Московский авиационный институт
(национальный исследовательский университет)

Институт № 8 «Информационные технологии и прикладная математика»

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Дискретный анализ»

Студент: Л. В. Короткевич
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19
Дата:
Оценка:
Подпись:

Москва 2020

Лабораторная работа №4. Вариант 2-1.

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

Вариант алгоритма: Поиск одного образца при помощи алгоритма Бойера-Мура.

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

1. Описание

Алгоритм Бойера-Мура — один из самых распространенных алгоритмов общего назначения, предназначенный для поиска подстроки в строке. Основное его преимущество над другими алгоритмами поиска подстроки в строке состоит в том, что шаблон (образец) сравнивается с исходным текстом не во всех позициях — часть проверок пропускается как заведомо не дающие результата.

Общая оценка вычислительной сложности — O(n + m), если не используется таблица стоп-символов, и O(n + m + |E|), если используется. Здесь n — длина строки, m — образца, E — алфавит, на котором производится сравнение.

Итак, основные идеи алгоритма:

1. *Сканирование текста слева направо, а сравнение с шаблоном — справа налево.* В случае несовпадения, сдвиг образца вправо определяется следующими эвристиками:

2. *Эвристика стоп-символа.* Определим крайнее справа вхождение каждого символа образца, и в случае несовпадения pattern[i] с text[i + j] сдвинем шаблон вправо так, чтобы ближайший слева от «места несовпадения» символ text[i + j] в образце встал на место символа pattern[i].

3. *Эвристика хорошего суфикса.* При несовпадении в позиции i, сдвинем шаблон вправо так, чтобы его суффикс pattern[i + 1 … m] совпал с точно такой же подстрокой, ближайшей слева от «места несовпадения», притом слева этой подстроки обязательно стоит символ, не равный pattern[i].

С помощью данных правил подсчитаем возможные сдвиги и выберем из них максимальный.

2. Исходный код

main.cpp:

```
#include <algorithm>
#include <iostream>
#include <map>
#include <sstream>
#include <vector>

void BadCharacter(const std::vector<std::string> &pattern, std::map<std::string, int> &badCharacter)
{
    for (int i = 0; i < (int)pattern.size() - 1; ++i)
        badCharacter[pattern[i]] = i;
}

void GoodSuffix(const std::vector<std::string> &pattern, std::vector<int> &goodSuffix)
{
    int n = (int)pattern.size();
    int left = 0;
    int right = 0;

    std::vector<int> zFunction(n, 0);
    for (int i = 1; i < n; ++i) {
        if (i <= right)
            zFunction[i] = std::min(right - i + 1, zFunction[i - left]);

        while (i + zFunction[i] < n && (pattern[n - 1 - zFunction[i]] == pattern[n - 1 - (i + zFunction[i])]))
            zFunction[i]++;

        if (i + zFunction[i] - 1 > right) {
            left = i;
            right = i + zFunction[i] - 1;
        }
    }

    std::vector<int> N(zFunction.rbegin(), zFunction.rend());

    std::vector<int> L(n + 1, n);

    int j;
    for (int i = 0; i < n - 1; ++i) {
```

```
        j = n - N[i];
        L[j] = i;
    }

    std::vector<int> l(n + 1, n);
    for (int i = n - 1; i >= 0; i--) {

        j = n - i;
        if (N[j - 1] == j)

            l[i] = (j - 1);
        else

            l[i] = l[i + 1];
    }

    for (int i = 0; i < n + 1; ++i)
        if (L[i] == n)
            L[i] = l[i];

    for (auto &x : L)
        if (x != n)
            x = n - 1 - x;

    goodSuffix = L;
}

void Lowercase(std::string &str)
{
    for (int i = 0; i < (int)str.size(); ++i)
        if (str[i] >= 'A' && str[i] <= 'Z')
            str[i] += 'a' - 'A';
}

void Read(std::vector<std::string> &pattern, std::vector<std::string> &txt, std::vector<std::pair<int, int>> &rowCol)
{

    std::string curLine;
    getline(std::cin, curLine, '\n');

    std::istringstream stringStream(curLine);
```

```
    std::string curWord;
    while (stringStream >> curWord) {
        Lowercase(curWord);
        pattern.push_back(curWord);
    }

    int lineNum = 1;
    int wordNum = 0;
    int lineLen;
    int left;
    std::pair<int, int> lineNumWordNum;
    while (getline(std::cin, curLine, '\n')) {
        wordNum = 1;
        lineLen = curLine.size();
        for (int curPos = 0; curPos < lineLen;) {
            if (isalpha(curLine[curPos])) {
                left = curPos;
                while (isalpha(curLine[curPos]) && curPos < lineLen)
                    ++curPos;

                lineNumWordNum.first = lineNum;
                lineNumWordNum.second = wordNum++;
                rowCol.push_back(lineNumWordNum);

                curWord = curLine.substr(left, curPos - left);
                Lowercase(curWord);
                txt.push_back(curWord);
            }
            ++curPos;
        }
        ++lineNum;
    }
}

int main()
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr), std::cout.tie(nullptr);

    std::vector<std::string> pattern;
    std::vector<std::string> txt;
    std::vector<std::pair<int, int>> rowCol;
```

```
    Read(pattern, txt, rowCol);

    std::map<std::string, int> badCharacter;
    BadCharacter(pattern, badCharacter);

    std::vector<int> goodSuffix(pattern.size() + 1, pattern.size());
    GoodSuffix(pattern, goodSuffix);

    std::vector<int> occurPos;
    int j, bound = 0, shift = 0;
    for (int i = 0; i < 1 + (int)txt.size() - (int)pattern.size();) {
        for (j = pattern.size() - 1; j >= bound; j--) {
            if (pattern[j] != txt[i + j]) {
                break;
            }
        }
        if (j < bound) {
            occurPos.push_back(i);
            bound = pattern.size() - goodSuffix[0];
            j = -1;
        }
        else {
            bound = 0;
        }
        if (j < bound) {
            shift = goodSuffix[j + 1];
        }
        else {
            shift = std::max(goodSuffix[j + 1], j - badCharacter[txt[i + j]]);
        }
        i += shift;
    }

    for (const auto &oc : occurPos)
        std::cout << rowCol[oc].first << ", " << rowCol[oc].second << '\n';

    return 0;
}
```

3. Консоль

```
[leo@pc final]$ cat input1
guard
thread annoying
rhythm fry
fry bee
thread humorous
sassy turn
[leo@pc final]$ ./prog <input1
[leo@pc final]$ cat input 2
lopsided cynical
lopsided shaggy labored lopsided
lopsided cynical
labored sore cynical cynical
lopsided thread actually responsible
languid even auspicious adventurous
lopsided cynical
humorous incredible scarf scarf
milky neighborly humorous sore
sassy sassy cynical auspicious
actually responsible
cat: 2: No such file or directory
[leo@pc final]$ ./prog <input2
4, 1
8, 1
[leo@pc final]$ cat input3
actually fool shelf
scarf responsible direful adventurous fretful fool
auspicious momentous milky lopsided turn report
rhythm neighborly sore
[leo@pc final]$ ./prog <input3
[leo@pc final]$ cat input4
guard shelf rebel even
cynical lopsided scarf bee shaggy momentous door neighborly
[leo@pc final]$ ./prog <input4
```

4. Тест производительности

Достаточно взглянуть на два теста, чтобы уловить преимущества и недостатки алгоритма Бойера-Мура одновременно.

Сравнение производится с std::find в строке-тексте.

1. Текст, состоящий из 10^6 слов, часто повторяющихся.

[leo@pc final]$ ./brute <input1e6

std::find: **45379ms**

[leo@pc final]$ ./prog <input1e6

BM: **26371ms**

2. Текст, состоящий из 10^6 слов, редко повторяющихся.

[leo@pc final]$ ./brute <input1e6_u

std::find: **10460ms**

[leo@pc final]$ ./prog <input1e6_u

BM: **13056ms**

Не удивительно: во втором случае сдвиг почти всегда был мизерным из-за отсутствия повторений подстрок, благодаря чему время поиска сильно увеличилось.

В первом же случае, уверен, сыграла свою роль эвристика хорошего суффикса: постоянно происходили сдвиги на приличное число шагов.

5. Вывод

По мере выполнения данной лабораторной работы я ознакомился с теоретической информацией о поиске образцов в тексте, описал алгоритм Бойера-Мура, используя обе его эвристики: стоп-символов и хорошего суффикса; в силу этого, временная сложность мною написанного алгоритма, как было сказано во «введении», есть O(n + m + |E|).

Что выяснилось по мере тестирования: алгоритм Бойера-Мура на «хороших» данных очень быстр, а вероятность появления «плохих» данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск. Разве что на коротких текстах выигрыш не оправдает предварительных вычислений.

Из недостатков можно отметить, что алгоритмы семейства Бойера-Мура не расширяются до приблизительного поиска, поиска любой строки из нескольких. Также, на больших алфавитах таблица стоп-символов может занимать много памяти. В таких случаях либо обходятся хэш-таблицами, либо дробят алфавит, рассматривая, например, 4-байтовый символ как пару двухбайтовых, либо (что проще всего) пользуются вариантом алгоритма Бойера-Мура без эвристики стоп-символов.

И все же, алгоритм Бойера-Мура считается наиболее эффективным алгоритмом поиска шаблонов в стандартных приложениях и командах, таких как Ctrl+F в браузерах и текстовых редакторах.