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-Карасик

Студент гр. 7304 _____ Абдульманов Э.М

Преподаватель _____ Филатов А.Ю

г. Санкт-Петербург
2019

Цель работы

Освоить алгоритм Ахо – Карасик, который реализует поиск множества подстрок из словаря в данной строке.

Задачи

1. Разработать программу, решающую задачу точного поиска набора образцов.
2. Используя реализацию точного множественного поиска, решить задачу точного поиска для одного образца с *джокером*.

Теоретические сведения

Бор (trie)— структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах. Строки получаются последовательной записью всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной. Размер бора линейно зависит от суммы длин всех строк, а поиск в бору занимает время, пропорциональное длине образца.

Детерминированный конечный автомат — набор из пяти элементов $\langle \Sigma, Q, s \in Q, T \subset Q, \delta: Q \times \Sigma \to Q \rangle \langle \Sigma, Q, s \in Q, T \subset Q, \delta: Q \times \Sigma \to Q \rangle$, где Σ — алфавит (англ. alphabet), Q — множество состояний (англ. finite set of states), s — начальное (стартовое) состояние (англ. start state), T — множество допускающих состояний (англ. set of accept states), δ — функция переходов (англ. transition function).

Пусть $x\alpha$ обозначает произвольную строку, где x — её первый символ, а α — оставшаяся подстрока (возможно пустая). Если для внутренней вершины v с путевой меткой $x\alpha$ существует другая вершина $s(v)$ с путевой меткой α, то ссылка из v в $s(v)$ называется суффиксной ссылкой (англ. suffix link).

Ход работы

1. Был реализован алгоритм Ахо - Карасика, который решает задачу точного поиска набора образцов. Алгоритм был реализован следующим образом:

    a. Была реализована структура вершины в боре. Где flagEnd – это флаг, который показывает, вершина удовлетворяет шаблону или нет, и если удовлетворяет, то какому шаблону именно. Map<char,int> next – это все ребра, по которым можно пройти из данной вершины в другие. Map<char,int> auto_move – это все возможно переходы из одного состояния в другое по какому-то символ. Parent - это предок данной вершины. Symbol – это по какому символу происходит переход от предка к данной вершине. Suffix_link – суффиксальная ссылка.

```
struct FlagEnd{
    bool flag;
    int count_pattern;
};

struct Vertex{
    FlagEnd flagEnd;
    map<char,int> next;
    map<char,int> auto_move;
    int suffix_link;
    int parent;
    char symbol;

    Vertex(int parent,char symbol) {
        this->flagEnd.flag=false;
        this->flagEnd.count_pattern=0
        this->suffix_link=-1;
        this->parent=parent;
        this->symbol=symbol;
    }
};
```

    b. Была реализована функция, которая добавляет строку в бор. Сначала встаем на нулевую вершину, которая является корнем. Потом идем по символам данной строки, если по данному символу можно перейти, то просто переходит, если нельзя. То создаем вершину, создаем переход и переходим.

```
void addPattern(const string &pattern){
    int cur_ver=0;//стоим в корне
    this->pattern.push_back(pattern);
    for(char symbol:pattern){
        if(borh[cur_ver].next.find(symbol)==borh[cur_ver].next.end()){
            borh.push_back(Vertex(cur_ver,symbol));
            borh[cur_ver].next[symbol]=count++;
        }
        cur_ver=borh[cur_ver].next[symbol];
    }
    borh[cur_ver].flagEnd.flag=true;
    borh[cur_ver].flagEnd.count_pattern=this->pattern.size();
}
```

c. Были реализованы два метода, которые строят по данному бору конечный детерминированный автомат, он строится по ходу работы алгоритма. Первый метод – это получение суффиксальной ссылки для данной вершины. Если суффиксальная ссылка еще не определена (= - 1), то мы сначала проверяем не корень ли это или не предок ли корень, если да, то суффиксальная ссылка будет равна 0, иначе суффиксальная ссылка равна = переходим на предка, далее переходим по суффиксальной ссылки предка и пытаемся пройти по данному символу, если нельзя, то повторяем. Все это происходит во втором методе get_auto_move. Который по данной вершине и поступившему символу выдает следующее состояние.

```
int get_suffix_link(int vertex){
    if(borh[vertex].suffix_link==-1)
        if(vertex==0||borh[vertex].parent==0)
            borh[vertex].suffix_link=0;
        else
            borh[vertex].suffix_link=get_auto_move(get_suffix_link(borh[vertex].parent),borh[vertex].symbol);
    return borh[vertex].suffix_link;
}

int get_auto_move(int vertex,char symbol){
    if(borh[vertex].auto_move.find(symbol)==borh[vertex].auto_move.end())
        if(borh[vertex].next.find(symbol)!=borh[vertex].next.end())
            borh[vertex].auto_move[symbol]=borh[vertex].next[symbol];
        else
            borh[vertex].auto_move[symbol]= ((vertex==0) ? 0:get_auto_move(get_suffix_link(vertex),symbol));
    return borh[vertex].auto_move[symbol];
}
```

d. Был реализован метод поиска шаблонов в строке. Для каждого символа, вызываем метод get_auto_move от данной вершины. Далее начинаем идти по суффиксальным ссылкам, пока они не приведут корень. Если встречается вершина, которая показывает, что строка в шаблоне кончилась, то выводим ответ, иначе продолжаем идти к корню. Затем переходим к следующему символу и повторяем процесс.

```
void searchString(const string& str){
    int cur_ver=0;
    for(int i=0;i<str.size();i++){
        cur_ver=get_auto_move(cur_ver,str[i]);
        for(int ver=cur_ver;ver!=0;ver=get_suffix_link(ver)){
            if(borh[ver].flagEnd.flag){
                int count_pattern=borh[ver].flagEnd.count_pattern;
                int position=i-pattern[count_pattern-1].size()+2;
                cout<<position<<" "<<count_pattern<<endl;
            }
        }
    }
}
```

2. Была решена задача, которая используя реализацию точного множественного поиска, решит задачу точного поиска для одного образца с *джокером*.

a. Идея заключается в следующем:

1. C — вектор длины T, инициализированный нулями.
2. $\mathbb{P} = \{P_1, P_2, \ldots, P_k\}$ — набор максимальных подстрок P без джокеров. $l_1, l_2, \ldots, l_k$ — начальные позиции этих подстрок в P. Для $P = ab??c?ab??$ $\mathbb{P} = \{ab, c, ab\}$ и $l_1 = 1$, $l_2 = 5$ и $l_3 = 7$
3. Алгоритмом Ахо-Корасик найти все вхождения P_i в T. Для каждого вхождения P_i в j-й позиции текста увеличить счётчик $C[j - l_i + 1]$ на единицу.
4. Вхождение P в T, начинающиеся в позиции p, имеется в том и только том случае, если $C(p) = k$.
5. Время поиска $O(km)$ из-за использования массива C, если k ограничено константой, не зависящей от $|P|$, то время поиска — линейно.

b. Для этого была изменена структура FlagEnd. Теперь одна вершина, может показывать, какие одинаковые шаблоны в ней кончаются.

```
struct FlagEnd{
    bool flag;
    vector<int> count_pattern;
};

struct Vertex{
    FlagEnd flagEnd;
    map<char,int> next;
    map<char,int> auto_move;
    int suffix_link;
    int parent;
    char symbol;

    Vertex(int parent,char symbol) {
        this->flagEnd.flag=false;
        this->suffix_link=-1;
        this->parent=parent;
        this->symbol=symbol;
    }
};
```

c. Была изменена реализация поиска. Создается вектор C, о котором говорилось в пункте 2a. Начинаем идти по строке. Все отличие заключается в том момента, когда текущая вершина является конечной. Начинаем проходить по всем шаблонам, которые оканчиваются в этой вершине и делаем следующие вычисления. Position (позиция шаблона в данной строке) – patterns[count_pattern-1].second (на какой позиции находится данный шаблон в строке с джокерами),если это >=0,то если это первый шаблон или если не первый, но C[Position – patterns[count_pattern-1].second]!=0 делаем следующие C[Position

– patterns[count_pattern-1].second]++; В самом конце, мы идем по массиву C. И если C[i] = числу шаблонов, то это i- позиция является началом строки с джокерами.

```
void searchString(const string& str){
    vector<int>C(str.size(),0);
    int cur_ver=0;
    for(int i=0;i<str.size();i++){
        cur_ver=get_auto_move(cur_ver,str[i]);
        for(int ver=cur_ver;ver!=0;ver=get_suffix_link(ver)){
            if(borh[ver].flagEnd.flag){
                for(int index=0;index<borh[ver].flagEnd.count_pattern.size();index++) {
                    int count_pattern = borh[ver].flagEnd.count_pattern[index];
                    int position = i - patterns[count_pattern - 1].first.size() + 1;
                    if(position - patterns[count_pattern - 1].second>=0) {
                        if (count_pattern == 1) {
                            C[position - patterns[count_pattern - 1].second]++;
                        } else if (C[position - patterns[count_pattern - 1].second] != 0) {
                            C[position - patterns[count_pattern - 1].second]++;
                        }
                    }
                }
            }
        }
    }
    for(int i=0;i<C.size();i++){
        if(C[i]==patterns.size()&&((i+strWithJoker.size()<=C.size()))){
            cout<<i+1<<endl;
        }
    }

}
```

Примеры работы программы

1.

```
CCCA
1
CC
1 1
2 1
```

2.

```
ACT
A$
$
1
```

Вывод

В ходе данной лабораторной работы был реализован алгоритм Ахо-Карасика на языке c++. Данный алгоритм делает точный поиск набора образцов в строке. В нем используются такие понятия, как бор, конечный детерминированный автомат, суффиксальные ссылки. Построение бора происходит за время O(n*logK), так как map<char,int> next, а не массив. Поиск происходит за время O(n*Len),где Len – это длина строки. Можно сделать оптимизацию, введя понятие хороших суффиксальных ссылок, тогда поиск будет занимать время O(Len).