Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Дискретный анализ" №5

Студент: Кострюков Е.С.

Преподаватель: Макаров Н.К.

Группа: М8О-307Б-22

Дата:

Оценка:

Подпись:

Москва 2024 г.

Оглавление

Цель работы

Найти образец в тексте используя статистику совпадений.

Постановка задачи

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.

Общий алгоритм решения

Программа строит суффиксное дерево по паттерну, используя алгоритм Укконена, и с помощью статистики совпадений определяет позиции вхождения паттерна в текст.

1. Построение суффиксного дерева:
- Для каждого суффикса строки строится суффиксное дерево с использованием алгоритма Укконена. Дерево создаётся за линейное время относительно длины строки;
- Узлы дерева содержат информацию о начале (*start*) и конце (*end*) фрагмента строки, который представлен ребром, а также имеют ссылки на потомков (*children*) и суффиксную ссылку (*suffixLink*), которая помогает эффективно находить суффиксы в процессе построения.

2. Алгоритм Укконена:
- Суффиксное дерево строится итеративно. На каждом шаге рассматривается новый символ строки, и добавляются все суффиксы, заканчивающиеся этим символом.
- Алгоритм поддерживает три переменные:
  - *activeNode* — текущий узел, из которого осуществляется добавление нового суффикса.
  - *activeEdge* — ребро, по которому мы двигаемся от *activeNode*.
  - *activeLength* — количество символов, совпадающих с ребром *activeEdge*.
- Если добавляемый символ не совпадает с символом на *activeEdge*, ребро разрезается, создается новый узел, и алгоритм продолжает обрабатывать оставшиеся суффиксы.

3. Суффиксные ссылки:
- Суффиксные ссылки используются для того, чтобы ускорить добавление новых суффиксов. Если текущий суффикс частично совпадает с каким-то уже добавленным суффиксом, можно перескочить по суффиксной ссылке и начать добавление с уже построенной части дерева.

4. Поиск строки в тексте:
- После построения суффиксного дерева, функция *search* выполняет поиск всех вхождений строки (паттерна) в текст;

- Для каждого стартового индекса текста проверяется, начинается ли с этого индекса подстрока, совпадающая с паттерном. Это делается с помощью функции *checkSubstringAtPos*, которая последовательно спускается по дереву и сравнивает символы на рёбрах.

5. Результат:
- Все позиции вхождений паттерна в текст выводятся на экран.

Основные шаги алгоритма:
1. Построение суффиксного дерева за ($O(m)$), где (m) — длина паттерна.
2. Поиск паттерна в тексте, используя суффиксное дерево, за ($O(n)$), где (n) — длина текста.

Реализация

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

using namespace std;

const int ALPHABET_SIZE = 128;

// *** Узел суффиксного дерева ***
struct Node {
    int start, *end; // end в листовом узле указывает на leafEnd
    Node *suffixLink;
    std::unordered_map<char, Node*> children;

    Node(int start, int *end) : start(start), end(end) {
        suffixLink = nullptr;
    }
};

class SuffixTree {
    string pattern;
    Node *root;
    int remainingSuffixCount; // Счётчик недобавленных суффиксов
    int activeLength;
    Node *activeNode;
    int activeEdge;
    int leafEnd; // Общая переменная, указывающая концы листовых узлов
    int *rootEnd; // Указатель на конец ребра для корневого узла
    int *splitEnd;
    int size;

    Node* newNode(int start, int *end) {
        Node* node = new Node(start, end);
        node->suffixLink = root;  // Суффиксная ссылка по умолчанию указывает на корень
        return node;
    }

    int getEdgeLength(Node *n) {
        return *(n->end) - n->start + 1;
    }
```

```
// Функция спуска по ребру узла
bool walkDown(Node *currNode) {
    if (activeLength >= getEdgeLength(currNode)) {
        activeEdge += getEdgeLength(currNode);
        activeLength -= getEdgeLength(currNode);
        activeNode = currNode;
        return true;
    }
    return false;
}

// Функция, которая добавляет новый суффикс в дерево
void extendSuffixTree(int pos) {
    leafEnd = pos;
    ++remainingSuffixCount;
    Node *lastNewNode = nullptr;

    while (remainingSuffixCount > 0) {
        if (activeLength == 0) {
            activeEdge = pos;
        }

        // Проверяем, есть ли ребро для текущего символа среди детей
        if (activeNode->children[pattern[activeEdge]] == nullptr) {
            // Создаем новый листовой узел
            activeNode->children[pattern[activeEdge]] = newNode(pos, &leafEnd);

            // Если есть предыдущий узел, создаем суффиксную ссылку
            if (lastNewNode != nullptr) {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = nullptr;
            }
        } else {
            Node *next = activeNode->children[pattern[activeEdge]];
            if (walkDown(next)) {
                continue;
            }

            // Если символ совпадает, увеличиваем длину совпадения
            if (pattern[next->start + activeLength] == pattern[pos]) {
                if (lastNewNode != nullptr && activeNode != root) {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = nullptr;
                }
                activeLength++;
                break;
            }

            // Разделяем ребро
            splitEnd = new int(next->start + activeLength - 1);
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[pattern[activeEdge]] = split;
```

```
            // Создаем новый лист и корректируем существующее ребро
            split->children[pattern[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[pattern[next->start]] = next;

            // Устанавливаем суффиксную ссылку
            if (lastNewNode != nullptr) {
                lastNewNode->suffixLink = split;
            }

            lastNewNode = split;
        }

        --remainingSuffixCount;

        // Переходим по суффиксной ссылке
        if (activeNode == root && activeLength > 0) {
            --activeLength;
            activeEdge = pos - remainingSuffixCount + 1;
        } else if (activeNode != root) {
            activeNode = activeNode->suffixLink;
        }
    }
}

// Функция для поиска подстроки в тексте с использованием прыжков по счётчику
bool checkSubstringAtPos(const string &text, int startPos) {
    int i = 0;
    Node *currentNode = root;

    while (i < size && startPos + i < text.size()) {
        char currentChar = text[startPos + i];
        if (currentNode->children[currentChar] == nullptr) {
            return false;
        }

        Node *nextNode = currentNode->children[currentChar];
        int edgeLen = getEdgeLength(nextNode);

        // Сравниваем символы на ребре
        int j = 0;
        while (i < size && j < edgeLen && startPos + i < text.size() && pattern[i] == text[startPos + i]) {
            ++i;
            ++j;
        }

        // Если не все символы совпали
        if (j < edgeLen) {
            return false;
        }

        currentNode = nextNode;
    }

    return (i == size);
}
```

```
public:
    SuffixTree(const string &pat) {
        pattern = pat;
        size = pattern.size();
        rootEnd = new int(-1);
        root = newNode(-1, rootEnd);
        activeNode = root;
        leafEnd = -1;
        activeEdge = -1;
        activeLength = 0;
        remainingSuffixCount = 0;

        for (int i = 0; i < size; ++i) {
            extendSuffixTree(i);
        }
    }

    // Функция, которая ищет все вхождения паттерна в текст
    void search(const string &text) {
        vector<int> result;
        for (int i = 0; i <= text.size() - size; ++i) {
            if (checkSubstringAtPos(text, i)) {
                result.push_back(i + 1);
            }
        }

        // Выводим результат
        for (int pos : result) {
            cout << pos << endl;
        }
    }
};

int main() {
    cin.tie(0);
    ios::sync_with_stdio(false);

    string pattern, text;
    cin >> pattern >> text;

    SuffixTree tree(pattern);
    tree.search(text);

    return 0;
}
```

Пример работы

Input	Output
aba qababaz	2 4
aa aaaaa	1 2 3 4
hello ohhellotherehello	3 13

Вывод

В ходе лабораторной работы была успешно реализована программа на языке C++, которая эффективно ищет вхождения паттерна в текст.

Мой код реализует построение суффиксного дерева для паттерна и поиск всех его вхождений в текст. Суффиксное дерево строится за линейное время ($O(m)$), что делает поиск подстроки эффективным — за ($O(n)$), где (m) и (n) соответственно длины паттерна и текста.

Преимущества:
- Быстрый поиск подстрок в тексте за линейное время;
- Универсальность: можно использовать для решения множества задач работы со строками.

Недостатки:
- Высокие требования к памяти;
- Сложность реализации по сравнению с другими алгоритмами, такими как Кнута-Морриса-Пратта или Бойера-Мура.

Таким образом, суффиксное дерево эффективно для поиска в больших текстах, но может быть избыточным для коротких строк и простых задач.