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: И. П. Моисеенков
Преподаватель: Н. С. Капралов
Группа: М8О-208Б-19

Дата: 19.12.2020
Оценка: отлично
Подпись:

Москва, 2020

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` $2^{32} - 1$.

Искомый образец задаётся на первой строке входного файла.

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.

В выходной файл нужно вывести информацию о всех вхождениях искомых образцов в обрабатываемый текст: по одному вхождению на строку.

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

Согласно [1], алгоритм Бойера-Мура последовательно прикладывает образец P к тексту T и проверяет совпадение символов P с прилежащими символами T. Когда проверка завершается, P сдвигается вправо по T. Но алгоритм Бойера-Мура в отличие от других подобных алгоритмов использует три идеи: просмотр справа налево, правило сдвига по плохому символу и правило сдвига по хорошему суффиксу. Совместный эффект этих идей позволяет алгоритму проверять меньше, чем $m + n$ символов и после некоторых улучшений работать за линейное время в худшем случае.

Правило плохого символа: Предположим, что при некотором сопоставлении P с T крайние правые $m-i$ символов P совпадают со своими парами в T, но следующий слева символ *P(i)* не совпадает со своей парой, скажем, в позиции k строки T. Правило плохого символа гласит, что P следует сдвинуть вправо на $max(1, i - R(T(k))$ мест, где $R(i)$ - индекс самого правого вхождения символа i в образце P. Таким образом, если крайнее правое вхождение в P символа $T(k)$ занимает позицию $j < i$, то P сдвигается так, чтобы символ j в P поравнялся с символом k в T. В противном случае P сдвигается на одну позицию.

Правило хорошего суффикса: Пусть строка P приложена к T и подстрока t из T совпадает с суффиксом P, но следующий левый символ уже не совпадает. Найдём, если она существует, крайнюю правую копию t' строки t в P такую, что t' не является суффиксом P и символ слева от t' в P отличается от символа слева от t в P. Сдвинем P вправо, приложив подстроку t' в P к подстроке t в T. Если t' не существует, то сдвинем левый конец P за левый конец t в T на наименьший сдвиг, при котором префикс сдвинутого образца совпал бы с суффиксом t в T. Если такого сдвига не существует, то сдвинем P на m позиций вправо. Если найдено вхождение P, то сдвинем P на наименьший сдвиг, при котором собственный префикс сдвинутого P совпадает с суффиксом вхождения P в T. Если такой сдвиг невозможен, то нужно сдвинуть P на m мест, то есть сдвинуть P за t в T.

2 Исходный код

Всю работу с паттерном я решил вынести в отдельный класс $TPattern$. Класс хранит std::vector<unsigned int> Pattern - сам паттерн, std::map<unsigned int, unsigned int> BCR - индексы самых правых вхождений символов алфавита в паттерне, std::vector<int> N, L, l - массивы с N-функцией, L'-функцией и l'-функцией соответственно.

В классе реализован метод для определения максимального сдвига. Максимальный сдвиг вычисляется как максимум из сдвига, предлагаемого правилом плохого символа, сдвига, предлагаемого правилом хорошего суффикса, и единицы.

```
class TPattern {
private:
    std::vector<unsigned int> Pattern;
    std::map<unsigned int, unsigned int> BCR;
    std::vector<int> N;
    std::vector<int> L;
    std::vector<int> l;

    void ReadPattern();
    void BadCharRule();
    std::vector<int> ZFunction(std::vector<unsigned int>& pattern);
    void NFunction();
    void LFunction();
    void lfunction();

    long long GetBCRMove(unsigned int symbol, long long pos);
    long long GetGSRMove(long long pos);
public:

    void Initialize();

    unsigned int& operator[](long long id);
    long long Size() const;
    long long GetMove(unsigned int symbol, long long pos);
    long long GetFinalMove();
};
```

Алгоритм поиска построен по правилам, описанным выше. Вместо сдвига мы увеличиваем индекс до значения, соответствующего правому концу сдвинутого паттерна.

```
long long k = m - 1;
    while (k < text.size()) {
        long long i = k;
        long long j = m - 1;
        while (j >= 0 && pattern[j] == text[i]) {
            --i;
            --j;
        }
        if (j == -1) {
```

```
            std::cout << positions[k - m + 1].first + 1 << ", " << positions[k - m +
                1].second + 1 << "\n";
            long long move = pattern.GetFinalMove();
            k += move;
        }
        else {
            long long move = pattern.GetMove(text[i], j);
            k += move;
        }
    }
```

3 Консоль

```
mosik@LAPTOP-69S778GL:~/da_lab4$ make
g++ -pedantic -Wall -std=c++11 -Werror -Wno-sign-compare -O2 -lm main.cpp -o
solution
mosik@LAPTOP-69S778GL:~/da_lab4$ cat test.txt
11 45 11 45 90
0011 45 011 0045 11 45 90    11
45 11 45 90
mosik@LAPTOP-69S778GL:~/da_lab4$ ./solution <test.txt
1,3
1,8
mosik@LAPTOP-69S778GL:~/da_lab4$ cat test.txt
15 30 15 15 30
15
15 30 30 15


30 15 15 30 15 15
30
mosik@LAPTOP-69S778GL:~/da_lab4$ ./solution <test.txt
2,4
5,3
mosik@LAPTOP-69S778GL:~/da_lab4$ cat test.txt
0 0 0
1 1 1
mosik@LAPTOP-69S778GL:~/da_lab4$ ./solution <test.txt
```

4 Тест производительности

Я решил сравнить свою реализацию алгоритма Бойера-Мура с наивным алгоритмом поиска.

Тестирование происходило на тестах с примерно миллионом слов. В первом тесте образец часто встречался в тексте, во втором - не встречался вообще.

```
// Test 1
mosik@LAPTOP-69S778GL:~/da_lab4$ ./solution <test.txt >result.txt
Time: 0.164362 seconds
mosik@LAPTOP-69S778GL:~/da_lab4$ ./naive <test.txt >result1.txt
Time: 0.397422 seconds
mosik@LAPTOP-69S778GL:~/da_lab4$ diff result.txt result1.txt

// Test 2
mosik@LAPTOP-69S778GL:~/da_lab4$ ./solution <test.txt >result.txt
Time: 0.0644277 seconds
mosik@LAPTOP-69S778GL:~/da_lab4$ ./naive <test.txt >result1.txt
Time: 0.148165 seconds
mosik@LAPTOP-69S778GL:~/da_lab4$ diff result.txt result1.txt
```

По результатам тестирования видно, что в мой алгоритм значительно выиграл по времени у наивного алгоритма. Благодаря правилам плохого символа и хорошего суффикса алгоритму Бойера-Мура удалось выполнять достаточно большие сдвиги и не выполнять лишних проверок.

Стоит отметить, что у асимптотическая сложность алгоритмов различается. Сложность алгоритма Бойера-Мура - O(m + n), а наивного алгоритма - O(m * n), где m и n - длины паттерна и текста соответственно.

5 Выводы

Выполнив четвёртую лабораторную работу по курсу «Дискретный анализ», я изучил алгоритмы поиска образца в строке: алгоритм Кнута-Морриса-Пратта, алгоритм Апостолико-Джанкарло, алгоритм Ахо-Корасик и реализовал алгоритм Бойера-Мура.

Алгоритм Бойера-Мура на хороших данных довольно 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. Но на искусственно подобранных неудачных текстах скорость алгоритма Бойера-Мура серьёзно снижается. Также на больших алфавитах (например, Юникод) алгоритм Бойера-Мура может занимать много памяти. В таких случаях либо обходятся хэш-таблицами, либо дробят алфавит, рассматривая, например, 4-байтовый символ как пару двухбайтовых.

Список литературы

[1] Ден Гасфилд. *Строки, деревья и последовательности в алгоритмах.* — Издательский дом «Невский диалект», 2003. Перевод с английского: И. В. Романовский. — 654 с. (ISBN 5-7940-0103-8)