Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Дискретный анализ»

Студент: А. О. Ларченко
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

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 Апостолико-Джанкарло.

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

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

1 Описание

Требуется написать реализацию алгоритма Апостолико-Джанкарло для поиска одного образца в тексте.
Алгоритма Апостолико-Джанкарло построен на основе алгоритма Бойера-Мура, но является его улучшенной версией. Он не меняет то, что уже использовалось в алгоритме Бойера-Мура (проход по образу справа-налево, использование эвристик плохого символа и хорошего суффикса для эффективного сдвига при несовпадении), а добавляет к этому эффективный сдвиги, которые позволяют нам избежать лишних сравнений. [1]

Основная идея заключается в использовании большей полезной информации из правила хорошего суффикса. Для реализации алгоритма будем использовать дополнительный массив M длиной равной длине исходного текста, а также массив N, взятый из правила хорошего суффикса, который показывает, что $N_i(P)$ - длина наибольшего суффикса $P[1..i]$, совпадающего с суффиксом P. [1]
Также как и в алгоритме Бойера-Мура мы идем по тексту T указателем j, а по образцу P указателем i, но теперь при каждом сдвиге смотрим на значения N_i и M_j. Тут возможно 4 различных случая:

1) M_j не определено или N_i==0==M_j. Тогда если P_i==T_j и i>0, то декрементируем i и j, если i==0, то запоминаем вхождение образца и M_h присваиваем значение m, где h - правая граница текста, к которой в текущий момент приложен правый конец образца, а m - длина образца. Иначе - сдвиг по правилу Бойера-Мура, а M_h присваиваем значение $h - j$.

2) $M_j < N_i$. Эта ситуация означает что если подставить к T_j часть совпадет, т.к. совпадает суффикс. Поэтому уменьшаем i и j на значение M_j.

3) $M_j == N_i$. Такой же смысл как и в пункте 2), поэтому уменьшаем i и j на значение M_j. Но при этом появляется условие, что если $N_i == i$, то мы фиксируем вхождение.

4) $M_j > N_i$. Эта ситуация означает что есть совпадение суффиксу, но следующий символ обязательно не совпадет (если это не начало паттерна). Поэтому если $N_i == i$, то мы фиксируем вхождение, иначе M_h присваиваем значение $h-j$ и делаем сдвиг по правилу Бойера-Мура.

2 Исходный код

Т.к. алгоритм Апостолико-Джанкарло основывается на алгориме Бойера-Мура, и использует нотации хорошего суффикса и плохого символа, то удобно сразу инициализировать эти нотации. Это позволит нам получать сдвиги за O(1). Поэтому создадим класс $TPatern$.

```
template<class T>
class TPatern{
    private:
        vector<T> pattern_data;
        unordered_map<T, vector<uint64_t>> bad_symbol_pos;
        vector<uint64_t> good_suf_pos;

        void bad_symbols_init();
        void good_suffix_init();

        uint64_t bad_symbol_shift(uint64_t cur_idx, T mismatch_val);
        uint64_t good_suffix_shift(uint64_t cur_idx);
    public:
        vector<uint64_t> N_array;

        TPatern(vector<T> input_array){
            pattern_data = move(input_array);
            bad_symbols_init();
            good_suffix_init();
        }
        uint64_t size(){
            return pattern_data.size();
        }
        T operator [] (uint64_t idx) const{
            return this->pattern_data[idx];
        }
        T& operator [] (const uint64_t idx){
            return this->pattern_data[idx];
        }
        uint64_t get_shift(uint64_t cur_pattern_idx, T mismatch_val);

        void print_bad_symbol_array();
        void print_good_suffix_array();
        ~TPatern(){};
};
```

Функция	Описание
void bad_symbols_init()	Инициализирует правило плохого символа (заполняет словарь unordered_map<T, vector<uint64_t» bad_symbol_pos)
void good_suffix_init	Инициализирует правило хорошего суффикса (заполняет массив vector<uint64_t> N_array и vector<uint64_t> good_suf_pos)
uint64_t bad_symbol_shift(uint64_t cur_idx, T mismatch_val)	Возвращает сдвиг по правилу плохого символа для текущей позиции
uint64_t good_suffix_shift(uint64_t cur_idx)	Возвращает сдвиг по правилу хорошего суффикса для текущей позиции
TPatern(vector<T> input_array)	Вызывает инициализацию эвристик
uint64_t get_shift(uint64_t cur_pattern_idx, T mismatch_val)	Возвращает наибольший сдвиг для текущей позиции

Фнкция vector<T> pattern_parser() отвечает за парсинг образца, а функция vector<TVal<T» text_parser(uint64_t start_num_line, uint64_t start_num_val) за парсинг текста. Эти функции реализованы с использованием шаблона, что делает их универсальными и позволяет производить поиск образца разных типов данных без переписывания кода.

```
template<class T>
vector<T> pattern_parser(){
    string str_input_data;
    getline(cin, str_input_data);
    istringstream input_stream(str_input_data);
    vector<T> v_output_data;
    T cur_val;
    while(input_stream>>cur_val){
        v_output_data.push_back(cur_val);
    }
    return v_output_data;
}

template<class T>
vector<TVal<T>> text_parser(uint64_t start_num_line, uint64_t start_num_val){
    vector<TVal<T>> v_output_data;
    string str_line;
    uint64_t cnt_line = start_num_line;
    while(getline(cin, str_line)){
        istringstream input_stream(str_line);
        T cur_val;
        uint64_t cnt_val = start_num_val;
        while(input_stream>>cur_val){
```

```
            v_output_data.push_back(pair(cur_val, pair(cnt_line, cnt_val)));
            cnt_val++;
        }
        cnt_line++;
    }

    return v_output_data;
}
```

Реализация основного тела алгоримта представлена в в функции *main*.

```
    while(r_border<text_data.size()){
        uint64_t i_t=r_border, i_p=p_size-1;
        while(i_p>0 and text_data[i_t].first == pattern[i_p]){
            if(M_vector[i_t]==uint64_t(-1) or (M_vector[i_t]==0 and M_vector[i_t]==
                pattern.N_array[i_p])){
                if (text_data[i_t].first == pattern[i_p]){
                    i_p--;
                    i_t--;
                } else{
                    M_vector[r_border] = r_border - i_t;
                    break;
                }
            } else if (M_vector[i_t]!=uint64_t(-1)){
                if (M_vector[i_t]<pattern.N_array[i_p]){
                    i_p-=M_vector[i_t];
                    i_t-=M_vector[i_t];
                } else if(M_vector[i_t]==pattern.N_array[i_p]){
                    if (pattern.N_array[i_p]==i_p){
                        M_vector[r_border] = r_border-i_t;
                    }
                    i_p-=M_vector[i_t];
                    i_t-=M_vector[i_t];
                } else{
                    if (pattern.N_array[i_p]==i_p){
                        M_vector[r_border] = r_border-i_t;
                        i_p-=M_vector[i_t];
                        i_t-=M_vector[i_t];
                    } else {
                        M_vector[r_border] = r_border - i_t;
                        break;
                    }
                }
            }
        }
        if (i_p==0 and text_data[i_t].first == pattern[i_p]){
            v_answ.push_back(text_data[i_t].second);
            r_border++;
        } else{
```

```
38              r_border+= pattern.get_shift(i_p, text_data[i_t].first);
39          }
40      }
```

3 Консоль

```
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ g++ -std=c++20 main.cpp -o
main
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ cat <test
11 45 11 45 90
0011 45 011 0045 11 45 90    11
45 11 45 90
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./main <test
1,3
1,8arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$
```

4 Тест производительности

Тест производительности представляет из себя следующее: алгоритм Апостолико-Джанкарло сравнивается с наивным алгоритмом поиска подстроки в строке. В тест производитеельности также включен препроцессинг. Для поиска зависимостей было подготовлен 21 тест, которые условно можно разделить на 3 группы: тестирование зависимости скорости от размера паттерна, от размера текста и от частоты выпадения паттерна в тексте. Ниже представлены пара таких тестов.

```
// Тестирование зависимости скорости от частоты выпадения паттерна в тексте
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/01.t
Probability = 0.1
Pattern size = 7354
Text size = 1001817
Apostolico_Giancarlo algorithm time: 0.016757281 s
Dumm algorithm time:      0.007159800 s
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/05.t
Probability = 0.98
Pattern size = 7354
Text size = 1004459
Apostolico_Giancarlo algorithm time: 0.014491209 s
Dumm algorithm time:      0.007348549 s

// Тестирование зависимости скорости от размера паттерна
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/06.t
Probability = 0.59
Pattern size = 3
Text size = 1000004
Apostolico_Giancarlo algorithm time: 0.053340360 s
Dumm algorithm time:      0.005611640 s
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/07.t
Probability = 0.59
Pattern size = 2252
Text size = 1000839
Apostolico_Giancarlo algorithm time: 0.012560133 s
Dumm algorithm time:      0.006874890 s
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/10.t
Probability = 0.59
Pattern size = 8999
Text size = 1005222
Apostolico_Giancarlo algorithm time: 0.014650538 s
```

```
Dumm algorithm time:      0.006812823 s

// Тестирование зависимости скорости от размера текста
Probability = 0.59
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/11.t
Pattern size = 7354
Text size = 14846
Apostolico_Giancarlo algorithm time: 0.005261117 s
Dumm algorithm time:      0.000103084 s
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/16.t
Probability = 0.59
Pattern size = 7354
Text size = 511085
Apostolico_Giancarlo algorithm time: 0.009006845 s
Dumm algorithm time:      0.003601026 s
arsenii@PC-Larcha14:~/Documents/C_pp_uk/DA/Lab_4$ ./benchmark <tests/21.t
Probability = 0.59
Pattern size = 7354
Text size = 1006971
Apostolico_Giancarlo algorithm time: 0.014268859 s
Dumm algorithm time:      0.007304772 s
```

Если не брать тот факт, что алгоритм Апостолико-Джанкарло проигрывает на всех тестах наивному алгоритму, что очень странно (т.к. срость наивного алгоритма - O(n*m), а теоретическая скорость алгоримт Апостолико-Джанкарло - O(m), т.к. в отличие от алгоритма Бойера-Мура (на основе которого построен алгоримт Апостолико-Джанкарло), который в худшем случае работает за O(n*m), алгоримт Апостолико-Джанкарло использует всю полезною информацию о сдвигах. n - длина паттерна, а m - длина текста),то можно изучить закономерности. Например, что при увеличении вероятности выпадения паттерна, время алгоритма уменьшается, тогда как скорость наивного алгоритма повышается. При увеличении длинны паттерна происходит аналогичный процесс. А вот при увеличении размера текста время выполнения обоих алгоритмом повышается.

У меня есть теория, что такие плохие результаты по сравнению с наивным алгоритмом получены из-за искуственно-составленных неидеальных тестов. Т.к. по условию задачи мы используем алфавит из $2^{32} - 1$ элементов, вероятность случаного выпадения одинаковых чисел в паттерне, длинной от 3 до 10000 из этого диапазона мала, следовательно основные нотации, которые использует алгоритм (правило хорошего суффикса и правило плохого символа) не играют большой роли и не помогают ускорить алгоритм, в то время как их препроцессинг, который, стоит отметить, занимает

const*O(n), требует время. Моей ошибкой в данном случае является игнорирование повышения вероятности выпадения одиноковых символов в паттерне.

5 Выводы

В этой лабораторной работе я познакомился с алгоритмами поиска образца в тексте. Получил много новых знаний об алгоритме Боейра-Мура, в частности об его эвристиках (правилах плохого символа и хорошего суффикса, а также об их сильных реализациях) и об его улучшенном варианте - алгоритме Апостолико-Джанкарло, а также реализовал эти два алгоритма.

Данная лабораторная сложностей у меня не вызвала, кроме тестирования производительности алгоритма, которое сложно назвать удачным и связанно это, как я полагаю, с плохими искуственными данными. В реальной жизни, при работе с алфавитами меньшей мощности алгоримт Апостолико-Джанкарло показывает себя гораздо лучше и работает за O(m).

Список литературы

[1] Гасфилд Дэн
Строки, деревья и последовательности в алгоритмах: Информатика и вычисвычислительная биология / Пер. с англ. И. В. Романовского. — СПб.: Невский Диа- Диалект; БХВ-Петербург, 2003. — 654 с: ил.