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: Ахо-Корасик

Студент гр. 9383 _____ Крейсманн К.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритм Ахо-Корасика, реализовать с его помощью 2 программы, первая из которых решает задачу точного поиска набора образцов, а вторая решает задачу точного поиска для одного образца с джокером.

Задание.

1) Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,1<=|T|<=100000).

Вторая – числа n(1<=n<=3000), каждая следующая из n строк содержит шаблон из набора P = {p1,….,pn} 1<=pi<=75.

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

Выход:

Все вхождения образцов из P в T.

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

Где i – позиция в тексте, с которой начинается вхождение образца с номером p.

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

Пример входных данных:

NTAG

3

TAGT

TAG

T

Пример выходных данных:

2 2

2 3

2) 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?a*b*??c? с джокером ?? встречается дважды в тексте xabvccbababcax*xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в T*T*. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A,C,G,T,N}.
Вход:

Текст (T,1<=|T|<=100000)

Шаблон (P,1<=|P|<=40)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона.

Номера в порядке возрастания.

Пример входных данных:

ACTANCA

A$$A$

$

Пример выходных данных:

1

Теория

Бор – дерево, образованное последовательным добавлением всех образцов посимвольно. При добавлении символа создается вершина, если соответствующий подобразец еще не добавлялся. Иначе просто осуществляется переход к ранее созданной вершине. Изначально бор состоит из корня.

Автомат – это бор, дополненный суффиксными и конечными ссылками. Это дополнение может происходить заранее или во время обработки текста.

Суффиксная ссылка из вершины A – это ссылка на вершину, соответствующую максимально длинному подобразцу в автомате, являющемуся несобственным суффиксов подобразца A. Для корня и его сыновей суффиксная ссылка указывает на корень.

Конечная ссылка из вершины A – это ссылка на вершину, соответствующую максимально длинному образцу, который может быть получен при выполнении нескольких переходов по суфф. Ссылкам, начиная с A. Если образцов получить нельзя, то конечная ссылка пуста.

Алгоритм точного поиска набора подобразцов

1. Строится бор из образцов.
2. В бор добавляются суффиксные ссылки:

Происходит обход в ширину. Сначала корню и его сыновьям добавляется суффиксная ссылка на корень.

Далее для каждой вершины:

А) Переход к вершине-родителю

Б) Переход по суффиксным ссылкам, пока не появится путь по ребру или до попадания в корень.

В) Переход по ребру. Если мы в корне и нужного ребра из него нет, то остаемся в корне.

3. Происходит посимвольная обработка текста. Для каждого символа:

3.1. Совершается переход из текущей вершины по рассматриваемому символу:

3.1.1. Если есть нужное ребро, происходит переход по нему.

3.1.2. Если нет, то если находимся не в корне, то происходит переход по суффиксной ссылке и повторяется 3.1. Если находимся в корне, то ничего не происходит.

3.2. Если перешли в конечную вершину, то добавляется в результат вхождение образца.

3.3. Обходятся конечные ссылки, сохраняя результаты.

Алгоритм точного поиска для одного образца с джокером

1. Строится автомат Ахо-Корасик из образцов, полученных выделением максимальных безджокерных подстрок из шаблонной подстроки.
2. Для каждого образца записывается смещения, по которым образец находится в шаблонной строке.
3. Инициализируется массив, заполненный нулями, длиной, совпадающей с текстом.
4. Выполняется поиск по тексту с использованием автомата. При обнаружении образца инкрементируются ячейка массива по адресу, образованному разностью номера начального символа образца в текста и смещением образца. Если у образца несколько смещений, то инкрементируются все соответствующие ячейки массива.

Шаблонная подстрока будет начинаться в тех местах текста, для которых соответствующая ячейка массива содержит количество образцов.

Описание основных функций и структур данных

Vertex – вершина бора.

Input() – ввод входных данных.

addSuffixLinks() – добавление суффиксных ссылок.

findPos() – реализация алгоритмов точного поиска набора подобразцов и точного поиска для одного образца с джокером.

addString() - добавление строки в бор.

addInstances() – разбиение шаблонной строки на подобразцы и добавление их в бор.

nextVertex() – переход к следующей вершине по суффиксным ссылкам.

Тестирование (задание 1)

Входные данные	Выходные данные
NTAG 3 TAGT TAG T	2 2 2 3
AAAAA 3 A C AAA	1 1 1 3 2 1 2 3 3 1 3 3 4 1 5 1
ACGTNAAA 3 ACGTN AVGTNN	1 1 4 3

TNAA	
CGGGCCCCNNN 4 NNNN CCGGG GC CNN	4 3 8 4

Тестирование (задание 2)

Входные данные	Выходные данные
ACTANCA A$$A$ $	1
AAAAA *A* *	1 2 3
AOOACCCOAOAOCAOGAO !OA !	2 **7** 9
ACGTNACGACTNACNACGGGGGACTGGGGNNN ***GAC********* *	5

Вывод.

В процессе выполнения лабораторной работы был изучен и реализован алгоритм Ахо-Корасик. С помощью данного алгоритма построены программы для точного поиска вхождений подобразцов в текст и для точного поиска одного образца с джокером.

Приложение А

main.cpp

```
#include "task1.hpp"

int main()
{
   Vertex* root = new Vertex;
   std::string Text;
   input(std::cin, root, Text);
   addSuffLinks(root);
   Result result = findPos(root, Text);
   std::sort(result.begin(), result.end(), [](auto i, auto j)
      {
         if (i.first != j.first)
         {
            return i.first < j.first;
         }
         return i.second < j.second;
      });
   for (auto i : result)
   {
      std::cout << i.first << ' ' << i.second << '\n';
   }
   std::cout << '\n';

   return 0;
}
```

task1.hpp

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

struct Vertex;
using Links = std::map<char, Vertex*>;
using Result = std::vector<std::pair<int, int>>;
struct Vertex
{
   Links links;
   Vertex* suffixLink = nullptr;
   int terminal = 0;
   int level = 0;
};

void addString(Vertex* root, std::string string, int number);
```

```
void addSuffLinks(Vertex* root);
void input(std::istream& in, Vertex* root, std::string& Text);
Vertex* nextVertex(Vertex*vertex,Vertex*root,char symb);
Result findPos(Vertex* root, std::string Text);
```

task1.cpp

```
#include "task1.hpp"

void addString(Vertex* root, std::string string, int number)
{
   Vertex* vertex = root;
   int lvl = 1;
   std::for_each(string.begin(), string.end(), [&vertex, &lvl](char c)
      {
         if (!vertex->links[c])
         {
            vertex->links[c] = new Vertex;
         }
         vertex = vertex->links[c];
         vertex->level = lvl++;
      });
   vertex->terminal = number;
}

void addSuffLinks(Vertex* root)
{
   std::queue<Vertex*> vertexesQueue;

   root->suffixLink = root;
   std::for_each(root->links.begin(), root->links.end(), [&root, &vertexesQueue](auto link)
      {
         link.second->suffixLink = root;
         vertexesQueue.push(link.second);
      });

   while (!vertexesQueue.empty())
   {
      Vertex* current = vertexesQueue.front();
      vertexesQueue.pop();

      for (auto link : current->links)
      {
         char name = link.first;
         Vertex* child = link.second;
         Vertex* suffixLink = current->suffixLink;
         while (suffixLink != root && !suffixLink->links[name])
         {
            suffixLink = suffixLink->suffixLink;
         }
         if (suffixLink == root && !suffixLink->links[name])
         {
```

```
            child->suffixLink = suffixLink;
        }
        else
        {
            suffixLink = suffixLink->links[name];
            child->suffixLink = suffixLink;
        }
        vertexesQueue.push(child);
    }
  }
}

void input(std::istream& in, Vertex* root, std::string& Text)
{
  in >> Text;

  int quantity;
  in >> quantity;

  int number = 1;
  for (int i = 0; i < quantity; i++)
  {
    std::string currentString;
    in >> currentString;
    addString(root, currentString, number++);
  }
}

Vertex* nextVertex(Vertex*vertex,Vertex*root,char symb)
{
  while(!vertex->links[symb]&&vertex!=root)
  {
    vertex=vertex->suffixLink;
  }
  if(vertex->links[symb])
  {
    vertex=vertex->links[symb];
  }
  return vertex;
}


Result findPos(Vertex* root, std::string Text)
{
  Result result;
  Vertex* currentVert = root;
  for (int i = 0; i < Text.length(); i++)
  {
    char currentSymb = Text[i];
    currentVert = nextVertex(currentVert,root,currentSymb);

    Vertex *v = currentVert;
```

```
        while(v!=root)//переходим по конечным ссылкам
        {
            if(v->terminal!=0)
            {
                result.push_back(std::pair<int, int>(i + 2 - v->level, v->terminal));
            }
            v=v->suffixLink;
        }
    }
    return result;
}
```

main.cpp

```
#include "task2.hpp"

int main()
{
    Vertex* root = new Vertex;
    std::string Text, pattern;
    char joker;
    int count = 0;
    input(std::cin, root, Text, pattern, joker);
    addInstances(root, pattern, joker, count);
    addSuffLinks(root);
    Result result = findPos(root, Text, count, pattern.size());
    for (auto i : result)
    {
        std::cout << i << '\n';
    }
    return 0;
}
```

task2.hpp

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

struct Vertex;
using Links = std::map<char, Vertex*>;
using Result = std::vector<int>;
struct Vertex
{
    Links links;
    Vertex* suffixLink = nullptr;
    std::vector<int> offsets;
};
void addString(Vertex* root, std::string string, int offset);
void addSuffLinks(Vertex* root);
void input(std::istream& in, Vertex* root, std::string& Text, std::string& pattern, char& joker);
Vertex* nextVertex(Vertex* vertex, Vertex* root, char symb);
```

```
Result findPos(Vertex* root, std::string Text, int count, int patternSize);
void addInstances(Vertex* root, std::string pattern, char joker, int& count);

task2.cpp
#include "task2.hpp"

void addString(Vertex* root, std::string string, int offset)
{
   Vertex* vertex = root;
   std::for_each(string.begin(), string.end(), [&vertex](char c)
      {
         if (!vertex->links[c])
         {
            vertex->links[c] = new Vertex;
         }
         vertex = vertex->links[c];
      });
   vertex->offsets.push_back(offset);
}

void addSuffLinks(Vertex* root)
{
   std::queue<Vertex*> vertexesQueue;

   root->suffixLink = root;
   std::for_each(root->links.begin(), root->links.end(), [&root, &vertexesQueue](auto link)
      {
         link.second->suffixLink = root;
         vertexesQueue.push(link.second);
      });

   while (!vertexesQueue.empty())
   {
      Vertex* current = vertexesQueue.front();
      vertexesQueue.pop();

      for (auto link : current->links)
      {
         char name = link.first;
         Vertex* child = link.second;
         Vertex* suffixLink = current->suffixLink;
         while (suffixLink != root && !suffixLink->links[name])
         {
            suffixLink = suffixLink->suffixLink;
         }
         if (suffixLink == root && !suffixLink->links[name])
         {
            child->suffixLink = suffixLink;
         }
         else
         {
            suffixLink = suffixLink->links[name];
```

```
            child->suffixLink = suffixLink;
        }
        vertexesQueue.push(child);
    }
  }
}

void input(std::istream& in, Vertex* root, std::string& Text, std::string& pattern, char& joker)
{
  in >> Text;
  in >> pattern;
  in >> joker;
}

Vertex* nextVertex(Vertex* vertex, Vertex* root, char symb)
{
  while (!vertex->links[symb] && vertex != root)
  {
    vertex = vertex->suffixLink;
  }
  if (vertex->links[symb])
  {
    vertex = vertex->links[symb];
  }
  return vertex;
}


Result findPos(Vertex* root, std::string Text, int count, int patternSize)
{
  Result result;
  std::vector<int> numberMatches(Text.size(), 0);
  Vertex* currentVert = root;
  for (int i = 0; i < Text.length(); i++)
  {
    char currentSymb = Text[i];
    currentVert = nextVertex(currentVert, root, currentSymb);

    Vertex* v = currentVert;
    while (v != root)//переходим по конечным ссылкам
    {
      if (!v->offsets.empty())
      {
        for (auto offset : v->offsets)
        {
          if(i-offset >=0)
            numberMatches[i - offset]++;
        }
      }
      v = v->suffixLink;
    }
  }
```

```
    for (int i = 0; i < numberMatches.size(); i++)
    {
        if (numberMatches[i] == count && i + patternSize <= numberMatches.size() )
        {
            result.push_back(i + 1);
        }
    }
    return result;
}

void addInstances(Vertex* root, std::string pattern, char joker, int& count)
{
    std::string instance;
    int offset = 0;
    for (auto c : pattern)
    {
        if (c == joker)
        {
            if (!instance.empty())
            {
                addString(root, instance, offset - 1);
                count++;
            }

            instance.clear();
        }
        else
        {
            instance.push_back(c);
        }
        offset++;
    }
    if (!instance.empty())
    {
        addString(root, instance, offset - 1);
        count++;
    }
}
```