного веса w . Предположим, W — целое число, а все запросы $i = 1, \dots, n$ имеют целые веса w_i . Тогда подзадача создается для каждого $i = 0, 1, \dots, n$ и каждого целого числа $0 \leq w \leq W$. Для оптимального решения, использующего подмножество элементов $\{1, \dots, i\}$ с максимально допустимым весом w , будет использоваться решение $OPT(i, w)$, то есть

$$OPT(i, w) = \max_S \sum_{j \in S} w_j$$

где максимум определяется по подмножествам $S \subseteq \{1, \dots, i\}$, удовлетворяющим условию $\sum_{j \in S} w_j \leq w$. С этим новым множеством подзадач мы сможем представить значение $OPT(i, w)$ в виде простого выражения, использующего значения меньших подзадач. Более того, $OPT(n, W)$ — значение, которое мы ищем в конечном итоге. Как и прежде, пусть O обозначает оптимальное решение исходной задачи.

- Если $n \notin O$, то $OPT(n, W) = OPT(n - 1, W)$, так как элемент n можно просто игнорировать.
- Если $n \in O$, то $OPT(n, W) = w_n + OPT(n - 1, W - w_n)$, так как теперь ищется вариант оптимального использования оставшейся емкости $W - w_n$ между элементами $1, 2, \dots, n - 1$.

Разработка алгоритма

Когда n -й элемент слишком велик, то есть $W < w_n$, должно выполняться условие $OPT(n, W) = OPT(n - 1, W)$. В противном случае мы получаем оптимальное решение, допускающее все n заявок, выбирая лучший из этих двух вариантов. Применяя аналогичные рассуждения для подзадачи с элементами $\{1, \dots, i\}$ и максимальным допустимым весом w , получаем следующее рекуррентное отношение:

Утверждение 2.8. Если $w < w_i$, то $OPT(i, w) = OPT(i - 1, w)$. В противном случае $OPT(i, w) = \max(OPT(i - 1, w), w_i + OPT(i - 1, w - w_i))$.

Как и прежде, мы хотим создать алгоритм, который строит таблицу всех значений $OPT(i, w)$, вычисляя каждое из них не более одного раза.

Subset-Sum(n, W)

Массив $M[0 \dots n, 0 \dots W]$

Инициализировать $M[0, w] = 0$ для всех $w = 0, 1, \dots, W$

For $i = 1, 2, \dots, n$

For $w = 0, \dots, W$

 Использовать рекуррентное отношение (2.8) для вычисления $M[i, w]$

Конец For

Конец For

Вернуть $M[n, W]$

При помощи (2.8) можно немедленно доказать посредством индукции, что возвращаемое значение $M[n, W]$ является значением оптимального решения для заявок $1, \dots, n$ и доступного веса W .

Анализ алгоритма

n	0													
	0													
	0													
	0													
i	0													
$i-1$	0													
	0													
	0													
	0													
2	0													
1	0													
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	2	$w - w_i$			w	W						

В качестве примера выполнения этого алгоритма рассмотрим экземпляр с предельным весом $W = 6$ и $n = 3$ элементами с размерами $w_1 = w_2 = 2$ и $w_3 = 3$. Оптимальное значение $\text{OPT}(3, 6) = 5$ достигается при использовании третьего элемента и одного из первых двух элементов.

Размер рюкзака $W = 6$, элементы $w_1 = 2, w_2 = 2, w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Исходные значения

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Заполнение значений для $i = 1$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Заполнение значений для $i = 2$

3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Заполнение значений для $i = 3$

Двумерная таблица значений OPT. Крайний левый столбец и нижняя строка всегда содержат 0. Значение $\text{OPT}(i, w)$ вычисляется по двум другим элементам — $\text{OPT}(i - 1, w)$ и $\text{OPT}(i - 1, w - w_i)$ в соответствии со стрелками

Анализ алгоритма

Следующий вопрос — время выполнения алгоритма. Как и прежде в задаче взвешенного интервального планирования, мы строим таблицу решений M и вычисляем каждое из значений $M[i, w]$ за время $O(1)$ с использованием предыдущих значений. Следовательно, время выполнения пропорционально количеству элементов в таблице.

Утверждение 2.9. Алгоритм $\text{Subset-Sum}(n, W)$ правильно вычисляет оптимальное значение для задачи и выполняется за время $O(nW)$.

Обратите внимание: этот метод менее эффективен, чем динамическая программа для задачи взвешенного интервального планирования. В самом деле, его время выполнения не является полиномиальной функцией n ; это полиномиальная функция n и W — наибольшего целого, задействованного в определении задачи. Такие алгоритмы называются псевдополиномиальными. Псевдополиномиальные алгоритмы могут обладать неплохой эффективностью при достаточно малых числах $\{w_i\}$ во входных данных; с увеличением этих чисел их практическая применимость снижается.

Чтобы получить оптимальное множество элементов S , можно выполнить обратное отслеживание по массиву M по аналогии с тем, как это делалось в предыдущих разделах:

Утверждение 2.10. Для таблицы M оптимальных значений подзадач оптимальное множество S может быть найдено за время $O(n)$.

Расширение: задача о рюкзаке

Задача о рюкзаке немного сложнее задачи планирования, которая рассматривалась нами ранее. Возьмем ситуацию, в которой каждому элементу i поставлен в соответствие неотрицательный вес w_i , как и прежде, и отдельное значение v_i . Цель — найти подмножество S с максимальным значением $\sum_{i \in S} v_i$, в котором общий вес множества не превышает W : $\sum_{i \in S} w_i \leq W$.

Алгоритм динамического программирования несложно расширить до этой более общей задачи. Аналогичное множество подзадач $\text{OPT}(i, w)$ используется для представления значения оптимального решения с использованием подмножества элементов $\{1, \dots, i\}$ и максимального доступного веса w . Рассмотрим оптимальное решение O и определим два случая в зависимости от того, выполняется ли условие $n \in O$:

- Если $n \notin O$, то $\text{OPT}(n, W) = \text{OPT}(n - 1, W)$.
- Если $n \in O$, то $\text{OPT}(n, W) = v_n + \text{OPT}(n - 1, W - w_n)$.

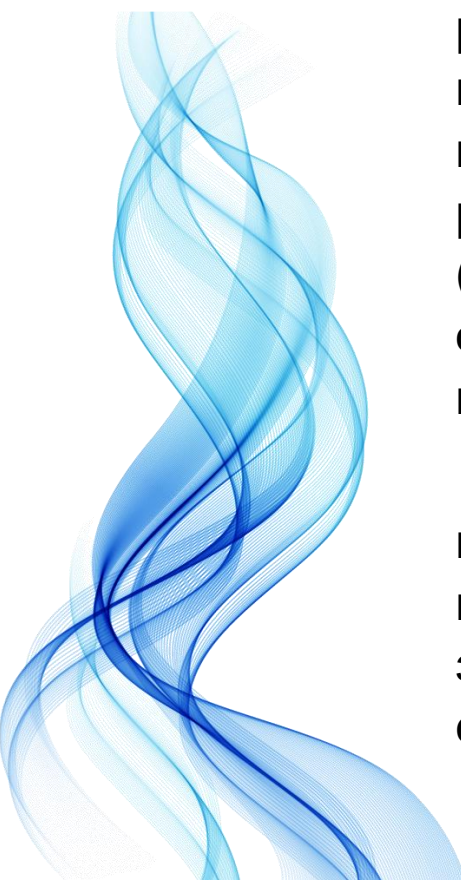
Применение этой аргументации к подзадачам приводит к следующей аналогии с (2.8).

Утверждение 2.11. Если $w < w_i$, то $\text{OPT}(i, w) = \text{OPT}(i - 1, w)$. В противном случае $\text{OPT}(i, w) = \max(\text{OPT}(i - 1, w), v_i + \text{OPT}(i - 1, w - w_i))$.

При помощи этого рекуррентного отношения можно записать полностью аналогичный алгоритм динамического программирования, из чего вытекает следующий факт:

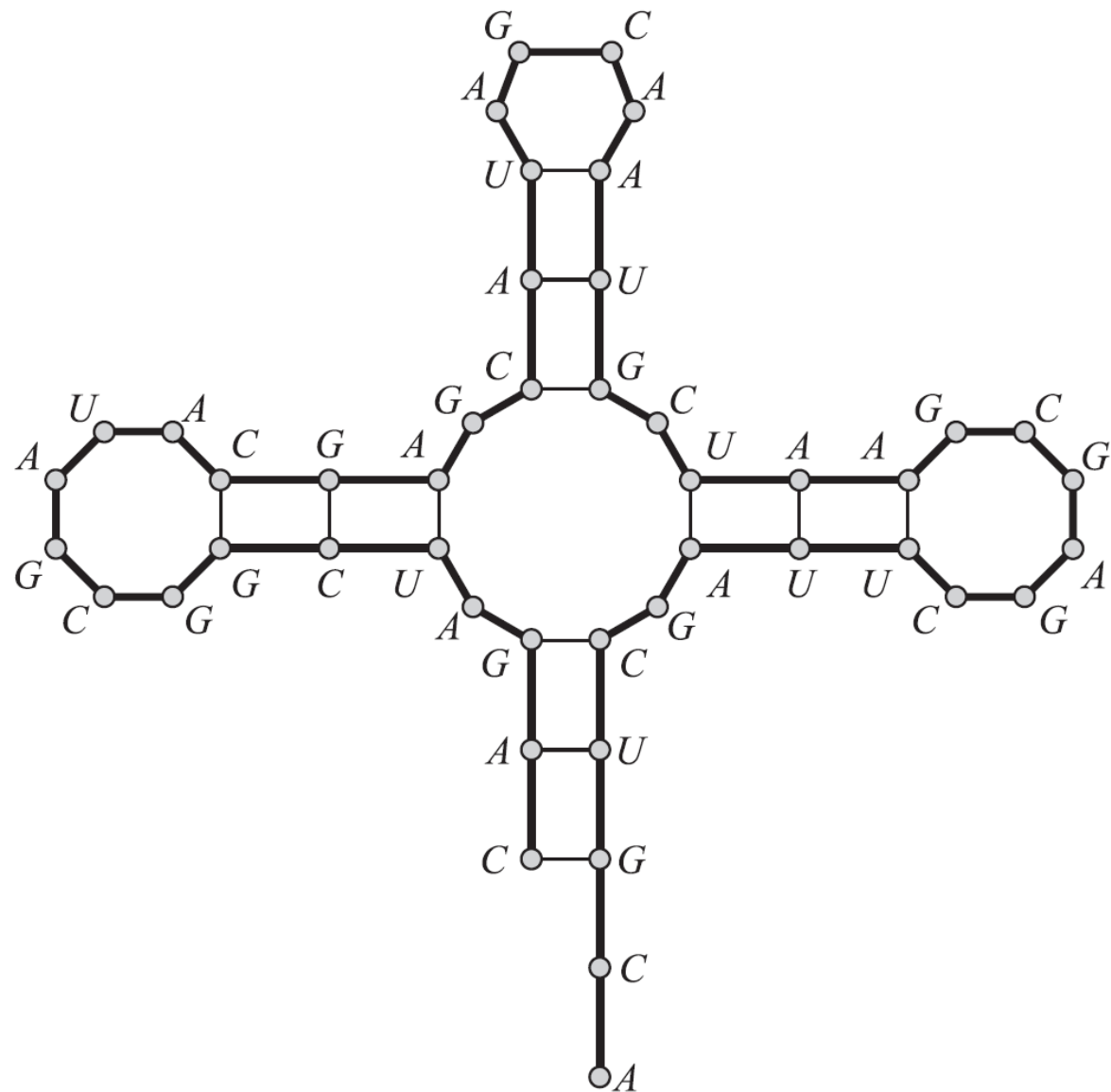
Утверждение 2.12. Задача о рюкзаке решается за время $O(nW)$.

Вторичная структура РНК: динамическое программирование по интервалам



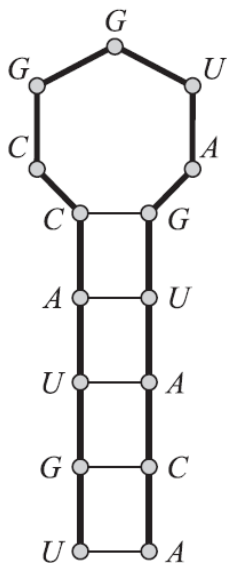
В задаче о рюкзаке нам удалось сформулировать алгоритм динамического программирования при добавлении новой переменной. Существует и другой очень распространенный сценарий, в котором в динамическую программу добавляется переменная. Мы начинаем с рассмотрения множества подзадач $\{1, 2, \dots, j\}$ для всех возможных j , но найти естественное рекуррентное отношение не удастся. Тогда мы рассматриваем большее множество подзадач из $\{i, i + 1, \dots, j\}$ для всех возможных i и j (где $i \leq j$) и находим естественное рекуррентное отношение для этих подзадач. Таким образом, в задачу добавляется вторая переменная i ; в результате рассматривается подзадача для каждого непрерывного интервала в $\{1, 2, \dots, n\}$.

Существует ряд канонических задач, соответствующих этому описанию; студенты, изучавшие алгоритмы разбора для контекстно-независимых грамматик, вероятно, видели хотя бы один алгоритм динамического программирования в этом стиле. А здесь мы сосредоточимся на задаче предсказания вторичной структуры РНК, одной из фундаментальных проблем в области вычислительной биологии.

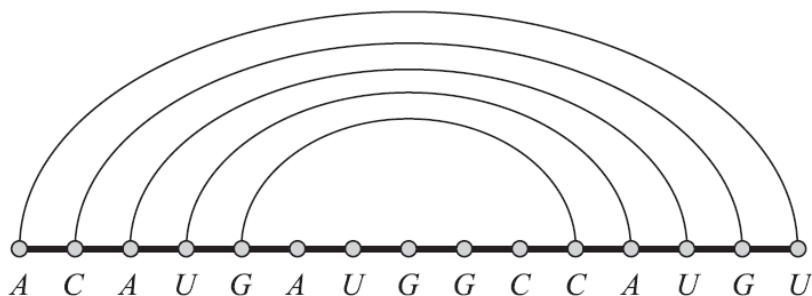


75

Задача



a



b

В первом приближении вторичная структура может моделироваться следующим образом. Как обычно, основание A образует пары с U, а основание C образует пары с G; также требуется, чтобы каждое основание могло образовать пару с не более чем одним другим основанием, иначе говоря, множество пар оснований образует паросочетание. Также вторичные структуры (снова в первом приближении) не образуют узлов, что ниже формально определяется как своего рода условие отсутствия пересечений.

А если более конкретно, вторичная структура V представляет собой множество пар $S = \{(i, j)\}$, где $i, j \in \{1, 2, \dots, n\}$, удовлетворяющих следующим условиям:

- (i) (Отсутствие крутых поворотов) Концы каждой пары в S разделяются минимум четырьмя промежуточными основаниями; иначе говоря, если $(i, j) \in S$, то $i < j - 4$.
- (ii) Любая пара в S состоит из элементов $\{A, U\}$ или $\{C, G\}$ (в произвольном порядке).
- (iii) S является паросочетанием; никакое основание не входит более чем в одну пару.
- (iv) (Отсутствие пересечений) Если (i, j) и (k, l) — две пары в S , то невозможна ситуация $i < k < j < l$.

Разработка и анализ алгоритма

Динамическое программирование: первая попытка

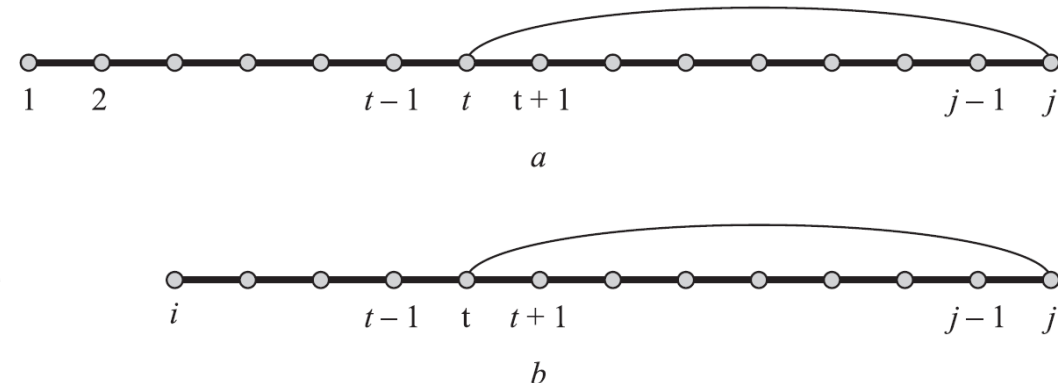
Вероятно, первая естественная попытка применения динамического программирования будет основана на следующих подзадачах: мы говорим, что $OPT(j)$ — максимальное количество пар оснований во вторичной структуре b_1, b_2, \dots, b_j . Согласно приведенному выше запрету на резкие повороты, мы знаем, что $OPT(j) = 0$ для $j \leq 5$; также известно, что $OPT(n)$ — искомое решение.

Проблемы начинаются тогда, когда мы пытаемся записать рекуррентное отношение, выражающее $OPT(j)$ в контексте решений меньших подзадач. Здесь можно пройти часть пути: в оптимальной вторичной структуре b_1, b_2, \dots, b_j возможно одно из двух:

- j не участвует в паре; или
- j участвует в паре с t для некоторого $t < j - 4$.

В первом случае необходимо обратиться к решению для $OPT(j - 1)$. Второй случай изображен на рисунке, а; из-за запрета на пересечения мы знаем, что не может существовать пары, один конец которой находится между 1 и $t - 1$, а другой — между $t + 1$ и $j - 1$. Таким образом, мы фактически выделили две новые подзадачи: для оснований $b_1 b_2 \dots b_{t-1}$ и для оснований $b_{t+1} \dots b_{j-1}$.

Включение пары (i, j) порождает две независимые подзадачи



Первая решается как $OPT(t - 1)$, а вторая в наш список подзадач не входит, потому что она не начинается с b_1 . Это соображение наводит на мысль о том, что в задачу следует добавить переменную. Нужно иметь возможность работать с подзадачами, не начинающимися с b_1 ; другими словами, необходимо иметь возможность рассматривать подзадачи для $b_i b_{i+1} \dots b_j$ при любых $i \leq j$.

Разработка и анализ алгоритма

Динамическое программирование по интервалам

После принятия этого решения приведенные выше рассуждения ведут прямо к успешной рекурсии. Пусть $OPT(i, j)$ — максимальное количество пар оснований во вторичной структуре $b_i b_{i+1} \dots b_j$. Запрет на резкие повороты позволяет инициализировать $OPT(i, j) = 0$ для всех $i \geq j - 4$.

(Для удобства записи мы также разрешим ссылки $OPT(i, j)$ даже при $i > j$; в этом случае значение равно 0).

Теперь в оптимальной вторичной структуре $b_i b_{i+1} \dots b_j$ существуют те же альтернативы, что и прежде:

- j не участвует в паре; или
- j находится в паре с t для некоторого $t < j - 4$.

В первом случае имеем $OPT(i, j) = OPT(i, j - 1)$. Во втором случае, порождаются две подзадачи $OPT(i, t - 1)$ и $OPT(t + 1, j - 1)$; как было показано выше, эти две задачи изолируются друг от друга условием отсутствия пересечений. Следовательно, мы только что обосновали следующее рекуррентное отношение:

Утверждение 2.13. $OPT(i, j) = \max(OPT(i, j - 1), \max_t (1 + OPT(i, t - 1) + OPT(t + 1, j - 1)))$, где \max определяется по t , для которых b_t и b_j образуют допустимую пару оснований (с учетом условий (i) и (ii) из определения вторичной структуры).

Теперь нужно убедиться в правильном понимании порядка построения решений подзадач. Из (2.13) видно, что решения подзадач всегда вызываются для более коротких интервалов: тех, для которых $k = j - i$ меньше. Следовательно, схема будет работать без каких-либо проблем, если решения строятся в порядке возрастания длин интервалов.

Разработка и анализ алгоритма

Теперь нужно убедиться в правильном понимании порядка построения решений подзадач. Из (2.13) видно, что решения подзадач всегда вызываются для более коротких интервалов: тех, для которых $k = j - i$ меньше. Следовательно, схема будет работать без каких-либо проблем, если решения строятся в порядке возрастания длин интервалов.

Инициализировать $OPT(i, j) = 0$ для всех $i \geq j - 4$

For $k = 5, 6, \dots, n - 1$

For $i = 1, 2, \dots, n - k$

 Присвоить $j = i + k$

 Вычислить $OPT(i, j)$ с использованием рекуррентного отношения из (2.13)

Конец For

Конец For

Вернуть $OPT(1, n)$

Найти границу для времени выполнения несложно: существуют $O(n^2)$ подзадач, которые необходимо решить, а вычисление рекуррентного отношения в (2.13) занимает время $O(n)$ для каждой задачи. Следовательно, время выполнения будет равно $O(n^3)$.

РНК последовательность ACCGGUAGU

4	0	0	0	
3	0	0		
2	0			
i = 1				

j = 6 7 8 9

Исходные значения

4	0	0	0	0
3	0	0	1	
2	0	0		
i = 1	1			

j = 6 7 8 9

Заполнение значений для $k = 5$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	
i = 1	1	1		

j = 6 7 8 9

Заполнение значений для $k = 6$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	

j = 6 7 8 9

Заполнение значений для $k = 7$

4	0	0	0	0
3	0	0	1	1
2	0	0	1	1
i = 1	1	1	1	2

j = 6 7 8 9

Заполнение значений для $k = 8$

Выравнивание последовательностей

Задача

Словари в Интернете становятся все более удобными: часто бывает проще открыть сетевой словарь по закладке, чем снимать бумажный словарь с полки. К тому же многие электронные словари предоставляют возможности, отсутствующие в бумажных словарях: если вы ищете определение и введете слово, отсутствующее в словаре (например, *osurrance*), словарь поинтересуется: «Возможно, вы имели в виду *occurrence*?» Как он это делает? Он действительно знает, что происходит у вас в голове?

В начале 1970-х годов молекулярные биологи Нидлман и Вунш предложили определение сходства, которое практически в неизменном виде стало стандартным определением, используемым в наши дни.

Допустим, имеются две строки X и Y ; строка X содержит последовательность символов x_1, x_2, \dots, x_m , а строка Y — последовательность символов y_1, y_2, \dots, y_n . Рассмотрим множества $\{1, 2, \dots, m\}$ и $\{1, 2, \dots, n\}$, представляющие разные позиции в строках X и Y , и рассмотрим паросочетание этих множеств. Паросочетание M этих двух множеств называется выравниванием при отсутствии «пересекающихся» пар: если $(i, j), (i', j') \in M$ и $i < i'$, то $j < j'$. На интуитивном уровне выравнивание предоставляет способ параллельного расположения двух строк: оно сообщает, какие пары позиций будут находиться друг напротив друга. Например,

stop

-tops

соответствует выравниванию $\{(2, 1), (3, 2), (4, 3)\}$.

Выравнивание последовательностей

Задача

Наше определение сходства будет основано на нахождении оптимального выравнивания между X и Y по следующему критерию. Предположим, M — заданное выравнивание между X и Y .

- Во-первых, должен существовать параметр $\delta > 0$, определяющий штраф за разрыв. Для каждой позиции X или Y , не имеющей пары в M (то есть разрыва), вводится штраф δ .
- Во-вторых, для каждой пары букв p, q в нашем алфавите существует штраф за несоответствие α_{pq} за совмещение p с q . Таким образом, для всех $(i, j) \in M$ совмещение x_i с y_j приводит к выплате соответствующего штрафа за несоответствие $\alpha_{x_i y_j}$. В общем случае предполагается, что $\alpha_{pp} = 0$ для любого символа p , то есть совмещение символа со своей копией не приводит к штрафу за несоответствие — хотя это предположение не является обязательным для каких-либо дальнейших рассуждений.
- Стоимость выравнивания M вычисляется как сумма штрафов за разрывы и несоответствия. Требуется найти выравнивание с минимальной стоимостью.

В литературе по биологии процесс минимизации стоимости часто называется выравниванием последовательностей. Величины δ и $\{\alpha_{pq}\}$ представляют собой внешние параметры, которые должны передаваться программе для выравнивания последовательностей; и действительно, значительная часть работы связана с выбором значений этих параметров. При разработке алгоритма выравнивания последовательностей мы будем рассматривать их как заданные извне. Возвращаясь к первому примеру, обратите внимание на то, как эти параметры определяют, какое из выравниваний *occurrence* и *occurrence* следует предпочесть: первый вариант строго лучше других в том и только в том случае, если $\delta + \alpha_{ae} < 3\delta$.

Разработка алгоритма

Для оценки сходства между строками X и Y имеется конкретное числовое определение: это минимальная стоимость выравнивания между X и Y . Чем ниже эта стоимость, тем более похожими объявляются строки. Теперь обратимся к задаче вычисления этой минимальной стоимости и оптимального выравнивания, которое обеспечивает эту стоимость для заданной пары строк X и Y . Для решения этой задачи можно попытаться применить динамическое программирование, руководствуясь следующей базовой дихотомией.

- В оптимальном выравнивании M либо $(m, n) \in M$, либо $(m, n) \notin M$. (То есть два последних символа двух строк либо сопоставляются друг с другом, либо нет.)

Самого по себе этого факта недостаточно для того, чтобы предоставить решение методом динамического программирования. Однако предположим, что он дополняется следующим базовым фактом:

Утверждение 2.14. Пусть M — произвольное выравнивание X и Y . Если $(m, n) \notin M$, то либо m -я позиция X , либо n -я позиция Y не имеют сочетания в M .

Доказательство. Действуя от обратного, предположим, что $(m, n) \notin M$ и существуют числа $i < m$ и $j < n$, для которых $(m, j) \in M$ и $(i, n) \in M$. Но это противоречит нашему определению выравнивания: имеем $(i, n), (m, j) \in M$ с $i < m$, но $n > j$, поэтому пары (i, n) и (m, j) пересекаются. ■

Существует эквивалентный вариант записи (2.14) с тремя альтернативными возможностями, приводящий напрямую к формулировке рекуррентного отношения.

Разработка алгоритма

Утверждение 2.15. В оптимальном выравнивании M истинно хотя бы одно из следующих трех условий:

- (i) $(m, n) \in M$; или
- (ii) m -я позиция X не имеет сочетания; или
- (iii) n -я позиция Y не имеет сочетания.

Теперь пусть $OPT(i, j)$ обозначает минимальную стоимость выравнивания между $x_1x_2 \cdots x_i$ и $y_1y_2 \cdots y_j$. Если выполняется условие (i) из (2.15), мы платим и выравниваем $x_1x_2 \cdots x_{m-1}$ с $y_1y_2 \cdots y_{n-1}$ настолько хорошо, насколько это возможно; получаем $OPT(m, n) = \alpha_{x_my_n} + OPT(m-1, n-1)$. Если выполняется условие (ii), мы платим штраф за разрыв δ , потому что m -я позиция X не имеет сочетания, и выравниваем $x_1x_2 \cdots x_{m-1}$ с $y_1y_2 \cdots y_{n-1}$ настолько хорошо, насколько это возможно. В этом случае получаем $OPT(m, n) = \delta + OPT(m-1, n)$. Аналогичным образом для случая (iii) получаем $OPT(m, n) = \delta + OPT(m, n-1)$.

Используя аналогичные рассуждения для подзадачи нахождения выравнивания с минимальной стоимостью для $x_1x_2 \cdots x_i$ и $y_1y_2 \cdots y_j$, приходим к следующему факту:

Утверждение 2.16. Стоимости минимального выравнивания удовлетворяют следующему рекуррентному отношению для $i \geq 1$ и $j \geq 1$:

$$OPT(i, j) = \min[\alpha_{x_iy_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)].$$

Кроме того, (i, j) принадлежит оптимальному выравниванию M для этой подзадачи в том и только в том случае, если минимум достигается на первом из этих значений.

Мы добрались до точки, в которой алгоритм динамического программирования стал ясен: мы строим значения $OPT(i, j)$, используя рекуррентное отношение в (2.16). Существуют только $O(mn)$ подзадач, и $OPT(m, n)$ дает искомое значение.

Разработка алгоритма

Теперь определим алгоритм для вычисления значения оптимального выравнивания. Для выполнения инициализации заметим, что $OPT(i, 0) = OPT(0, i) = i\delta$ для всех i , так как совместить i -буквенное слово с 0-буквенным словом возможно только с использованием i разрывов.

Alignment(X, Y)

Массив $A[0...m, 0...n]$

Инициализировать $A[i, 0] = i\delta$ для всех i

Инициализировать $A[0, j] = j\delta$ для всех j

For $j = 1, \dots, n$

For $i = 1, \dots, m$

 Использовать рекуррентное отношение из (6.16) для вычисления $A[i, j]$

Конец For

Конец For

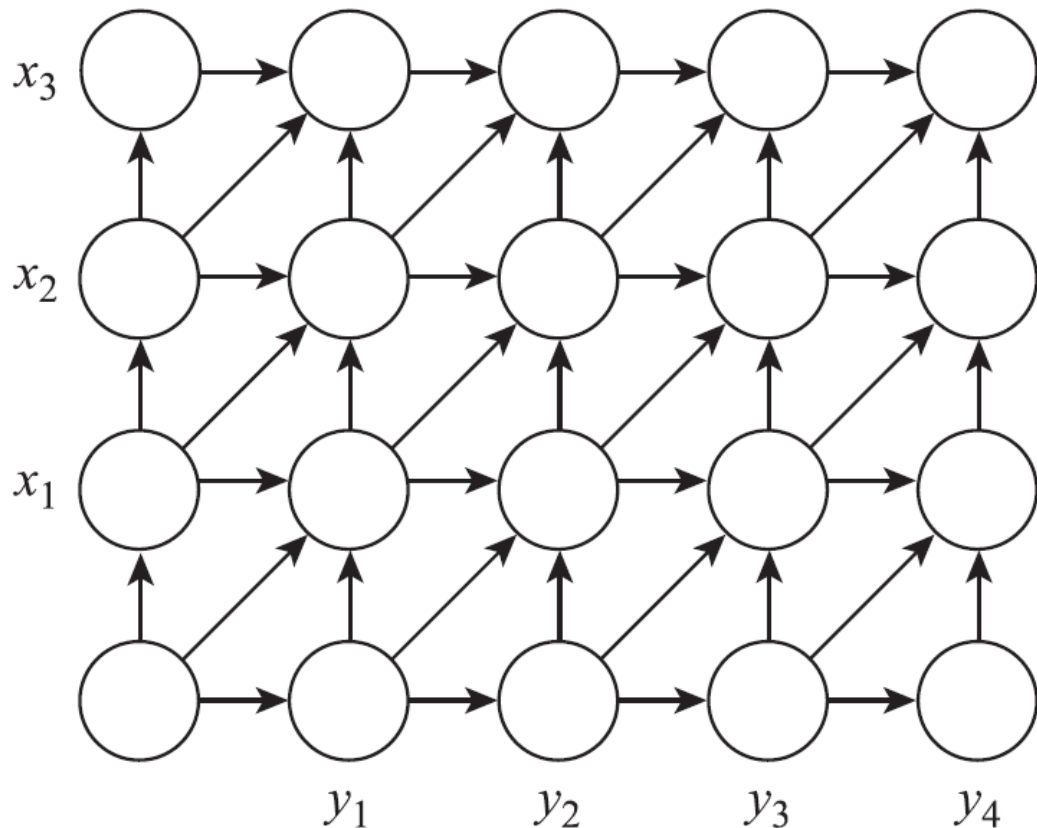
Вернуть $A[m, n]$

Как и в случае с предыдущими алгоритмами динамического программирования, построение самого выравнивания осуществляется обратным отслеживанием по массиву A с использованием второй части факта (2.16).

Анализ алгоритма

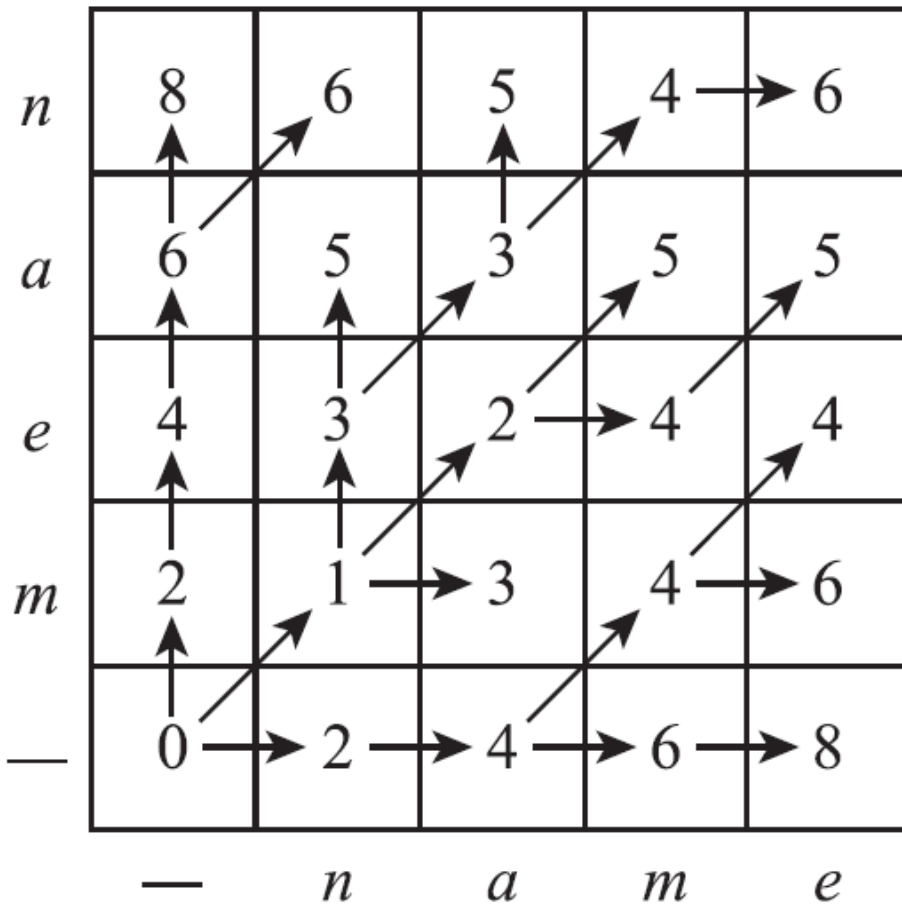
Правильность алгоритма следует непосредственно из (2.16). Время выполнения равно $O(mn)$, так как массив A содержит только $O(mn)$ элементов и в худшем случае на каждый тратится постоянное время.

У алгоритма выравнивания последовательностей существует элегантное визуальное представление. Допустим, имеется двумерный решетчатый граф GXY с размерами $m \times n$; строки помечены символами X , столбцы помечены символами Y , а ребра направлены так, как показано на рисунке



Строки нумеруются от 0 до m , а столбцы от 0 до n ; узел на пересечении i -й строки и j -го столбца обозначается (i, j) . Ребрам GXY назначаются стоимости: стоимость каждого горизонтального и вертикального ребра равна δ , а стоимость диагонального ребра из $(i - 1, j - 1)$ в (i, j) равна $\alpha_{x_i y_j}$.

Анализ алгоритма



Значения OPT для задачи
выравнивания слов mean и name

Теперь смысл этой схемы становится понятным: рекуррентное отношение для $OPT(i, j)$ из (2.16) в точности соответствует рекуррентному отношению для пути минимальной стоимости от $(0, 0)$ до (i, j) в GXY. Следовательно, мы можем показать

Утверждение 2.17. Пусть $f(i, j)$ — стоимость минимального пути из $(0, 0)$ в (i, j) в GXY. Тогда для всех i, j выполняется условие $f(i, j) = OPT(i, j)$.

Доказательство. Утверждение легко доказывается индукцией по $i + j$. Если $i + j = 0$, значит, $i = j = 0$, и тогда $f(i, j) = OPT(i, j) = 0$.

Теперь рассмотрим произвольные значения i и j и предположим, что утверждение истинно для всех пар (i', j') с $i' + j' < i + j$. Последнее ребро кратчайшего пути к (i, j) проходит либо из $(i - 1, j - 1)$, либо из $(i - 1, j)$, либо из $(i, j - 1)$. Следовательно, имеем

$$\begin{aligned}
 f(i, j) &= \min[\alpha_{x_i y_j} + f(i - 1, j - 1), \delta + f(i - 1, j), \delta + f(i, j - 1)] \\
 &= \min[\alpha_{x_i y_j} + OPT(i - 1, j - 1), \delta + OPT(i - 1, j), \delta + OPT(i, j - 1)] \\
 &= OPT(i, j),
 \end{aligned}$$

Переход от первой строки ко второй осуществляется по индукционной гипотезе, а переход от второй к третьей — по (2.16).