Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: А. А. Каримов
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б-22

Дата:
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №4

Задача: Необходимо реализовать алгоритм Укконена, который строит суффиксное дерево за O(n*k), где n - длина строки, а k - размер алфавита.

Вариант алгоритма: `Найти самую длинную общую подстроку двух строк с использованием суфф. дерева.`

1 Описание

Наивное построение суффиксного дерева работает за время O(n^3), где n - длина строки. В наивной реализации добавляются для каждого префикса все его суффиксы. Алгоритм Укконена предлагает построение суффиксного дерева за линейной время, при условии ограниченного алфавита. Этот алгоритм предлагает не добавлять на каждой итерации в отдельности всем листьям букву на конец, а хранить пары чисел - левая и правая граница. На итерации для листьев происходит инкремент правой границы, так дописывания, которые раньше занимали линейное время, стали константой. Также предлагается ввести суффиксные ссылки для вершин xa и a (то есть суффиксная ссылка указывает на ребро, где окончивается такая же строка, но без первого символа).

Для нахождения наибольшей общей подстроки соединим строки через сентинел, и каждуый лист пометим в соответствии с его позицией вхождения либо как "длинный суффикс либо как "короткий". Затем также пометим остальные вершины, основываясь на листах. Те вершины, которые будут помечены как короткая и как длинная, будут общей подстрокой. Среди таких вершин пройдем от корня максимально глубоко. Найденная подстрока будет ответом к задаче. Сложность также будет линейной.

2 Исходный код

Здесь располагается реализация алгоритма Укконена.

```
#include <vector>
#include <iostream>
#include <string>
#include <map>
#include <algorithm>


namespace NSuffixTree {

const static std::size_t INF = 1e9;
const static std::size_t undefinedNode = std::size_t(-1);

const static int SHIFT = 4;

const static int SHORT_SUFFIX = 1;
const static int LONG_SUFFIX = 2;
const static int LONG_SHORT_SUFFIX = 3;

class SuffixTree {

private:


    struct SuffixNode {
        std::size_t left = 0;
        std::size_t length = 0;

        std::size_t suffixLink = 0;

        std::map<char, std::size_t> childs;

        SuffixNode(
                   std::size_t _l,
                   std::size_t _len
                ) : left(_l), length(_len) { }
    };


private:

    std::vector<SuffixNode> nodes;
    std::string text;
    std::size_t size = 0;

    std::size_t currentNode = 0;
    std::size_t reminder = 0;
```

```
47
48      std::size_t move(std::size_t node, char c) {
49          if(!nodes[node].childs.contains(c)) {
50              return undefinedNode;
51          }
52          return nodes[node].childs[c];
53      }
54
55      void move_node() {
56          while(1) {
57              char c = text[size - reminder];
58              std::size_t nextNode = move(currentNode, c);
59
60              if(nextNode == undefinedNode) return;
61
62              if(nodes[nextNode].length < reminder) {
63                  reminder -= nodes[nextNode].length;
64                  currentNode = nextNode;
65              } else return;
66          }
67      }
68
69      std::size_t create_suffix_node(std::size_t left = 0, std::size_t length = INF) {
70          nodes.push_back(SuffixNode(left, length));
71          return nodes.size() - 1;
72      }
73
74  public:
75
76      SuffixTree(const std::string& s) : text(s) {
77          size = 0;
78          create_suffix_node(0, 0);
79          for(char c : text) {
80              add_symbol(c);
81          }
82      }
83
84      void add_symbol(char symbol) {
85
86          ++reminder;
87          ++size;
88          std::size_t previousNode = 0;
89
90          while(reminder > 0) {
91
92              move_node();
93              char c = text[size - reminder];
94              std::size_t nextNode = move(currentNode, c);
95
```

```
96              if(nextNode == undefinedNode) {
97
98                  nodes[currentNode].childs[c] = create_suffix_node(size - reminder);
99                  nodes[previousNode].suffixLink = currentNode;
100                 previousNode = currentNode;
101
102             } else {
103
104                 char matchingSymbol = text[nodes[nextNode].left + reminder - 1];
105
106                 if(matchingSymbol != symbol) {
107                     std::size_t internalNode = create_suffix_node(nodes[nextNode].left,
                            reminder - 1);
108                     std::size_t leaf = create_suffix_node(size - 1, INF);
109
110                     nodes[nextNode].left += reminder - 1;
111                     nodes[nextNode].length -= reminder - 1;
112
113                     nodes[currentNode].childs[c] = internalNode;
114
115                     nodes[internalNode].childs[matchingSymbol] = nextNode;
116                     nodes[internalNode].childs[symbol] = leaf;
117
118                     nodes[previousNode].suffixLink = internalNode;
119                     previousNode = internalNode;
120                 } else {
121                     nodes[previousNode].suffixLink = currentNode;
122                     return;
123                 }
124
125             }
126             if(currentNode == 0) {
127                 --reminder;
128             } else {
129                 currentNode = nodes[currentNode].suffixLink;
130             }
131         }
132     }
133
134
135     std::vector<int> suffixTypes;
136     std::vector<std::pair<std::size_t, std::size_t>> entries;
137
138     void get_longest_common_substring(std::size_t firstLength) {
139         suffixTypes.resize(nodes.size(), 0);
140         get_suffix_types(0, 0, firstLength);
141
142         find_common_substrings(0, 0);
143         if(entries.size() == 0) {
```

```
144             std::cout << "0\n";
145             return;
146         }
147 
148         std::size_t maxLen = (*std::max_element(entries.begin(), entries.end(),
149             [](const std::pair<std::size_t, std::size_t>& fst, const std::pair<std::
                    size_t, std::size_t>& scd)
150                 { return fst.second < scd.second; } )).second;
151 
152         std::cout << maxLen << '\n';
153         for(const std::pair<std::size_t, std::size_t>& entry : entries) {
154             if(entry.second == maxLen) {
155                 for(std::size_t i{0}; i < entry.second; ++i)
156                     std::cout << text[i + entry.first];
157                 std::cout << '\n';
158             }
159         }
160 
161     }
162 
163     void find_common_substrings(std::size_t cur, std::size_t localLength) {
164         bool hasExpands = false;
165         for(std::pair<char, std::size_t> next : nodes[cur].childs) {
166             if(suffixTypes[next.second] == LONG_SHORT_SUFFIX) {
167                 hasExpands = true;
168                 find_common_substrings(next.second, localLength + nodes[cur].length);
169             }
170         }
171 
172         if(!hasExpands) {
173             entries.emplace_back(std::make_pair(nodes[cur].left - localLength,
                        localLength + nodes[cur].length));
174         }
175     }
176 
177     int get_suffix_types(std::size_t cur, std::size_t localLength, std::size_t
            firstLength) {
178 
179         for(std::pair<char, std::size_t> next : nodes[cur].childs) {
180             suffixTypes[cur] |= get_suffix_types(next.second, localLength + nodes[cur].
                    length, firstLength);
181         }
182 
183         if(nodes[cur].childs.empty()) { // Leaf
184             if(nodes[cur].left - localLength < firstLength) {
185                 suffixTypes[cur] = LONG_SUFFIX;
186             } else {
187                 suffixTypes[cur] = SHORT_SUFFIX;
188             }
```

```
189         }
190 
191         return suffixTypes[cur];
192     }
193 };
194 
195 }
196 
197 
198 
199 
200 int main() {
201     std::ios::sync_with_stdio(false);
202     std::cin.tie(0);
203 
204     std::string s1, s2;
205     std::cin >> s1 >> s2;
206     std::string s = s1 + '!' + s2 + '$';
207     NSuffixTree::SuffixTree tr(s);
208     tr.get_longest_common_substring(s1.size());
209 }
```

3 Консоль

```
karseny99@karseny99:/mnt/study/DA/lab4/src$ cat 2.in
xabay
xabcbay
karseny99@karseny99:/mnt/study/DA/lab4/src$ ./lab5 <2.in
3
bay
xab
```

4 Тест производительности

Производительность оценивается так: на один и тех же тестовых данных запускается наивный алгоритм и алгоритм, использующий суффиксное дерево. Тесты на 10^2, 10^3 и 10^4 слов в тексте.

```
karseny99@karseny99:/mnt/study/DA/lab5$ ./benchmark <tests/test_3.i
Suffix tree: 188us
Naive: 356360us
karseny99@karseny99:/mnt/study/DA/lab5$ ./benchmark <tests/test_2.i
Suffix tree: 37us
Maive: 975us
karseny99@karseny99:/mnt/study/DA/lab5$ ./benchmark <tests/test_4.i
Suffix tree: 2096us
Naive: 47633200us
```

5 Выводы

В ходе выполнения лабораторной работы я смог реализовать алгоритм Укконена построения суффиксного дерева по строке за линейное время. Существуют и другие алгоритмы построения, в том числе имеющие линейную сложность, не зависящую от размера алфавита. Тем не менее, алгоритм Укконена относительно других алгоритмов считается несложным в реализации и понимании.

Список литературы

[1] Гасфилд Дэн. *Строки, деревья и последовательности в алгоритмах.* — 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] *Алгоритм Бойера-Мура*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бойера-Мура`