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.Д. Волков
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата: 28.05.2024
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 Кнута-Морриса-Пратта.

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

1 Описание

Требуется написать реализацию алгоритма Кнута Морриса Пратта. Алгоритм работает за линейное время. Есть множество способов для реализации данного алгоритма. При написании своей лабораторной работы я решил реализовать этот алгоритм с помощью построения сильной префикс функции на основе Z функции.

Алгоритм заключается в сдвиге паттерна относительно текста и посимвольного сравнения паттерна с текстом. Различия всех реализаций в основном касаются сдвигов паттерна, которые могут ускорить алгоритм. Я реализовал сдвиг с помощью сильной префикс функции. Она ищет максимальный по длине префикс совпадающий с суффиксом. С помощью этого мы гарантируем что мы не пропустим ни одного вхождения нашего паттерна в текст. [**?**].

2 Исходный код

Опишем функции. std::vector<size_t> ZFunction(std::vector<uint32_t>& string) - функция для вычисления Z функции. Определяем левые и правые границы покрытия. Идем циклом по строке. Если текущее слово в паттерне находится внутри покрытия, то для него есть два пути. Первый, если значение Z функции в соответствующем месте в префиксе меньше чем количество слов до правой границы. Тогда мы присваиваем значение текущей Z функции и идем дальше. Если же нет, то мы пропускаем символы до правой границы, и смотрим символы за правой границей. Если мы вычислили Z функцию уже за правую границу, то мы должны перенести наше покрытие и продвигаться дальше. std::vector<size_t> StrongSPFunction(std::vector<uint32_t>& string) - функция для вычисления сильной префикс функции на основе Z функции. С помощью формулы мы можем вычислить значение сильной префикс функции на основе Z функции. Проходимся по паттерну справа налево чтобы находить самые большие префиксы и суффиксы. std::vector<size_t> KMPSearch(std::vector<uint32_t>& pattern, std::vector<std::pair<uint32_t, std::pair<size_t, size_t»>& text) - функция для поиска подстроки в строке с помощью алгоритма Кнута Морриса Пратта. Проходимся по тексту, сопоставляя с ним паттерн. Когда нашли совпадение, записываем его в результирующий вектор. Если же не нашли, то выполняем сдвиг, ориентируясь на значение сильной префикс функции. int main() - точка вхождения программы. Считываем первую строку и записываем ее в вектор паттерна. Далее считываем строки текста. Для оптимизации программы, мы будем выполнять поиск подстроки во время чтения текста, если количество слов в тексте будет в два раза больше чем количество слов в паттерне. Далее просто булем отсекать просмотренные части текста и добавлять к ним новые. В конце опять выполним поиск оставшейся части текста и выведем результат.

```
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
#include <cstdint>
#include <iterator>

std::vector<size_t> ZFunction(std::vector<uint32_t>& string)
{
    std::vector<size_t> zfunc(string.size());
    size_t left = 0;
    size_t right = 0;
    for (size_t i = 1; i < string.size(); ++i)
    {
        if (i <= right)
        {
            zfunc[i] = std::min(zfunc[i - left], right - i);
        }
```

```
19          while ((i + zfunc[i] < string.size()) && (string[zfunc[i]] == string[i + zfunc[
                i]]))
20          {
21              zfunc[i]++;
22          }
23          if (i + zfunc[i] > right)
24          {
25              left = i;
26              right = i + zfunc[i];
27          }
28      }
29      return zfunc;
30  }
31
32  std::vector<size_t> StrongSPFunction(std::vector<uint32_t>& string)
33  {
34      std::vector<size_t> spfunc(string.size());
35      std::vector<size_t> zfunc = ZFunction(string);
36      for (size_t i = string.size() - 1; i > 0; --i)
37      {
38          spfunc[i + zfunc[i] - 1] = zfunc[i];
39      }
40      return spfunc;
41  }
42
43  std::vector<size_t> KMPSearch(std::vector<uint32_t>& pattern, std::vector<std::pair<
        uint32_t, std::pair<size_t, size_t>>>& text)
44  {
45      std::vector<size_t> spfunc = StrongSPFunction(pattern);
46      std::vector<size_t> result;
47      if (pattern.size() <= text.size())
48      {
49          size_t text_iterator = 0;
50          while (text_iterator < text.size() - pattern.size() + 1)
51          {
52              size_t pattern_iterator = 0;
53              while ((pattern_iterator < pattern.size()) && (pattern[pattern_iterator] ==
                     text[text_iterator + pattern_iterator].first))
54              {
55                  pattern_iterator++;
56              }
57              if (pattern_iterator == pattern.size())
58              {
59                  result.push_back(text_iterator);
60              }
61              else
62              {
63                  if ((pattern_iterator >= 1) && (pattern_iterator > spfunc[
                        pattern_iterator - 1]))
```

```
64                {
65                    text_iterator += (pattern_iterator - spfunc[pattern_iterator - 1] -
                          1);
66                }
67            }
68            text_iterator++;
69        }
70    }
71    return result;
72 }
73
74 int main()
75 {
76    std::string input_pattern;
77    std::getline(std::cin, input_pattern);
78    std::istringstream pattern_stream(input_pattern);
79    std::vector<uint32_t> pattern;
80    std::copy(std::istream_iterator<uint32_t>(pattern_stream), std::istream_iterator<
          uint32_t>(), std::back_inserter(pattern));
81    std::string input_text;
82    std::vector<std::pair<uint32_t, std::pair<size_t, size_t>>> text;
83    size_t lc = 1;
84    std::vector<size_t> result;
85    while (std::getline(std::cin, input_text))
86    {
87        std::istringstream text_stream(input_text);
88        uint32_t x;
89        size_t wc = 1;
90        while (text_stream >> x)
91        {
92            text.push_back({x, {wc, lc}});
93            wc++;
94        }
95        lc++;
96        if (text.size() > 2 * pattern.size())
97        {
98            result = KMPSearch(pattern, text);
99            for (size_t i = 0; i < result.size(); ++i)
100           {
101               std::cout << text[result[i]].second.second << ", " << text[result[i]].
                      second.first << "\n";
102           }
103           std::vector<std::pair<uint32_t, std::pair<size_t, size_t>>> tmp;
104           for (size_t i = text.size() - pattern.size() + 1; i < text.size(); ++i)
105           {
106               tmp.push_back(text[i]);
107           }
108           text = tmp;
109       }
```

```
110     }
111     result = KMPSearch(pattern, text);
112     for (size_t i = 0; i < result.size(); ++i)
113     {
114         std::cout << text[result[i]].second.second << ", " << text[result[i]].second.
                first << "\n";
115     }
116 }
```

main.c	
std::vector<size$_t$ > $ZFunction()$	Функция подсчета Z функции
std::vector<size$_t$ > $StrongSPFunction()$	Функция подсчета сильной префикс функции
std::vector<size$_t$ > $KMPSearch()$	Функция поиска подстроки в строке
int main()	Точка входа программы

3 Консоль

```
lexasy@MSI$ g++ main.cpp
lexasy@MSI$ cat test
001 020 000000004294967295
```

```
001
020
001
020
000000004294967295
001
020
001
001
020
000000004294967295
001
020
001
00000000000000012345678
000000004294967295
```

```
001
020
000000004294967295
001
020
001
00000000000000012345678
000000004294967295
00000000000000012345678
000000004294967295
000000004294967295
001
020
001
00000000000000012345678
001
020
000000004294967295
lexasy@MSI$ ./a.out <test
7,1
13,1
21,1
36,1
```

4 Тест производительности

Тест производительности представляет из себя следующее: сравнение поиска подстроки в строке с помощью стандартных инструментов языка и поиска подстроки в строке с помощью алгоритма КМП.

```
lexasy@MSI$ g++ main.cpp
lexasy@MSI$ ./a.out <test >out
lexasy@MSI$ cat out | grep "time"
time: 2707ms
lexasy@MSI$ g++ benchmark.cpp
lexasy@MSI$ ./a.out <test >out
lexasy@MSI$ cat out | grep "time"
time: 2053ms
```

Как видно, КМП немного отстает. Скорее всего это связано с тем, что я в своем алгоритме пытался потимизировать его по памяти. Поэтому пришлось немного пожертвовать временем. Такое различие во времени некритично.

5 Выводы

Выполнив 4 лабораторную работу по курсу Дискретный Анализ, я научился реализовывать один из алгоритмов для линейного поиска подстроки в строке. Но при этом, реализовав один алгоритм, я понял как можно реализовать алгоритм Z функции. Также я узнал элегантный способ экономии памяти в таких алгоритмах, что помогло выполнить лабораторную работу быстрее и который точно мне поможет при выполнении следующих лабораторных работ.

Список литературы

[1] Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. — СПб.: Невскии диалект, 2003.