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. 8383 _____ Федоров И.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать алгоритм Ахо-Корасика для поиска всех вхождений всех образцов в заданную строку. Реализовать вариант алгоритма для поиска вхождений одного образца с "джокером".

Задание.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, $1 \le |T| \le 10000$).

Вторая - число n ($1 \le n \le 3000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \ldots, p_n\}$ $1 \le |p_i| \le 75$.

Все строки содержат символы из алфавита $\{A, C, G, T, N\}$.

Выход:

Все вхождение образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p.

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1).

Строки выхода должна быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

NTAG

3

TAGT

TAG

1

Sample Output:

2 2

2 3

Вариант №2: Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*. В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу *P* необходимо найти все вхождения *P* в текст *T*.

Например, образец *ab??c?* с джокером *?* встречается дважды в тексте *xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в *T*. Каждый джокер соответствует одному символу, а не подстроке неопределенной длины. В шаблон входит хотя бы один символ не джокер, т.к. шаблоны вида *???* недопустимы. Все строки содержат символы из алфавита $\{A,C,G,T,N\}$.

Вход:

Текст (*T*, $1 \le |T| \le 10000$).

Шаблон $(P, 1 \le |P| \le 40)$.

Символ джокера.

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Номера должны выводиться в порядке возрастания.

Sample Input:

ACTANCA

A$$A$

$

Sample Output:

1

Описание алгоритма для поиска вхождений образцов.

На вход алгоритм имеет набор строк-образцов и исходный текст.

В начале алгоритма строится бор для образцов. Изначально есть лишь одна вершина (корень, соответствует пустой строке). Далее добавляя поочередно образцы происходит обработка: начиная в корне, двигаемся по бору, выбирая ребро, соответствующее считанному символу строки. Если такого ребра нет, то создаем его. В случае, если при добавлении строки не было создано ни одного ребра, то есть процесс остановился на внутренней вершине, а не на листе, то такая вершина помечается как терминальная (по сути каждой вершине соответствует своя строка, но помечаются лишь те вершины, которые соответствуют образцам).

После построения бора происходит посимвольная обработка текста: для очередного символа либо выполняем обычный переход по дереву, если у текущей вершин есть сын с соответствующим ребром, либо переходим в корень если текущая вершина является корнем или сыном корня, либо переходим по суффиксной ссылке (функция перехода определена через суффиксную ссылку, а суффиксная ссылка через функцию перехода, поэтому поиск перехода реализован "ленивым" образом через взаимную рекурсию, автомат строится по ходу выполнения, а не заранее).

Оказавшись после перехода в новом состоянии, проверяем по "сжатым" суффиксным ссылкам строки, которые были обнаружены, если новое состояние является терминальным, то соответствующую ему строку тоже отмечаем. "Сжатая" суффиксная ссылка либо переходит по суффиксной ссылке, если она ведет к терминальной вершине или в корень, либо у полученной вершины вызывает рекурсивно переход по "сжатой" суффиксной ссылке.

Например: образцы {abc, c, cab}. Если мы находимся в терминальной вершине 3, то мы отмечаем соответствующий образец abc, затем, запускаем функцию "хождения" по "сжатым" суффиксным ссылкам: сначала по суффиксной ссылке произойдет переход в вершину 5, а из нее аналогично в терминальную вершину 6. Таким образом не будет пропущен образец *c*. После

этого, вершина 3 уже будет хранить "сжатую" суффиксную ссылку сразу на вершину 6, поэтому повторное хождение не требуется.

После окончания работы алгоритма выполняется индивидуализация: выводится число вершин бора, а после, сравнивая последние позиции вхождение образцов в текст, выводятся пересекающиеся.

Сложность построения бора по времени будет $O(m)$, m – суммарная длина всех образцов, т.к. алгоритм просто последовательно обрабатываем символы образцов. Обработка текст происходит за $O(N + Ans)$, где N - длина текста, Ans - число всех вхождений, т.к. для каждой вершины за $O(1)$ можно найти следующую в суффиксном пути терминальную вершину (то есть следующее совпадение). Поэтому суммарная сложность по времени будет $O(m + N + Ans)$.

Сложность алгоритма по памяти будет $O(m \cdot k), k$ – количество символов алфавита, т.к. алгоритму нужно хранить бор, максимально в нем может быть m вершин, и для каждой хранится информация переходов по каждому символу из алфавита.

Описание для поиска вхождения образца с джокером.

После считывания образца и символа джокера, происходит обработка: образец разбивается на максимальные подстроки без джокера, позиции подстрок запоминаются в массиве. Таким образом будет полученный обычный

список образцов $\{q_1,...,q_z\}$ и список стартовых позиций $\{l_1,...,l_z\}$. После этого по тому же алгоритму происходит поиск всех вхождений подстрок в тексте. Когда находим очередную подстроку на позиции *j*, то увеличиваем на единицу $C[j-l_i+1]$. После этого последовательно проходимся по массиву *C* и, каждое *i*, для которого $C[i]=z$, является стартовой позицией появления образца с джокером в тексте.

Сложность по времени алгоритма поиска вхождений подстрок будет такой же, то есть $O(m+N+Ans)$. Кроме этого, в конце происходит линейны проход по массиву *C* за время $O(N)$. Таким образом суммарная сложность $O(m+N+Ans+N)$.

Сложность по памяти будет $O(m\cdot k+N)$, т.к. помимо бора нужно хранить массив *C*.

Описание функций и структур данных

```
struct Node{
  int neighbours[Alphabet_length]; // соседние вершины
  int movePath[Alphabet_length];   // массив переходов
  int parrent;                     // вершина-предок
  int patternNumber;               // номер строки-образца
  int suffLink;                    // суффиксная ссылка
  int upSuffLink;                  // "сжатая" суффиксная ссылка
  char charToParrent;              // символ ведущий к предку
  bool terminal;                   // является ли терминальной
};                                 // (совпадает со строкой)
```

Структура `Node` - структура для хранения вершины бора. Хранит в себе следующие поля:

- `neighbours` - массив смежных вершин, размер равен количеству символов в алфавите.
- `movePath` - массив переходов в автомате.
- `parrent` - вершина-предок.
- `charToParrent` - символ, ведущий от родителя к текущей вершине.

- `suffLink` - суффиксная ссылка.
- `upSuffLink` - "сжатая" суффиксная ссылка.
- `terminal` - признак того, является ли вершина терминальной.
- `patternNumber` - номер строки-образца, соответствующий данной вершине.

```
using Trie = std::vector<Node>;     //хранение бора
```

`Trie` - хранит в себе бор как вектор структур-вершин.

```
using Patterns = std::vector<std::string>;  //вектор образцов
```

`Patterns` - хранит в себе список строк-образцов.

```
using Output = std::vector<std::pair<int,int>>; //вывод позиция - образец
```

- `Node makeNode(int parrent, char transfer);`

Функция `makeNode()` создает новую вершину, суффиксная ссылка изначально заполняется -1, т.к. еще не вычислена, аналогична для массива переходов автомата. Принимает на вход вершину-родитель и символ перехода.

- `void initTrie(Trie& trie);`

  Функция `initTrie()` инициализирует бор, создавая корень.

- `void addString(std::string& str, Trie& trie, int count_patterns);`

Функция `addString()` добавляет строку в бор, посимвольно считывая ее и если нужно, добавляя нужное ребро, начиная проходить от корня. Получает на вход ссылку на строку, ссылку на бор и номер строки-образец. Номер будет записан в терминальную вершину.

- `int getSuffLink(int vert, Trie& trie);`

Функция `getSuffLink()` для вычисления суффиксной ссылки. Если текущая вершина `vert` является корнем, или сыном корня, то функция вернет 0 (ссылку на корень), т.к. максимальный суффикс у такой вершин нулевой. Иначе, функция вернет результат работы рекурсивного вызова функции вычисления перехода для суффиксной ссылки родителя и символа перехода.

- `int getLink(int vert, int index, Trie& trie);`

Функция `getLink()` для вычисления перехода по символу. Если есть обычный переход, то функция вернет сына текущей вершины, если она

является корнем, то функция вернет соответственно корень. Иначе функция вернет результат рекурсивного вызова функции перехода для суффиксной ссылки текущей вершины и символа перехода.

- `int getUpSuffLink(int vert, Trie& trie);`

Функция `getUpSuffLink()` для вычисления сжатой суффиксной ссылки. Возвращает ближайшую терминальную вершину, до которой можно дойти по суффиксным ссылкам.

- `void checkUpLink(int vert, int index, Trie& trie, Output& output, Patterns& pattern);`

Функция `checkUpLink()` функция "хождения" по "сжатым" суффиксным ссылкам. Проходит по ним до тех пор, пока не дойдет до корня, добавляя встреченные строки-образцы в `output`.

- `void processText(std::string& text, Trie& trie, Patterns& pattern, Output& output);`

Функция `processText()` функция для обработки текста и поиска в нем образцов. По очереди просматривает символы текста. Для очередного символа с помощью функции перехода перемещается в новое состояние. Оказавшись в новом состоянии, с помощью функции `checkUpLink()` проходит по "сжатым" суффиксным ссылкам, отмечая встреченные образцы, если текущая вершина тоже терминальная, то соответствующий ему образец тоже отмечается.

- `void splitString(std::string& str, Patterns& patterns, std::vector<int>& positions, char joker);`

Функция `splitString()` используется для вычисления вхождений образца с джокером, разбивает переданную строку на максимальные подстроки без джокера, которые записывает в `patterns`.

Тестирование.

Таблица 1 - тестирование поиска вхождений образцов

Входные данные	Результат

NNTAGATNTAG 5 TAGAT AGAT GAT T TAG	3 1 3 4 3 5 4 2 5 3 7 4 9 4 9 5
ACAGAG 4 A CAGA CA AG	1 1 2 2 2 3 3 1 3 4 5 1 5 4
NTAG 3 TAGT TAG T	2 2 2 3
N 1 N	1 1
ACGGTNAAGGCNTGTNC 9 ACGTN GAA GTNCA GTNC GTN	4 5 4 6 5 7 6 8 13 7 14 4 14 5

GT T NAAGG CNTGG	14 6 15 7

Таблица 2 - тестирование поиска вхождений образца с джокером

Входные данные	Результат
ACTANCA A$$A$ $	1
TCA A$ $	Вхождений нет.
ACACGGG ACXXG X	1 3
ACAACACCCACCACA ACAXXCAXXACA X	4
CABAABBABABCA AB$A $	2 3 4 5 6 7 8 10

Пример работы с подробными выводами представлен на рис 1-2.:

```
ACAGAG
4
A
CAGA
CA
AG

------------Fill trie start--------------
Add pattern: A
        create node for transition: 0--A->1
Add pattern: CAGA
        create node for transition: 0--C->2
        create node for transition: 2--A->3
        create node for transition: 3--G->4
        create node for transition: 4--A->5
Add pattern: CA
Add pattern: AG
        create node for transition: 1--G->6
--------------Fill trie end----------------
------------Proccesing Text star-----------
        Node 1 is terminal, at pos 1 find: A
        Node 3 is terminal, at pos 2 find: CA
        3--UpLink-->1
        Node 1 is terminal, at pos 3 find: A
        Node 6 is terminal, at pos 3 find: AG
        Node 5 is terminal, at pos 2 find: CAGA
        5--UpLink-->1
        Node 1 is terminal, at pos 5 find: A
        Node 6 is terminal, at pos 5 find: AG
------------Proccesing Text end------------
1 1
2 2
2 3
3 1
3 4
5 1
5 4
-------------------------------------
```

Рисунок 1

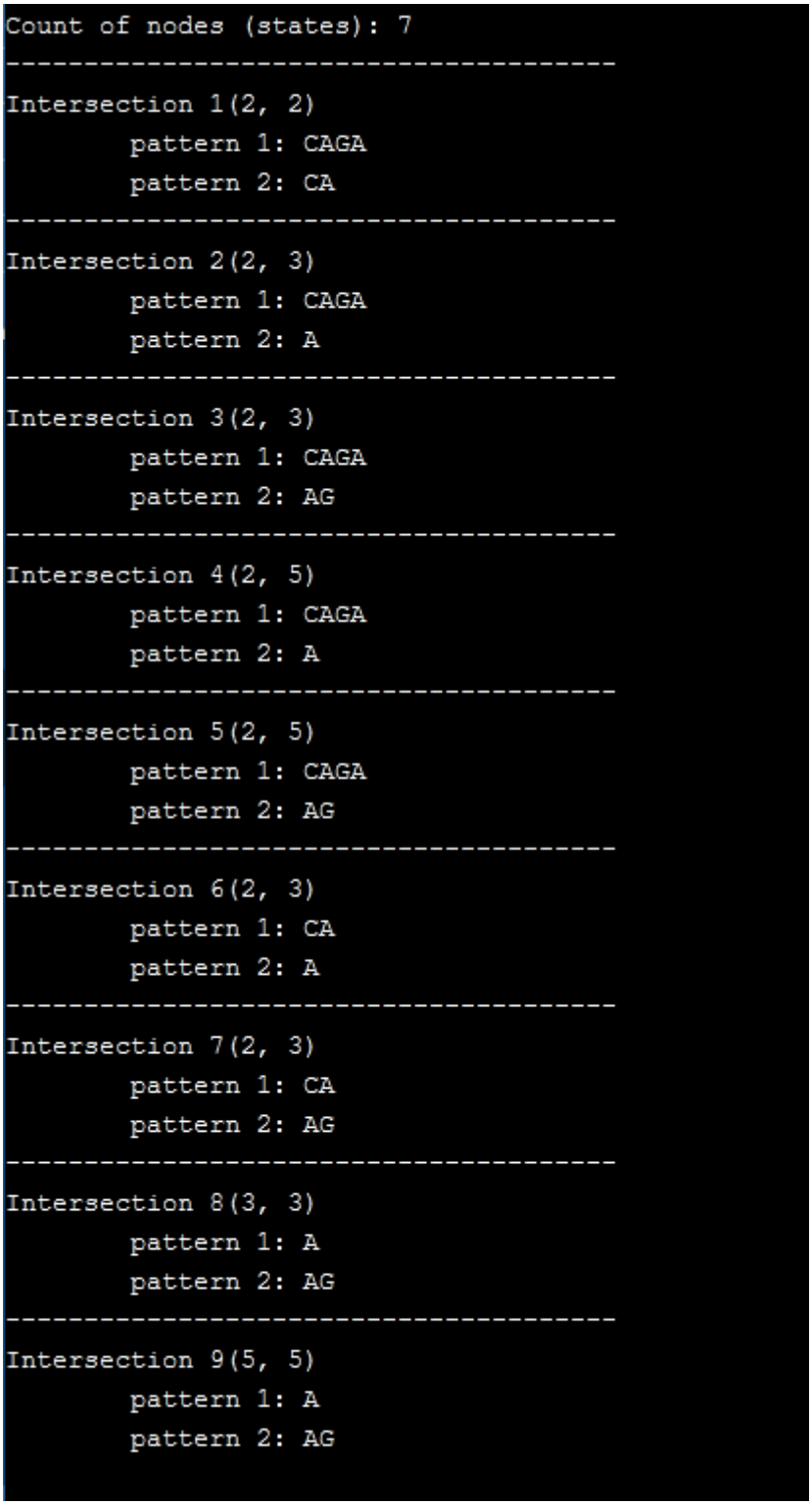

Рисунок 1

Выводы.

В ходе выполнения лабораторной работы был изучен и реализован алгоритм Ахо-Корасика для поиска вхождений всех строк-образцов, а также версия, для поиска образца с джокером.

Приложение

Код поиск образцов

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

struct Node;
using Trie = std::vector<Node>;                          //хранение бора
using ALphabet = std::map<char, int>;                    //алфавит, по
условию состоит из {A,C,G,T,N}
using Patterns = std::vector<std::string>;               //вектор образцов
using Output = std::vector<std::pair<int,int>>;          //вывод позиция -
образец

const int Alphabet_length = 5;
ALphabet trie_alphabet = { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N',
4} };

int getIndex(char symb){
    return trie_alphabet[symb];
}

bool Compare(std::pair<int, int> a, std::pair<int, int> b)
{                                                        //компаратор
для вывода результата
    if (a.first == b.first)
        return a.second < b.second;
    else
        return a.first < b.first;
}


// структура вершины
struct Node{
  int neighbours[Alphabet_length];      // соседние вершины
  int movePath[Alphabet_length];        // массив переходов
  int parrent;                          // вершина-предок
  int patternNumber;                    // номер строки-образца
  int suffLink;                         // суффиксная ссылка
  int upSuffLink;                       // "сжатая" суффиксная ссылка
  char charToParrent;                   // символ ведущий к предку
  bool terminal;                        // является ли терминальной
};                                      // (совпадает со строкой)
```

```
Node makeNode(int parrent, char transfer){                    // создание
новой вершины
    Node newNode;
    memset(newNode.neighbours, 255, sizeof(newNode.neighbours));
    memset(newNode.movePath, 255, sizeof(newNode.neighbours));
    newNode.suffLink = newNode.upSuffLink = -1;               //
изначально нет ссылки
    newNode.parrent = parrent;
    newNode.charToParrent = transfer;
    newNode.terminal = false;
    newNode.patternNumber = -1;
    return newNode;
}



void initTrie(Trie& trie){
    // создаем бор, изначально только корень
    trie.push_back(makeNode(0, '#'));
}


void addString(std::string& str, Trie& trie, int count_patterns){
    int index = 0;
    std::cout << "Add pattern: " << str << std::endl;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1){        // если нет ребра
по символу
            trie.push_back(makeNode(index, str[i]));
            trie[index].neighbours[curr] = trie.size() - 1;
            std::cout << "\tcreate node for transition: " << index << "--
" << str[i] << "->" << trie.size()-1 << std::endl;
        }
         index = trie[index].neighbours[curr];
    }
    trie[index].terminal = true;
    trie[index].patternNumber = count_patterns;
}


bool findString(std::string& str, Trie& trie){       //функция поиска
строки в боре
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1)
            return false;
        index = trie[index].neighbours[curr];
    }
    return true;
}
```

```
int getLink(int vert, int index, Trie& trie);

// Функция для вычисления суффиксной ссылки
int getSuffLink(int vert, Trie& trie){
    if (trie[vert].suffLink == -1)                    // еще не искали
       if (vert == 0 || trie[vert].parrent == 0)      // корень или
родитель корень
          trie[vert].suffLink = 0;                    // для корня
ссылка в корень
       else
           trie[vert].suffLink = getLink(getSuffLink(trie[vert].parrent,
trie), getIndex(trie[vert].charToParrent), trie);
    return trie[vert].suffLink;
}

// Функция для вычисления перехода
int getLink(int vert, int index, Trie& trie){
   if (trie[vert].movePath[index] == -1)
// если переход по данному
      if (trie[vert].neighbours[index] != -1)
// символу ещ не вычислен
         trie[vert].movePath[index] = trie[vert].neighbours[index];
      else if (vert == 0)
          trie[vert].movePath[index] = 0;
      else
          trie[vert].movePath[index] = getLink(getSuffLink(vert, trie),
index, trie);
   return trie[vert].movePath[index];
}


// Функция для вычисления сжатой суффиксной ссылки
int getUpSuffLink(int vert, Trie& trie){
   if (trie[vert].suffLink == -1){             // если сжатая суффиксная
ссылка ещ не вычислена
      int tmp = getSuffLink(vert, trie);
      if (trie[tmp].terminal)                  // если ссылка на
терминальную, то ок
          trie[vert].upSuffLink = tmp;
      else if (tmp == 0)                       // на корень = 0
          trie[vert].upSuffLink = 0;
      else
          trie[vert].upSuffLink = getUpSuffLink(tmp, trie);   //поиск
ближайшей
   }
   return trie[vert].upSuffLink;
}


//проверка сжатых суффиксных ссылок
```

```
void checkUpLink(int vert, int index, Trie& trie, Output& output,
Patterns& pattern){
    int n = 0; int prev;
    for (int i = vert; i != 0; i = getUpSuffLink(i, trie)){
        if (trie[i].terminal){ // если является терминальной, то нашли
соответствующий образец
            if (n) std::cout << "\t" << prev << "--UpLink-->" << i <<
std::endl;
            std::cout << "\tNode " << i << " is terminal, at pos " <<
index-pattern[trie[i].patternNumber].length()+1 << " find: " <<
pattern[trie[i].patternNumber] << std::endl;
            output.push_back(std::pair<int,int> ((index-
pattern[trie[i].patternNumber].length()+1), trie[i].patternNumber+1));
            n++; prev = i;
        }
    }
}

//Функция для процессинга текста
void processText(std::string& text, Trie& trie, Patterns& pattern,
Output& output){
    int current = 0;
    for (int i = 0; i < text.length(); i++){
        //std::cout << "Current: ";
        current = getLink(current, getIndex(text[i]), trie);
        checkUpLink(current, i+1, trie, output, pattern);
    }
}


int main()
{
    int count_patterns;
    Trie trie;
    std::string text;
    Output output;
    std::cin >> text;
    std::cin >> count_patterns;
    while (count_patterns <= 0 ){
        std::cout << "\nError, try again: ";
        std::cin >> count_patterns;
    }
    Patterns pattern(count_patterns);
    initTrie(trie);
    for (int i = 0; i < count_patterns; i++){
        std::cin >> pattern[i];
    }

    std::cout << "\n------------Fill trie start--------------" <<
std::endl;
    for (int i = 0; i < count_patterns; i++){
        addString(pattern[i], trie, i);
```

```
    }
    std::cout << "--------------Fill trie end----------------" <<
std::endl;

    std::cout << "------------Proccesing Text star-----------" <<
std::endl;
    processText(text, trie, pattern, output);
    std::cout << "------------Proccesing Text end------------" <<
std::endl;
    std::sort(output.begin(), output.end(), Compare);
    for (auto &curr: output){
        std::cout << curr.first << " " << curr.second << std::endl;
    }

    std::cout << "-------------------------------------\n" <<
std::endl;
    std::cout << "Count of nodes (states): " << trie.size() << std::endl;
    int pos1, pos2;
    int count_inter = 1;
       if (pattern.size() == 1 || text.size() < 2) return 0;
    for (int i = 0; i <= output.size()-2; i++){
        for (int j = i+1; j <= output.size()-1; j++){
            pos1 = pattern[output[i].second-1].size() + output[i].first -
1;
            pos2 = output[j].first;
            if (pos1 >= pos2 && output[i].second!=output[j].second){
                std::cout << "-------------------------------------" <<
std::endl;
                std::cout << "Intersection " << count_inter++ << "(" <<
output[i].first << ", " << output[j].first << ")" << std::endl;
                std::cout << "\tpattern 1: " << pattern[output[i].second-
1] << std::endl;
                std::cout << "\tpattern 2: " << pattern[output[j].second-
1] << std::endl;
            }
        }
    }


    return 0;
}
```

Приложение

Код поиск образца с джокером

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

struct Node;
```

```
using Trie = std::vector<Node>;                          //хранение бора
using ALphabet = std::map<char, int>;                    //алфавит, по
условию состоит из {A,C,G,T,N}
using Patterns = std::vector<std::string>;               //вектор образцов
using Output = std::vector<std::pair<int,int>>;          //вывод позиция -
образец

const int Alphabet_length = 5;
int Word_Len = 0;

ALphabet trie_alphabet = { {'A', 0}, {'C', 1}, {'G', 2}, {'T', 3}, {'N',
4} };

int getIndex(char symb){
    return trie_alphabet[symb];
}

bool Compare(std::pair<int, int> a, std::pair<int, int> b)
{                                                        //компаратор
для вывода результата
    if (a.first == b.first)
        return a.second < b.second;
    else
        return a.first < b.first;
}

// структура вершины
struct Node{
  int neighbours[Alphabet_length]; // соседние вершины
  int movePath[Alphabet_length];   // массив переходов
  int parrent;                     // вершина-предок
  std::vector<int> patternNumber;  // номер строки-образца
  int suffLink;                    // суффиксная ссылка
  int upSuffLink;                  // "сжатая" суффиксная ссылка
  char charToParrent;              // символ ведущий к предку
  bool terminal;                   // является ли терминальной
};                                 // (совпадает со строкой)



Node makeNode(int parrent, char transfer){               // создание
новой вершины
    Node newNode;
    memset(newNode.neighbours, 255, sizeof(newNode.neighbours));
    memset(newNode.movePath, 255, sizeof(newNode.neighbours));
    newNode.suffLink = newNode.upSuffLink = -1;          // изначально
нет ссылки
    newNode.parrent = parrent;
    newNode.charToParrent = transfer;
    newNode.terminal = false;
    //newNode.patternNumber = -1;
    return newNode;
```

```
}



void initTrie(Trie& trie){
    // создаем бор, изначально только корень
    trie.push_back(makeNode(0, '#'));
}

void addString(std::string& str, Trie& trie, int count_patterns){
    int index = 0;
    std::cout << "Add pattern: " << str << std::endl;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1){        // если нет ребра
по символу
            trie.push_back(makeNode(index, str[i]));
            trie[index].neighbours[curr] = trie.size() - 1;
            std::cout << "\tcreate node for transition: " << index << "--
" << str[i] << "->" << trie.size()-1 << std::endl;
        }
         index = trie[index].neighbours[curr];
    }
    trie[index].terminal = true;
    trie[index].patternNumber.push_back(count_patterns);
}


bool findString(std::string& str, Trie& trie){       //функция поиска
строки в боре
    int index = 0;
    for (int i = 0; i < str.length(); i++){
        int curr = getIndex(str[i]);
        if (trie[index].neighbours[curr] == -1)
            return false;
        index = trie[index].neighbours[curr];
    }
    return true;
}


int getLink(int vert, int index, Trie& trie);

// Функция для вычисления суффиксной ссылки
int getSuffLink(int vert, Trie& trie){
    if (trie[vert].suffLink == -1)                    // еще не искали
       if (vert == 0 || trie[vert].parrent == 0)      // корень или
родитель корень
          trie[vert].suffLink = 0;                    // для корня
ссылка в корень
       else
```

```
           trie[vert].suffLink = getLink(getSuffLink(trie[vert].parrent,
trie), getIndex(trie[vert].charToParrent), trie);
    return trie[vert].suffLink;
}


// Функция для вычисления перехода
int getLink(int vert, int index, Trie& trie){
   if (trie[vert].movePath[index] == -1)
// если переход по данному
      if (trie[vert].neighbours[index] != -1)
// символу ещ не вычислен
         trie[vert].movePath[index] = trie[vert].neighbours[index];
      else if (vert == 0)
          trie[vert].movePath[index] = 0;
      else
          trie[vert].movePath[index] = getLink(getSuffLink(vert, trie),
index, trie);
   return trie[vert].movePath[index];
}


// Функция для вычисления сжатой суффиксной ссылки
int getUpSuffLink(int vert, Trie& trie){
   if (trie[vert].suffLink == -1){             // если сжатая суффиксная
ссылка ещ не вычислена
      int tmp = getSuffLink(vert, trie);
      if (trie[tmp].terminal)                  // если ссылка на
терминальную, то ок
          trie[vert].upSuffLink = tmp;
      else if (tmp == 0)                       // на корень = 0
          trie[vert].upSuffLink = 0;
      else
          trie[vert].upSuffLink = getUpSuffLink(tmp, trie);   //поиск
ближайшей
   }
   return trie[vert].upSuffLink;
}


//проверка сжатых суффиксных ссылок
void checkUpLink(int C[], int vert, int index, Trie& trie, std::string&
text, Patterns& patterns, std::vector<int>& positions){
    int n = 0; int prev;
    for (int i = vert; i != 0; i = getUpSuffLink(i, trie)){
        if (trie[i].terminal){
            if (n) std::cout << "\t" << prev << "--UpLink-->" << i <<
std::endl;
            n++; prev = i;
            for (auto& in: trie[i].patternNumber){
                int tmp = index + 1 - patterns[in].size() -
positions[in];
```

```
                if (tmp >= 0 && tmp <= text.length() - Word_Len){
                    C[tmp]++;
                    std::cout << "\tNode " << i << " is terminal, at pos
" << index-patterns[in].length()+1 << " find: " << patterns[in] <<
std::endl;
                }
            }
        }
    }
}

//Функция для процессинга текста
//в качестве набора образцов - множество подстрок образца без джокера
void processText(int C[], std::string& text, Trie& trie, Patterns&
patterns, Output& output, std::vector<int>& positions){
    int current = 0;
    for (int i = 0; i < text.length(); i++){
        current = getLink(current, getIndex(text[i]), trie);
        checkUpLink(C, current, i, trie, text, patterns, positions);
    }
    std::cout << "\nResult: " << std::endl;
    int num = 0;
    for (int i = 0; i < text.size(); i++){
        if (C[i] == patterns.size()){
            std::cout << i+1 << std::endl;
            num++;
        }
    }
    if (num == 0) std::cout << "\nNothing" << std::endl;
}

// Функция разбиение образца на максимальные подстроки без джокера
void splitString(std::string& str, Patterns& patterns, std::vector<int>&
positions, char joker){
    int index = 0;
    int position = 0;
    int count = 0;
    for (int i = 0; i < str.length(); i = index){
        std::string buff = "";
        while (index < str.length() && str[index] == joker)   //
пропускаем джокера
            index++;
        if (index == str.length()) return;
        position = index;
        while(index < str.length() && str[index] != joker)    // новоя
подстрока
            buff += str[index++];
         if(!buff.empty()){
            count++;
            positions.push_back(position);
//запоминаем позиции подстрок
            patterns.push_back(buff);
```

```
        }
    }

}


int main()
{
    int count_patterns;
    Trie trie;
    std::string text;
    std::string word;
    std::vector<int> positions;
    char joker;
    Output output;

    std::cin >> text;
    std::cin >> word;
    std::cin >> joker;

    auto it = trie_alphabet.find(joker);
    while (it != trie_alphabet.end()){
        std::cout << "\nError, joker can`t be ALphabet symbol: ";
        std::cin >> joker;
        it = trie_alphabet.find(joker);
    }

    Word_Len = word.length();
    Patterns patterns;
    initTrie(trie);
    splitString(word, patterns, positions, joker);
    int i = 0;
    std::cout << "\n------------Fill trie start--------------" <<
std::endl;
    for (auto& buff: patterns){
        addString(buff, trie, i++);
    }
    std::cout << "--------------Fill trie end----------------" <<
std::endl;

    int C[text.size()] = {0};

    std::cout << "Count of nodes: " << trie.size() << std::endl;
    std::cout << "Parts of pattern without joker " << joker << std::endl;
    for (auto& buff: patterns){
        std::cout << "\t" << buff << std::endl;
    }
    std::cout << "Array of starts positions for parts" << std::endl;
    for (auto& buff: positions){
        std::cout << " " << buff;
    }
```

```
    std::cout << "\n------------Proccesing Text star-----------" <<
std::endl;
    processText(C,text, trie, patterns, output, positions);
    std::cout << "------------Proccesing Text end------------" <<
std::endl;
    return 0;
}
```