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: М. В. Леухин
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-20

Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №7

Задача:

Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

1 Описание

Суффиксный массив представляет собой лексикографически отсортированный массив всех суффиксов строки. Он был предложен в качестве более экономной по памяти альтернативы суффиксному дереву. Однако, для построения суффиксного массива мы будем исползовать алгоритм, в основе которого лежит построение суффиксного дерева.

Суффиксное дерево представляет собой дерево, в котором содержатся все возможные суффиксы строки. Для его построения существует наивный алгоритм, имеющий сложность $O(n^3)$. Так как он крайне неэффективен, мы будем использовать алгоритм Укконена, который имеет сложность $O(n)$. В основе этого алгоритма лежит идея использования суффиксных ссылок, которые позволяют бещ лишних обходов переходить на уже добавленные вершины при вставке очередного суффикса.

```
1
2  #include <iostream>
3  #include <vector>
4  #include <map>
5  #include <string>
6  #include <algorithm>
7
8  using namespace std;
9
10 class TSufArr;
11
12 class TNode {
13 public:
14     map<char, TNode *> dest;
15     string::iterator begin, end;
16     TNode *sufLink;
17
18     TNode(string::iterator start, string::iterator end) : begin(start), end(end),
           sufLink(nullptr) {};
19
20     ~TNode() = default;
21 };
22
23 class TSufTree {
24 public:
25
26     explicit TSufTree(string s) : text(move(s)), root(new TNode(text.end(), text.end())
           ), isReady(0) {
27         activeEdge = text.begin();
28         activeLen = 0;
29
30         activeNode = root;
31         SLmarker = root;
```

```
32          root->sufLink = root;
33
34          for (string::iterator i = text.begin(); i != text.end(); ++i) {
35              TrExtend(i);
36          }
37      }
38
39      ~TSufTree() = default;
40
41      friend TSufArr;
42
43  private:
44      string text;
45      TNode *root;
46      int isReady;
47      TNode *SLmarker, *activeNode;
48      int activeLen;
49      string::iterator activeEdge;
50
51      int EdgeLen(TNode *node, string::iterator cur_pos) {
52          return static_cast<int>(min(node->end, cur_pos + 1) - node->begin);
53      }
54
55      void TreeDestroy(TNode *node) {
56          for (auto &it: node->dest) {
57              TreeDestroy(it.second);
58          }
59          delete node;
60      }
61
62      bool WalkDown(string::iterator cur_pos, TNode *node) {
63          if (activeLen >= EdgeLen(node, cur_pos)) {
64              activeEdge += EdgeLen(node, cur_pos);
65              activeLen -= EdgeLen(node, cur_pos);
66              activeNode = node;
67              return true;
68          }
69          return false;
70      }
71
72      void SufLinkActivate(TNode *node) {
73          if (SLmarker != root) {
74              SLmarker->sufLink = node;
75          }
76
77          SLmarker = node;
78      }
79
80      void TrExtend(string::iterator current) {
```

```
81          SLmarker = root;
82          ++isReady;
83
84          while (isReady) {
85              if (!activeLen) {
86                  activeEdge = current;
87              }
88              auto tree_it = activeNode->dest.find(*activeEdge);
89              TNode *next = (tree_it == activeNode->dest.end()) ? NULL : tree_it->second;
90              if (!next) {
91                  TNode *leaf = new TNode(current, text.end());
92                  activeNode->dest[*activeEdge] = leaf;
93                  SufLinkActivate(activeNode);
94              } else {
95                  if (WalkDown(current, next)) {
96                      continue;
97                  }
98
99                  if (*(next->begin + activeLen) == *current) {
100                     ++activeLen;
101                     SufLinkActivate(activeNode);
102                     break;
103                 }
104
105                 TNode *split = new TNode(next->begin, next->begin + activeLen);
106                 TNode *leaf = new TNode(current, text.end());
107                 activeNode->dest[*activeEdge] = split;
108
109                 split->dest[*current] = leaf;
110                 next->begin += activeLen;
111                 split->dest[*next->begin] = next;
112                 SufLinkActivate(split);
113             }
114             --isReady;
115             if (activeNode == root && activeLen) {
116                 --activeLen;
117                 activeEdge = current - isReady + 1;
118             } else {
119                 activeNode = (activeNode->sufLink) ? activeNode->sufLink : root;
120             }
121         }
122     }
123
124     void DFS(TNode *node, vector<int> &result, int depth) {
125         if (node->dest.empty()) {
126             result.push_back(static_cast<int &&>(text.size() - depth));//found
127             return;
128         }
129         for (auto &it: node->dest) {
```

```
130             int tmp = depth;
131             tmp += it.second->end - it.second->begin;
132             DFS(it.second, result, tmp);
133         }
134     }
135 };
136 
137 class TSufArr {
138 public:
139 
140     explicit TSufArr(TSufTree tree) : text(tree.text), array() {
141         tree.DFS(tree.root, array, 0);
142         tree.TreeDestroy(tree.root);
143     }
144 
145     vector<int> Find(string pattern) {
146         pair<vector<int>::iterator, vector<int>::iterator> bounds(array.begin(), array.
                end());
147         for (int i = 0; i < pattern.size() && bounds.first != bounds.second; ++i) {
148             bounds = equal_range(bounds.first, bounds.second, numeric_limits<int>::max
                    (),
149                                  [this, &pattern, &i](int l, int r) -> bool {
150                                      bool tmp;
151                                      (l == numeric_limits<int>::max()) ? tmp = (pattern[i]
                                           < text[i + r]) : tmp = (
152                                             text[i + l] < pattern[i]);
153                                      return tmp;
154                                  });
155         }
156 
157         vector<int> result(bounds.first, bounds.second);
158         sort(result.begin(), result.end());
159 
160         return result;
161     }
162 
163     ~TSufArr() = default;
164 
165 private:
166     string text;
167     vector<int> array;
168 };
169 
170 int main() {
171     string text, pattern;
172     cin >> text;
173 
174     TSufTree tree(text + "$");
175     TSufArr array(tree);
```

```
176
177     for (int pattern_num = 1; cin >> text; ++pattern_num) {
178         vector<int> result = array.Find(text);
179         if (!result.empty()) {
180             cout << pattern_num << ": ";
181             for (int i = 0; i < result.size(); ++i) {
182                 cout << result[i] + 1; // from 0
183                 if (i < result.size() - 1) cout << ", ";
184             }
185             cout << '\n';
186         }
187     }
188     return 0;
189 }
```

2 Консоль

```
C:/Users/leyhi/CLionProjects/contest/cmake-build-debug/lab5.exe
abcdabc
abcd
bcd
bc
>1: 1
>2: 2
>3: 2,6
```

3 Тест производительности

Для теста производительности сравним наивный алгоритм построения суффиксного дерева за $O(n^3)$ и алгоритм Укконена:

```
C:/Users/leyhi/CLionProjects/contest/cmake-build-debug/lab7.exe
text with 1000 symbols
Ukkonen: 39 mcs.
Naive: 88 mcs.
C:/Users/leyhi/CLionProjects/contest/cmake-build-debug/lab7.exe
text with 10000 symbols
Ukkonen: 534 mcs.
Naive: 2753 mcs.
C:/Users/leyhi/CLionProjects/contest/cmake-build-debug/lab7.exe
text with 100000 symbols
Ukkonen: 9629 mcs.
Naive: -mcs.
```

Как видно из тестов, навинвый алгоритм значительно уступает по производительности алгоритму Укконена, особенно с ростом длины входного текста. В частности, для входного текста длины 100000 наивный алгоритм требует неприлично много времени для построения суффиксного дерева.

4 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я познакомился с такими подходами поиска паттернов в тексте, как использование суффиксного дерева и суффиксного массива. Суффиксное дерево эффективно использовать в том случае, если текст не меняется, а на вход постоянно поступают новые паттерны для поиска. Однако оно крайне неэффективно по памяти, что привело к идее построения суффиксного массива, с помощью которого можно выполнять те же функции, однако используя $O(n)$ памяти (n — длина текста).

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] *Суффиксное дерево — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Суффиксное-дерево` (дата обращения: 09.12.2022).

[3] *Суффиксный массив — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Суффиксный-массив` (дата обращения: 09.12.2022).

[4] *Алгоритм Укконена — Викиконспекты.*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм-Укконена` (дата обращения: 09.12.2022).