Лабораторная работа № 5 по курсу Дискретный Анализ. Суффиксные деревья

Выполнил студент группы 08-307 МАИ *Павлов Иван.*

Условие

Кратко описывается задача:

1. Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки, с использованием суфф. дерева.

2. Вариант задания: 4. Линеаризация циклической строки.

Описание алгоритма

Для линеаризации циклической строки необходимо написать алгоритм, использующий структуру данных Суффиксное Дерево.

Суффиксное дерево - это структура данных, представляющая собой дерево, в котором каждый путь от корня до листа соответствует некоторому суффиксу входной строки. Алгоритм Укконена позволяет реализовать СД за линейное время.

Алгоритм Укконена:

1. Для быстрой работы алгоритма в узлах дерева не должны лежать сами суффиксы. Вместо этого можно хранить в узлах индексы начала и конца суффикса. Кроме этого в структуре узла должны находиться ребра для доступа к следующим вершинам, суффиксная ссылка - указатель на вершину и номер суффикса (для алгоритма линеаризации циклической строки он не нужен, но нужен для отладки).

2. В структуре СД хранится ссылка на текст, указатель на корень и несколько дополнительных переменных: счетчик суффиксов cnt, текущая вершина currentNode, текущее ребро currentEdge, текущая длина суффикса currentLength, счетчик максимального конца суффикса на текущей итерации globalEnd и последняя созданная внутренняя вершина lastCreatedNode.

3. При создании дерева создаем корень, в котором индексы не указывают на символы в тексте. Изначально currentLength = 0 (длина в пустом дереве), currentEdge = -1 (нет ребер), globalEnd = -1 (нет вершин), lastCreatedNode = nullptr (вершин еще не создавали), а к тексту добавляем sentinel, котрый больше в ASCII чем $'z'$, так как алгоритм Укконена не достраивает суффиксы для последнего символа, а сам sentinel нам не нужен. После объявления идем по тексту и на каждом шаге добавляем (с помощью функции) суффикс для каждого элемента текста.

4. Функция добавления суффикса. Применяется одно из трех правил:

    1. Если находимся в листе - дописываем букву
    2. Если пути нет - создаем новый лист
    3. Если в дереве уже есть вставляемая строка - ничего не делаем

    В начале каждой итерации делаем инкремент переменной globalEnd (правило 1), на которую ссылаются все end в листьях. Таким образом "добавляем"следующий символ в уже существующий лист. Далее увеличим cnt, который показывает, сколько суффиксов осталось создать на текущей итерации. Пока cnt больше нуля, выполняем следующие действия:

    1. Если currentLength = 0, выполняем поиск текущего символа из корня. Индекс текущего символа в тексте определит ребро, по которому будем двигаться.
    2. Ищем текущий символ среди меток ребер currentNode. Если не нашли - создаем новый лист (правило 2), при этом установив на нее суффиксную ссылку из lastCreatedNode (если она не nullptr). Иначе в lastCreatedNode запишем currentNode.
    3. Если мы нашли текущий символ среди меток ребер currentNode, то идем по этому ребру (прибавляем длину найденной вершины (в которую ведет ребро) к currentEdge, вычитаем ее же из currentLength, и записываем в currentNode эту вершину), пока длина суффикса в вершине не будет меньше currentLength.
    4. Если текущий символ уже есть на ребре (правило 3), то увеличим currentLength и смотрим на lastCreatedNode. Если она не nullptr, то она уже была создана ранее, значит надо создать из нее в currentNode суффиксную ссылку. После этого выходим из цикла.
    5. Если текущего символа нет на ребре, то создаем новую внутреннюю вершину. Также устанавливаем на нее суффиксную ссылку из lastCreatedNode, если она не nullptr. Записываем в lastCreatedNode новую внутерннюю вершину.
    6. После этого уменьшаем cnt на 1 (суффикс рассмотрели) и переходим к началу поиска для нового суффикса: если мы в корне - уменьшаем currentLength и увеличиваем currentEdge (начинаем поиск с корня), иначе переходим по суффиксной ссылке и начинаем поиск с нее.

Алгоритм линеаризации циклической строки:

1. Удваиваем входную строку и дописываем sentinel, лексиографически наибольший.
2. Строим из этой строки суффиксное дерево приведенным выше алгоритмом.
3. Проходим по суффиксному дереву, каждый раз переходя по ребру с лексиографически наименьшей меткой и записывая в результирующую строку все ребро, если его длина меньше чем длина исходного текта. Иначе возвращаем результат.

Описание программы

Программа состоит из класса узла СД, класса СД и функции main.
Класс узла:

- Ребра хранятся в std::map с меткой в ключе.
- Чтобы удобнее было применять правило 1, храним указатель на End, который ссылается либо на globalEnd (листья), либо на nullptr (при удалении), либо на выделенную память (внутренние вершины).
- При вызове деструктора уничтожаем все ребра.

Класс дерева:

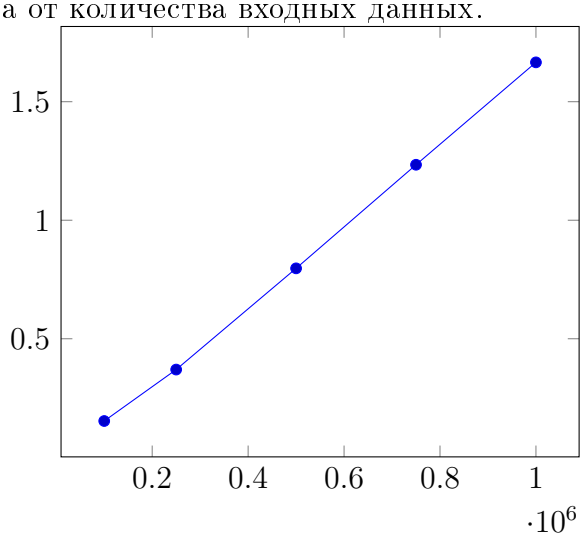
- Исходный текст при создании не копируется, так как передаем по ссылке
- При вызове деструктора уничтожаем всю память, выделенную на end во внутренних вершинах, затем вызываем delete от корня (деструкторы узлов сам реализуют рекурсивное уничтожение).

Дневник отладки

WA4 Неправильно разбивал циклическую строку.

Тест производительности

Оценим производительность построения суффиксного дерева с помощью алгоритма Укконена. Синим цветом выделена зависимость времени построения в алгоритме Укконена от количества входных данных.



Из графика видна линейная зависимость.

Выводы

В ходе данной лабораторной работы я изучил алгоритм Укконена и его применение для построения суффиксных деревьев. Алгоритм Укконена является эффективным методом построения суффиксного дерева за линейное время от размера входной строки.

В процессе выполнения лабораторной работы я реализовал алгоритм Укконена. Сложность алгоритма Укконена зависит от размера входной строки. В наихудшем случае сложность составляет $O(n^2)$, где n - длина строки. Однако, с использованием оптимизаций, таких как структуры данных для хранения промежуточных результатов, сложность алгоритма может быть снижена до $O(n)$.