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. О. Знай
Преподаватель: С. А. Михайлова
Группа: М8О-301Б-21

Дата: 25.09.2023
Оценка:
Подпись:

Москва, 2023

Лабораторная работа №5

Задача: Необходимо реализовать поиск паттерна в тексте с использованием суффиксного дерева.

Вариант алгоритма: Найти образец в тексте используя статистику совпадений.

1 Описание

Суффиксное дерево — бор, содержащий все суффиксы некоторой строки (и только их). Позволяет выяснять, входит ли строка w в исходную строку t, за время O(|w|), где |w| — длина строки w.

2 Описание программы

Построение суффиксного дерева будем осуществлять с помощью алгоритма Укконена за O(n). Мы должны добавить все суффиксы каждого префикса строки, но делать это используя несколько эвристик: 1) Мы должны использовать линейное количество памяти, поэтому на ребрах хранятся не символы строки, а 2 числа - позиция самого левого и самого правого ее символов в тексте. 2) Автоматическое продление всех листов за O(1) благодаря увеличения переменной текущего конца на единицу. 3) Использование суффиксных ссылок: если есть суффикс xa, то есть и суффикс a, который мы можем пропустить перейдя по суффиксной ссылки между двумя вершинами 4) Переход по счетчику внутри ребер

В данном варианте для поиска строки в тексте используется статистика совпадений. Ее смысл заключается в следующем: создается массив ms размером с текст, где в i ячейке хранится число k, которое означает, что начиная с i позиции в тексте совпадут k символов какой-то подстроки образца. Для заполнения такого массива необходимо: 1) построить суффиксное дерево не для текста, а для паттерна 2) считывать символы текста до тех пор пока есть совпадения в дереве и увеличиваем k, если произошла неудача, то записываем число в массив, откатываемся до ближайшей внутренней вершины, переходим по суффиксной ссылке и накатываем k-1 символов для текущего символа в тексте и продолжаем искать совпадения 3) пройтись по массиву и вывести все индексы, в которых лежат k, равные длине паттерна

Сложность алгоритма поиска статистикой совпадений - O(m), где m - длина текста. [1]

Код

```
#include <string>
#include <iostream>
#include <vector>
#include <stack>
#include <memory>

struct SuffixTree {
    struct Edge {
        int left;
        std::shared_ptr<int> right;
        int nodeTarget;

        Edge(int left, std::shared_ptr<int> right, int node):
            left(left),
            right(right),
            nodeTarget(node) {}
    };

    std::vector< std::vector<Edge> > Tree;
    std::string Text;
    std::vector<int> VSuffix;
    std::vector<int> LenToNode;

    int FindEdge(int node, char ch) {
        for (int i = 0; i < Tree[node].size(); ++i) {
            if (Text[Tree[node][i].left] == ch) {
                return i;
            }
        }
        return -1;
    };

    SuffixTree(const std::string & str): Text(str),
        Tree(str.size() * 2),
        VSuffix(str.size() * 2),
        LenToNode(str.size() * 2) {
            int left = 0;
            std::shared_ptr<int> end(new int);
            *end = 0;

            bool newNode = false;
            int newNodeNumber = 0;

            int eqFromRoot = 0;
            int eqCur = 0;

            int curNode = 0;
            int curEdge = -1;
```

```
49
50              for (int i = 0; i < str.size(); ++i) {
51                  ++*end;
52
53                  int curNewNode = -1;
54                  int lastNewNode = 0;
55
56                  while (*end > left) {
57                      if (curEdge == -1) {
58                          int nextEdge = FindEdge(curNode, Text[left + eqFromRoot + eqCur])
                                ;
59
60                          if (nextEdge == -1) {
61                              newNode = true;
62                          }
63                          else {
64                              eqCur = 1;
65                          }
66
67                          curEdge = nextEdge;
68                      }
69                      else if (
70                              Text[Tree[curNode][curEdge].left + eqCur] ==
71                              Text[left + eqFromRoot + eqCur ]
72                      ) {
73                              ++eqCur;
74                      }
75                      else {
76                          newNode = true;
77                      }
78
79                      if (eqCur > 0 and Tree[curNode][curEdge].left + eqCur == *Tree[
                            curNode][curEdge].right) {
80                          curNode = Tree[curNode][curEdge].nodeTarget;
81                          curEdge = -1;
82
83                          eqFromRoot += eqCur;
84                          eqCur = 0;
85                      }
86                      if (newNode) {
87                          ++newNodeNumber;
88
89                          if (eqCur == 0 and curEdge == -1) {
90                              Tree[curNode].push_back(Edge(left + eqFromRoot, end,
                                    newNodeNumber));
91                          }
92                          else {
93                              curNewNode = newNodeNumber;
94                              Edge tempCurEdge = Tree[curNode][curEdge];
```

```
95
96                          std::shared_ptr<int> newRight(new int);
97                          *newRight = Tree[curNode][curEdge].left + eqCur;
98                          Tree[curNode][curEdge].right = newRight;
99
100                         Tree[curNode][curEdge].nodeTarget = newNodeNumber;
101                         tempCurEdge.left = *newRight;
102                         Tree[newNodeNumber].push_back(tempCurEdge);
103
104                         ++newNodeNumber;
105                         Tree[newNodeNumber - 1].push_back(Edge(left + eqCur +
                                eqFromRoot, end, newNodeNumber));
106
107                         LenToNode[newNodeNumber - 1] = LenToNode[curNode] + *Tree[
                                curNode][curEdge].right - Tree[curNode][curEdge].left;
108
109                         if (lastNewNode > 0) {
110                             VSuffix[lastNewNode] = curNewNode;
111                         }
112                         lastNewNode = curNewNode;
113                     }
114
115                     int nextNode = VSuffix[curNode];
116                     int nextEdge = -1;
117                     int nextEqFromRoot = LenToNode[nextNode];
118                     int nextEqCur = eqCur + eqFromRoot - nextEqFromRoot - 1;
119
120                     while (nextEqCur > 0) {
121                         nextEdge = FindEdge(nextNode, Text[left + nextEqFromRoot +
                                1]);
122
123                         int r = *Tree[nextNode][nextEdge].right;
124                         int l = Tree[nextNode][nextEdge].left;
125
126                         if (r - l <= nextEqCur) {
127                             nextEqFromRoot += r - l;
128                             nextEqCur -= r + l;
129                             nextNode = Tree[nextNode][nextEdge].nodeTarget;
130                             nextEdge = -1;
131                         }
132                         else {
133                             break;
134                         }
135                     }
136
137                     if (nextEdge != -1) {
138                         eqCur = nextEqCur;
139                     }
140                     else {
```

```
                        eqCur = 0;
                    }

                    eqFromRoot = nextEqFromRoot;
                    curNode = nextNode;
                    curEdge = nextEdge;
                    left++;
                    newNode = false;
                }

            }
        }
    }

    std::vector<size_t> MatchStatistics(const std::string &text) {
        size_t len = text.size();
        std::vector<size_t> ms(len);

        int curNode = 0;
        int curEdge = FindEdge(curNode, text[0]);
        int eqFromRoot = 0;
        int eqCur = 0;

        std::stack<int> path;
        path.push(0);

        for (int i = 0; i < len; i++) {
            int nextNode = curNode;

            while(VSuffix[nextNode] == 0 and !path.empty()) {
                nextNode = path.top();
                path.pop();
            }
            curNode = VSuffix[nextNode];

            path.push(curNode);
            eqCur = std::max(0, eqFromRoot + eqCur - 1 - LenToNode[curNode]);
            eqFromRoot = LenToNode[curNode];
            curEdge = -1;

            while (eqCur >= 0) {
                int nextEdge = FindEdge(curNode, text[i + eqFromRoot]);

                if(nextEdge != -1) {
                    int edgeLen = *Tree[curNode][nextEdge].right - Tree[curNode][
                        nextEdge].left;

                    if (eqCur >= edgeLen) {
                        eqCur -= edgeLen;
```

```
189                         eqFromRoot += edgeLen;
190                         curNode = Tree[curNode][nextEdge].nodeTarget;
191                         nextEdge = -1;
192                     }
193                     else{
194                         curEdge = nextEdge;
195                         break;;
196                     }
197                 }
198                 else break;
199             }
200             while (
201                 curEdge != -1 and i + eqFromRoot + eqCur < len and text[i + eqCur +
                        eqFromRoot]
202                 ==
203                 Text[Tree[curNode][curEdge].left + eqCur]
204              ) {
205                 ++eqCur;
206                 if (Tree[curNode][curEdge].left + eqCur == *Tree[curNode][curEdge].right
                        ) {
207                     int nextNode = Tree[curNode][curEdge].nodeTarget;
208                     int nextEdge = FindEdge(nextNode, text[i + eqFromRoot + eqCur]);
209                     int edgeLen = eqCur;
210                     eqFromRoot += edgeLen;
211                     eqCur = 0;
212                     curNode = nextNode;
213                     path.push(curNode);
214                     curEdge = nextEdge;
215                 }
216             }
217             ms[i] = eqFromRoot + eqCur;
218         }
219         return ms;
220     }
221 };
222
223 int main() {
224     std::ios_base::sync_with_stdio(false);
225     std::cin.tie(nullptr);
226     std::cout.tie(nullptr);
227
228     std::string pattern;
229     std::string text;
230
231     std::cin >> pattern;
232     pattern += "$";
233     SuffixTree tree(pattern);
234
235     std::cin >> text;
```

```
236     std::vector<size_t> ms(text.size());
237     ms = tree.MatchStatistics(text);
238 
239     for (int i = 0; i < ms.size(); ++i) {
240         if (ms[i] == pattern.size() - 1) {
241            std::cout << i + 1 << "\n";
242         }
243     }
244 }
```

3 Консоль

```
artemznaj@MacBook-Air-Artem lab5 % ./a.out
ba
baobabbaobaoba
1
4
7
10
13
artemznaj@MacBook-Air-Artem lab5 % ./a.out
aba
qababaz
2
4
artemznaj@MacBook-Air-Artem lab5 % ./a.out
bao
bacbaobabbaobabbac
4
10
artemznaj@MacBook-Air-Artem lab5 % ./a.out
baobab
baobabaobab
1
6
```

4 Тест производительности

Тест производиительности представляет собой сравнение реализованного мной алгоритма статистики совпадений и STL find. Сравнение производится путем запуска поиска 400000 паттернов в тексте. Результат работы:

```
artemznaj@MacBook-Air-Artem lab5 % ./a.out <test.txt
ms: 86481861 ms
STL find: 202546125 ms
```

5 Выводы

Выполнив пятую лабораторную работу, мною была изучена такая структура данных, как суффиксное дерево. Научился строить его за O(n), используя алгоритм Укконена, и искать строку в тексте (за O(m), где m - длина текста) с использованием приложения к суффиксным деревьям - статистику совпадений.

Несмотря на очевидный плюс в виде сложности, у алгортима Укконена есть и минусы, например, большие затраты по памяти относительно размера текста, также для простых задач поиска подстроки в строке лучше использовать другие алгоритмы.

Список литературы

[1] String Algorithms and Algrorithms in Computational Biology - Gusfield