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: А. А. Боглаев
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-22
Дата:
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №5

Задача: Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построить суффиксное дерево для входной строки для решения своего варианта задания.

Вариант: Найти образец в тексте используя статистику совпадений.

Алфавит строк: Строчные буквы латинского алфавита (от a до z).

1 Описание

Требуется реализовать алгоритм Укконена для построения суффиксного дерева за линенйное время.

Согласно [1]: «Алгоритм Укконена строит неявное суффиксное дерево τ_i для каждого префикса $S[1..i]$ строки S, начиная с τ_1 и увеличивая i на единицу, пока не будет построено τ_n. Настоящее суффиксное дерево для S получается из τ_n, и вся работа требует времени $O(n)$.»

Начинают знакомство с алгоритмом Укконена с наивной реазизации, имеющей сложность $O(n^3)$. А после пытаются ускорить алгоритм, чтобы сложность стала $O(n)$.

В чем же проблема алгоритма за $O(n^3)$?

Вместо того, чтобы вставить все суффиксы строки, происходит вставка суффиксов всех префиксов строки. Для поиска и вставки элемента каждый раз осуществляем проход из корня. Алгоритм не эффективен по памяти, так как на ребре хранятся символы строки.

Для ускорения алгоритма будем строить все суффиксы строки, чтобы избежать большого числа повторений, как это было в алгоритме за $O(n^3)$.

Допустим, в суффиксном дереве есть строка $x\alpha$, где x - первый символ строки, а α - оставшаяся подстрока (возможно пустая). Значит в суффиксном дереве есть и строка α. Пусть для строки $x\alpha$ есть внутренняя вершина v, тогда для строки α существует внутреняя вершина p. Значит из v в p поможно постоить «путь», который называется суффиксной ссылкой. В программной реализации суффиксная ссылка будет являться указателем из v в p.

Суффиксные ссылки позволяют перемещаться по дереву быстрее, так как нам известно, какие символы будут находится на ребре до внутренней вершины, значит можно не делать проход из корня (можно не делать повторное сравнение). Чтобы построить суффиксную ссылку, нужно хранить указатель на последную внутренню вершину, и, когда мы создадим новую внутреннюю вершину, необходимо связать ее с последней.

Но это еще не все, для ускорения алгоритма, надо избегать повторных сравнений.

Чтобы перейти по суффиксной сслыке, нужно перейти по ребру до ближайшей внутренней вершины, а после пройти такое же количество символов, после перемещения по ссылке. Избежать лишних сравнений помогут «прыжки по счётчику». Будем считать, сколько символов на ребре было пройдено, до перехода по суффиксной ссылке, и столько же символов «перепрыгнем» после перехода.

Теперь сложность алгоритма будет составлять $O(n^2)$. Причиной всему память, которая затрачивается на хранение символов на ребре. На данных момент на ребре хранися подстрока. Значит символы идут подряд. Тогда можно хранить только индекс начала и конца подстроки на каждом ребре. При добавлении нового символа на

ребро будем увеличивать индекс конца подстроки для всего дерева. Такое «продление» суффиксов в среднем работает за O(1).

Если использовать все вышеописанные эвристики, то алгоритм Укконена будет иметь сложность O(n).

2 Исходный код

Для реализации аллгоритма Укконена опишем необходимо написать класс для суффиксного дерева $TSuffixTree$. Дерево состоит из узлов, значит напишим класс для узла $TNode$. Также у дерева есть параметры, котрые мы должны хранить для правильного построения. Напишем структуру для хранения параметров $TreeData$.

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>


const char SENTINEL = '$';
const int TERMPOS = -1;
const int TERMVAL = 0;
const bool ISLEAF = true;


class TSuffixTree {
   private:
       class TNode {
           public:
               int begin; // start curve
               int *end; // end curve, ptr, for change
               TNode *suffix_link; // suffix link
               bool is_leaf; // node status
               std::unordered_map<char, TNode *> child; // child

               TNode(int start, int *finish, TNode* s_link, bool leaf); // node
                   constructor
       };

       struct TreeData {
           TNode *current_node;
           int current_index;
           int jump_counter;
           int plannedSuffixs;
           TNode *last_inner_node;
       };


       TNode *root;
       std::string str;
       int suffixTreeEnd;
       TreeData params;

       void CreateTree();
       void AddSuffix(int position);
```

```
42          void DestroyTree(TNode *node);
43          int CurveLength(TNode *node);
44          void SplitCurve(TNode *node, int position);
45
46      public:
47          TSuffixTree(std::string &input_str);
48          ~TSuffixTree();
49          void MatchStatistic(std::vector<int> &ms, const std::string &str);
50  };
```

Стоит отметить, что строение узла суффиксного дерева, параметры дерева и ряд методов (разделение ребра, подсчет длины подстроки на ребре) доступны только внутри класса. Вне класса пользователю доступен только конструктор с деструктором и метод для подсчета статистики совпадений.

Для начала определим, что информация о ребре находится в узле, в который оно приходит.

Также заметим, что в классе, описывающем узел, есть статус узла, он принимает два значения ($true$, $false$). Так мы сможем отличить внутренний узел от листового. Конечно, вместо булевых значений можно использовать целочисленные и внутреннему узлу присвоить занчение -1, а листовому номер суффикста в строке.

Корень является внутренней вершиной, его статус будет равен $false$. У корня есть только исходящие ребра, значит позиции начала и конца подстроки будет равно -1.

Опишем методы для класса суффиксного дерева. Так как полный листинг методов займет много места, опимем их в таблице ниже.

suffix_tree.hpp	
void CreateTree();	Создание и построение дерева
void AddSuffix(int position)	Добавление нового суффикса в дерево
void DestroyTree(TNode *node)	Удаление дерева
int CurveLength(TNode *node)	Подсчет длины подстроки на ребре
void SplitNode(TNode *node, int position)	Разделение ребра на два
TSuffixTree(std::string &input_str)	Конструктор для суффиксного дерева
TSuffixTree()	Деструктор дерева
void MatchStatistic(std::vector<int> &ms, const std::string &str)	Подсчет статистики совпадений

При создании дерева в конструкторе будет вызван метод $CreateTree$. Но перед этим, чтобы не копировать входную строку в переменную класса str, переместим ее. Так будет затрачено меньше памяти на построение суффиксного дерева.

Конец подстроки на ребре удобно сделать указателем, так можно удобно добавить символ на каждое ребро.

Метод разделения ребра $SplitCurve$ создает внутреннюю вершину и «прикрепляет» к ней два листа, старый и новый. Также не забываем прикрепить суффиксную ссылку, если она есть.

Основной метод для конструирования дерева это $AddSuffix$. Тут надо воссоздать все эвристики, которые были описаны выше.

В этой функции ведется подсчет суффиксов, которые нужно создать. В качетсве такого счетчика выступает переменная $plannedSuffixs$ из структуры $TreeData$. В обычном случае эта пременная сначала увеличивается на 1, а в конце метода уменьшается. Но, если символ, который мы вставляем уже есть, то происходит выход из функции и значение счетчика не уменьшается. Это нужно для последующего прохода по остальным суффиксам и деления ребер.

Основная логика прописана в цикле. Тут мы пытаемся найти новый символ в дереве. Если его нет, то создаем новый лист. Но если символ есть (точнее такой путь), то нужно проверить, есть ли он на ребре, если да, то по правилу 3 в [1], ничего не делаем. Чтобы сделать эффективную реализаию, будем увеличивать счетчик свопадений $jump\_counter$.

Если такой путь не нашли, то считаем длину ребра и проверяем, что больше: счетчик прыжков или длина ребра. Если счетчик стал больше, то нужно его уменьшить на длину ребра, а текущию позицию увеличить. И, соответсвенно, перейти на другое ребро.

После всех преобразований остается разделить ребра, чтобы сохранялось свойство $compactTrie$.

Далее важно понять, в каком узле мы находимся, чтобы подготовиться к новой итерации. Если мы в корне, то увеличим текущую позицию и уменьшим счетчик совпадений. Важно не забыть, что количетсво суффиксов стало меньше, значит уменьшим значение счетчика суффиксов.

Если же оставновились не в корневом узле, то нужно прейти по суффиксной ссылке.

Как сказано в [1]: «Определим $ms[i]$ как длину наибольшей подстроки , начинающейся с позиции i, которая совпадает где-то (но мы не знаем, где) с подстрокой . Эти значения называются статистикой совпадений.»

В данной реализации был создан вектор ms длины текста, куда будет записываться статистика совпадений. Статистика совпадений это тот же поиск по суффиксному дереву, только с некоторомы улучшениями (ускорениями). Ведь не просто так было построено суффиксное дерево при помощи алгоритма Укконена. Для статистики совпадений будем задействовать суффиксные ссылки и идею с счетчиком совпадений и прыжками.

Для начала будем искать последоватьельность совпадающий символов. Как только это процесс прервется (встретим различные символы), сохраним длину такой последовательности в вектор ms. И продолжим поиск.

Чтобы поиск был эффективным, перейдем по суффиксной ссылке, ведь как мы знаем, символы до внутренней вершины у нас сопадают. Более того, мы знаем, сколько символов совпадает после внутренней вершины, ведь мы только что из сравнили на предыдущем шаге. Тогда в следующую позицию текста можно записать значение из предыдущей ячейки меньшее на 1. При этом проверив, нет ли еще совпадений.

Метод подсчета статистики использует вышеописанные преимущества. Используется три цикла. Берется цикл по тексту. Далее создается перемнная $j = i$. Главное условие, чтобы j не вышло за пределы текста.

Внутри цикла делаем поиск символа. Если символ найден, то проходим по ребру и равниваем сиволы.

Если все символы совпали, то переходим на следующее ребро, иначе нужно перейти по суффиксной ссылке и проверить другое ребро. Так будут сохранены все совпадающие подстроки паттрена и текста.

Если символ не найден, то выходим из цикла.

3 Консоль

```
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ ./lab5
aba
qababaz
2
4
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ ./lab5
baobab
aobbaobabababba
4
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ cat <tests/01.t
fdvmldnvlswa
agtkgdifdvmldnvlswattgtidmnai
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ ./lab5 <tests/01.t
8
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ cat <tests/05.t
hnvlovfjgz
kvzubkvfswhgkxqtxwvtpqzjmlkhnvlovfjgzyq
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_05$ ./lab5 <tests/05.t
28
```

4 Тест производительности

Реализованный алгоритм Укконена и алгоритм статистики совпадений сравним с наивным алгоритмом суффиксного дерева.

Будем тестировать на следующих строках и паттренах: (100, 2), (100, 50), (1000, 5), (1000, 100), (1000, 50), (10000, 1000). В скобках сначала длина строки, потом длина паттерна.

```
[info] [2024-10-08 23:20:13] Running tests/01.t
Ukkonen+MS: 0.057 ms
Naive Suffix Tree: 2.394 ms
[info] [2024-10-08 23:21:19] Running tests/01.t
Ukkonen+MS: 0.045 ms
Naive Suffix Tree: 1.791 ms
[info] [2024-10-08 23:21:53] Running tests/01.t
Ukkonen+MS: 0.085 ms
Naive Suffix Tree: 123.346 ms
[info] [2024-10-08 23:22:34] Running tests/01.t
Ukkonen+MS: 0.198 ms
Naive Suffix Tree: 128.511 ms
[info] [2024-10-08 23:23:18] Running tests/01.t
Ukkonen+MS: 0.257 ms
Naive Suffix Tree: 121.333 ms
[info] [2024-10-08 23:24:11] Running tests/01.t
Ukkonen+MS: 0.851 ms
Naive Suffix Tree: 13044.072 ms
```

Как можно заметить, реализованный алгоритм работает быстрее, чем наивный. За счет суффиксных ссылок и прыжков по счётчику алгоритм является более производительным.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я изучил и реализовал алгоритм Укконена для построения суффиксного дерева за линейное время. Ознакомился с приложениями суффиксного дерева, а точнее со статистикой совпадений.

Статистика совпадений позволяет искать совпадающие подстроки в паттерне и тексте, при этом суффиксное дерево требует меньше памяти, потому что строиться по паттерну. Статистика совадений активно используется в базах данных для ускоренного поиска.

Суффиксные деревья лучше всего подходят для поиска нескольких шаблонов в одном тексте, так как остальные алгоритмы (Кнута-Морриса-Пратта, Бойера-Мура) не смогут также эффективно справиться с этой задачей. Суффиксные деревья активно применяются в химии и биологии для работы с ДНК.

Список литературы

[1] Гасфилд Дэн
Строки, деревья и последовательности в алгоритмах: Информатика и вычисвычислительная биология. Пер. с англ. И. В. Романовского. — СПб.: Невский Диа- Диалект; БХВ-Петербург, 2003. — 654 с: ил. ISBN 5-7940-0103-8 ("Невский Диалект"), ISBN 5-94157-321-9 ("БХВ-Петербург")

[2] *Алгоритм Укконена — Викиконспекты.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 05.10.2024).

[3] *Алгоритм Укконена — MAXimal.*
URL: `http://www.e-maxx-ru.1gb.ru/algo/ukkonen` (дата обращения: 05.10.2024).