FСанкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материала и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Алгоритм Укконена

Выполнил студент гр. 3331506/20102 Жданов И.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

Оглавление

1. Введение

Данная работа посвящена изучению и практическому применению алгоритма Укконена, который представляет собой эффективный метод построения суффиксного дерева. Алгоритм Укконена значительно оптимизирует процесс создания суффиксных деревьев для произвольных строк, позволяя осуществлять это за линейное время относительно длины входной строки.

Постановка задачи:

В задачах, связанных с обработкой строк и текстов, часто возникает необходимость быстрого извлечения информации обо всех подстроках, их частотности или о наибольших общих подстроках. Стандартные подходы к построению суффиксных деревьев продвигаются с временной сложностью O(n^2), что делает их неэффективными для длинных строк. Алгоритм Укконена решает эту проблему, снижая временные затраты до O(n), что является значительным улучшением в задачах с большими объемами данных. Целью данной работы является изучение и анализ алгоритма Укконена, а также его практическое применение.

Актуальность и значимость:

Алгоритм Укконена является неотъемлемым инструментом для быстрого анализа строковых данных, особенно в области обработки текстов и поиска подстрок. Его линейная временная сложность делает его надежным выбором в задачах, где критично важна эффективность. По сравнению с другими методами, алгоритм Укконена обеспечивает большую скорость работы без значительных затрат дополнительных ресурсов.

Область применения: [2]

Алгоритм Укконена широко применяется в различных областях, таких как:

- **Текстовая аналитика**: Поиск и анализ текстовых данных, определение уникальных подстрок и их частот.
- **Биоинформатика**: Сравнение геномов и поиск совпадений в последовательностях ДНК.
- **Обработка естественного языка**: Задачи по индексации и поиска в больших объемах текстовой информации.
- **Системы управления базами данных**: Оптимизация запросов к строковым данным.
- **Программирование игр**: Реализация механизмов поиска и сопоставления текстовых данных в играх

2. Алгорит Укконена

2.1 Теоретические сведения

Алгоритм Укконена предназначен для построения суффиксного дерева [1] из строки. На момент создания (в 1995 году) он был самым эффективным алгоритмом в своем роде:

- При наивном подходе суффиксное дерево можно создать лишь за O(n3), в лучшем случае за O(n2), а решение Укконена выполняется за линейное время.
- Существующие алгоритмы построения суффиксных деревьев, предложенные Вайнером (в 1973 году) и МакКрейтом (в 1976 году) [3], относятся к оффлайновым, то есть для начала работы им нужна вся строка целиком, а алгоритм Укконена – онлайновый и работает последовательно.

Алгоритм не свободен от недостатков – суффиксное дерево должно быть полностью загружено в оперативную память, что может привести к проблемам при очень больших входных данных или алфавитах. Для обработки небольшого объема текстовых данных проще использовать префикс-функцию, а для работы с очень большими объемами данных лучше подходят кэш-эффективные алгоритмы, оптимизированные для выполнения на современных процессорах. [6]

Суффиксное дерево – это компактное, сжатое древовидное представление всех суффиксов данной строки. Каждый путь от корня к листу соответствует одному суффиксу исходной строки. Суффиксные деревья позволяют очень быстро находить вхождения подстрок в строку, и благодаря этому:

- Существенно упрощают решение некоторых важных задач, связанных с обработкой текста (поиск паттернов и сжатие).
- Находят широкое применение в вычислительной биологии и биоинформатике (анализ белковых последовательностей, ДНК, поиск точных совпадений).

2.2 Пример

Продемонстрируем компактность суффиксных деревьев на конкретном примере.

Так выглядит концептуальное древовидное представление строки MISSISSIPPI$, которая имеет 12 непустых суффиксов: [4,5]

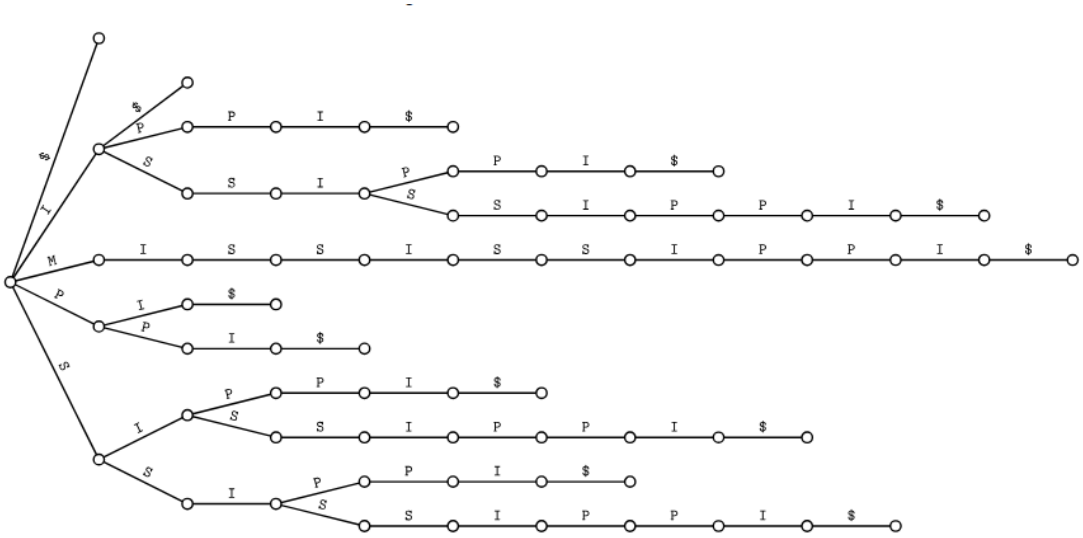

Рисунок 1 – Не сжатое суффиксальное древо MISSISSIPPI$

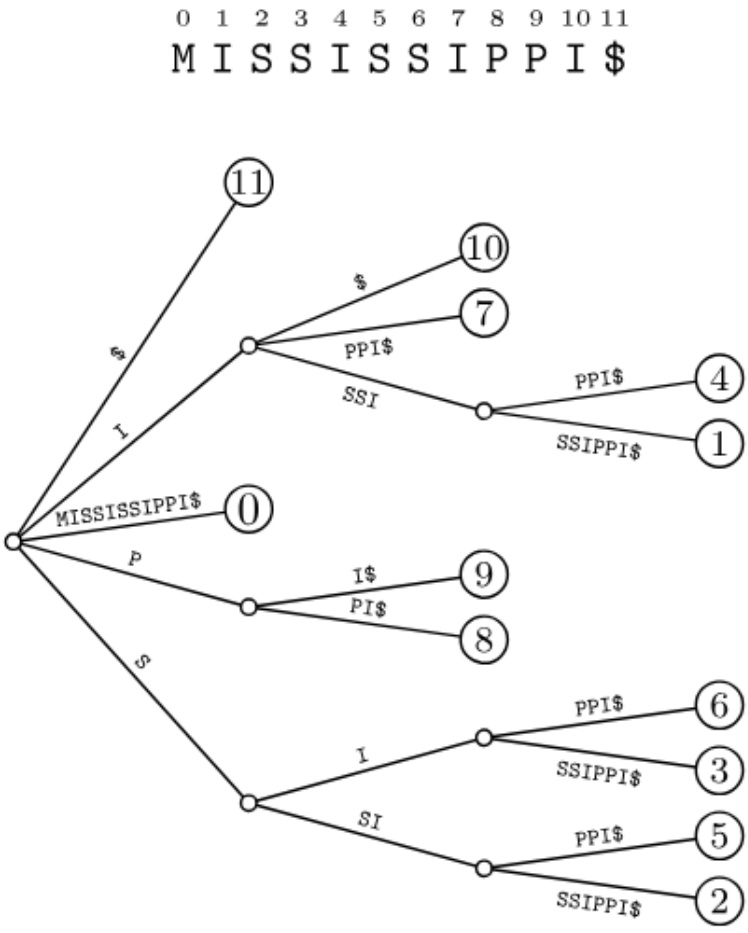

Рисунок 2 – Сжатое суффиксальное древо MISSISSIPPI$

3. Заключение

В ходе работы был изучен и продемонстрирован алгоритм Укконена, который является эффективным методом для построения суффиксных деревьев. Алгоритм позволяет динамически обрабатывать строки с логарифмической временной сложностью, что значительно превышает производительность наивных методов.

Было показано, что алгоритм Укконена способен решать задачи поиска подстрок, вычисления наибольших общих префиксов и других задач обработки текстов с высоким объемом данных. Это делает его незаменимым инструментом в области алгоритмов обработки строк и приложений, требующих быстрой и эффективной работы с текстовой информацией.

4. Литература

1. Левин, А. А. "Построение суффиксного дерева". 2013 г.
2. Кнут Д.Э. "Искусство программирования. Т. 4, А: Комбинаторные алгоритмы.2019
3. Скиена Стивен, Алгоритмы. Руководство по разработке, 2022 г.
4. Генри С. Уоррен мл. – Алгоритмические трюки для программистов, 2014 г.
5. Джордж Хайнеман, Гари Поллис, Стэнли Селков – Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017 г.
6. Лукин, А. Г. "Алгоритмы и структуры данных" 2018 г.

5. Приложение

```
#include <iostream>

#include <map>

#include <string>

#include <vector>



class Node {

public:

    std::map<char, Node*> children;

    int start;

    int* end;  // Указатель для конца

    Node* suffix_link;



    Node(int start, int* end) : start(start), end(end), suffix_link(nullptr) {}

};



class SuffixTree {

private:

    Node* root;

    std::string text;
```

```
    int *leafEnd;
    int size;
    int activeNode;
    int activeEdge;
    int activeLength;

public:
    SuffixTree(const std::string &str) {
        text = str;
        size = str.size();
        leafEnd = new int(-1);
        root = new Node(-1, leafEnd);
        root->suffix_link = root;
        activeNode = 0;  // Индекс активного узла
        activeEdge = -1; // Индекс активного ребра
        activeLength = 0; // Длина активного ребра
        build();
    }

    ~SuffixTree() {
```

```
        delete root; // Указатель на корень освобождается; необходимо реализовать рекурсивное удаление

        delete leafEnd;

    }



    void build() {

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

            extend(i);

        }

    }



    void extend(int pos) {

        leafEnd = new int(pos); // Обновляем конец листа

        int lastCreatedNode = -1; // Узел, созданный на предыдущем шаге

        bool continueProcessing = true; // Флаг для продолжения обработки



        while (pos >= 0 && continueProcessing) {

            if (activeLength == 0) activeEdge = pos; // Если длина активного ребра 0, устанавливаем его на текущую позицию



            char currentChar = text[activeEdge]; // Текущий символ

            if (root->children.find(currentChar) == root->children.end()) {
```

```
    // Если символ не найден, создаем новое ребро
    Node* leafNode = new Node(pos, leafEnd);
    root->children[currentChar] = leafNode; // Добавляем новое ребро
    if (lastCreatedNode != -1)
        root->children[char(lastCreatedNode)]->suffix_link = root;

    lastCreatedNode = currentChar; // Обновляем последний созданный узел
} else {
    Node* existingNode = root->children[currentChar];
    if (activeLength >= existingNode->end - existingNode->start) {
        // Если длина активного ребра больше, чем длина существующего ребра
        activeEdge += activeLength; // Переходим к следующему
        activeLength = 0;
        continue; // продолжаем с новым активным ребром
    }

    if (text[existingNode->start + activeLength] == text[pos]) {
        // Если символ совпадает
        if (lastCreatedNode != -1 && lastCreatedNode != currentChar) {
            root->children[char(lastCreatedNode)]->suffix_link = existingNode;
```

```
            lastCreatedNode = -1;

        }

        activeLength++;

    } else {

        // Если нет совпадения, создаем новое ребро

        Node* newInternalNode = new Node(existingNode->start, new int(existingNode->start + activeLength));

        root->children[currentChar] = newInternalNode; // Добавляем внутренний узел

        newInternalNode->children[text[existingNode->start + activeLength]] = existingNode; // Присоединяем старый узел

        existingNode->start += activeLength; // Обновляем состояние старого узла

        newInternalNode->children[text[pos]] = new Node(pos, leafEnd); // Новое ребро для текущего символа



        if (lastCreatedNode != -1)

            root->children[char(lastCreatedNode)]->suffix_link = newInternalNode;



        lastCreatedNode = currentChar;

    }



    if (activeNode == 0) {

        if (activeLength == 0) {
```

```
                continueProcessing = false; // Прекращаем обработку, если ничего не осталось
            } else {
                activeLength--; // уменьшаем длину
                activeEdge = pos - 1; // устанавливаем обратно активный край
            }
        }
    }
  }
};

  void display(Node* node, int level = 0) {
    if (node == nullptr) return;

    for (auto &child : node->children) {
        for (int i = 0; i < level; i++)
            std::cout << "  ";
        std::cout << child.first << ": ";
        for (int i = child.second->start; i <= *(child.second->end); i++)
            std::cout << text[i];
```

```
            std::cout << std::endl;
            display(child.second, level + 1); // Рекурсивный вызов для дочернего узла
        }
    }

    void display() {
        display(root);
    }

int main() {
    std::string text = "MISSISSIPPI$";
    SuffixTree suffixTree(text);
    suffixTree.display(); // Выводим дерево
    return 0;
}
```