Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: Р. С. Лисин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-20
Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск большого количества образцов при помощи алгоритма Ахо-Корасик.

Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые).

Формат входных данных: Искомый образец задаётся на первой строке входного файла. В случае, если в задании требуется найти несколько образцов, они задаются по одному на строку вплоть до пустой строки. Затем следует текст, состоящий из слов или чисел, в котором нужно найти заданные образцы. Никаких ограничений на длину строк, равно как и на количество слов или чисел в них, не накладывается.

Формат результата: В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку. Для заданий, в которых требуется найти только один образец, следует вывести два числа через запятую: номер строки и номер слова в строке, с которого начинается найденный образец. В заданиях с большим количеством образцов, на каждое вхождение нужно вывести три числа через запятую: номер строки; номер слова в строке, с которого начинается найденный образец; порядковый номер образца. Нумерация начинается с единицы. Номер строки в тексте должен отсчитываться от его реального начала (то есть, без учёта строк, занятых образцами).Порядок следования вхождений образцов несущественен.

1 Описание

Как сказано в [1]: «Алгоритм Ахо-Корасик - алгоритм поиска подстроки, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, реализует поиск множества подстрок из словаря в данной строке. Широко применяется в системном программном обеспечении, например, используется в утилите поиска grep.».

Сам алгоритм заключается в следующем. Сначала из паттернов строим бор (префиксное дерево). Бор - структура данных, которая является деревом и содержит корень, равный пустой строке, ребра, по которым можно перейти к следующей вершине по символу, и другие вершины, которые по сути являются префиксами паттернов.

Далее в боре для каждого узла нужно расставить суффиксные ссылки, то есть ссылки на узлы дерева, в которых лежат самые длинные суффиксы строк исходных узлов. Для корня и вершин первого уровня суффиксной ссылкой будет ссылка на корень. Для любой другой вершины нужно перейти по суффиксной ссылке её родителя в узел n. Если из узла n можно перейти в другой узел x по последнему символу строки исходной вершины, тогда ставим суффиксную ссылку на узел x. Если нельзя перейти, тогда переходим по суффиксной ссылке узла n, пока не упрёмся в корень и делаем то же самое. Если дошли до корня, тогда ставим суффиксную ссылку на корень.

Наконец бор готов для поиска подстроки в строке. Начинаем в корне. Считав очередной символ, пытаемся из текущего узла перейти по этому символу к другому узлу бора. Если это возможно - переходим. Если из какого-то узла нельзя никуда перейти, тогда он является листом бора. Это значит, что паттерн в этом узле входит в текст. Если узел не является листом и нет перехода по нужному символу, тогда переходим по суффиксной ссылке текущего узла и ищем переход по нужному символу. Если перехода нет, тогда снова переходим по суффиксной ссылке, пока не дойдём до корня (это значит, что паттерна нет в тексте) или не найдём нужный переход.

Общая сложность алгоритма будет складываться из сложностей построения бора из строк суммарной длины k в алфавите размером n ($O(n * k)$) и поиска по нему ($O(h + m)$), где h - длина текста, m - общая длина всех совпадений.

2 Исходный код

Каждый узел бора $TBohr$ будет представлять собой структуру $TBohrItem$, в которой поле $path$ нужно для перехода по символу в другой узел дерева, $sufflink$ является суффиксной ссылкой, вектор $success$ нужен для того, чтобы запомнить какой именно это паттерн, так как их может быть больше одного по заданию. Все методы класса $TBohr$ реализованы в соответсвии с описанием.

```
#include <map>
#include <sstream>
#include <vector>
#include <iostream>
#include <string>
#include <queue>

using namespace std;

typedef struct TBohrItem {
  map<string, TBohrItem*> path;
  TBohrItem* sufflink = nullptr;
  vector<int> success; // pattern id
} TBohrItem;


class TBohr {
  public:
  TBohr();
  TBohrItem* Next(TBohrItem* item, string letter); // find next item by letter
  TBohrItem* Move(TBohrItem* item, string letter); // move by means of letter or item's sufflink
  void Push(const vector<string>&);
  TBohrItem* FindSufflink(TBohrItem* child, TBohrItem* parent, const string letter);
  void Search(vector<string>&, vector<pair<int, int>>&);
  void Linkate(); // make suffix links
  vector<int> pieceIndex; // how many patterns in each line
  private:
  TBohrItem* root;
  int patternSize;
  int pieces;
};


TBohr::TBohr() :
patternSize(0), pieces(0) {
  root = new TBohrItem;
  root->sufflink = root;
}


```

```
41 TBohrItem* TBohr::Move(TBohrItem* item, string letter) {
42
43   try {
44     item = item->path.at(letter);
45   }
46   catch (out_of_range&) {
47
48     if (item == root) {
49       item = root;
50     }
51     else {
52       item = Move(item->sufflink, letter);
53     }
54   }
55   return item;
56 }
57
58
59 TBohrItem* TBohr::Next(TBohrItem* item, string letter) {
60
61   if (!item) {
62     return nullptr;
63   }
64   try {
65     item = item->path.at(letter);
66   }
67   catch (out_of_range&) {
68     item = nullptr;
69   }
70   return item;
71 }
72
73
74 void TBohr::Linkate() {
75
76   TBohrItem* node, * child;
77   queue<TBohrItem*> queue;
78   queue.push(root);
79   while (!queue.empty()) {
80     node = queue.front();
81     queue.pop();
82     map<string, TBohrItem*>::iterator childPairIt;
83     for (childPairIt = node->path.begin(); childPairIt != node->path.end(); ++
         childPairIt) {
84       child = childPairIt->second;
85       queue.push(child);
86       child->sufflink = FindSufflink(child, node, childPairIt->first);
87
88
```

```
      child->success.insert(child->success.end(),
      child->sufflink->success.begin(),
      child->sufflink->success.end());


      child->success.shrink_to_fit();
    }
  }
}


TBohrItem* TBohr::FindSufflink(TBohrItem* child, TBohrItem* parent,
const string letter) {
  TBohrItem* linkup = parent->sufflink, * check;
  while (true) {
    check = Next(linkup, letter);
    if (check) {
      return (check != child) ? check : root;
    }
    if (linkup == root) {
      return root;
    }
    linkup = linkup->sufflink;

  }
}


void TBohr::Push(const vector<string>& str) {
  TBohrItem* bohr = root, * next;
  for (size_t i = 0; i < str.size(); ++i) {
    next = Next(bohr, str[i]);
    if (!next) {
      next = new TBohrItem;
      next->sufflink = root;
      bohr->path.insert(pair<string, TBohrItem*>(str[i], next));
    }
    bohr = next;
  }
  bohr->success.push_back(pieces);
  pieces++;
}


void TBohr::Search(vector<string>& text, vector<pair<int, int>>& grid) {
  Linkate();
  int m = text.size();
  TBohrItem* node = root;
  int occurrence;
```

```
138   int c;
139   for (c = 0; c < m; ++c) {
140     for (size_t i = 0; i < node->success.size(); ++i) {
141       occurrence = c - pieceIndex[node->success[i]];
142       cout << grid[occurrence].first << ", " << grid[occurrence].second << ", " << node
          ->success[i] + 1 << '\n';
143     }
144
145     node = Move(node, text[c]);
146
147   }
148
149   for (size_t i = 0; i < node->success.size(); ++i) {
150     occurrence = c - pieceIndex[node->success[i]];
151     cout << grid[occurrence].first << ", " << grid[occurrence].second << ", " << node->
          success[i] + 1 << '\n';
152   }
153 }
154
155
156 int main() {
157   vector<pair<int, int>> grid;
158   vector<string> input;
159   vector<string> patterns;
160   string line, word;
161   string pattern;
162   TBohr b;
163   int index = 0;
164   while (getline(cin, pattern) && !pattern.empty()) {
165     stringstream lineStream(pattern);
166     while (lineStream >> word) {
167       for (unsigned int i = 0; i < word.length(); ++i) {
168         word[i] = tolower(word[i]);
169       }
170
171       patterns.push_back(word);
172       index++;
173     }
174
175     b.Push(patterns);
176     b.pieceIndex.push_back(index);
177
178     index = 0;
179     pattern.clear();
180     patterns.clear();
181
182   }
183
184   int lineIndex = 1, wordIndex = 1;
```

```
185
186   while (getline(cin, line)) {
187     stringstream lineStream(line);
188     while (lineStream >> word) {
189       for (unsigned int i = 0; i < word.length(); ++i) {
190         word[i] = tolower(word[i]);
191       }
192       input.push_back(word);
193       grid.push_back(make_pair(lineIndex, wordIndex));
194       wordIndex++;
195     }
196
197     wordIndex = 1;
198     lineIndex++;
199
200   }
201   input.shrink_to_fit();
202   grid.shrink_to_fit();
203   b.Search(input, grid);
204   return 0;
205 }
```

3 Консоль

```
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ g++ -Wall main.cpp
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ cat test1
cat dog cat dog
CAT dog CaT
Dog doG dog dOg

Cat doG cat dog  cat dog cat Parrot
doG dog DOG DOG   dog
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ ./a.out <test1
1,1,2
1,1,1
1,3,2
1,3,1
1,5,2
2,1,3
2,2,3
```

4 Тест производительности

Я решил сравнить свою реализацию алгоритма Ахо-Корасик с наивным алгоритмом поиска. Тестирование происходило на тестах с миллионом слов. В первом тесте образец часто встречался в тексте, во втором - не встречался вообще.

```
// Test 1
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ ./a.out <test.txt >result.txt
Time: 0.164362 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ ./naive <test.txt >result1.txt
Time: 0.397422 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ diff result.txt result1.txt

// Test 2
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ ./a.out <test.txt >result.txt
Time: 0.0644277 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ ./naive <test.txt >result1.txt
Time: 0.148165 seconds
roma@DESKTOP-JD58QU2:~/Diskran/lab4$ diff result.txt result1.txt
```

По результатам тестирования видно, что алгоритм Ахо-Корасик значительно выиграл по времени у наивного алгоритма благодаря использованию префиксного дерева и суффиксных ссылок.

5 Выводы

Выполнив четвёртую лабораторную работу по курсу «Дискретный анализ», я научился реализовывать алгоритм поиска подстроки в строке Ахо-Корасик. Данный алгоритм обычно сравнивают с суффиксными деревьями. У них одинаковая ассимптотика, но разное потребление памяти, так как в алгоритме с суффиксными деревьями мы кладём в бор текст, а не паттерны как в Ахо-Корасик. Следовательно, если текст слишком большой, то целесообразнее использовать алгоритм Ахо-Корасик, а если много паттернов и надо давать быстрый ответ, тогда лучше использовать алгоритм с суффиксными деревьями.

Список литературы

[1] *Алгоритм Ахо-Корасик - Википедия.*
URL: `https://ru.wikipedia.org/wiki/Алгоритм_Ахо_-_Корасик` (дата обращения: 25.05.2022).