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: А. Ю. Голов
Преподаватель: С. А. Михайлова
Группа: М8О-301Б-21

Дата:
Оценка:
Подпись:

Москва, 2023

Лабораторная работа №5

Формулировка задания: Найти длину наибольшей общей подстроки двух струк, используя суффиксное дерево, построенное алгоритмом Укконена.

Формат ввода: Две строки.

Формат вывода: 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

Решающее действие при решении поставленной задачи - определение точного метода использования суффиксного дерева при нахождении наибольшей общей подстроки. Известно, что одно из свойств суффиксного дерева - обеспечение доступа ко всем подстрокам строки за линейное время. Таким образом, получаем, что, если объединить две исходные строки, разделив их уникальным символом, в построенном для этой строки суффиксном дереве будет в относительно явном виде, за исключением абстракций, вводимых при использовании алгоритма Укконена, существовать наибольшая общая подстрока.

Основой программы-решения послужила парадигма объектно-ориентированного программирования. Были реализованы два класса - узел дерева и само дерево, связывающее узлы. Для соблюдения принципов инкапсуляции как атрибуты, так и методы, в том чиссле и конструкторы и деструкторы были описаны в качестве приватных методов, а для упрощения структуры классов в целях избежания сверхинжиниринга классов путём добавления модификаторов доступа и "геттеров"классы были сделаны "дружественными".

2 Исходный код

```
1
2  #include <bits/stdc++.h>
3
4  class TNode
5  {
6    friend class TSufTree;
7
8    public:
9
10   ~TNode() {
11     for (std::pair <char, TNode *> node : this->edges) {
12       delete node.second;
13     }
14   }
15
16   private:
17
18   TNode* link;
19
20   int start;
21   int idx = -1;
22
23   std::shared_ptr<int> end;
24   std::map <char, TNode* > edges;
25 };
26
27 class TSufTree
28 {
29   public:
30
31   TSufTree(std::string& text, int size)
32   {
33     data = text;
34     firstSize = size;
35
36     Init();
37     SetIndex(root, 0);
38   }
39
40   std::pair<int, std::vector<std::string>> FindLCP()
41   {
42     std::vector<int> startIdx;
43     Walk(root, 0, &curRes, startIdx);
44
45     std::vector<std::string> result;
46     std::string tmp;
```

```
47
48      for (size_t i = 0; i < startIdx.size();++i)
49      {
50        tmp.clear();
51
52        for (int k = 0; k < curRes; k++) {
53          tmp += data[k + startIdx[i]];
54        }
55
56        result.push_back(tmp);
57      }
58
59      std::sort(result.begin(), result.end());
60
61      return std::make_pair(this->curRes, result);
62    }
63
64    ~TSufTree() {
65      delete root;
66    }
67
68
69    protected:
70
71    std::string data;
72
73    TNode* root = NULL;
74    TNode* prevNode = NULL;
75    TNode* curNode = NULL;
76
77    int curEdge = -1;
78    int curLen = 0;
79    int curRes = 0;
80
81    int sufCnt = 0;
82
83    std::shared_ptr<int> leafEnd = std::make_shared<int>(-1);
84
85    int size = -1;
86    int firstSize = 0;
87
88    private:
89
90    TNode* AddNode(int start, std::shared_ptr<int> end)
91    {
92      TNode* node = new class TNode();
93
94      node->link = root;
95      node->start = start;
```

```
96      node->end = end;
97
98      return node;
99    }
100
101   void Init()
102   {
103     size = data.size();
104     root = AddNode(-1, std::make_shared<int>(-1));
105     curNode = root;
106
107     for (int i = 0; i < size; i++) {
108       Extend(i);
109     }
110   }
111
112   int GetEdgeLen(TNode* n)
113   {
114     if (n == root) {
115       return 0;
116     }
117
118     return *(n->end) - (n->start) + 1;
119   }
120
121
122   int WalkDown(TNode* currTNode)
123   {
124     if (curLen >= GetEdgeLen(currTNode))
125     {
126       curEdge += GetEdgeLen(currTNode);
127       curLen -= GetEdgeLen(currTNode);
128       curNode = currTNode;
129
130       return 1;
131     }
132
133     return 0;
134   }
135
136   void Extend(int pos)
137   {
138     ++*leafEnd;
139     sufCnt++;
140     prevNode = NULL;
141
142     while (sufCnt > 0)
143     {
144       if (curLen == 0) {
```

```
145         curEdge = pos;
146       }
147
148       if (!curNode->edges[data[curEdge]])
149       {
150         curNode->edges[data[curEdge]] = AddNode(pos, leafEnd);
151
152         if (prevNode != NULL)
153         {
154           prevNode->link = curNode;
155           prevNode = NULL;
156         }
157       }
158       else
159       {
160         TNode* next = curNode->edges[data[curEdge]];
161
162         if (WalkDown(next)) {
163           continue;
164         }
165
166         if (data[next->start + curLen] == data[pos])
167         {
168           if (prevNode != NULL && curNode != root)
169           {
170             prevNode->link = curNode;
171             prevNode = NULL;
172           }
173
174           curLen++;
175           break;
176         }
177
178         int splitEnd = next->start + curLen - 1;
179         TNode* split = AddNode(next->start, std::make_shared<int>(splitEnd));
180         curNode->edges[data[curEdge]] = split;
181
182         split->edges[data[pos]] = AddNode(pos, leafEnd);
183         next->start += curLen;
184         split->edges[data[next->start]] = next;
185
186         if (prevNode != NULL) {
187           prevNode->link = split;
188         }
189
190         prevNode = split;
191       }
192
193       sufCnt--;
```

```
194
195       if (curNode == root && curLen > 0)
196       {
197         curLen--;
198         curEdge = pos - sufCnt + 1;
199       }
200       else if (curNode != root) {
201         curNode = curNode->link;
202       }
203     }
204   }
205
206   void SetIndex(TNode* n, int curHeight)
207   {
208     if (n == NULL) {
209       return;
210     }
211
212     int leaf = 1;
213
214     for (auto child : n->edges)
215     {
216       leaf = 0;
217       SetIndex(child.second, curHeight + GetEdgeLen(child.second));
218     }
219
220     if (leaf == 1) {
221       n->idx = size - curHeight;
222     }
223   }
224
225   int Walk(TNode* node, int curHeight, int* maxHeight, std::vector<int>& startIdx)
226   {
227     if (node == NULL) {
228       return 0;
229     }
230
231     int ret = -1;
232
233     if (node->idx < 0)
234     {
235       for (auto child : node->edges)
236       {
237         ret = Walk(child.second, curHeight + GetEdgeLen(child.second), maxHeight,
              startIdx);
238
239         if (node->idx == -1) {
240           node->idx = ret;
241         }
```

```
242         else if (
243         (node->idx == -2 && ret == -3)
244         || (node->idx == -3 && ret == -2)
245         || node->idx == -4
246         || ret == -4
247         )
248         {
249           node->idx = -4;
250 
251           if (*maxHeight < curHeight)
252           {
253             *maxHeight = curHeight;
254 
255             startIdx.clear();
256             startIdx.push_back(*(node->end) - curHeight + 1);
257           }
258           else if (
259           *maxHeight == curHeight
260           && !startIdx.empty()
261           && startIdx.back() != *(node->end) - curHeight + 1
262           ) {
263             startIdx.push_back(*(node->end) - curHeight + 1);
264           }
265         }
266       }
267     }
268     else if (node->idx > -1 && node->idx < firstSize) {
269       return -2;
270     }
271     else if (node->idx >= firstSize) {
272       return -3;
273     }
274 
275     return node->idx;
276   }
277 };
278 
279 int main()
280 {
281   std::ios_base::sync_with_stdio(false);
282   std::cin.tie(NULL);
283   std::cout.tie(NULL);
284 
285   std::string first, second;
286   std::cin >> first >> second;
287 
288   first += "#";
289   second += "$";
290   std::string data = first + second;
```

```
291
292   TSufTree tree(data, first.size());
293   std::pair<int, std::vector<std::string>> result = tree.FindLCP();
294
295   std::cout << result.first << '\n';
296
297   for (std::string& elem : result.second) {
298     std::cout << elem << '\n';
299   }
300
301   return 0;
302 }
```

3 Тест производительности

В качестве тестирующего инструмента были выбраны гугл-тесты, интегрированные в программу при помощи CMake. Были применены методы «table-driven tests» и «fuzzy-test», ниже приведены табличные тестовые случаи.

Ввод	Вывод
abacaba bacabac	6 bacaba
abacaba b	1 b
abacaba cbc	1 b c
axerbxerx xerx	4 xerx
abacaba rerroro	0

4 Выводы

По итогам выполнения пятой лабораторной работы по курсу «Дискретный анализ», стало очевидно, что грамтоный подход к разработке алгоритма, не оглядывающийся на сложность реализации, приведёт к лучшему кнечному результату.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] *Сортировка подсчётом — Википедия.*
URL: `http://ru.wikipedia.org/wiki/Сортировка_подсчётом` (дата обращения: 16.12.2013).

[3] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008