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: Е. А. Айрапетова
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2021

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 $2^{32} - 1$.

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

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

Требуется написать реализацию алгоритма Кнута-Морриса-Пратта для поиска подстроки в строке для чисел в диапазоне от 0 до $2^{32} - 1$

Как сказано в [3]: «КМП-алгоритм — эффективный алгоритм, осуществляющий поиск подстроки в строке. Время работы алгоритма линейно зависит от объёма входных данных». Суть этого алгоритма в том, чтобы делать больший сдвиг шаблона по тексту. Для этого существуют префикс-функция и функция неудач.

Префикс-функция от строки S и позиции i в ней - длина k наибольшего собственного (не равного всей подстроке) префикса подстроки $S[1..i]$, который одновременно является суффиксом этой подстроки.

Для каждого места i от 1 до n+1 определим функцию неудач F(i), как π(i-1)+1, причем π(01) приняты равными 0. Мы используем указатель p на место в P и один указатель c на место в T. После несовпадения в позиции i+1 $>$ 1 строки P алгоритм КМП "сдвигает"P так, что следующим будут сравниваться символы в позиции c строки T и в позиции π[i]+1 строки P. Но π[i] + 1 = F(i+1), так что общий "сдвиг"можно выполнить за константное время, просто полагая p равным F(i+1). Когда несовпадение нашлось в позиции 1 из P, p полагается равным F(1) = 1, а c увеличивается на 1. Когда находится вхождение P, то P сдвигается вправо на n - π(n) мест. Этот сдвиг реализуется тем, что F(n+1) полагается равным π(n)+1.

Вычисление префикс-функции происходит за O(m), где m - длина шаблона. Алгоритм поиска, соответственно, работает за O(n + m), где n - длина текста. ПО памяти алгоритм оценивается как O(m).

2 Исходный код

Заголовочный файл KMP.h:

```
#pragma once

#include <vector>
#include <string>

std::vector<int> CalculateSP(std::vector<int> &z, std::vector<std::string> &pat);
std::vector<int> ZFunction(std::vector<std::string> &pat);
std::vector<int> FailFunction(std::vector<std::string> &pattern);
void KMP(std::vector<std::string> &pat);
```

В файле KMP.cpp реализуем вычисление префикс-функции для шаблона:

```
#include "KMP.h"

std::vector<int> CalculateSP(std::vector<int> &zArray, std::vector<std::string> &pat)
    {
    const unsigned long psize = pat.size();
    std::vector<int> spArray(psize);

    for (unsigned long j = psize - 1; j > 0; --j) {
        unsigned long i = j + zArray[j] - 1;
        spArray[i] = zArray[j];
    }

    return spArray;
}
```

Далее реализуем функцию неудач и Z-функцию:

```
std::vector<int> ZFunction(std::vector<std::string> &pat) {
    const unsigned long psize = pat.size();
    std::vector<int> zArray(psize);

    for (int i = 1, l = 0, r = 0; i < psize; ++i) {
        if (i <= r) {
            zArray[i] = std::min(r - i + 1, zArray[i - l]);
        }
        while (i + zArray[i] < psize && pat[zArray[i]] == pat[i + zArray[i]]) {
            ++zArray[i];
        }
        if (i + zArray[i] - 1 > r) {
            l = i;
            r = i + zArray[i] - 1;
        }
    }
    return zArray;
}
```

```
19
20 std::vector<int> FailFunction(std::vector<std::string> &pattern) {
21     unsigned long psize = pattern.size();
22
23     std::vector<int> zArray = ZFunction(pattern);
24     std::vector<int> spArray = CalculateSP(zArray, pattern);
25     std::vector<int> f(psize + 1);
26     f[0] = 0;
27
28     for (int k = 1; k < psize; ++k) {
29         f[k] = spArray[k - 1];
30     }
31     f[psize] = spArray[psize - 1];
32     return f;
33 }
```

Затем реализуем непосредственно KMP-алгоритм, в который включим считывание строки:

```
1  void KMP(std::vector<std::string> &pat) {
2      std::vector<std::pair<std::pair<int, int>, std::string>> text;
3
4      char c = '$';
5      bool wordAdded = true;
6      std::pair<std::pair<int, int>, std::string> temp;
7      const unsigned long n = pat.size();
8      temp.first.first = 1;
9      temp.first.second = 1;
10     int p = 0;
11     int t = 0;
12     std::vector<int> f = FailFunction(pat);
13     do {
14         while (text.size() < pat.size() && c != EOF) {
15             c = getchar_unlocked();
16             if (c == '\n') {
17                 if (!wordAdded) {
18                     text.push_back(temp);
19                     temp.second.clear();
20                     wordAdded = true;
21                 }
22                 ++temp.first.second;
23                 temp.first.first = 1;
24             }
25             else if (c == ' ' || c == '\t' || c == EOF) {
26                 if (wordAdded) {
27                     continue;
28                 }
29                 text.push_back(temp);
30                 temp.second.clear();
31                 ++temp.first.first;
```

```
            wordAdded = true;
        }
        else {
            if (temp.second.front() == '0') {
                temp.second.erase(temp.second.begin());
            }
            temp.second.push_back(c);
            wordAdded = false;
        }
    }
    if (text.size() < pat.size()) {
        return;
    }
    while (p < n && pat[p] == text[t].second ) {
        ++p;
        ++t;
    }
    if (p == n) {
        printf("%d, %d\n", text[0].first.second, text[0].first.first);
    }
    if (p == 0) {
        ++t;
    }
    p = f[p];
    text.erase(text.begin(), text.begin() + t - p);
    t = p;
} while (c != EOF);
}
```

Далее напишем main-функцию, в которой будет считываться шаблон и вызываться функция KMP:

```
#include <iostream>
#include <vector>
#include <string>
#include "KMP.h"

int main() {
    std::string buf;
    std::vector<std::string> pattern;

    char c;
    while (true) {
        c = getchar_unlocked();
        if (c == ' ') {
            if (!buf.empty()) {
                pattern.push_back(buf);
            }
            buf.clear();
        }
```

```
19          else if (c == '\n' || c == EOF) {
20              if (!buf.empty()) {
21                  pattern.push_back(buf);
22              }
23              break;
24          }
25          else {
26              buf.push_back(c);
27          }
28      }
29
30      KMP(pattern);
31      return 0;
32  }
```

3 Консоль

```
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$ make
g++ -std=c++17 -O2 -Wextra -Wall -Wno-sign-compare -Wno-unused-result -pedantic
-c main.cpp
g++ -std=c++17 -O2 -Wextra -Wall -Wno-sign-compare -Wno-unused-result -pedantic
-c KMP.cpp
g++ -std=c++17 -O2 -Wextra -Wall -Wno-sign-compare -Wno-unused-result -pedantic
main.o KMP.o -o solution
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$ ./solution <f.txt
1,3
1,8
3,1
4,1
14,2
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$ ./solution
11 45 11 45 90
0011 45 011 0045 11 45 90    11
1,3
45 11 45 90
1,8
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$
```

4 Тест производительности

Сравним алгоритм Кнута - Морриса - Пратта с наивным алгоритмом, время на считывание данных не учитывается. Тест состоит из текста длиной 1000 символов.

```
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$ ./benchmark <test.txt
KMP algorithm: 0.049 ms
Naive algorithm: 0.803 ms
jane@Evgenia:/mnt/c/Files/ДА/ЛР4/solution$
```

Как видно, КМП работает намного быстрее наивного алгоритма, скорость работы которого $O(m * (n - m))$, где n - длина текста, m - длина образца,

5 Выводы

Выполнив четвертую лабораторную работу по курсу «Дискретный анализ», я познакомилась с эффективными алгоритмами поиска подстроки в строке, в частности, с алгоритмом Кнута-Морриса-Пратта. Этот алгоритм не всегда является наиболее эффективным, по сравнению с другими, однако он довольно прост в понимании и поэтому широко применяется в таких задачах, как, например, поиск слова по тексту.

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 КМП - Хабр.*
URL: `https://habr.com/ru/post/307220/` (дата обращения: 16.06.2021).

[3] *Алгоритм КМП — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Алгоритм_Кнута_-_Морриса_-_Пратта` (дата обращения: 16.06.2021).