Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 5 по курсу дискретного анализа: Суффиксные деревья

Студент: А. О. Тояков
Преподаватель: А. Н. Ридли
Группа: М8О-307Б-18
Дата:
Оценка:
Подпись:

Москва
2021

Условие

1. Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из выходных строк, необходимо воспользоваться полученным суффисным деревом для решения своего варианта задания.
   Алфавит строк: строчные буквы латинского алфавита (т.е. от a до z).

2. Вариант: Найти в заранее известном тексте поступающие на вход образцы с использование суффиксного массива.

Метод решения

1. Для начала нужно построить суффиксное дерево с помощью алгоритма Укконена сложностью O(m), где m - колчисетво элементов в строке.

   Существуют несколько правил добавления, а также оптимизаций данного алгоритма для приведения ко времени O(m):

   (a) Суффиксная связь + прыжки по счётчику: суффиксная связь необходима, чтобы за константное время переходить от одной вершины к другой. Предположим, что есть строки Xa и a. Тогда необходимо построить суффиксную ссылку из Xa в a, также суффиксная свзяь может быть направлена к корню. Прыжки по счётчику также позволяют за константное время проходить по вершинам, которые находятся на одном ребре.

   (b) Был листом, листом и останешься (Достаточно просто увеличивать end или присвоить бесконечность, чтобы не возвращаться к данному листу).

   (c) На рёьрах информация хранится в виде [start, end], где start - индекс начала строки, а end - индекс конца.

   (d) Также если при добавлении элемента X, есть строка, у которой элемент X является последним, то достаточно просто закончить фазу.

2. Затем необходмо построить суффиксный массив из дерева, выполнив поиск в глубину.

3. В конце концов выполняем поиск подстрок в строке, используя построенный ранее суффиксный массив.

Описание программы

Алгоритма Укконена реализован следующим образом. Каждый узел содержит итераторы, указывающие на начало и конец этой подстроки в тексте, суффиксную ссылку,

которая либо указывает на вершину с таким же суффиксом как и в этой, только без первого символа, либо при отсутствии такой вершины на корень. В дереве храним текст (в конце которого терминальный символ), по которому ищем, указатель на корень, переменную remainder, которая показывает сколько суффиксов ещё надо вставить. Структура activePoint указывает на вершину node, которая имеет ребро edge, в котором мы сейчас находимся и длину len, которая показывает на каком расстроянии от этой вершины мы находимся.

При создании дерева итеративно проходим по тексту. на каждой итерации начинается новая фаза и remainder увеличивается на 1. Далее пока все невставленные суффиксы не помещены в дерево выполняем цикл. Если в той вершине, в которой мы остановились ещё нет ребра, начинающегося с первой буквы обрабатываемого суффикса, то по правилу 1 продолжений создаём новую вершину, которая будет листом. Если это необходимо, создаём суффиксную ссылку (если до этого в этой фазе была создана вершина по 2 правилу продолжений). Если в той вершине, в которой мы остановились, уже есть такое ребро, то нужно пройти вниз по рёбрам на len и обновить activePoint. Если некоторой путь на этом ребре начинается со вставляемого символа, значит по правилу 3 продолжений нам ничего делать не надо, заканчиваем фазу, оставшиеся суффиксы будут добавлены неявно. Увеличивем len на 1 (т. к. этот символ уже появился на нашем пути), по необходимости строим суффиксную ссылку. Если никакой путь не начинается со вставляемого символа, то нужно разделить ребро в этом месте, вставив 2 новые вершины - одну листовую и одну разделяющую ребро. Далее по необходимости добавляем суффиксную ссылку. Уменьшаем remainder на 1, если вставили суффикс в цикле. Если после всех этих действий active Point указывает на корень и len больше 0, то уменьшаем len на 1, а edge устанавливаем на первый символ нового суффикса, который нужно вставить. Если activePoint не коренью то переходим по суффиксной ссылке.

После конструирования дерева строим суффиксный массив. В нём расположен вектор, в котором находятся все начальные позиции суффиксов и все эти суффиксы лексикографически упорядочены. Массив строим из дерева, выполняя обход в глубину. Т. к. словарь который находится в каждой вершине, это упорядоченный контейнер, то номера позиции после обхода в глубину будут также лексикографически упорядочены.

Поиск вхождений в массиве осуществляется с помощью бинарного поиска. В зависимости от того, лексикографически меньше или больше буква в паттерне и буква в тексте, границы поиска в массиве сужаются наполовину. В конце возвращается диапазон начальных позиций, в которых найдены вхождения.

Исходный код

```
#ifndef SUFF_TREE_H
#define SUFF_TREE_H
```

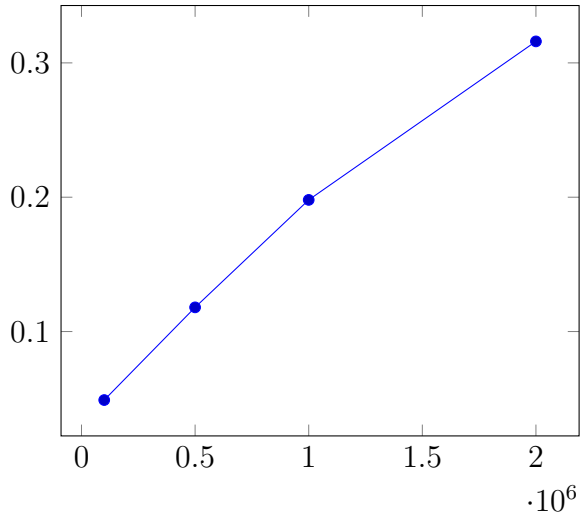
```
#include <map>
#include <vector>
#include <algorithm>
#include <memory>
#include <string>
#include <iostream>
#include <exception>

namespace NSuff {

    class TNode {
    public:
        TNode(const size_t&, const std::string&, size_t, size_t);
        ~TNode() { };

        TNode* FindChildByChar(const char);
        void   AddLeaf(const size_t);
        TNode* AddNode(const size_t, const size_t, const char);

        TNode* GetSuffixLink();
        void   SetSuffixLink(TNode*);

        size_t GetLower();
        void   SetLower(size_t);

        size_t GetUpper();

        size_t      GetLength();
        char        GetChar(size_t);
        std::string GetString();


        void PrintNode(size_t);

        void FillArray(std::vector<size_t>&, size_t);

    private:
        typedef std::unique_ptr<TNode>      TUniquePtr;
        typedef std::map<char, TUniquePtr>  TMap;
        typedef std::pair<const char, TUniquePtr> TPair;

        TMap   children;
```

```
    TNode* suffixLink;
    struct {
        size_t lower;
        size_t upper;
    } bounds;
    const size_t&      globalUpper;
    const std::string& strRef;

};

class TSuffTree {
public:
    TSuffTree();
    TSuffTree(std::string);

    TSuffTree& SetString(const std::string);
    TSuffTree& Construct();

    void       PrintTree();

    std::vector<size_t> CreateSuffArray();

private:
    typedef std::unique_ptr<TNode> TUniquePtr;

    TNode*      prevCreated;
    TUniquePtr  root;
    std::string originalStr;
    size_t      globalUpper;
    size_t      remainder;

    struct {
        TNode* node;
        char   edge;
        size_t length;
    } activePoint;

    void IncrementGlobal();
    void TryLink(TNode*);
    void WalkDown(size_t, TNode*);
};
```

```
    class TSuffArray {
    public:
        TSuffArray(std::vector<size_t>, std::string);

        std::vector<int> Find(std::string);

        void PrintArray();
        void PrintLcp();

    private:
        typedef std::vector<size_t> TArray;

        TArray suffArray;
        TArray lcp;
        std::string originalStr;

        void   CalcLcp();
        size_t GetLCP(size_t min, size_t max);
    };
}

#endif
```

Тест производительности

Тесты представляют из себя набор строк, где размер строки равен 10 символам и данный размер одинаков для всех тестов. Увеличивается только количество элементов в тесте.
В каждом тесте текст состоит из 1000 символов исходного входного алфавита (a-z).



Пояснения к графику: Ось y - время в секундах. Ось x - количество строк.

1. За 0.049 обрабатывается 100000 образцов.
2. За 0.118 обрабатывается 500000 слов.
3. За 0.198 обрабатывается 1000000 слов.
4. За 0.198 обрабатывается 1000000 слов.

Выводы

Существует несколько алгоритмов построения суффиксного дерева за линейное время. В ходе работы я реализовал алгоритм Укконена. Суффиксное дерево показалось мне одной из самых сложных структур данных для понимания. Построив суффиксное дерево, можно найти линеаризацию, количество вхождений, общие подстроки или построить суффиксный массив. Данный факт доказывает универсальность суффиксного дерева, а также его пользу. Однако в некоторых ситуациях суффиксное дерево может занимать много памяти, поэтому на практике оно не всегда применимо. В таких случаях логичнее использовать суффиксный массив или простой вектор.