Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: И. А. Мариничев
Преподаватель: Н. С. Капралов
Группа: М8О-208Б

Дата: 8.04.21
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №5

Задача: Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из выходных строк, необходимо воспользоваться полученным суффисным деревом для решения своего варианта задания.

Алфавит строк: строчные буквы латинского алфавита (т.е. от a до z).

Вариант: Найти самую длинную общую подстроку двух строк.

Формат входных данных

Две строки.

Формат результата

На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

1 Описание

Согласно [1], алгоритм Укконена за $O(m)$, где m - длина входной строки, достигается последовательными дополнениями и улучшениями наивного алгоритма построения суффиксного дерева за $O(m^3)$.

Сперва поговорим о правилах расширения или дополнения:

1. Если конец данного суффикса заканчивается в листе, продлить данный лист этим символом.

2. Если пусть $S[j..i]$ заканчивается не в листе и нет продолжения, создается новый потомок лист, если же данный суффикс заканчивается не в конце, а в середине данного, создается внутренняя вершина, также как и новый лист, который является разницей между одинаковой частью и отличной.

3. Если суффикс $S[j..i]$ заканчивается не в листе, допустим сейчас мы в символе $S[i]$ а символ $S[i+1]$ уже присутствует, ничего делать не нужно

Также введем понятие **суффиксной ссылки**: если суффикс помеченый $x\alpha$ заканчивается в данной вершине, где x - какой то символ, а α строка, возможно пустая, суффиксная ссылка указывается на вершину, в которой заканчивается суффикс α на единицу меньший по длинне. Они пригодятся нам для быстрого перехода по вершинам при добавлении, вместо того чтобы идти от корня в поиске очередного суффикса.

Скачок по счетчику (skip/count): чтобы не сравнивать всю строчку с уже имеющейся в дереве при спуске, можно делать проще, зная длину подстроки γ например, проверять, если полная длинна больше чем длина имеющейся в вершине подстроки, можно сразу пропускать всю это подстроку, так как при поиске мы будем двигаться от добавления наибольшего суффикса к наименьшему, это поможет нам пробегать вниз после перехода по суффиксной ссылке, делая данный пробег не за колличество символов в подстроке, а за кол-во промежуточных вершин в пути, что почти константно.

Что порождает трюк 1: если путь больше длины ребра, перескочи дальше.

Еще один важный пункт, **сжатие дуговых меток** - позволяет хранить в вершинах не копии подстрок, а всего лишь указатели на итераторы строки, или же начало и конец подстроки в вершине, где начало и конец указывают с какой буквы входной строчки и по какую вершина хранит подстроку.

Наблюдение 1: Правило 3 - остановка процесса.

Если в итерации мы встретили правило 3, следовательно прододлжать дальнейшую вставку текущего символа бессмысленно, так как все дальнейшие суффиксы уже хранят данный суффикс.

Отсюда трюк 2: остановка процесса, сразу как происходит правило 3.

Наблюдение 2: Однажды лист, всегда лист.

Таким образом, проще всего при создании листа, сразу продлить его до конца строки, так как алгоритм не содерджит в себе каких-то модификаций листов.

Отсюда трюк 3: Создание глобальной переменной для концов листа.

Если раньше в фазе $i+1$, когда создается листовая дуга, которая нормально помечается подстрокой $S[p..i+1]$, вместо записи индеков этой дуги $(p, i+1)$ использовать (p, end), где end – указатель на глобальную переменную, которая увеличивается на 1 каждую итерацию, продлевая за $O(1)$, все листы.

Таким образом сейчас алгоритм выглядит так:

По правилу 1 продлеваем за константу все листы, дальше по правилу 2 добавляем все суффиксы которые можем, если встречаем правило 3, сразу выходим и начинаем добавлять новый элемент.

Введем понятие $activeNode$, $activeEdge$, $activeLength$, что образует собой в тройке $activePoint$.

1. **activePoint**: это может быть корень, любая внутренняя вершина или любая точка внутри вершины. Это указатель, который показывает, откуда алгоритм начнет свой путь при начале каждой итерации. Для начала activePoint установлен на корень. Все другие итерации будут ставить activePoint на правильное место, основываясь на предыдущей итерации (исключение APCFALZ о нем поговорим ниже) и обязанность текущей итерации менять activePoint в конце каждой итерации для использования в дальнейшем, где правило 2 или 3 будут применяться (в текущей или следующей фазе).

   Следовательно нам надо понять как хранить activePoint. Мы храним их как три переменные: activeNode, activeEdge, activeLength.

2. **activeNode**: указатель на вершину, которой может быть корень или любая внутренняя вершина.

3. **activeEdge**: когда мы находимся в какой то вершине, мы должны знать по какой букве нам надо идти вниз, если идти все таки приходится. activeEdge будет хранить эту информацию. Пример, activeNode это вершина с которой мы начинаем путь, в то время как activeEdge будет поставлен на следующий добавляющийся символ.

4. **activeLength**: показывает сколько букв нам надо пройти вниз (по пути который говорит нам activeEdge) из activeNode чтобы достигнуть activePoint откуда алгоритм начнет путь. Как пример, если activeNode это откуда мы начнем путь, то activeLength будет равна нулю.

После фазы i, если у нас j листовых вершин тогда в фазе i+1, первые j расширений будут сделаны за константу с помощью трюка 3. activePoint нужна для расширений с j+1 по i+1 и activePoint может быть изменена (или не изменена) на основании предыдущих действий или этапов.

activePoint change for extension rule 3 (APCFER3) [4]:

Когда применяется правило 3 в любой фазе i, прежде чем пойти в фазу i+1, увеличиваем activeLength на 1. Никаких изменений activeNode и activeEdge. Почему? Потому что после правила 3, текущий символ из строки S совпадает с путем в текущем activePoint, т.е. для будущего расширения activePoint, activeNode и activeEdge остаются теми же, только activeLength увеличивается на 1 (потому что символ совпал). Данный activePoint (тот же node, тот же edge и увеличенная length) будут использованы в фазе i+1.

activePoint change for walk down (APCFWD) [4]:

activePoint изменится на основании применимого правила расширения. activePoint также может меняться когда мы спускаемся по ребрам. Представим activePoint как (N, s, 6), если длина текущего activeNode меньше чем activeLength, тогда идем по activeEdge до тех пор пока не упремся в внутреннюю вершину где длина вершины будет меньше нашей activeLength, важно заметить, пока спускаемся, вычитаем из activeLength длину вершины, а также меняем activeEdge на следующий на длину вершины. Нам надо как можно более сократить поиск начала отсчета для будущией итерации. Что делать если мы не встретили никакой внутренней вершины по пути, следовательно, мы остаемся на месте, делать нам больше ничего не нужно.

activePoint change for Active Length ZERO (APCFALZ) [4]:

Если activeLength равна нулю, то пускай следующая буква, которую мы читаем это s, мы меняем наш activeEdge на данный символ, и никуда не идем, так как activeLength равна нулю и спускаться никуда не надо.

В коде мы будем пробегаться по символам строки, добавляя новые суффиксы один за другим. Каждый цикл, пока число оставшихся суффиксов больше 0, для фазы i мы будем добавлять суффиксы с буквой i. Мы будем добавлять не не все суффиксы (первые несклько добавит трюк 3, расширяя наши листы, а также правило 3, которое обрывает наш цикл, также что касается того если кол-во оставшихся к добавлению суффиксов не равно нулю к концу итерации, значит это то, что мы добавили данные суффиксы неявно, что называется неявным суффиксным деревом, а еще то, что мы добавим их явно в следующих итерациях).

Наибольшая общая подстрока двух строк

Классическая задача анализа строк — найти наибольшую подстроку, общую для двух заданных строк S_1 и S_2. Это задача о наибольшей общей подстроке (отличная от задачи о наибольшей общей подпоследовательности).

Эффективный и концептуально простой способ нахождения наибольшей общей подстроки открывается, если построить обобщенное суффиксное дерево для S_1 и S_2. Каждый лист этого дерева представляет собой либо суффикс одной из этих двух строк, либо суффикс их обеих. Пометим каждую внутреннюю вершину v числом 1 (или 2), если в поддереве v существует лист, представляющий собой суффикс строки S_1 (или, соответственно, S_2). Путевая метка любой внутренней вершины с пометкой 1 или 2 есть подстрока, общая для S_1 и S_2, и самая длинная такая строка и есть наибольшая общая подстрока. Так что алгоритм должен только найти вершину, имеющую наибольшую строковую глубину (число символов в пути до нее) среди вершин с пометками 1 и 2. Построение суффиксного дерева может быть выполнено за линейное время (пропорциональное суммарной длине S_1 и S_2), а пометка вершин и вычисления строковой глубины — с помощью стандартных методов обхода дерева, линейных по времени.

В итоге: наибольшая общая подстрока двух строк с использованием обобщенного суффиксного дерева может быть найдена за линейное время.

2 Исходный код

Проект состоит из 3 файлов:

- **main.cpp**: главный файл, в котором происходит считывание строк и их конкатенация с добавлением терминальных символов;

- **Vertex.hpp**: файл, содержащй структуру вершины суффиксного дерева;

- **SuffixTree.hpp**: файл, содержащий структуру суффиксного дерева;

Таблица методов и полей классов

Vertex.hpp	
Методы и поля	Значение
map<char, Vertex *> child	Поле, хранящее всех детей данной вершины (ключ – перый символ ребра, значение – указетль на вершину).
string::iterator begin, end	Поле, хранящее диапазон символов ребра, входящего в данную вершину.
Vertex * suffixLink	Поле, хранящее указатель на вершину по суффиксной ссылке.
set<int> stringNumber	Поле, хранящее множество цифр, характеризующих принадлежность данной вершины к определенной строке.
Vertex::Vertex()	Конструктор вершины.
Vertex:: Vertex()	Деструктор вершины.
SuffixTree.hpp	
Методы и поля	Значение
string pattern	Поле, хранящее конкатенированную строку.
string str2	Поле, хранящее втррую строку.
Vertex * root, * activeNode, * link	Поля, хранящие указатели на корень, текущую вершину и текущую суффиксную связь.
int remaining	Поле, хранящее количество оставшихся несозданных листов.
int activeLen	Поле, хранящее текущее продвижение по ребру.
string::iterator activeEdge	Поле, хранящее текущее ребро, а именно первый символ.

struct LCS	Структура, хранящая параметры наибольшей общей подстроки.
int lengthOfLCS	Поле, хранящее текущую длину наибольшей общей подстроки.
vector <LCS> answers	Вектор, хранящий наибольшие общие подстроки.
LCS ans	Поле, хранящее наибольшую общую подстроку.
TSuffixTree::TSuffixTree()	Контсруктор суффиксного дерева, выполняющий все задание.
void TSuffixTree::Add()	Метод, строяющий суффиксное дерево по алгоритму Укконена.
void TSuffixTree::SuffixLink()	Метод, работающий с созданием суффиксных ссылок и запоминанием вершин.
int TSuffixTree::MarkUp()	Метод, выполняющий обход в глубину и помечающий принадлежность вершины к определенной строке.
void TSuffixTree::FindMaxHeight()	Метод, выполняющий обход в глубину и находящий длину наибольшей общей подстроки.
void TSuffixTree::FindLongestSubstrings()	Метод, выполняющий обход в глубину и находящий все наибольшие общие подстроки.
TSuffixTree:: TSuffixTree()	Деструктор суффиксного дерева.

3 Тест производительности

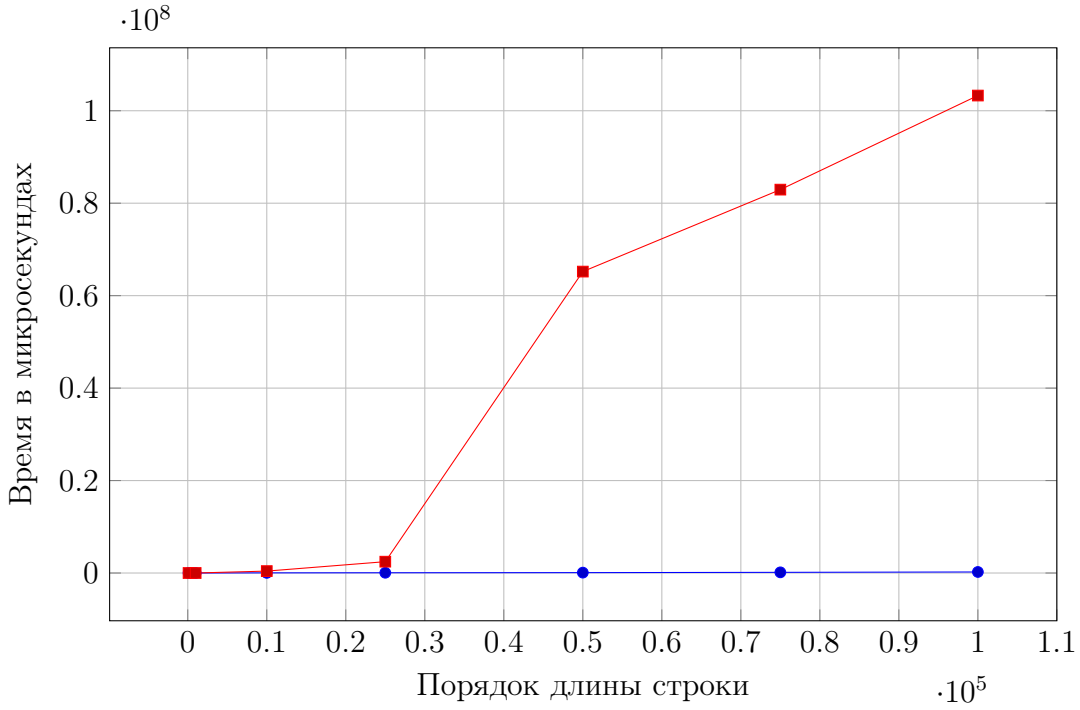
Я решил сравнить свою реализацию алгоритма Укконена с решением при помощи метода динамического программирования. Тестирование происходило на строках с размерами порядка 100, 1000, 10000, 25000, 50000, 75000, 100000.
Время выводится микросекундах. Для измерения времени использовалась бибилиотека **chrono**.

```
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_100.t
Dynamic Programming answer: 53
Time of Dynamic Programming method: 303 microsecond(s)

Ukkonen's Algorithm answer: 53
Time of Ukkonen's Algorithm: 168 microsecond(s)
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_1000.t
Dynamic Programming answer: 152
Time of Dynamic Programming method: 3976 microsecond(s)

Ukkonen's Algorithm answer: 152
Time of Ukkonen's Algorithm: 1579 microsecond(s)
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_10000.t
Dynamic Programming answer: 4193
Time of Dynamic Programming method: 416407 microsecond(s)

Ukkonen's Algorithm answer: 4193
Time of Ukkonen's Algorithm: 33586 microsecond(s)
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_25000.t
Dynamic Programming answer: 7566
Time of Dynamic Programming method: 2453599 microsecond(s)

Ukkonen's Algorithm answer: 7566
Time of Ukkonen's Algorithm: 56532 microsecond(s)
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_50000.t
Dynamic Programming answer: 25014
Time of Dynamic Programming method: 65206836 microsecond(s)
```

```
Ukkonen's Algorithm answer: 25014
Time of Ukkonen's Algorithm: 84897 microsecond(s)
ivan@Laptop-IM:/mnt/c/Users/Иван/projects/da_labs/da_lab5/benchmark$ ./solution
<../tests/test_75000.t
Dynamic Programming answer: 38000
Time of Dynamic Programming method: 103287478 microsecond(s)

Ukkonen's Algorithm answer: 38000
Time of Ukkonen's Algorithm: 145030 microsecond(s)
```

Синий: Поиск наибольшей общей подстроки при помощи суффиксного дерева.

Красный: Поиск наибольшей общей подстроки методом динамического программирования.



Метод динамического программирования можно использовать для поиска самой длинной общей подстроки за время $O(m * n)$. Идея состоит в том, чтобы найти длину самого длинного общего суффикса для всех подстрок обеих строк и сохранить эти длины в таблице.

Решение этой же задачи при помощи построения обобщенного суффиксного дерева по алгоритму Укконена происходит за $O(m + n)$

Тесты создавались с помощью программы на языке Python:

```
1 import string
2 import random
```

```
3
4  ALPHABET = string.ascii_lowercase
5
6  def get_random_text(text_len):
7      return "".join([random.choice(ALPHABET) for _ in range(text_len)])
8
9  NUMBER_OF_LINES = [100, 1000, 10000, 25000, 50000, 75000, 100000]
10
11 length_of_list = len(NUMBER_OF_LINES)
12
13 for enum in range(length_of_list):
14     with open(f'test_{NUMBER_OF_LINES[enum]}.t', 'w') as file:
15         text = get_random_text(NUMBER_OF_LINES[enum])
16         pattern_count = random.randint(1, 9)
17         answer = ""
18         file.write( "{}\n".format(text))
19         for cnt in range(pattern_count):
20             use_real_pattern = random.choice([True, False])
21             if use_real_pattern:
22                 start_pos = random.randint(0, len(text) - 3)
23                 end_pos= random.randint( start_pos+1, len(text) - 1)
24                 pattern = text[start_pos:end_pos]
25             else:
26                 pattern = get_random_text(random.randint(1, 10))
27             answer += pattern
28         file.write( "{}\n".format(answer))
```

4 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я изучил алгоритм Укконена. Познакомился с такой мощной структурой данных, как суффиксное дерево, позволяющей неожиданно эффективно решать множество сложных поисковых задач. К сожалению, известные алгоритмы построения суффиксного дерева (главным образом алгоритм, предложенный Эско Укконеном) достаточно сложны для понимания и трудоёмки в реализации. Лишь относительно недавно, в 2011 году, стараниями Дэни Бреслауэра и Джузеппе Италиано был придуман сравнительно несложный метод построения, который фактически является упрощённым вариантом алгоритма Питера Вайнера – человека, придумавшего суффиксные деревья в 1973 году.

Список литературы

[1] Ден Гасфилд. *Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология* — Издательский дом «Невский Диалект», 2003. Перевод с английского: И. В. Романовский. — 654 с. (ISBN 5-7940-0103-8)

[2] *Suffix Tree using Ukkonen's algorithm*
URL: `https://youtu.be/aPRqocoBsFQ` (дата обращения: 05.04.2021).

[3] *Суффиксное дерево. Алгоритм Укконена*
URL: `https://e-maxx.ru/algo/ukkonen` (дата обращения: 05.04.2021).

[4] *Ukkonen's Suffix Tree Construction – Part 1-4*
URL: `https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/`
(дата обращения: 05.04.2021).

[5] *Suffix Tree Application 5 – Longest Common Substring*
URL: `https://www.geeksforgeeks.org/suffix-tree-application-5-longest-common-substring-2/` (дата обращения: 06.04.2021).