Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет «Компьютерные науки и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу

«Дискретный анализ»

Тема работы

“Суффиксные деревья”

Студент : Ю.И. Катаев
Группа : М8О-310Б-21
Преподаватель : Н.К. Макаров
Оценка : _____
Дата : _____
Подпись : _____

Москва, 2023

1. Постановка задачи

Реализовать поиск подстрок в тексте с использование суффиксного дерева. Суффиксное дерево можно построить за $O(n^2)$ наивным методом.

Формат ввода:
Текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

Формат вывода:
Для каждого образца, найденного в тексте, нужно распечатать строку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

2. Описание

Суффиксное дерево (сжатое суффиксное дерево) T для строки s (где |s|=n) — дерево с n листьями, обладающее следующими свойствами:

- каждая внутренняя вершина дерева имеет не меньше двух детей;
- каждое ребро помечено непустой подстрокой строки s;
- никакие два ребра, выходящие из одной вершины, не могут иметь пометок, начинающихся с одного и того же символа;
- дерево должно содержать все суффиксы строки s причем каждый суффикс заканчивается точно в листе и нигде кроме него.

Вот шаги построения суффиксного дерева с помощью наивного алгоритма:
1. Создайте пустое дерево, состоящее из одного корня.
2. Добавьте всю строку в дерево как первый суффикс по значению первой буквы. Если символ уже существует, просто продлите ребро, указывающее на него.
3. Повторяйте следующие шаги для каждого последующего символа:
   3.1 Начиная от корня, сравните текущий символ с каждым символом на ребре одновременно. Если совпадение найдено, перейдите к следующему символу.

3.2 Если совпадение не найдено, создайте новую ветвь, начиная с текущего символа, и добавьте ее к дереву.
3.3 Если ребро заканчивается, добавьте текущий символ как новое ребро.
4. Повторите шаги 2 и 3 до завершения строки.
5. Построение суффиксного дерева завершается, когда все символы строки были добавлены.

Наивный алгоритм может быть неэффективным для больших строк из-за квадратичной сложности его времени выполнения. Существуют более эффективные алгоритмы, такие как алгоритм Укконена, который имеет линейную сложность O(n), где n - длина исходной строки.

3. Разбор программы

N - длина текста
M - длина паттерна

Функция	Описание	Time complexity	Extra Space
void Node::Insert(int begin, int stance) {	Добавляет в суффиксное дерево строку, начинающуюся с позиции begin.	O(N)	O(1)
vector<int> Node::Find(string pattern)	Ищет переданный паттерн в суффиксном дереве и возвращает все индексы, начиная с которых паттерн входит в текст	O(M)	O(M)
void Node::AllPositions(vector<int> &entrances)	Накапливает в переданный вектор все литья, предком которых является текущая вершина	O(N)	O(1)

4. Исходный код

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

using namespace std;
```

```
string build_string;

struct Node {
    int start;
    int end;
    int position;
    vector<Node*> next;

    Node(int, int, int);
    ~Node();
    void Insert(int, int);
    vector<int> Find(string);
    void AllPositions(vector<int> &);
    void Print(int);
};

Node::Node(int start, int end, int position) {
    this->start = start;
    this->end = end;
    this->position = position;
    this->next.resize(27); // 26 букв + 1 терминал
}

Node::~Node() {
    for (auto child: next) {
        delete child;
    }
}

void Node::Insert(int begin, int stance) {
    if (next[build_string[begin] - 'a'] == nullptr) {
        next[build_string[begin] - 'a'] = new Node(begin,
(int)build_string.length() - 1, stance);
        return;
    }
    Node* current = next[build_string[begin] - 'a'];
    int match_shift = 0;
    while (current->start + match_shift <= current->end) {
```

```
        if (build_string[begin + match_shift] !=
build_string[current->start + match_shift]) {
            break;
        }
        ++match_shift;
    }
    // split node
    if (current->start + match_shift <= current->end) {
        Node* new_node = new Node(current->start, current->start +
match_shift - 1, -1);
        current->start += match_shift;
        new_node->next[build_string[current->start] - 'a'] = current;
        new_node->next[build_string[begin + match_shift] - 'a'] = new
Node(
                begin + match_shift,
                (int)build_string.length() - 1,
                stance
                );
        next[build_string[begin] - 'a'] = new_node;
    } else {
        current->Insert(begin + match_shift, stance);
    }
}

vector<int> Node::Find(string pattern) {
    bool is_contain = true;
    Node* current = this;
    size_t last_matched = 0;
    while(last_matched < pattern.size()) {
        current = current->next[pattern[last_matched] - 'a'];
        if (current == nullptr) {
            is_contain = false;
            break;
        }
        for (int i = current->start; i <= current->end && last_matched
< pattern.size(); ++i) {
            if (pattern[last_matched] != build_string[i]) {
                break;
```

```
            }
            ++last_matched;
        }

        if (last_matched < pattern.size() && current->start +
(int)last_matched < current->end) {
            is_contain = false;
            break;
        }
    }
    vector<int> entrances;
    if (is_contain) {
        current->AllPositions(entrances);
    }
    return entrances;
}

void Node::AllPositions(vector<int> &entrances) {
    if (position != -1) {
        entrances.push_back(position);
        return;
    }

    for (int i = 0; i < 27; ++i) {
        if (next[i] != nullptr) {
            next[i]->AllPositions(entrances);
        }
    }
}

void Node::Print(int depth) {
    if (depth != 0) {
        for (int i = 0; i < depth - 1; ++i) {
            cout << "\t";
        }
        cout << start << " "
             << end << " "
             << build_string.substr(start, end - start + 1) << "\t"
```

```
             << position << "\n";
    }
    for (int i = 0; i < 27; ++i) {
        Node* current = next[i];
        if (current != nullptr) {
            current->Print(depth + 1);
        }
    }
}

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr), cout.tie(nullptr);

    string text;
    cin >> text;
    build_string = text + "{";
    Node suffix_tree_root = Node(-1, -1, -1);
    for (size_t position = 0; position < build_string.length();
++position) {
        suffix_tree_root.Insert((int)position, (int)position);
    }

    int word_number = 0;
    string pattern;
    while (cin >> pattern) {
        ++word_number;
        vector<int> entrances = suffix_tree_root.Find(pattern);
        if (!entrances.empty()) {
            sort(entrances.begin(), entrances.end());
            cout << word_number << ": ";
            for (size_t i = 0; i < entrances.size(); ++i) {
                if (i > 0) {
                    cout << ", ";
                }
                cout << entrances[i] + 1;
            }
            cout << "\n";
```

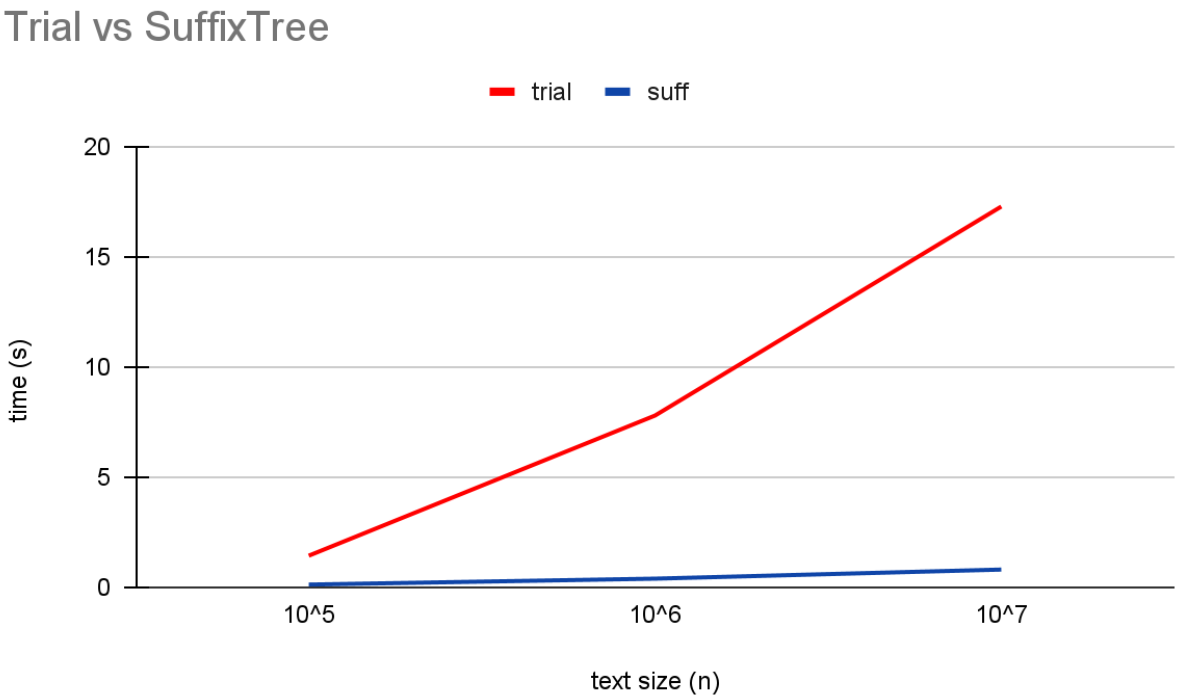
```
        }
    }
    return 0;
}
```

5. Демонстрация работы программы

```
➜  src git:(main) ✗ cat gen.txt
acababaaaa
aaadb
abbdda
ba
cdaa
bb
➜  src git:(main) make build
g++ -Wall -Waddress -Wextra -Werror -g main.cpp -o main
➜  src git:(main) ./main < gen.txt
3: 4, 6
```

6. Сравнение работы алгоритмов сортировки

Размер текста (N)	Количество паттернов (K)	Суффиксное дерево $O(N^2 + MK)$	Наивный алгоритм $O(NMK)$
1000	10^5	0.127104	1.43796
1000	10^6	0.396097	7.79441
1000	10^7	0.807794	17.2748



7. Выводы

В ходе выполнения данной лабораторной работы было реализовано суффиксное дерево с использованием наивного метода. Затем было выполнено задание по поиску всех вхождений паттернов в тексте, используя ранее построенное суффиксное дерево.

Таким образом, реализация суффиксного дерева с помощью наивного метода позволяет эффективно находить все вхождения паттернов в текст. Однако стоит отметить, что наивный метод не является самым оптимальным, и существуют более эффективные алгоритмы построения суффиксных деревьев.