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: В. А. Слесарчук
Преподаватель: Н. К. Макаро
Группа: М8О-301Б
Дата:
Оценка:
Подпись:

Москва, 2025

# Лабораторная работа №5

**Вариант:** 3

**Задача:**

Требуется разработать программу, осуществляющую ввод образца и текста, реализующую поиск образца в тексте используя статистику совпадений и вывод номеров позиций, начиная с которых встретился образец.

**Формат ввода:** На первой строке располагается образец, на второй — текст.

**Формат вывода:** Последовательность строк содержащих в себе номера позиций, начиная с которых встретился образец. Строки должны быть отсортированы в порядке возрастания номеров.

## 1 Описание

Для решения задачи был использован алгоритм, основанный на обобщенном суффиксном дереве. Суффиксное дерево — это сжатое префиксное дерево, содержащее все суффиксы заданной строки. Его ключевое преимущество — возможность построения за линейное время от длины строки с помощью алгоритма Укконена.

## 2 Исходный код

**Основная идея:**

Программа стоит обобщенное суффиксное дерево для образца используя алгоритм Укконена. Затем посимвольно производится поиск символов текста в суффиксном дереве образца. Если символ найден - то счетчик совпадений увеличивается на 1, а поиск следующего символа будет осуществляться с позиции последнего совпадения. Если невозможно найти символ в текущей итерации поиска - то счетчик совпадений уменьшается на 1 и производится откат по суффиксной ссылке и восстановление позиции на ребре дерева, откуда поиск будет продолжен для следующего символа.

**Структуры данных:**

struct Node — узел суффиксного дерева. Содержит:

- int start, end - пара указателей на исходный текст, определяющая ребро, ведущее в узел.
- Node* suffix_link - суффиксная ссылка.
- std::unordered_map<char, Node*> children; - переходы к дочерним узлам

struct staff - вспомогательная структура для обхода с использованием статистики совпадений.

- Node* prev_node - ссыллка на предыдущую вершину (используется для отката при несовпадении).
- Node* current_node - вершина в которой сейчас производится обход.
- int edge_start - началао ребра.
- int edge_pos - позиция на ребре.
- int counter - счетчик совпадений.
- int j - позиция совпадения в тексте.

**Построение дерева:**

Функция extend(int i) реализует одну фазу алгоритма Укконена, добавляя символ text[i] во все активные суффиксы. Для поддержания линейного времени работы активно используются суффиксные ссылки и концепция "активной точки"(active_node, active_edge, active_length).

**Поиск:**

Функция feed(char c, vector<int>& stat) принимает букву текста и осуществляет ее эффективный поиск в тексте.

Функция go_back(char c, vector<int>& stat) осуществляет корректный откат по суффиксной ссылке при обнаружении несовпадения.

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

using namespace std;

class SuffixTree {
private:
    struct Node {
        int start;
        int end;
        Node* suffix_link;
        std::unordered_map<char, Node*> children;

        Node(int start = -1, int end = -1)
            : start(start), end(end), suffix_link(nullptr) {}
    };

```

```
    struct ActivePoint {
        Node* node;
        int edge;
        int length;

        ActivePoint() : node(nullptr), edge(-1), length(0) {}
    };

    Node* root;
    ActivePoint active;
    int remaining;
    int global_end;
    std::string text;

    int edge_length(Node* node) {
        return std::min(node->end, global_end + 1) - node->start;
    }

    bool walk_down(Node* node) {
        int edge_len = edge_length(node);

        if (active.length >= edge_len) {
            active.edge += edge_len;
            active.length -= edge_len;
            active.node = node;
            return true;
        }
        return false;
    }

    void extend_tree(int pos) {
        global_end = pos;
        remaining++;
        Node* last_created_internal_node = nullptr;

        while (remaining > 0) {
            if (active.length == 0) {
                active.edge = pos;
            }

            char current_char = text[pos];
            auto it = active.node->children.find(text[active.edge]);

            if (it == active.node->children.end()) {
                Node* new_node = new Node(pos, text.length());
                active.node->children[text[active.edge]] = new_node;

                if (last_created_internal_node != nullptr) {
                    last_created_internal_node->suffix_link = active.node;
```

```
                last_created_internal_node = nullptr;
            }
        } else {
            Node* next_node = it->second;

            if (walk_down(next_node)) {
                continue;
            }

            if (text[next_node->start + active.length] == current_char) {
                if (last_created_internal_node != nullptr && active.node != root) {
                    last_created_internal_node->suffix_link = active.node;
                }
                active.length++;
                break;
            }

            Node* split_node = new Node(next_node->start, next_node->start + active.
                length);
            Node* new_leaf = new Node(pos, text.length());

            next_node->start += active.length;

            split_node->children[text[next_node->start]] = next_node;
            split_node->children[current_char] = new_leaf;

            active.node->children[text[active.edge]] = split_node;

            if (last_created_internal_node != nullptr) {
                last_created_internal_node->suffix_link = split_node;
            }
            last_created_internal_node = split_node;
        }

        remaining--;

        if (active.node == root && active.length > 0) {
            active.length--;
            active.edge = pos - remaining + 1;
        } else if (active.node != root) {
            active.node = (active.node->suffix_link != nullptr) ?
                        active.node->suffix_link : root;
        }
    }
}

void delete_subtree(Node* node) {
    if (node == nullptr) return;

```

```
117         for (auto& child : node->children) {
118             delete_subtree(child.second);
119         }
120         delete node;
121     }
122 
123     void print_tree(Node* node, int depth = 0) {
124         if (node == nullptr) return;
125 
126         for (int i = 0; i < depth; i++) {
127             std::cout << " ";
128         }
129 
130         std::cout << "Node(" << node->start << ", "
131                   << (node->end == text.length() ? "END" : std::to_string(node->end))
132                   << ")";
133 
134         if (node->suffix_link != nullptr) {
135             std::cout << " [suffix_link]";
136         }
137         std::cout << std::endl;
138 
139         for (auto& child : node->children) {
140             for (int i = 0; i < depth + 1; i++) {
141                 std::cout << " ";
142             }
143             std::cout << "Edge '" << child.first << "' -> ";
144             print_tree(child.second, depth + 2);
145         }
146     }
147 
148 public:
149     SuffixTree() : root(new Node()), remaining(0), global_end(-1) {
150         active.node = root;
151     }
152 
153     ~SuffixTree() {
154         delete_subtree(root);
155     }
156 
157     void build(const std::string& input_text) {
158         text = input_text + "$";
159         global_end = -1;
160         remaining = 0;
161         active = ActivePoint();
162         active.node = root;
163 
164         for (int i = 0; i < text.length(); i++) {
165             extend_tree(i);
```

```
166         }
167     }
168 
169     void print() {
170         std::cout << "Suffix Tree for: " << text << std::endl;
171         print_tree(root);
172     }
173 private:
174     struct staff
175     {
176         int counter = 0;
177         Node* prev_node;
178         Node* current_node;
179         int edge_start = -1;
180         int edge_pos = 0;
181         int j = 0;
182     };
183 
184     staff traversal;
185     void start_traversal() {
186         traversal.counter = 0;
187         traversal.prev_node = root;
188         traversal.current_node = root;
189         traversal.edge_start = -1;
190         traversal.edge_pos = 0;
191     }
192 
193 public:
194     int global_counter = 0;
195     int downgrade(int a){
196         a -= 1;
197         if (a < 0) return 0;
198         return a;
199     }
200 
201     void go_back(char c, vector<int>& stat){
202         if (traversal.edge_start != -1){
203             Node* suffix_target = (traversal.prev_node->suffix_link != nullptr)
204                                   ? traversal.prev_node->suffix_link
205                                   : root;
206             traversal.current_node = traversal.prev_node;
207             if (traversal.current_node == root){
208                 if (global_end == (traversal.edge_start + 1)){
209                     traversal.prev_node = root;
210                     traversal.current_node = root;
211                     traversal.edge_pos = 0;
212                     traversal.edge_start = -1;
213                     feed(c, stat);
214                 } else {
```

```
215                 traversal.edge_pos -= 1;
216                 if (traversal.edge_pos == 0){
217                     traversal.prev_node = root;
218                     traversal.current_node = root;
219                     traversal.edge_pos = 0;
220                     traversal.edge_start = -1;
221                     feed(c, stat);
222                 } else {
223                     traversal.prev_node = traversal.current_node;
224                     traversal.current_node = traversal.current_node->children[text[
                            traversal.edge_start + 1]];
225                     traversal.edge_start += 1;
226                     feed(c, stat);
227                 }
228             }
229         } else {
230             traversal.current_node = suffix_target;
231             traversal.prev_node = suffix_target;
232             traversal.current_node = traversal.current_node->children[text[traversal
                    .edge_start]];
233             feed(c, stat);
234         }
235     } else {
236         Node* suffix_target = (traversal.current_node->suffix_link != nullptr)
237                               ? traversal.current_node->suffix_link
238                               : root;
239         traversal.current_node = suffix_target;
240         feed(c, stat);
241     }
242 }
243
244 void feed(char c, vector<int>& stat){
245     global_counter += 1;
246     if ((traversal.edge_start == -1) || (traversal.edge_pos == 0)) {
247         traversal.edge_pos = 0;
248         traversal.edge_start = -1;
249         if (traversal.current_node->children.find(c) != traversal.current_node->
                children.end()){
250             traversal.prev_node = traversal.current_node;
251             traversal.edge_start = traversal.current_node->children[c]->start;
252             traversal.current_node = traversal.current_node->children[c];
253             traversal.edge_pos = 1;
254             traversal.counter += 1;
255         } else {
256             stat[traversal.j] = traversal.counter;
257             traversal.j += 1;
258             traversal.counter = downgrade(traversal.counter);
259             if (traversal.current_node == root){
260                 if (root->children.find(c) == root->children.end()) {
```

```
                    traversal.edge_start = -1;
                    traversal.edge_pos = 0;
                    traversal.prev_node = root;
                    traversal.current_node = root;
                } else {
                    feed(c, stat);
                }
            } else {
                go_back(c, stat);
            }
        }
    } else {
        bool stop = false;
        while (!stop){
            int edge_len = edge_length(traversal.current_node);
            if (traversal.edge_pos == edge_len) {
                bool stop = true;
                traversal.edge_start = -1;
                traversal.edge_pos = 0;
                feed(c, stat);
                break;
            } else if (traversal.edge_pos > edge_len) {
                traversal.edge_pos -= edge_len;
                traversal.prev_node = traversal.current_node;
                traversal.current_node = traversal.current_node->children[text[
                    traversal.edge_start + edge_len]];
                traversal.edge_start += edge_len;
            } else {
                bool stop = true;
                if (text[traversal.edge_start + traversal.edge_pos] == c){
                    traversal.edge_pos += 1;
                    traversal.counter += 1;
                } else {
                    stat[traversal.j] = traversal.counter;
                    traversal.j += 1;
                    traversal.counter = downgrade(traversal.counter);
                    go_back(c, stat);

                }
                break;
            }
        }
    }
}

vector<int> search(string T){
    vector<int> T_stat(T.length(), 0);

    int fin_pos = T.length();
```

```
309         //fin_pos = 5;
310         start_traversal();
311         for (size_t i = 0; i < fin_pos; i++)
312         {
313             char c = T[i];
314             feed(c, T_stat);
315         }
316         for (size_t i = 0; i < traversal.counter; i++)
317         {
318             T_stat[T.length() - (traversal.counter - i)] = (traversal.counter - i);
319         }
320         vector<int> finale;
321         for (size_t i = 0; i < T_stat.size(); i++)
322         {
323             if (T_stat[i] == (text.length() - 1)) {
324                 finale.push_back(i + 1);
325             }
326         }
327         return finale;
328     };
329 };
330 
331 std::vector<int> findPatternPositions(const std::string& image, const std::string&
        text) {
332     if ((image == "") && (text == "")){
333         vector<int> a(1, 1);
334         return a;
335     }
336     vector<int> stat(text.length(), 0);
337     SuffixTree tree;
338     tree.build(image);
339     return tree.search(text);
340 }
341 
342 int main(){
343     string image = "", text = "";
344     cin >> image >> text;
345     vector<int> stat(text.length(), 0);
346     stat = findPatternPositions(image, text);
347     for (size_t i = 0; i < stat.size(); i++)
348     {
349         cout << stat[i] << endl;
350     }
351 }
```

# 3 Консоль

Пример компиляции и демонстрация работы программы:

```
yugo@yugo-pc:~/Desktop/Labs/DA/lab 5$ g++ main.cpp -o main
yugo@yugo-pc:~/Desktop/Labs/DA/lab 5$ ./main
aba
qababacaba
2
4
8
```

# 4 Тест производительности

Методика тестирования заключалась в сравнении производительности реализованного алгоритма на суффиксных деревьях с классическим подходом, использующим динамическое программирование (ДП). Алгоритм ДП имеет квадратичную сложность $O(|S1| \cdot |S2|)$, в то время как поиск с использованием статистики совпадений обеспечивает линейное время $O(|S1| + |S2|)$

```
yugo@yugo-pc:~/Desktop/Labs/DA/lab 5$ g++ main.cpp -o main
yugo@yugo-pc:~/Desktop/Labs/DA/lab 5$ ./main
== == == ==
Pattern length = 10
Text lenght = 100000
MSA = 0.0364635
DP = 0.105447
== == == ==
Pattern length = 100
Text lenght = 1000000
MSA = 0.356132
DP = 6.29394
== == == ==
Pattern length = 1000
Text lenght = 10000000
MSA = 3.64518
DP = 528.938
```

Результаты тестов полностью подтверждают теоретические оценки сложности. Как видно поиск с использованием статистики совпадений на суффиксном дереве имеет линейное время.

# 5 Выводы

В ходе выполнения лабораторной работы была успешно реализована программа для поиска всех вхождений образца в текст с использованием статистики совпадений.

- Освоена структура суффиксного дерева и принципы его построения с помощью онлайн-алгоритма Укконена, включая ключевые концепции, такие как суффиксные ссылки и "активная точка".
- Реализован метод решения задачи на основе обобщенного суффиксного дерева.d
- Реализован поиск подстроки в строке с использованием статистики совпадений, что позволило выполнить задачу за линейное время.
- Проведено сравнительное тестирование производительности, которое на практике подтвердило асимптотическое преимущество линейного алгоритма на суффиксных деревьях (O(N)) над квадратичным решением на основе динамического программирования (O($N^2$ ))
- Получен практический опыт работы со сложными динамическими структурами данных на C++, включая ручное управление памятью и указателями, что является важным для понимания низкоуровневых аспектов программирования. Разработанное решение является эффективным и масштабируемым, а полученные знания о суффиксных структурах данных могут быть применены для решения широкого круга других задач в области биоинформатики, обработки текстов и анализа данных.

.

# Список литературы

[1] Ukkonen E. On-line construction of suffix trees // Algorithmica. — 1995. — Vol. 14, no. 3. — P. 249–260.

[2] Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[3] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ, 3-е изд. — М.: «Вильямс», 2013.