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. Ибежать лишних сравнений помогут «прыжки по счётчикам». 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 конеца 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 *root;
        std::string str;
        int suffixTreeEnd;
        TreeData params;

        void CreateTree();
        void AddSuffix(int position);
        void DestroyTree(TNode *node);
```

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

3 Консоль

4 Тест производительности

Реализованный алгоритм Z-функции сравним с наивным алгоритмом Z-функции. Будем тестировать на последовтельности из: 10^3, 10^4, 10^5, 10^6.

```
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_04$ g++ -std=c++17 -pedantic
-Wall -Wextra -Werror bechmark.cpp
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_04$ ./a.out <tests/1
Z_naive: 0.026 ms
Z_effective: 0.050 ms
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_04$ ./a.out <tests/2
Z_naive: 0.365 ms
Z_effective: 0.440 ms
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_04$ ./a.out <tests/3
Z naive: 21.339 ms
Z effective: 18.047 ms
alex@wega:~/Рабочий стол/вуз/2course/da_labs/Lab_04$ ./a.out <tests/4
Z naive: 40.037 ms
Z effective: 19.681 ms
```

На первых тестах наивная Z-функция работает быстрее. Это связано с тем, что в ней нет дополнительных проверок, как в эффективной Z-функции, где мы проверяем правую границу Z-блока. Видно, что стандартная Z-функция работает медлее с большими текстами, Так как она вычисляет значение для каждого символа. Эффективная Z-функция использует значения уже рассчитанные, посещая каждую позицию не более двух раз,что для больших текстов является более эффективным.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я изуил 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 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). *Алгоритм Укконена — MAXimal.*
URL: `http://www.e-maxx-ru.1gb.ru/algo/ukkonen` (дата обращения: 05.10.2024).