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: П. А. Харьков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-19

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №5

Задача: 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).

Вариант: Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

1 Описание

Требуется реализовать алгоритм Укконена, а затем линеаризовать циклическую строку. Суффиксное дерево - дерево, которое мы будем строить с помощью алгоритма Укконена является бором, который содержит в себе все суффиксы строки. Для того, чтобы сложность построения суффиксного дерева была линейной, нам необходимо использовать сжатое суффиксное дерево, на ребрах хранить не сами буквы, а только их индексы начала и конца в тексте и использовать суффиксные ссылки для быстрого перехода к суффиксу подстроки.

Для того, чтобы линеаризовать циклическую строку, нам необходимо прибавить к данной строке её же саму - в полученной строке будут содержаться все возможные разрезы этой строки. Затем для нее нужно построить суффиксное дерево и пройти по наименьшим ребрам.

2 Исходный код

Для начала нам необходимо поспользовать алгоритмом Укконена для построения дерева. Допустим нам надо вставить букву c:

1. Сначала увеличиваем количество суффиксов, которые нужно будет вставить, если это количество равно 1, то значит, мы в корне и следующее ребро должно начинаться с c.

2. Затем мы ищем, есть ли какое-либо ребро, начинающееся со буквы, на которой мы остановились.

3. Если такого ребра нет, то создаем его и рассматриваем узел, доступный по суффиксной ссылке.

4. Если такое ребро есть, но мы пришли к его концу, то переходим в следующее ребро.

5. Затем проверяем, совпала ли наша буква с буквой, которая находится следующей в ребре. Если совпала, то переходим к следующей букве текста.

6. Если же не совпала, то нам необходимо разбить узел на 2 узла, а затем рассмотреть узел, доступный по суффиксной ссылке.

После того, как мы построили суффиксное дерево, нам необходимо лишь пройтись по ребрам, которые начинаются с наименьшей буквы. В результате мы получим линеаризованную строку.

main.cpp	
int main()	Функция, которая считывает входную строку и вызывает функции построения дерева и поиска минимума.
tree.cpp	
TTree::TTree(string str)	Функция, инициализирующая дерево и вызывающая функцию, строящую дерево.
TNode* TTree::FindNode(TNode* NodeToFind, char c)	Функция, которая ищет ребро в текущем узле, начинающееся с c.
int TTree::EdgeLen(TNode* node)	Функция, вычисляющая длину ребра.
void TTree::Add(int i)	Функция, добавляющая i-ый символ строки в дерево.
string TTree::FindMin()	Функция, ищущая минимальный разрез строки.
void TTree::DeinitNode(TNode* node)	Функция, удаляющая узлы дерева.

Структуры и классы:

```
struct TNode{
    int Lpos, Rpos;
    TNode* Link;
    unordered_map<char, TNode*> Next;

    TNode(int lpos, int rpos):Lpos(lpos), Rpos(rpos){
        Link = nullptr;
    }
};

class TTree{
    private:
        string Text;
        TNode* Root;
        TNode* CurNode;
        int CurLen, End, Rem;
        int CurEdge;

        void DeinitNode(TNode* node);
        TNode* FindNode(TNode* NodeToFind, char c);
        int EdgeLen(TNode* node);
    public:
        void Add(int i);
        string FindMin();
        TTree(string str);
        ~TTree(){
            DeinitNode(Root);
        }
};
```

3 Консоль

```
p.kharkov$ make
g++ -std=c++14 -pedantic -O2 -c src/tree.cpp -o src/tree.o
g++ -std=c++14 -pedantic -O2 src/laba5.cpp src/tree.o -o laba5
p.kharkov$ cat tests/main
xabcd
p.kharkov$ ./laba5 <tests/main
abcdx
```

4 Тест производительности

Тест производительности представляет из себя следующее: поиск минимального в лексикографическом смысле разреза циклической строки с помощью алгоритма Укконена и с помощью наивного алгоритма. В первом тесте на вход подается строка длины 100000, состоящая из маленьких и больших латинских букв. Во втором тесте подается строка, длина которой в два раза больше.

```
p.kharkov$ ./banchmark <tests/01.t
Naive solution time: 1792ms
Ukkonen solution time: 203ms
p.kharkov$ ./banchmark <tests/02.t
Naive solution time: 5718ms
Ukkonen solution time: 492ms
```

Как видно, поиск, основанный на построении суффиксного дерева с помощью алгоритма Укконена гораздо эффективнее наивного алгоритма. Наивный алгоритм в данном случае реализован простым перебором всех возможных вариантов строки. В таком случае в худшем случае его сложность равна $O(n^2)$, а сложность с помощью построения дерева по алгоритму Укконена рана $O(n)$.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я изучил и написал алгоритм Укконена. Это очень полезный алгоритм, так как сложность построения суффиксного дерева составляет $O(n)$, где n - длина текста, а поиск образов занимает $O(m)$, где m - длина образца. Это очень большое ускорение, так как поиск образцов с помощью наивного сравнения занимал бы $O(n * m)$.

Также мне понравилось решение задачи, которое я написал. Мне кажется, что если бы задача не была по этой теме, было бы очень сложно предположить, что она решается с помощью суффиксных деревьев.

Список литературы

[1] *Алгоритм Укконена — ИТМО.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 10.05.2021).