МИНОБРНАУКИ РОССИИ
Санкт-Петербургский государственный
электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритм Ахо-Корасик

Студент гр. 7383 _____ Власов Р.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Содержание

Цель работы

Цель работы: познакомиться с алгоритмом поиска подстрок в строке Ахо-Корасик, создать программу, осуществляющую поиск набора подстрок в строке с помощью алгоритма Ахо-Корасик, а также программу, осуществляющую поиск подстроки с масками в строке.

Формулировка задачи: Разработайте программу, решающую задачу точного поиска набора образцов. Первая строка содержит текст (T,$1 \leq |T| \leq 100000$). Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора $P=\{p_1,\ldots,p_n\} 1 \leq |p_i| \leq 75$. Все строки содержат символы из алфавита {A,C,G,T,N}.

Реализация задачи

Программу было решено писать на языке программирования C++. Для реализации поставленной задачи был создан класс `Pattern_Shearch_Tree`.

```
class Pattern_Shearch_Tree
{
    typedef Pattern_Shearch_Tree PST;
    typedef Pattern_Shearch_Tree* PST_ptr;
private:
    std::vector<PST_ptr> next;
    PST_ptr prev;
    PST_ptr suffix;
    PST_ptr finish;
    PST_ptr root;
    PST_ptr self_reference;
    std::queue<PST_ptr> queue;
    std::vector<unsigned int> pattern_numbers;
    char name;

    void destroy();
    PST_ptr find_next(char name_to_go) const;
    PST_ptr find_suffix() const;
    PST_ptr find_finish() const;
    void visit_all(std::queue<PST_ptr> &q);
    void update_all_suffix_links();
    void update_all_finish_links();
public:
    Pattern_Shearch_Tree(char name = 0,  PST_ptr prev = nullptr, PST_ptr head
= nullptr);
    ~Pattern_Shearch_Tree();
    char get_name() const;
    unsigned int is_final() const;
    std::vector<unsigned int> get_final_marks();
    void add_pattern(std::string& pattern, unsigned int num);
  std::vector<std::pair<size_t, unsigned int>> search(std::string& str);
}
```

Функция `add_pattern(std::string& pattern, unsigned int num)` добавляет шаблон в дерево, с помощью которого в последствии будет осущетсвляться поиск. Функции `update_all_suffix_links()` и `update_all_finish_links()` расставляют суффиксные и финишные ссылки. Функция `search(std::string& str)` осуществляет поиск вхождений шаблонов в строке и возвращает пары: конец вхождения шаблона и его номер.

Исходный код программы представлен в приложении Б.

Исследование алгоритма

Сложность алгоритма Ахо-Корасик O(n|A|+H+k), где H – длина текста, в котором производится поиск, n – общая длина всех слов в словаре, |A| – размер алфавита, k – общая длина всех совпадений.

Тестирование

1. Процесс тестирования

Программа собрана в операционной системе Ubuntu 18.04.2 LTS bionic компилятором g++ version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04). В других ОС и компиляторах тестирование не проводилось.

2. Результаты тестирования

В результате тестирования программы ошибок выявлено не было. Тестовые случаи представлены в приложении А.

Вывод

В ходе выполнения данной работы был изучен метод поиска набора подстрок в строке с помощью алгоритма Ахо-Корасик. Была написана программа, применяющая алгоритм Ахо-Корасик для поиска набора подстрок в строке, а также осуществляющая поиск поиск подстроки с масками в строке. Сложность алгоритма Ахо-Корасик составляет O(n|A|+H+k).

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Результат
CCCA 1 CC	1 1 2 1
ACT A? ?	1

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <queue>

class Pattern_Shearch_Tree
{
    typedef Pattern_Shearch_Tree PST;
    typedef Pattern_Shearch_Tree* PST_ptr;
private:
    std::vector<PST_ptr> next;
    PST_ptr prev;
    PST_ptr suffix;
    PST_ptr finish;
    PST_ptr root;
    PST_ptr self_reference;
    std::queue<PST_ptr> queue;
    std::vector<unsigned int> pattern_numbers;
    //unsigned int pattern_number; //for pattern end
    char name;

    void destroy()
    {
        for (auto el : next)
        {
            el->destroy();
            delete el;
        }
    }
    PST_ptr find_next(char name_to_go) const
    {
        for (auto el : next)
            if (el->name == name_to_go)
                return el;
        return nullptr;
    }
    PST_ptr find_suffix() const
    {
        if (self_reference == root || prev == root)
        {
            return root;
        }
        else
        {
            PST_ptr link = prev->suffix;
            link = link->find_next(name);
            if (link)
                return link;
```

```
        else
        {
            link = prev->suffix;
            while(link != root)
            {
                link = link->suffix;
                PST_ptr tmp = link->find_next(name);
                if (tmp)
                    return tmp;
            }
        }
        return root;
    }
}
PST_ptr find_finish() const
{
    if (self_reference == root)
    {
        return nullptr;
    }
    else
    {
        PST_ptr link = suffix;
        if (link == root)
        {
            link = prev->suffix;
            while(link != root)
            {
                PST_ptr tmp;
                if ((tmp = link->find_next(name)))
                    if (tmp->is_final())
                        return tmp;
                link = link->suffix;

            }
            return nullptr;
        }
        else
        {
            if (link->is_final() != 0)
                return link;
            else
            {
                return link->finish;
            }
        }
    }
}
void visit_all(std::queue<PST_ptr> &q)
{
    PST_ptr link;
```

```
        std::queue<PST_ptr> tmp;
        tmp.push(self_reference);
        while(!tmp.empty())
        {
            link = tmp.front();
            tmp.pop();
            q.push(link);
            for (auto el : link->next)
            {
                tmp.push(el);
            }
        }

    }
    void update_all_suffix_links()
    {
        std::queue<PST_ptr> q = queue;
        while(!q.empty())
        {
            PST_ptr link = q.front();
            q.pop();
            link->suffix = link->find_suffix();
        }
    }
    void update_all_finish_links()
    {
        std::queue<PST_ptr> q = queue;
        while(!q.empty())
        {
            PST_ptr link = q.front();
            q.pop();
            link->finish = link->find_finish();
        }
    }

public:
    Pattern_Shearch_Tree(char name = 0,  PST_ptr prev = nullptr, PST_ptr head
= nullptr) : name(name)
    {
        this->prev = prev;
        suffix = head;
        finish = nullptr;
        self_reference = this;
        if (head)
            root = head;
        else
            root = self_reference;
        //pattern_number = 0;
    }
    ~Pattern_Shearch_Tree()
    {
```

```
    if (this == root)
        destroy();
}
char get_name() const
{
    return name;
}
unsigned int is_final() const
{
    return pattern_numbers.size();
}
std::vector<unsigned int> get_final_marks()
{
    return pattern_numbers;
}
void add_pattern(std::string& pattern, unsigned int num)
{
    if (pattern.size() == 0)
    {
        pattern_numbers.push_back(num);
        return;
    }
    char ch = pattern.front();
    pattern.erase(pattern.begin());
    for (auto el : next)
    {
        if (el->get_name() == ch)
        {
            el->add_pattern(pattern, num);
            return;
        }
    }
    next.push_back(new PST(ch, self_reference, root));
    next.back()->add_pattern(pattern, num);
    return;
}
std::vector<std::pair<size_t, unsigned int>> search(std::string& str)
{
    visit_all(queue);
    update_all_suffix_links();
    update_all_finish_links();
    std::vector<std::pair<size_t, unsigned int>> answer;
    size_t position = 0;
    PST_ptr state = root;
    PST_ptr tmp;
    while (position != str.size())
    {
        tmp = state->find_next(str[position]);
        if (tmp)
        {
            state = tmp;
```

```
            }
            else
            {
                state = state->suffix;
                while (state->is_final() && !state->next.size())
                    state = state->suffix;
                tmp = state->find_next(str[position]);
                while ((tmp == nullptr) && (state != root))
                {
                    state = state->suffix;
                    tmp = state->find_next(str[position]);
                }
                if (tmp)
                    state = tmp;
                else
                {
                    state = root;
                    position++;
                    continue;
                }
            }
            if (state->is_final())
            {
                for (auto el : state->pattern_numbers)
                {
                    answer.push_back(std::make_pair(position, el));
                }

            }
            tmp = state->finish;
            while (tmp)
            {
                for (auto el : tmp->pattern_numbers)
                {
                    answer.push_back(std::make_pair(position, el));
                }
                //answer.push_back(std::make_pair(position, tmp-
>is_final()));
                tmp = tmp->finish;
            }
            position++;
        }
        return answer;
    }
};
bool mycomp(const std::pair<size_t, unsigned int>& a, const std::pair<size_t,
unsigned int>& b)
{
    if (a.first == b.first)
        return a.second < b.second;
    return a.first < b.first;
```

```
}

int main()
{
    Pattern_Shearch_Tree tree;
    std::string str;
    std::string pattern;
    std::vector<size_t> pattern_sizes;
    std::cin >> str;
    /*
    std::cin >> n;
    for (unsigned int i = 1; i <= n; i++)
    {
        std::cin >> pattern;
        pattern_sizes.push_back(pattern.size());
        tree.add_pattern(pattern, i);
    }

    */
    std::cin >> pattern;
    char universal_symb;
    std::cin >> universal_symb;

    std::vector<size_t> C;
    std::vector<size_t> L;
    for (size_t i = 0; i <= pattern.size(); i++)
    {
        if ((i == pattern.size() || pattern[i] == universal_symb) &&
pattern[i-1] != universal_symb)
            L.push_back(i + 1);
    }
    C.push_back(L[0] == 1 ? 0 : 1);
    for (size_t i = 0; i < L.size() - 1; i++)
    {
        size_t t = L[i];
        while (pattern[t] == universal_symb)
        {
            t++;
        }
        C.push_back(t + 1);
    }

    for (unsigned int i = 1; i <= C.size(); i++)
    {
        std::string tmp;
        if (C[i-1] == 0)
        {
            pattern_sizes.push_back(0);
            continue;
        }
        for (size_t j = C[i-1]-1; j < L[i-1]-1; j++)
```

```
            tmp.push_back(pattern[j]);

        pattern_sizes.push_back(tmp.size());
        tree.add_pattern(tmp, i);
    }

    std::vector<std::pair<size_t, unsigned int>> ans = tree.search(str);
    for (size_t i = 0; i < ans.size(); i++)
        ans[i].first = ans[i].first - pattern_sizes[ans[i].second - 1] + 2;

    size_t *arr = new size_t[str.size()];
    for (size_t i = 0; i < str.size(); i++)
        arr[i] = C[0] == 0 ? 1 : 0;
    for (auto el : ans)
    {
        //std::cout << el.first - C[el.second - 1] << ' ' << str.size() <<
std::endl;
        //assert(el.first - C[el.second - 1] < str.size());
        //assert(el.first - C[el.second - 1] >= 0);

        if (el.first - C[el.second - 1] < str.size())
            arr[el.first - C[el.second - 1]] += 1;
    }
    for (size_t i = 0; i < str.size(); i++)
        if (i + pattern.size() <= str.size() && arr[i] == L.size())
            std::cout << i+1 << std::endl;


    /*
    std::sort(ans.begin(), ans.end(), mycomp);
    for (auto el : ans)
    {
        std::cout << el.first << ' ' << el.second << std::endl;
    }
    */
    delete[] arr;
    return 0;
}
```