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: Н. П. Ежов
Преподаватель: Н. С. Капралов
Группа: М8О-204Б
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 написать реализацию алгоритма Укконнена для построения суффиксного дерева и при помощи него решить поставленную задачу. Мы будем строить явное суффиксное дерево, терминирующим символом будет \$. Опишем т.н. **правила продолжения суффиксов:**
Укажем точные правила для *продолжения суффикса*. Пусть $S[j..i] = \beta$ - суффикс $S[1..i]$. В продолжении j, когда алгоритм находит конец β в текущем дереве, он продолжает β, чтобы обеспечить присутствие суффикса $\beta S(i+1)$ в дереве. Алгоритм действует по одному из следующих трёх правил:
Правило 1. В текущем дереве путь β кончается в листе. Это значит, что путь от корня с меткой β доходит до конца некоторой "листовой" дуги (дуги, входящие в лист). При изменеии дерева нужно добавить к концу метки этой листовой дуги $S(i+1)$.
Правило 2. Ни один путь из конца строки β не начинается символом $S(i+1)$, но по крайне мере один начинающийся оттуда путь имеется. В этом случае должна быть создана новая листовая дуга, начинающаяся в конце β и помеченная символом $S(i+1)$. При этом, если β кончается внутри дуги, должна быть создана новая вершина. Листу в конце новой листовой дуги сопоставляется номер j.
Правило 3. Некоторый путь из конца строки β начинается символом $S(i+1)$. В этом случае строка $\beta S(i+1)$ уже имеется в текущем дереве, так что ничего не надо делать.
Для ускорения алгоритма используем суффиксные связи. Пусть $x\alpha$ обозначает произвольную строку, где x - ее первый символ, а α оставшаяся подстрока (возможно, пустая). Если для внутренней вершины v с путевой меткой $x\alpha$ существует другая вершина $s(v)$ с путевой отметкой α, то указатель из v в $s(v)$ называется *суффиксной связью*. Суффиксную связь будем обозначать парой $(v, s(v))$. Если строка α пуста, связь идёт в корневую вершину. Но это лишь уменьшает кол-во передвижений от корня в каждом продолжении.
Для оптимизации алгоритма используем 3 приёма
Приём 1.
"Перескок". Суть этого приёма заключается в том, что если мы хотим найти место для вставки вхождения строки в дерево, то мы не идём посимвольно, если длина строки γ больше длины текущей дуги. В таком случае мы её просто пропускаем, а идти начинаем от $|\gamma|-$ длина дуги.
Приём 2.
Заканчивать каждую фазу $i+1$ после первого же использования правила 3. Если это случится в продолжении j, то уже не требуется явно находить концов строк $S[k..i]$ с $k > j$. Также стоит учесть, что дуги мы "сжимаем" до дуговых меток, т.е. это у нас не массив суффиксов, а пара чисел (p, e), где e - "текущий конец"
Приём 3.
Если дуга стала листовой - она таковой и останется. Из этого следует, что e для

листов можно ввести как глобальную переменную, и работать с ней за константу.

TNode	
TNode(TNode *link, int start, int *end)	конструктор дуги без индекса, используется при создании новых внутренних вершин
TNode(TNode *link, int start, int *end, int int)	Конструктор с индексом используется при создании лисьтев
map<char, TNode *> Children	Массив "детей".
TNode* SuffixLink	Суффиксная ссылка.
int Start	Индекс первого символа вершины в тексте.
int *End	Индекс последнего символа вершины в тексте
int SuffixIndex	Индекс суффикса.

TSuffixTree	
TSuffixTree(string &text)	Конструктор дерева из строки.
void BuildSuffixTree()	Функция построения суффиксного дерева.
std::string Linearization(int n)	Функция линеаризации среза циклической строки.
TSuffixTree()	Деструктор
void LinHelp(TNode *curr, int n, std::string &res)	Вспомагательная рекурсивная функция для вычисления линеаризованной строки.
void ExtendSuffixTree(int pos)	Расширение дерева.
void DeleteSuffixTree(TNode *TNode)	Вспомагательная функция для удаления суффиксного дерева.
int EdgeLength(TNode *TNode)	Вычисляет длину дуги.
TNode *Root	Указатель на корневой узел дерева.
TNode *LastCreatedInternalNode	Последняя созданная внутренняя вершина.
string Text	Текст, на основе которого было построено суфф. дерево.
TNode *ActiveNode	То, откуда начнется расширение на следующей фазе.
int ActiveEdge	Индекс символа, который задает движение из текущей ноды
int ActiveLength	на сколько символов идём в направлении activeEdge.

int RemainingSuffixCount	Сколько суффиксов осталось создать.
int LeafEnd	Глобальная переменная, "конец" листовой дуги.

Для того, чтобы линеаризовать срез циклической строки, надо понять, сколько срезов нам понадобится для нахождения минимального лексигографического среза. Допустим, что самый "маленький"символ находится в конце среза, и он в нём единственный. Это будет худший случай, для которого потребуется $n + n - 1$ символов. Возьмём два среза для простоты, т.е. $2n$. Как найти минимальный срез при помощи суфф. дерева: от каждой вершины у нас идут дуги, нужно просто каждый раз выбирать минимальную дугу и доходить до конца суффикса. Если суффикс, например, размера меньше n, то надо вернуться на шаг назад и взять следующую по возрастанию дугу и т.д., пока мы не найдём нужный нам минимальный срез. Не во всех случаях найденный суффикс будет являться минимальным срезом строки, т.к. он может быть размера больше n, поэтому от первого такого найденного суффикса мы возвращаем первые n символов, это и будет наш ответ.

2 Исходный код

```
1  //
2  // suffTree.cpp
3  //
4
5  #pragma once
6
7  #include <iostream>
8  #include <string>
9  #include <vector>
10 #include <algorithm>
11 #include <map>
12
13 #define TERMINATION_SYMBOL '$'
14
15 using namespace std;
16
17 class TSuffixTree;
18
19 class TNode
20 {
21 private:
22     map<char, TNode *> Children;
23     TNode *SuffixLink;
24     int Start;
25     int *End;
26     int SuffixIndex;
27 public:
28     friend TSuffixTree;
29
30
31     TNode(TNode *link, int start, int *end) : TNode(link, start, end, -1){
32     }
33
34
35     TNode(TNode *link, int start, int *end, int ind) :
36     SuffixLink(link),Start(start),End(end),SuffixIndex(ind){
37     }
38 };
39
40 class TSuffixTree
41 {
42 private:
43     void LinHelp(TNode *curr, int n, std::string &res){
44         if (!curr)
45             return;
46         for (auto it : curr->Children) {
47             LinHelp(it.second, n, res);
```

```
48              if(res != "") {
49                  return;
50              }
51          }
52          if (curr->SuffixIndex != -1 && n <= *curr->End - curr->SuffixIndex + 1
53              && Text[curr->SuffixIndex + n - 1] != TERMINATION_SYMBOL) {
54              res = Text.substr(curr->SuffixIndex, n);
55          }
56      }
57      void ExtendSuffixTree(int pos){
58
59          LastCreatedInternalNode = nullptr;
60          LeafEnd++;
61          RemainingSuffixCount++;
62          while (RemainingSuffixCount > 0){
63              if (ActiveLength == 0) {
64                  ActiveEdge = pos;
65              }
66
67              auto find = ActiveNode->Children.find(Text[ActiveEdge]);
68
69              if (find == ActiveNode->Children.end()){
70                  ActiveNode->Children.insert(make_pair(Text[ActiveEdge],
71                                                        new TNode(Root, pos, &LeafEnd, pos -
                                                              RemainingSuffixCount + 1)));
72
73                  if (LastCreatedInternalNode != nullptr){
74                      LastCreatedInternalNode->SuffixLink = ActiveNode;
75                      LastCreatedInternalNode = nullptr;
76                  }
77              }
78              else{
79
80                  TNode *next = find->second;
81                  int edge_length = EdgeLength(next);
82
83                  if (ActiveLength >= edge_length){
84                      ActiveEdge += edge_length;
85                      ActiveLength -= edge_length;
86                      ActiveNode = next;
87                      continue;
88                  }
89
90                  if (Text[next->Start + ActiveLength] == Text[pos]){
91                      if (LastCreatedInternalNode != nullptr && ActiveNode != Root) {
92                          LastCreatedInternalNode->SuffixLink = ActiveNode;
93                      }
94                      ActiveLength++;
95                      break;
```

```
96                }
97
98                TNode *split = new TNode(Root, next->Start, new int(next->Start +
                      ActiveLength - 1));
99                ActiveNode->Children[Text[ActiveEdge]] = split;
100               next->Start += ActiveLength;
101               split->Children.insert(make_pair(Text[pos], new TNode(Root, pos, &
                      LeafEnd, pos - RemainingSuffixCount + 1)));
102               split->Children.insert(make_pair(Text[next->Start], next));
103               if (LastCreatedInternalNode != nullptr) {
104                   LastCreatedInternalNode->SuffixLink = split;
105               }
106               LastCreatedInternalNode = split;
107           }
108
109           RemainingSuffixCount--;
110
111           if (ActiveNode == Root && ActiveLength > 0){
112               ActiveLength--;
113               ActiveEdge++;
114           }
115           else if (ActiveNode != Root) {
116               ActiveNode = ActiveNode->SuffixLink;
117           }
118       }
119   }
120   void DeleteSuffixTree(TNode *TNode){
121       for (auto it : TNode->Children) {
122           DeleteSuffixTree(it.second);
123       }
124       if (TNode->SuffixIndex == -1)
125           delete TNode->End;
126       delete TNode;
127   }
128   int EdgeLength(TNode *TNode){
129       return *TNode->End - TNode->Start + 1;
130   }
131
132
133   TNode *Root = new TNode(nullptr, -1, new int(-1));
134   TNode *LastCreatedInternalNode = nullptr;
135   string Text;
136
137   TNode *ActiveNode = nullptr;
138   int ActiveEdge = -1;
139   int ActiveLength = 0;
140   int RemainingSuffixCount = 0;
141   int LeafEnd = -1;
142public:
```

```
143     TSuffixTree(string &text){
144         this->Text += text+TERMINATION_SYMBOL;
145         BuildSuffixTree();
146     }
147     void BuildSuffixTree(){
148         ActiveNode = Root;
149         for (size_t i = 0; i < Text.length(); i++) {
150             ExtendSuffixTree(i);
151         }
152     }
153     std::string Linearization(int n){
154         TNode *current = Root;
155         std::string res = "";
156         for (int i = 0; i < n; ++i){
157             auto it = current->Children.begin();
158             for (;it != current->Children.end(); ++it){
159                 LinHelp(it->second, n, res);
160                 if(res != ""){
161                     break;
162                 }
163             }
164         }
165         return res;
166     }
167 
168     ~TSuffixTree() {
169         DeleteSuffixTree(Root);
170     }
171 };
```

```
1  [language=C++]
2  //
3  // main.cpp
4  //
5  #include <iostream>
6  #include "suffTree.hpp"
7  int main(){
8      ios::sync_with_stdio(0);
9      cout.tie(0), cin.tie(0);
10     string text, pattern;
11     cin >> text;
12     int n = text.size();
13     text = text+text;
14     TSuffixTree suffixTree(text);
15     std::string res = suffixTree.Linearization(n);
16     std::cout << res << "\n";
17     return 0;
18 }
```

3 Консоль

```
nezhov@killswitch:~/CLionProjects/Diskran/lab5$ make
g++ -std=c++14 -O3 -Wextra -Wall -Werror -Wno-sign-compare -Wno-unused-result
-pedantic -o solution main.cpp
nezhov@killswitch:~/CLionProjects/Diskran/lab5$  cat tests/01.t && ./solution
<tests/01.t
ixtvoqfzlx
fzlxixtvoq
```

4 Тест производительности

Алгоритм мы будем сравнивать с наивным поиском минимальнго разреза, т.е. мы будем от каждого $i[0..n]$ проходить максимум n раз, чтобы найти минимальный срез. Худшая оценка данного алгоритма $O(n^2)$, поэтому, казалось бы, суфф. дерево должно выигрывать всухую, но...

```
[info] [2021-05-12 08:39:23] Running tests/05.t
Size of string is 100000
naive linearization 0 ms
suffTree linearization 104 ms
[info] [2021-05-12 08:39:23] Running tests/06.t
Size of string is 1000000
naive linearization 3 ms
suffTree linearization 1572 ms
[info] [2021-05-12 08:39:26] Running tests/07.t
Size of string is 10000000
naive linearization 19 ms
suffTree linearization 20802 ms
```

Глядя на результаты я сначала сильно удивлися, а потом вспомнил, что все тесты у меня рандомно генерируемые, т.е. в строках очень много различающихся символов, а т.к. второй проход можно остановить, если один из сравниваемых элементов больше или меньше другого (в случае, если символ, перебираемый по циклу, меньше, то мы обновляем минимальный индекс и останавливаем цикл, в ином же случае, если перебираемый по циклу символ больше того, что принадлежит текущему минимальному срезу, то просто останавливаем цикл). А значит, что второй вложенный цикл проходит сравнительно небольшое кол-во итерация, из-за чего он почти не влияет на результат.

5 Выводы

В ходе выполнения данной работы по дискретному анализу я познакомился с суффиксными деревьями, а благодаря /citeGusfield реализовал алгоритм Укконена построения суффиксного дерева за $O(n)$. Конечно, я удивлён результатами теста производительности, но это можно легко объяснить рандомом, который генерирует очень простые для "наивного" алгоритма тесты и который имеет оч. малую вероятность выдать что-то близкое к крайнему случаю.

Список литературы

[1] Гасфилд Дэн. *Строки, деревья и последовательности в алгоритмах* — Издательский дом «БХВ-Петербург». Перевод с английского: И.В. Романовский — 126-142 с.(дата обращения: 05.04.2021).