Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Выполнил студент группы М8О-308Б-20 *Ядров Артем.*

Условие

Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из входных строк, необходимо воспользоваться полученным суффиксным деревом для решения своего варианта задания.
Алфавит строк: строчные буквы латинского алфавита (т.е., от a до z).

Вариант 4: Линеаризация циклической строки
Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.
Входные данные: некий разрез циклической строки.
Выходные данные: минимальный в лексикографическом смысле разрез.

Метод решения

В отличие от четвертой лабораторной сейчас перед нами стоит немного другая задача: если раньше нам приходил/и паттерн/ы и мы искали его/их в тексте, то сейчас нам нужен алгоритм, в котором мы можем взять текст, как-то его обработать $O(n)$, где n – длина текста, а потом запустить функцию поиска для паттерна за $O(m)$, где m – длина паттерна. В итоге мы решим поставленную задачу за $O(n+m)$. Такой алгоритм есть. Именно он использует структуру данных "Суффиксное дерево".

Суффиксное дерево – это *compacttrie*, который построен по всем суффиксам текущей строки, начиная от несобственного суффикса и заканчивая суффиксом, состоящем из одной буквы, обладающий следующими свойствами:

- в нем должно быть такое же количество листьев, как и символов в нашей строке (как и суффиксов в нашей строке)
- этот *trie* должен быть сжатый – если есть внутренняя вершина, то у нее должно быть как минимум два потомка
- не может быть такого, что из 1 вершины выходят два ребра, помеченных строками, начинающимися на одну и ту же букву

Суффиксное дерево можно построить наивным образом за $O(n^2)$, где n – длина текста. Достаточно просто поочередно вставлять все суффиксы строки. Действительно, пусть есть строка $a_1a_2a_3\dots a_n$. Тогда, чтобы построить суффиксное дерево, мы должны поочередно вставлять $a_1a_2a_3\dots a_n$, $a_2a_3\dots a_n$, $\dots$, $a_{n-1}a_n$, a_n символов, что в итоге даст $n+n-1+\dots 3+2+1 = O(n^2)$ по сложности. Очевидно, что это не самый

быстрый способ. Чтобы строить суффиксное дерево максимально быстро – за линейную сложность – существует алгоритм Укконена, в котором есть 3 важных правила:

- добавляем в лист – дописать букву на ребро
- нет пути – создать новый лист
- есть строка – ничего не делаем

А также присутствуют некоторые важные эвристики, которые ускоряют алгоритм: вместо строк мы на ребрах храним две интовые переменные, обозначающие начало и конец текста на ребре, при этом используя глобальную переменную end и инкрементируя ее на каждой итерации. Помимо этого, есть такое понятие, как суффиксная ссылка, которая перемещает нас из строки вида $x\alpha$ к строке α, где x - символ, а α - строка в уже построенном суффиксном дереве, и, так называемые, прыжки по счетчику, которые помогают нам при переходе между ребрами не проверять несколько символов и перешагивать по ним. Именно все эти эвристики и правила делают алгоритм Укконена единственно верным выбором, если перед нам нужно решить задачу при помощи суффиксного дерева.

Описание программы

Для реализации алгоритма Укконена я создал класс $TSuffixTree$.

```
#ifndef SUFF_TREE_SUFFTREE_H
#define SUFF_TREE_SUFFTREE_H

#include <memory>
#include <string>
#include <utility>
#include <vector>
#include <map>

/*!
 * Класс суффиксного дерева. Конструируется из строки.
 */
class TSuffixTree {
public:
    explicit TSuffixTree(std::string s);

    std::string
    lexMinString();

    ~TSuffixTree() = default;
```

```
private:
    /*!
     * структура ребра суффиксного дерева.
     */
    struct Node {
        Node *suffix_link;
        int start;
        int *end;
        std::map<char, Node *> children;
        bool is_leaf;

        Node(int start, int *end, bool is_leaf) {
            this->start = start;
            this->end = end;
            suffix_link = nullptr;
            this->is_leaf = is_leaf;
        }

        ~Node() {
            for (auto &it: children) {
                delete (it.second);
            }
        }
    };

    void
    insert(const int &pos);

    Node *root;
    Node *active_node;
    Node *last_created_node;
    int active_edge;
    int active_length;
    int suffixes_to_add;
    std::string text;
    int global_end;
    static const int DEFAULT_VALUE = -1;
};

#endif //SUFF_TREE_SUFFTREE_H
```

Реализацию функций, описанных выше, я поместил в отдельный файл.

```cpp
#include "SuffTree.h"

#include <utility>

/*!
 * Конструктор суффиксного дерева для строки s
 * @param s - исходная строка
 */
TSuffixTree::TSuffixTree(std::string s) :
        text(std::move(s)) {
    root = new Node(DEFAULT_VALUE, new int(DEFAULT_VALUE), false);
    active_node = root;
    global_end = DEFAULT_VALUE;
    active_edge = DEFAULT_VALUE; //activeEdge is represented as input string
↪   character index
    active_length = 0; //this tells how many characters we need to walk down
↪   for find symbol that needed
    suffixes_to_add = 0;
    for (int i = 0; i < text.size(); ++i) {
        insert(i);
    }
}

std::string
TSuffixTree::lexMinString() {
    std::string result_string;
    size_t length = (text.size() - 1) / 2;
    Node *next = root;
    while (result_string.size() < length) {
        auto it = next->children.begin();
        if (it->first == '$') {
            ++it;
        }
        next = it->second;
        for (int i = next->start; i <= *(next->end); ++i) {
            result_string += text[i];
            if (result_string.size() == length) {
                break;
            }
        }
    }
    return result_string;
```

```
}

/*!
 * Функция вставки суффикса, начинающегося в позиции pos
 * @param pos - позиция начала суффикса (0-индексация)
 */
void
TSuffixTree::insert(const int &text_position) {
    last_created_node = nullptr; /*indicating there is no internal node
↪   waiting for it's suffix link reset in current phase*/
    ++suffixes_to_add; //indicating that a new suffix ready to be added in
↪   tree
    ++global_end; //
    while (suffixes_to_add > 0) {
        //If activeLength is ZERO, set activeEdge to the current character
        if (active_length == 0) {
            active_edge = text_position;
        }
        // A new leaf edge gets created
        if (!active_node->children[text[active_edge]]) {
            active_node->children[text[active_edge]] = new
↪   Node(text_position, &global_end, true);
            if (last_created_node != nullptr) {
                last_created_node->suffix_link = active_node;
                last_created_node = nullptr;
            }
        } else {
            // Get the next node at the end of edge starting with
↪   activeEdge
            Node *next_node = active_node->children[text[active_edge]];
            int edge_length = *(next_node->end) - next_node->start + 1;
            //tricks
            if (active_length >= edge_length) {
                active_edge += edge_length;
                active_length -= edge_length;
                active_node = next_node;
                continue;
            }
            //(current character being processed is already on the edge
            if (text[next_node->start + active_length] ==
↪   text[text_position]) {
                //If a newly created node waiting for it's suffix link to be
↪   set, then set suffix link of that waiting node to current active node
```

```
                if (last_created_node != nullptr) {
                    last_created_node->suffix_link = active_node;
                    last_created_node = nullptr;
                }
                ++active_length;
                break;
            }
            //new leaf edge and a new internal node get created - if there
↪   is no way, create a new nodes
            Node *to_add = new Node(next_node->start, new
↪   int(next_node->start + active_length - 1), false);
            active_node->children[text[active_edge]] = to_add;
            next_node->start += active_length;
            std::pair<char, Node *> first =
↪   std::make_pair(text[text_position],
                                                                new
↪   Node(text_position, &global_end, true));
            std::pair<char, Node *> second =
↪   std::make_pair(text[next_node->start], next_node);
            to_add->children.insert(first);
            to_add->children.insert(second);
            //suffixLink of lastNewNode points to current newly created
↪   internal node
            if (last_created_node != nullptr) {
                last_created_node->suffix_link = to_add;
            }
            last_created_node = to_add;
        }
        --suffixes_to_add; //decrement remaining suffixes because we
↪   finished inserting
        if (active_node == root && active_length > 0) {
            ++active_edge;
            --active_length;
        } else if (active_node != root) {
            active_node = active_node->suffix_link;
        }
    }
}

//std::string
//TSuffixTree::lexMinString(const size_t &n, const std::shared_ptr<TNode>&
↪   node)
```

```
//{
//        // конец рекурсии
//        if (n == 0){
//                return {};
//        }
//        int len = node->length();
//        if (len > n){
//                return text.substr(node->left, n);
//        } else {
//                return text.substr(node->left, len) + lexMinString(n -
↪  len, node->children.begin()->second);
//        }
//}

#include <iostream>
#include <chrono>

#include "SuffTree.h"

const std::string SENTINEL = "$";


int main() {
    auto startPoint = std::chrono::system_clock::now();
    std::string input;
    std::cin >> input;
    input = input + input + SENTINEL;
    auto *tree = new TSuffixTree(input);
    std::string answer = tree->lexMinString();
//    std::cout << answer << "\n";
    delete tree;
    auto endPoint = std::chrono::system_clock::now();
    std::cout <<
↪  std::chrono::duration_cast<std::chrono::nanoseconds>(endPoint -
↪  startPoint).count() << " ms"
              << std::endl;
    return 0;
}
```

Дневник отладки

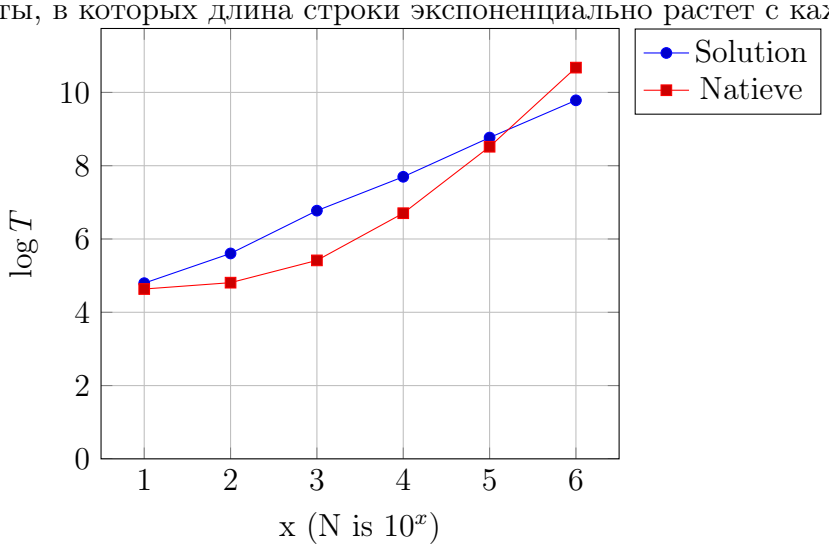
Проблемы в лабораторной были связаны, в основном, непосредственно с алгоритмом Укконена. Из-за некорректного построения суффиксного дерева функция минимально-

го разреза на больших тестах давала неправильные ответы, однако это было исправлено, и теперь программа работает корректно.

Тест производительности

Сравним написанную программу с наивным поиском минимального лексикографического среза: для того, чтобы найти минимальный лексикографический срез, удвоим строку, затем найдем минимум из всех подстрок длины n, начинающихся с позиции $start = 0, 1, 2, \ldots n - 1$. Этот наивный алгоритм будет работать за время $O(n^2)$, так как на каждой итерации $start$ мы ищем подстроку длины n, что занимает $O(n)$ времени.

Для сравнения я сгенерировал тесты с различной длиной строки. Очевидно, что на коротких строках будет выигрывать наивная реализация, так как построение суффиксного дерева хоть и линейно, но имеет большую константу. Поэтому я генерировал тесты, в которых длина строки экспоненциально растет с каждым тестом.



Выводы

Суффиксное дерево довольно мощная структура данных, имеющая множество приложений в задачах обработки строк. Среди них нахождение включения одной строки в текст, поиск наибольшей общей подстроки для набора строк, и т.д . Так же суффиксное дерево является вспомогательным, например, при построении суффиксных массивов, поиске статистики совпадений и скорее всего где-нибудь еще.

Написание суффиксного дерева мне показалось довольно сложным занятием. Мало того, что алгоритм сам по себе не очень простой, но, даже если разобраться на бумаге и начать писать с нуля, то приходишь к выводу, что в этом алгоритме очень много мелких деталей. Неправильно поняв (закодив) одну из них суффиксное дерево отказывается запускаться либо совсем, либо ломается на частных данных.