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: Д. А. Тарпанов
Преподаватель: Н. С. Капралов
Группа: М8О-307Б-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:

Найти самую длинную общую подстроку двух строк.

Формат входных данных

Две строки.

Формат результата
На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

1 Описание

Требуется реализовать алгоритм Укконена за время $O(n)$. Идея заключается в том, чтобы взять наивный алгоритм, работающий за $O(n^3)$ и оптимизировать его до линии. Наивный алгоритм будет добавлять все суффиксы каждого преффикса строки. Эвристика 1: пусть в суффиксном дереве есть строка xa (x – первый символ, a – оставшаяся строка). Тогда a тоже будет в дереве, т.к. является суффиксом xa. Если для строки xa существует некоторая вершина u, то вершина u существует и для строки a. Ссылка из u в v называется суффиксной ссылкой. Она позволяет не проходить каждый раз по дереву из корня. Для построения суффиксных ссылок достаточно хранить номер последней созданной вершины. Если на этой фазе мы создаем еще одну новую вершину, то нужно построить суффиксную ссылку из предыдущей в текущую. Это улучшение позволяет сократить временную сложность до $O(n^2)$.
Эвристика 2: для ускорения до линии необходимо оптимизировать хранение данных в дереве. Вместо строк будем хранить индексы начала и конца, причем переменная с индексом конца будет общей для всех вершин дерева, и её инкрементирование будет проходить за $O(1)$.
При использовании вышеописанных эвристик временная сложность сократится до $O(n)$.
Для нахождения наибольшей общей подстроки двух строк конкатенируем строки, добавив между ними символ-разделитель. Построим суффиксное дерево для получившейся строки. Потом необходимо обойти дерево и отметить вершины, принадлежащие одной первой строке, вершины, принадлежащие второй строке, и вершины, принадлежащие обоим строкам. Тогда наибольшей общей подстрокой будет самая длинная строка, принадлежащая обоим строкам.

2 Исходный код

В файле SuffTree.cpp содержится класс суффиксного дерева, в файле main.cpp содержится функция main, осуществляющая ввод дерева и вызов функции поиска наибольшей общей подстроки.

Листинг main.cpp

```
#include "SuffTree.cpp"

int main() {
    std::string a, b;
    std::cin >> a >> b;
    if (a.empty() || b.empty()) {
        exit(0);
    }
    TSuffTree tree(a, b);
    tree.LongestCommonSubstring();
}
```

Листинг SuffTree.cpp

```
#pragma once

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <set>

class TSuffTree {
private:
    class TNode {
    private:
        std::unordered_map<char, TNode*> children;
        TNode *link;
        int start, *end;
        int ind;
    public:
        friend TSuffTree;
        TNode(TNode *_link, int _start, int *_end, int _ind) : link(_link), start(
            _start),
                                                               end(_end), ind(_ind) {}
        TNode(TNode *_link, int _start, int *_end) : TNode(_link, _start, _end, -1) {}
    };
private:
```

```
25      void DeleteHelper(TNode* node) {
26          if (node == nullptr) {
27              return;
28          }
29          for (auto it : node->children) {
30              DeleteHelper(it.second);
31          }
32          if (node->ind < -1) {
33              delete node->end;
34          }
35          delete node;
36      }
37      int EdgeLength(TNode *node) {
38          return *(node->end) - (node->start) + 1;
39      }
40      void ExtendTree(int ind) {
41          internalNode = nullptr;
42          leafEnd++;
43          remainSuff++;
44          while (remainSuff > 0) {
45              if (activeLength == 0) {
46                  activeEdge = ind;
47              }
48              auto findSymb = activeNode->children.find(text[activeEdge]);
49              if (findSymb == activeNode->children.end()) {
50                  activeNode->children.insert(std::make_pair(text[activeEdge], new TNode(
                        root, ind, &leafEnd, ind - remainSuff + 1)));
51                  if (internalNode != nullptr) {
52                      internalNode->link = activeNode;
53                      internalNode = nullptr;
54                  }
55              } else {
56                  TNode *next = findSymb->second;
57                  int edgeLen = EdgeLength(next);
58                  if (activeLength >= edgeLen) {
59                      activeEdge += edgeLen;
60                      activeLength -= edgeLen;
61                      activeNode = next;
62                      continue;
63                  }
64                  if (text[next->start + activeLength] == text[ind]) {
65                      if (internalNode != nullptr && activeNode != root) {
66                          internalNode->link = activeNode;
67                      }
68                      activeLength++;
69                      break;
70                  }
71                  TNode *split = new TNode(root, next->start, new int(next->start +
                        activeLength - 1));
```

```
72                  activeNode->children[text[activeEdge]] = split;
73                  next->start += activeLength;
74                  split->children.insert(std::make_pair(text[ind], new TNode(root, ind, &
                        leafEnd, ind - remainSuff + 1)));
75                  split->children.insert(std::make_pair(text[next->start], next));
76                  if (internalNode != nullptr) {
77                      internalNode->link = split;
78                  }
79                  internalNode = split;
80              }
81              remainSuff--;
82              if (activeNode == root && activeLength > 0) {
83                  activeLength--;
84                  activeEdge++;
85              } else if (activeNode != root) {
86                  activeNode = activeNode->link;
87              }
88          }
89      }
90      void PrintHelper(TNode* p) {
91          static int level = 0;
92          level++;
93          if (p->start == -1) {
94              printf("\\_<root>");
95          }
96          else {
97              printf("_<%d, %d>[%d]", p->start, *p->end, p->ind);
98              //std::cout << text.substr(p->start, *p->end) << "[" << p->ind << "]";
99          }
100         for (int i = 0; i < (int)p->children.size(); i++) {
101             if (p->children[i]) {
102                 printf("\n");
103                 for (int j = 0; j < level; j++)
104                     printf(" ");
105                 printf("\\");
106                 PrintHelper(p->children[i]);
107             }
108         }
109         level--;
110     }
111
112     int doTraversal(TNode *n, int labelHeight, int* maxHeight, std::vector<int> &
            substringStartIndex) {
113         if(n == nullptr) {
114             return 0;
115         }
116         int ret = -1;
117         if(n->ind < 0) {
118             for (auto it : n->children) {
```

```
119                 TNode *temp = it.second;
120                 if(temp != nullptr) {
121                     ret = doTraversal(temp, labelHeight + EdgeLength(temp), maxHeight,
                            substringStartIndex);
122                     if(n->ind == -1) {
123                         n->ind = ret;
124                     }
125                     else if((n->ind == -2 && ret == -3) || (n->ind == -3 && ret == -2)
                            || n->ind == -4) {
126                         n->ind = -4;
127                         if(*maxHeight < labelHeight) {
128                             *maxHeight = labelHeight;
129                             substringStartIndex.clear();
130                             substringStartIndex.push_back(*(n->end) - labelHeight + 1);
131                         }
132                         else if(*maxHeight == labelHeight && !substringStartIndex.empty()
                                && substringStartIndex.back() != *(n->end) - labelHeight +
                                1) {
133                             substringStartIndex.push_back(*(n->end) - labelHeight + 1);
134                         }
135                     }
136                 }
137             }
138         }
139         else if(n->ind > -1 && n->ind < size1)
140             return -2;
141         else if(n->ind >= size1)
142             return -3;
143         return n->ind;
144     }
145
146
147
148     TNode *root = new TNode(nullptr, -1, new int(-1));
149     TNode *internalNode = nullptr;
150
151     std::string text;
152
153     TNode *activeNode = nullptr;
154     int activeEdge = -1;
155     int activeLength = 0;
156     int remainSuff = 0;
157     int leafEnd = -1;
158     int size1;
159 public:
160     void Print() {
161         PrintHelper(root);
162     }
163     TSuffTree(std::string &str1, std::string &str2) {
```

```
164         size1 = str1.size();
165         text = str1 + '#' + str2 + "$";
166         BuildSuffTree();
167     };
168     void LongestCommonSubstring() {
169         int maxHeight = 0;
170         std::vector<int> substringStartIndex;
171         doTraversal(root, 0, &maxHeight, substringStartIndex);
172         std::cout << maxHeight << "\n";
173         std::vector<std::string> ans;
174         std::string temp;
175         for (int i = 0; i < substringStartIndex.size();++i) {
176             int k = 0;
177             temp.clear();
178             for (k = 0; k < maxHeight; k++) {
179                 temp += text[k + substringStartIndex[i]];
180             }
181             ans.push_back(temp);
182         }
183         std::stable_sort(ans.begin(), ans.end());
184         for(auto i:ans) {
185             std::cout << i << "\n";
186         }
187     }
188     void BuildSuffTree() {
189         activeNode = root;
190         for (int i = 0; i < text.length(); ++i) {
191             ExtendTree(i);
192         }
193     }
194     ~TSuffTree() {
195         DeleteHelper(root);
196     }
197 };
```

3 Консоль

```
kng@kng-Legion-Y540-15IRH:~/CLionProjects/DISKRAN5/cmake-build-debug ./DISKRAN5
xabay
xabcbay
3
bay
xab
```

4 Тест производительности

Для изучения производетельности, сравним время работы программы на тестах разных размеров со строками длиной $10^4, 10^5, 10^6$

```
1. $10^4$
/home/kng/CLionProjects/DISKRAN5/cmake-build-debug/DISKRAN5
251279ms
Process finished with exit code 0

2.$10^5$
2791727ms
Process finished with exit code 0

3.$10^6$
/home/kng/CLionProjects/DISKRAN5/cmake-build-debug/DISKRAN5
34305311ms
Process finished with exit code 0
```

Видно, что рост времени работы практически линейный.

5 Выводы

Во время выполнения работы, я познакомился с такой структурой данных, как суффиксное дерево. Его применение на практике выглядит весьма специфическим, в чем я убедился, посмотрев варианты лабораторных работ. Алгоритм Укконена оказался весьма сложным как для понимания, так и для построения. Сам поиск наибольшей общей подстроки в сравнении с ним оказался простой задачей.

Список литературы