Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Студент: Н. С. Токарев
Преподаватель: Н. А. Зацепин
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва
2020

Условие

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:

Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

Метод решения

Для начала нужно построить суффиксное дерево с помощью алгоритма Укконена сложностью O(m), где m - количество элементов в строке. Затем же с помощью алгоритма за O(m), описанного в книге Дэна Гасфилда. Построение по линейному алгоритму Укконена:
Существуют несколько правил добавления, а также оптимизаций данного алгоритма для приведения ко времени O(m).

1. Суффиксная связь + прыжки по счетчику: Суффиксная связь необходима, чтобы за константное время переходить от одной вершине к другой. Предположим, что есть строки Xa и a, тогда необходимо построить суффиксную ссылку из Xa в a, также суффиксная связь может быть направлена к корню. Прыжки по счетчику также позволяют за константное время проходить по вершинам, которые находятся на одном ребре.

2. Был листом, листом и останешься(Достаточно просто увеличивать end или присвоить бесконечность, чтобы не возвращаться к данному листу).

3. На ребрах информация хранится в виде [start, end], где start индекс начала строки, а end индекс начала.

4. Также если при добавлении элемента X, есть строка у которой элемент X является последним, то достаточно просто закончить фазу.

Линеаризация циклической строки:

1. Строю суффиксное дерево по удвоенной строке SS.

2. Прохожу дерево таким образом, что в каждой вершине прохожу по дуге с первым символом, наименьшим среди заданного упорядочения среди всех дуг, выходящих из вершины.

При корректной реализации всех оптимизаций получается функция построения суффиксного дерева за линейное время.

Описание программы

Я решил использовать массив из умных указателей, так как это заметно упрощает проход по дугам с первыми символами. Например: children[0] - первый символ.
Реализация узла дерева:

```
std::shared_ptr<TSuffixTreeNode> children[ALPHABET_SIZE];
std::shared_ptr<TSuffixTreeNode> suffix_link; // суффиксная связь
int start; // первый элемент строки
int *end; // последний элемент
```

Для навигации в процессе построения у меня присутсвует структура активной(используемой) вершины, а также переменная REMINDER(отвечает за количество оставшихся несозданных вершин|дуг).
Реализация activePoint:

```
std::shared_ptr<TSuffixTreeNode> node; // текщая вершина
int symbol; // индекс добавляемого элемента
int length; // кол-во элементов до ближайшей вершины.
```

Функция ExtendSuffixTree(int pos) отвечает за добавление text[pos] в суффиксное дерево. Условие такого что пока REMINDER > 0 фаза продолжается, однако фаза может закончиться, если добавление происходит по правилу три или же REMINDER–, если создана новая вершина|дуга.
Создание новой листовой дуги:

```
if (!actPoint.node->children[symbol]) {
    actPoint.node->children[symbol] = std::make_shared<TSuffixTreeNode>(pos, END);
    if (link != root) {
    link->suffix_link = actPoint.node;
    link = root;
    } else link = actPoint.node;
    }
```

Скачок по счетчику:

```
if (actPoint.length >= actPoint.node->children[symbol]->EdgeLength()) {
    int length = actPoint.node->children[symbol]->EdgeLength();
    actPoint.length -= length;
    actPoint.node = actPoint.node->children[symbol];
    actPoint.symbol += length;
    continue;
    }
```

Функция GetLinearCircleString() - это функция линеаризации строки. Я прохожу по дугам с первыми элементами(в лексиграфическом смысле), пока размер меньше не дублированной строки. Проход по дугам с первыми элементами реализован:

```
for (i = 0; i < ALPHABET_SIZE; ++i) {
    if (cur->children[i]) { //поиск дуги с первым элементом
```

for (k = cur->children[i]->start; k <= *(cur->children[i]->end); ++k) { //проход по этой дуге

Исходный код

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
#include <time.h>


const int ALPHABET_SIZE = 28;
int REMINDER, END;
std::string text;
size_t SIZE;

class TSuffixTreeNode {
public:
    TSuffixTreeNode(int start, int *end);
    void ExtendSuffixTree(int pos);
    int EdgeLength();
    void GetLinearCircleString();
    void SetSuffixLink();
private:
    std::shared_ptr<TSuffixTreeNode> children[ALPHABET_SIZE];
    std::shared_ptr<TSuffixTreeNode> suffix_link;
    int start;
    int *end;
    int suffix_index;
};

struct ActivePoint {
    std::shared_ptr<TSuffixTreeNode> node;
    int symbol;
    int length;
};

ActivePoint actPoint;
std::shared_ptr<TSuffixTreeNode> root;

TSuffixTreeNode::TSuffixTreeNode(int start, int *end) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
```

```
        this->children[i] = nullptr;
    }
    this->suffix_link = root;
    this->start = start;
    this->end = end;
}

int TSuffixTreeNode::EdgeLength() {
    return *end - start + 1;
}

void TSuffixTreeNode::SetSuffixLink() {
    suffix_link = root;
}

void TSuffixTreeNode::ExtendSuffixTree(int pos) {
    std::shared_ptr<TSuffixTreeNode> cur = root;
    // объект, отвечающий за суффиксную связь.
    // В ходе работы link меняется на другие объекты(вершины)
    // link->suffix_link постоянно обновляется.
    std::shared_ptr<TSuffixTreeNode> link = root;
    std::shared_ptr<TSuffixTreeNode> next_node, cur_node;
    // кол-во несозданных вершин
    ++REMINDER;
    while (REMINDER > 0) {

        if (actPoint.length == 0) {
            actPoint.symbol = pos;
        }

        int symbol = text[actPoint.symbol] - 'a';
        //(новая листовая дуга)
        if (!actPoint.node->children[symbol]) {
            actPoint.node->children[symbol] = std::make_shared<TSuffixTreeNode>(pos, &EN
            if (link != root) {
                link->suffix_link = actPoint.node;
                link = root;
            } else link = actPoint.node;
        } else {
            // скачок по счетчику
            if (actPoint.length >= actPoint.node->children[symbol]->EdgeLength()) {
                int length = actPoint.node->children[symbol]->EdgeLength();
```

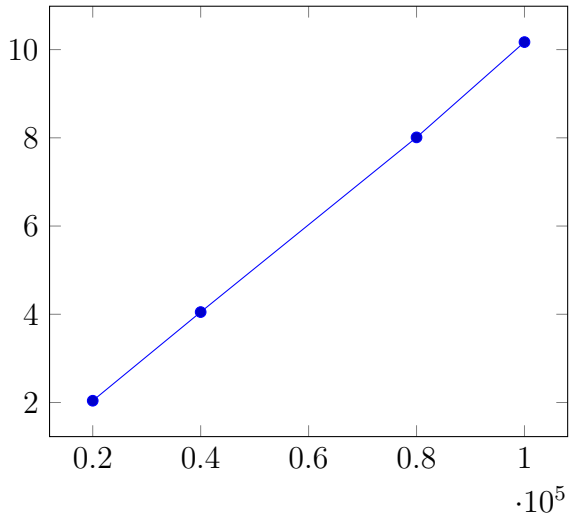
```
            actPoint.length -= length;
            actPoint.node = actPoint.node->children[symbol];
            actPoint.symbol += length;
            continue;
        }
        if (text[actPoint.node->children[symbol]->start + actPoint.length] == text[p
            actPoint.length++;

            if (link != root)
                link->suffix_link = actPoint.node;
            break;
        }

        cur = actPoint.node->children[symbol];

        int *position = new int;
        *position = cur->start + actPoint.length - 1;
        int next_pos = cur->start + actPoint.length;
        int next_symbol = text[next_pos] - 'a';
        int cur_symbol = text[pos] - 'a';

        cur_node = std::make_shared<TSuffixTreeNode>(pos, &END);
        next_node = std::make_shared<TSuffixTreeNode>(cur->start, position);

        actPoint.node->children[symbol] = next_node;
        cur->start = next_pos;
        next_node->children[next_symbol] = cur;
        next_node->children[cur_symbol] = cur_node;

        // ссылка на новую вершину и переход к ней.
        if (link != root)
            link->suffix_link = next_node;
        link = next_node;
    }
    REMINDER--;
    // возврат к элементу, которого нет в суффиксном дереве(продление от вершины или
    if (actPoint.node == root && actPoint.length > 0) {
        actPoint.length--;
        actPoint.symbol = pos - REMINDER + 1;
    } else { // переход по суффиксной связи
        actPoint.node = actPoint.node->suffix_link;
    }
```

```
    }
}

void TSuffixTreeNode::GetLinearCircleString() {
    TSuffixTreeNode *cur = root.get();
    int size = 0, k, i;
    while (true) {
        for (i = 0; i < ALPHABET_SIZE; ++i) {
            if (cur->children[i]) {
                for (k = cur->children[i]->start; k <= *(cur->children[i]->end); ++k) {
                    if (size < SIZE) {
                        std::cout << text[k];
                    } else {
                        std::cout << "\n";
                        return;
                    }
                    ++size;
                }

                cur = cur->children[i].get();
                break;
            }
        }
    }
}

int main() {
    int tmp = -1;
    int i;

    while (std::cin >> text) {
        END = -1;
        REMINDER = 0;

        root = std::make_shared<TSuffixTreeNode>(-1, &tmp);
        root->SetSuffixLink();

        actPoint.node = root;
        actPoint.length = 0;

        SIZE = text.size();
        text += text;
```

```
        for (i = 0; i < 2 * SIZE; ++i) {
            ++END;
            root->ExtendSuffixTree(END);
        }
        root->GetLinearCircleString();
    }
    return 0;
}
```

Тест производительности

Тесты представляют из себя набор из строк, где размер строки равен двадцати элементам и данный размер одинаков для всех тестов. Увеличивается только количество элементов в тесте.



Пояснения к графику: Ось y - время в секундах. Ось x - количество строк.
Таким образом видно, что алгоритм построения суффиксного дерева дублированной строки и ее линеаризация происходит за линейное время.

Выводы

Существуют несколько алгоритмов построения суффиксного дерва за линейное время.В ходе данной работы я реализовал алгоритм Укконена, а также алгоритм линеаризации циклической строки. Суффиксное дерево показалось мне одним из самых сложных структур данных для понимания. Построив суффиксное дерево можно найти линеаризацию, количесво вхождений, общие подстроки. Данный факт доказывает универсальность суффиксного дерева, а также его пользу.Также хотелось бы отметить, что существует суффиксное дерево для множества строк, которое как мне кажется является еще более универсальным.