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: А. В. Семин
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2022

Лабораторная работа №4

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

Вариант алгоритма: Поиск одного образца при помощи алгоритма Апостолико-Джанкарло.

Вариант алфавита: Числа в диапазоне от 0 до $2^{32} - 1$.

Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Формат ввода
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 вывода
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

Алгоритм Апостолико-Джанкарло это модификация алгоритма поиска подстроки в строке Бойера–Мура, основная задача которого - поиск вхождений паттерна в текст. Как и в случае других поисков вхождений на основе сравнения, это выполняется путем сдвига паттерна на определенное количество позиций и проверка совпадений в новой позиции. А затем в случае несовпадения, снова происходит сдвиг, и алгоритм повторяется. Применение правил сдвига Бойера-Мура часто приводит к тому, что большие фрагменты текста полностью пропускаются, что позволяет избежать лишних сравнений.

Что касается Апостолико–Джанкарло, по функциональности он эквивалентен алгоритму Бойера–Мура. Особенность Апостолико–Джанкарло состоит в том, чтобы ускорить операцию проверки соответствия для любого индекса. В Апостолико–Джанкарло имеется массив, по размеру совпадающий с текстом, в котором в позиции i хранится максимальный размер суффикса паттерна, который совпадает с текстом, если приложить конец паттерна в позицию текста i. Данный способ хранения уже совпавшего суффикса паттерна с символами текста в некоторой позиции позволяет пропускать сравнения букв, которые уже совпадали, совершая сдвиг на число, находящееся в данном массиве.

Рассмотрим также принцип работы алгоритма Бойера-Мура, так как именно он является основой для алгоритма Апостолико-Джанкарло. Алгоритм Боуйера-Мура основан на трёх идеях: (описание взято из интернет-ресура [2])

1. **Сканирование слева направо, сравнение справа налево**
Совмещается начало текста (строки) и шаблона, проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и выполняется поиск следующего вхождения подстроки.

Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на несколько символов вправо, и проверка снова начинается с последнего символа.

Эти «несколько», вычисляются по двум эвристикам.

2. **Эвристика стоп-символа**
(Замечание: эвристика стоп-символа присутствует в большинстве описаний алгорит-

ма Бойера — Мура, включая оригинальную статью Бойера и Мура, но не является необходимой для достижения оценки $O(n+m)O(n+m);,.)$

3. **Эвристика совпавшего суффикса**
Неформально, если при чтении шаблона справа налево совпал суффикс S, а символ b, стоящий перед S в шаблоне (то есть шаблон имеет вид PbS), не совпал, то эвристика совпавшего суффикса сдвигает шаблон на наименьшее число позиций вправо так, чтобы строка S совпала с шаблоном, а символ, предшествующий в шаблоне данному совпадению S, отличался бы от b (если такой символ вообще есть). Формально, для данного шаблона s[0..m-1] считается целочисленный массив suffshift[0..m], в котором suffshift[i] равно минимальному числу j>0, такому что s[i-j] != s[i-1] (если i>0 и i-j >= 0) и s[i-j+k] = s[i-1+k] для любого k>0, для которого выполняется 0 <= i-j+k<m и 0 <= i-1+k<m (для пояснения смотрите примеры ниже). Затем, если при чтении шаблона s справа налево совпало k-1 символов s[m-1],s[m-2], ... ,s[m-k+1], а символ s[m-k] не совпал, то шаблон сдвигается на suffshift[m-k] символов вправо.

2 Исходный код

Основная логика работы прогарммы состоит в следующем: двигаясь по входному тексту слева направо(начиная с символа, по индексу равному размеру входного паттерна), идет его сравнение с паттерном справаа налево.

На каждой итерации для позиции текста i и позиции паттерна j происходит сравнение значений массива M по индексу i и массива N-функции по индексу j. В зависимости от результатов сравнений (M[i] $>$ N[j], аналогично $<$, $==$ и также случай, когда M[i] неопределена или M[i] $==$ N[j] $==$ 0) выполняются определенные операции и сдвиги.

Если паттерн пройден полностью, значит зафиксировано вхождение подстроки. И программа печатает номер строки и номер индекса в этой строке, где было зафиксировано вхождение.

Листинг программы:

```
#include <iostream>
#include <vector>
#include <cstdint>
#include <string>
#include <algorithm>
#include <unordered_map>

const int UNDEFINED = -1;

void ParseStrToVec(const std::string& line, std::vector<int>& vec) {
  int tmp = 0;
  bool isSpaces = true;
  for (char c : line) {
    if ('0' <= c && c <= '9') {
      tmp = tmp * 10 + c - '0';
      isSpaces = false;
    }
    else {
      if (!isSpaces) {
        vec.push_back(tmp);
        tmp = 0;
      }
      isSpaces = true;
    }
  }
  if (!isSpaces) {
    vec.push_back(tmp);
```

```
  }
}

std::unordered_map<int, std::vector<int>> PFunction(const std::vector<int> pattern) {
    std::unordered_map<int, std::vector<int>> p_func;
// std::vector<std::vector<int>> p_func(10, std::vector<int>());
    int n = pattern.size();
    for (int i = 0; i < n; i++) {
        p_func[pattern[i]].push_back(i);
    }
    return p_func;
}

int UseRuleBadLetter(std::unordered_map<int, std::vector<int>>& p_func, int letter,
    int ind_patt, int patt_size) { // p-,   __,   ,
    auto it = p_func.find(letter);
    if (it == p_func.end()) {
        return ind_patt;
    }

    const std::vector<int> ind = it->second;
    auto it_bound = std::lower_bound(ind.begin(), ind.end(), ind_patt);
    return ind_patt - *(--it_bound);
}


std::vector<int> ZFunction(const std::vector<int>& pattern) {// Z-
  int n = pattern.size();
  std::vector<int> z_func(n, 0);
  int l = 0;
  int r = 0;
  for (int i = 1; i < n; ++i) {
    if (i <= r) {
      z_func[i] = std::min(r - i + 1, z_func[i - l]);
    }
    while (i + z_func[i] < n && pattern[z_func[i]] == pattern[i + z_func[i]]) {
      ++z_func[i];
    }
    if (i + z_func[i] - 1 > r) {
      l = i;
      r = i + z_func[i] - 1;
    }
  }
  return z_func;
}

std::vector<int> NFunction(std::vector<int> pattern) {// N-
    std::reverse(pattern.begin(), pattern.end());
    int n = pattern.size();
```

```
76      std::vector<int> n_func(n);
77      std::vector<int> z_func = ZFunction(std::move(pattern));
78      for (int i = 1; i < n; i++) {
79          if (z_func[i] != 0) {
80              n_func[n - i-1] = z_func[i];
81          }
82      }
83      return n_func;
84  }
85
86  std::vector<int> LFunction(const std::vector<int>& pattern,std::vector<int>& n_func,
        int& gp_size) {// L-
87      gp_size = 0;
88      int n = pattern.size();
89      n_func = NFunction(std::move(pattern));
90      std::vector<int> l_func(n, UNDEFINED);
91      for (int i = 0; i < n; i++) {
92          int j = n - n_func[i];
93          if (j != n) {
94              l_func[j] = i;
95              if (i == n - j -1) {
96                  gp_size = i+1;
97              }
98          }
99      }
100     return l_func;
101 }
102
103 int UseRuleGoodSuff(std::vector<int>& l_func, int ind_patt, int patt_size, int gp_size
        ) {
104     if (ind_patt == patt_size) {
105         return 1;
106     }
107     if (l_func[ind_patt] == UNDEFINED) {
108         return patt_size - gp_size;
109     }
110     return patt_size -1 - l_func[ind_patt];
111 }
112
113 int main() {
114     std::ios_base::sync_with_stdio(false);
115   std::cin.tie(nullptr);
116
117   std::string line;
118   getline(std::cin, line);
119   std::vector<int> pattern;
120   ParseStrToVec(std::move(line), pattern);
121   int n = pattern.size();
122
```

```
123     int word_count = 0;
124   std::vector<int> words_in_line;
125   std::vector<int> text;
126   while (getline(std::cin, line)) {
127     ParseStrToVec(std::move(line), text);
128     word_count += text.size() - word_count;
129     words_in_line.push_back(word_count);
130   }
131     int gp_size = 0;
132   bool equality = false;
133     std::unordered_map<int, std::vector<int>> p_func = PFunction(pattern);
134   std::vector<int> n_func;
135     std::vector<int> l_func = LFunction(pattern, n_func, gp_size);
136   std::vector<int> m_func(text.size(), UNDEFINED);
137     for (int k = pattern.size() - 1; k < text.size();) {
138     int i = k;
139     bool isOk = true;
140     for (int j = pattern.size() - 1; j >= 0; ) {
141
142       if (m_func[i] == UNDEFINED || (m_func[i] == 0 && n_func[j] == 0)) {
143         if (text[i] != pattern[j]) { //
144           int offset = std::max(UseRuleBadLetter(p_func, text[i], j, pattern.size()),
                UseRuleGoodSuff(l_func, j+1, pattern.size(),gp_size));
145           offset = std::max(offset, 1);
146           m_func[k] = k - i;
147           k += offset;
148           isOk = false;
149
150           break;
151         }
152         --i; //   , .
153         --j;
154       } else if (m_func[i] < n_func[j]) { //        N,     .      [i]
155         j -= m_func[i];
156         i -= m_func[i];
157       } else if (m_func[i] == n_func[j]) {
158         if (n_func[j] == j+1) { //    (.. )     ,    ,
159           isOk = true;
160           equality = true;
161         }
162         m_func[k] = k - i;
163         j -= m_func[i];
164         i -= m_func[i];
165
166       } else if (m_func[i] > n_func[j]) {
167         if (n_func[j] == j+1) {
168           isOk = true;
169           equality = true;
170           m_func[k] = k - i;
```

```
171           i -= m_func[i];
172           j -= n_func[j]; //  j -= m_func[i].     ,  j = -1;
173         } else { //   _ > n_func,    ,     -
174           m_func[k] = k - i;
175           j -= n_func[j];
176           i -= n_func[j];
177           if (text[i] != pattern[j]) { //
178             int offset = std::max(UseRuleBadLetter(p_func, text[i], j, pattern.size()),
                      UseRuleGoodSuff(l_func, j+1, pattern.size(),gp_size));
179             offset = std::max(offset, 1);
180             k += offset;
181             isOk = false;
182             break;
183           }
184         }
185       }
186     }
187     if (isOk) {
188       if (!equality) {
189         m_func[k] = pattern.size() - 1;
190       }
191       equality = false;
192       int ind = k - pattern.size() + 1;
193       auto it = std::upper_bound(words_in_line.begin(), words_in_line.end(), ind);
194       std::cout << distance(words_in_line.begin(), it) + 1 << ", ";
195       if (it == words_in_line.begin()) {
196         std::cout << ind + 1;
197       }
198       else {
199         std::cout << ind + 1 - *prev(it);
200       }
201       std::cout << '\n';
202       k += pattern.size() - gp_size;
203     // std::cout << "new k = " << k <<'\n';
204     }
205   }
206     return 0;
207 }
```

3 Консоль

test:

```
11 45 11 45 90
0011 45 011 0045 11 45 90    11
45 11 45 90
```

console output:

```
1,3
1,8
```

4 Тест производительности

Тест представлял из себя сравнение реализованного мной алгорима Апостолико-Джанкарло с бинарным поиском, реализованным в STL посредством функции upper_bound.

В результате работы *benchmark.cpp* видны следующие результаты:

```
a@WIN-THNQL51M105:~/Desktop/DA/lab4/bench# ./wrapper.sh
[info][Thu Jun  2 18:24:03 MSK 2022] Stage #1. Compiling...
g++ -std=c++17 -pedantic -Wall -Wextra -Wno-unused-variable benchmark.cpp -o
benchmark
[info][Thu Jun  2 18:24:03 MSK 2022] Compiling OK
[info][Thu Jun  2 18:24:04 MSK 2022] Stage #2. Benchmark generating...
[info][Thu Jun  2 18:24:05 MSK 2022] Benchmark generating OK
[info][Thu Jun  2 18:24:06 MSK 2022] Stage #3. Benchmark results:
BM_search: 220 ms
bin_search: 622 ms
[info][Thu Jun  2 18:24:07 MSK 2022] Benchmark OK
```

Из примера видно, что реализованный мной алгорим превосходит бинарный поиск из STL, потому что он обладает линейной сложностью, а бинарный поиск - операция за O(log(n)).

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», мною были изучены различные алгоритмы поиска подстроки в строке и реализован алгоритм Апостолико-Джанкарло.

Мой алгоритм является эффективной модификацией алгоритма Бойера-Мура за счет избегания проверки уже совпавших суффиксов.

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] Алгоритм Бойера—Мура. URL: https://ru.wikipedia.org/wiki/Алгоритм-