Московский авиационный институт (национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: А. А. Каримов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б-22

Дата:
Оценка:
Подпись:

Москва, 2024

Лабораторная работа №4

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

Вариант алгоритма: `Поиск одного образца при помощи алгоритма Бойера-Мура.`
Вариант алфавита: `Числа в диапазоне от 0 до` $2^{32} - 1$.

1 Описание

Требуется реализовать алгоритм Бойера-Мура для поиска подстроки в строке. Притом алфавит представляет собой множество четырех-байтных целых неотрицательных чисел. Алгоритм Бойера-Мура прикладывает образец к тексту и начинает проверку с конца. Также в алгоритме используется две эвристики(правила) для достижения наибольшей скорости. Первое правило - правило плохого символа. В случае, когда символ текста и образца не совпал, берется буква несовпавшая буква текста и ищется её самое правое вхождение, соответственно образец сдвигается так, чтобы совместить эти буквы. Если буква не присутствует в образце, происходит сдвиг на длина образца. Второе правило - правило хорошего суффикса. Снова, при несовпадении буквы текста и образца, ищется самая правая подстрока, которая равна совпавшему суффиксу, происходит сдвиг так, чтобы совместить эти подстроки. Если же такой подстроки нет, то ищется наибольший префикс, совпадающий с проверенным суффиксом, после сдвига префикс прикладывается на место совпавшего суффикса.

2 Исходный код

Здесь располагается реализация алгоритма Бойера-Мура. Поиск самого правого символа осуществляется с помощью бинарного поиска, нахождение совпадающих с суффиксом подстрок реализуется с помощью z-алгоритма.

```
template<class T, class Alph>
class Boyer_Moore {

    private:

        /* heuristic of good suffix

            N[j] - longest suffix of substring s[1..j] which is also a suffix of s

            l_[i] - largest position such that suffix of s matches the suffix of s
                [1..l_[i]] (strong one)

            l[i] - longest suffix of s[i..n] which is also a prefix of s
        */
        std::vector<int> N;
        std::vector<int> l_;
        std::vector<int> lp;
        std::map<Alph, std::vector<int>> entries;
        T pattern;

        int skipIfMatched;

        const int NOT_FOUND_LETTER = -1;

    private:

        std::vector<int> z_func(const T& s) {
            int n = s.size();
            std::vector<int> z(n);
            int l = 0, r = 0;

            for(int i = 1; i < n; ++i) {
                if(i <= r)
                    z[i] = std::min(z[i - l], r - i + 1);
                while(i + z[i] < n and s[z[i]] == s[i + z[i]]) ++z[i];

                if(r < i + z[i] - 1) {
                    l = i;
                    r = i + z[i] - 1;
                }
            }
            return z;
        }
```

```
43
44          private:
45
46              void get_letters_entry() {
47                  std::size_t i = 0;
48                  for(const Alph& el : pattern) {
49                      entries[el].push_back(i++);
50                  }
51              }
52
53              void suffix_equal_to_prefix() {
54                  int n = pattern.size();
55                  int i = 0;
56                  for (int j = n - 1; j >= 0; j--)
57                  {
58                      if (N[j] == j + 1)
59                      {
60                          while (i <= n - (j + 1))
61                              lp[i++] = j + 1;
62                      }
63                  }
64              }
65
66              void substr_suffix_equal_suffix() {
67                  T s_copy(pattern);
68                  reverse(s_copy.begin(), s_copy.end());
69                  N = z_func(s_copy);
70                  reverse(N.begin(), N.end());
71              }
72
73              void equal_suffixes() {
74                  substr_suffix_equal_suffix();
75                  int n = pattern.size();
76                  for(int j = 0; j < n - 1; ++j) {
77                      int i = n - N[j];
78                      if(i < n)
79                          l_[i] = j;
80                  }
81              }
82
83              int good_suffix_heuristic(int i) {
84
85                  // First comparison
86                  if(i == pattern.size() - 1)
87                      return 0;
88
89                  // shifting from mismatched letter
90                  ++i;
91                  if(l_[i] > 0)
```

```
92                  return pattern.size() - l_[i] - 1;
93              return pattern.size() - lp[i] - 1;
94          }
95
96          int bad_character_heuristic(int i, Alph c) {
97              int entry = utils::binary_search(entries[c], i);
98              // std::cout << '\n' << entry << ' ' << c << '\n';
99              if(entry == NOT_FOUND_LETTER)
100                 return i + 1;
101             return i - entries[c][entry];
102         }
103
104     public:
105
106         Boyer_Moore(const T& _pattern) : pattern(_pattern) {
107             lp.resize(pattern.size());
108             l_.resize(pattern.size());
109             get_letters_entry();
110             equal_suffixes();
111             suffix_equal_to_prefix();
112             skipIfMatched = std::max(1, static_cast<int>(pattern.size() - lp[1]));
113         }
114
115         std::vector<int> matches(const T& text) {
116             std::vector<int> entry;
117
118             if(text.size() < pattern.size()) {
119                 return entry;
120             }
121
122             int n = text.size();
123             int m = pattern.size();
124             int shift = 0;
125             bool matched;
126             int i = 0;
127
128             while(i < n - m + 1) {
129                 matched = true;
130                 for(int j = m - 1; j >= 0; --j) {
131                     if(pattern[j] != text[i + j]) {
132                         shift = std::max({bad_character_heuristic(j, text[i+j]),
                                good_suffix_heuristic(j), 1});
133                         matched = false;
134                         break;
135                     }
136                 }
137
138                 if(matched) {
139                     entry.push_back(i);
```

```
                    shift = skipIfMatched;
                }

                i += shift;
            }

            return entry;
        }
    };

    template<class T, class Alph>
    std::vector<int> boyer_moore_search(const T& text, const T& pattern) {
        Boyer_Moore<T, Alph> bm(pattern);
        return bm.matches(text);
    }
```

3 Консоль

```
karseny99@karseny99:/mnt/c/Users/Arsen/OneDrive/Документы/study/DA/lab4/src$
cat 2.in
4212930059 2980345134 2671036067 3694596163 2317633501
4212930059 2980345134 2671036067 3694596163 2317633501 4212930059 2980345134
2671036067 3694596163 2317633501 4212930059 2980345134 2671036067 3694596163
2317633501 4212930059 2980345134 2671036067 3694596163 2317633501 4212930059
2980345134 2671036067 3694596163 2317633501 4212930059 2980345134 2671036067
3694596163 2317633501 4212930059 2980345134 2671036067 3694596163 2317633501
4212930059 2980345134 2671036067 3694596163
2317633501 4212930059 2980345134 2671036067 3694596163
2317633501
karseny99@karseny99:/mnt/study/DA/lab4/src$ ./lab4 <2.in
1,1
1,6
1,11
1,16
1,21
1,26
1,31
1,36
2,2
```

4 Тест производительности

Производительность оценивается так: на один и тех же тестовых данных запускается наивный алгоритм и алгоритм Бойера-Мура. Тесты на 10^4, 10^5 и 10^6 слов в тексте.

```
karseny99@karseny99:/mnt/study/DA/lab4/src$ ./benchmark <tests/test0.t
Boyer-Moore algorithm: 6145 us
Naive: 98323 us
karseny99@karseny99:/mnt/study/DA/lab4/src$ ./benchmark <tests/test1.t
Boyer-Moore algorithm: 9625 us
Naive: 129312 us
karseny99@karseny99:/mnt/study/DA/lab4/src$ ./benchmark <tests/test2.t
Boyer-Moore algorithm: 16623 us
Naive: 258624 us
```

5 Выводы

В ходе выполнения лабораторной работы я смог реализовать алгоритм Бойера-Мура нахождения подстроки в строке. Существует много разных алгоритмов поиска подстрок, такое разнообразие обусловлено тем, что, к примеру, алгоритм Бойера-Мура будет долго искать вхождение разных паттернов в текст, когда как алгоритм Ахо-Корасик способен искать сразу несколько паттернов в тексте.

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`