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. 8382 _____ Щемель Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Задание

Точный поиск образцов

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:
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 $P = \{p_1, ..., p_n\} 1 \leq |p_i| \leq 75$
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 (нумерация начинается с 1), с которой начинается вхождение образца с номером *p*
(нумерация образцов начинается с 1). Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:
CCCA
1
CC

Sample Output:
1 1 2 1

Поиск образцов с джокером

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* xabvc-cbababcax.

Символ джокер не входит в алфавит, символы которого используются в *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 $(T, 1 \leq |T| \leq 100000)$
Шаблон $(P, 1 \leq |P| \leq 40)$
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

Номер варианта: 1. На месте джокера может быть любой символ, за исключением заданного.

Описание алгоритма

Точный поиск образцов

Решение задачи происходит в 3 этапа:

1. Считывание данных: текст для поиска, количество образцов для поиска и сами образцы.

2. Построение бора для введённых образцов. Создаётся корневая вершина. В неё следующим образом добавляется очередная строка: если очередного символа из строки нет в хэш-таблице дочерних вершин - создаётся новая и добавляется в таблицу. Если же есть - из хэш-таблицы достаётся уже существующая вершина. Для этой вершины рекурсивно вызывается этот же шаг для обработки следующего символа. В каждой вершине так же сохранятся информация для дальнейшей работы алгоритма: ссылка на родительскую и корневую вершины. В вершинах, соответствующих концу образца, сохраняется так же номер образца и его длина. Пример бора для набора строк {he, she, his, hers}:

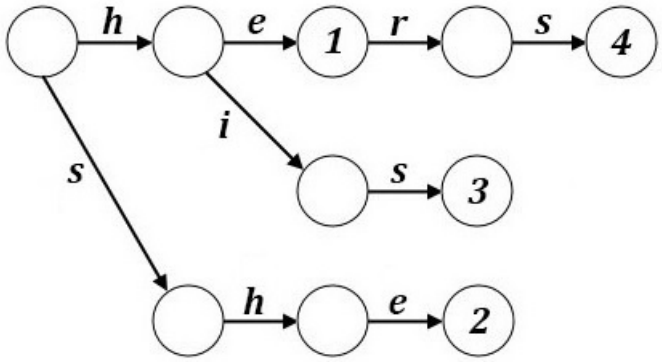

Figure 1: Бор

3. Построение автомата по бору и поиск вхождения образцов в текст: Здесь вводятся дополнительные понятия:

   - Функция перехода - функция, которая по вершине и символу возвращает одно из следующих состояний автомата(выполняет переход): сохранённое значение функции перехода, если оно уже вычислено или вершину, соответствующую символу в словаре дочерних дочерних вершин для заданной, если таковая имеется или корневая вершина, если заданная вершина - корневая и среди

дочерних нет вершины, соответствующей заданному символу в словаре дочерних вершин заданной(корневой) или значение рекурсивного вызова функции перехода, но уже для суффиксной ссылки заданной вершины и для заданного символа.

- Суффиксная ссылка - ссылка на вершину, с наибольший собственным суффиксом для [u], не равной u (u - вершина [u] - слово, соответствующее этой вершине в боре). Она вычисляется следующим образом для вершины(если еще не вычислено ранее. Если вычислено - используется сохранённое значение): если переданная вершина или родительская для переданной вершины - корневая - результатом будет корневая вершина. Если нет - результатом будет функция перехода для родительской вершины заданной и символа, соответствующего текущей вершине.
- Сжатая суффиксная ссылка - ссылка на вершину, в которой заканчивается наидлиннейший собственный суффикс строки, соответствующий переданной вершине. Она вычисляется следующим образом для вершины(если еще не вычислено ранее. Если вычислено - используется сохранённое значение): вычисляется суффиксная ссылка для заданной вершины. Если она (получившаяся вершина) - терминальная (это определяется тем, есть ли в ней информация об образцах) или корневая - это и будет результат (сжатая суффиксная ссылка). Если нет - результатом будет рекурсивный вызов функции вычисления сжатой суффиксной ссылки для суффиксной ссылки заданной вершины.

В совокупности новые сущности позволяют для одного терминального состояния автомата получить все другие терминальные состояния, соответствующие собственному суффиксу для текущего префикса. Автомат строится ленивым образом, во время обработки текста.

Пример автомата, построенного для бора из п.1:

Поиск вхождений образцов в текст выполняется следующим образом: для корневой вершины выполняется переход по первому символу из текста. Если новое состояние автомата - терминальное, то сохраняется результат вхождения образца в строке и рекурсивно выполняется переход по сжатым суффиксным ссылкам пока не будет достигнута корневая

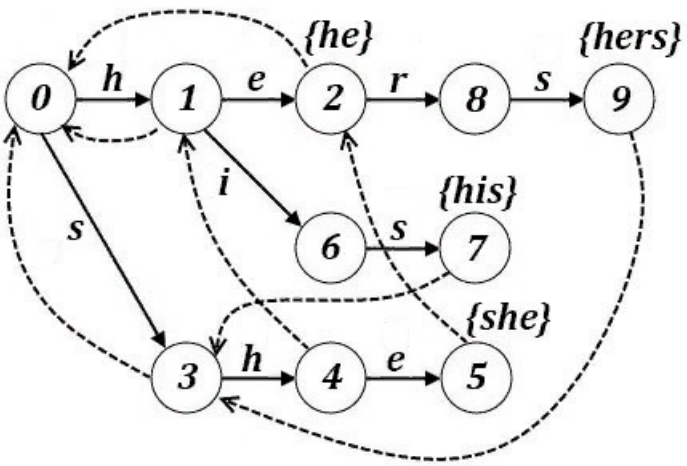

Figure 2: Конечный автомат

вершина и сохраняется результат (вхождение образца в строку). Дальше выполняется переход по следующему символу на нового полученного состояния пока не будет достигнут конец строки.

Поиск образцов с джокером

Алгоритм аналогичен тому, что был описан в предыдущем разделе. Единственное различие в функции перехода - если в словаре дочерних вершин есть вершина, соответствующая символу джокера (и, с учётом индивидуализации, переданный символ не равен символу-исключению), то можно осуществить переход по ней.

Сложности алгоритма:

Сложность одинаковая для обоих заданий.

По количеству операций

Так как для хранения дочерних вершин используется хэш-таблица, то сложность по количеству операций составляет $O(T + s + e)$, где T - длина текста, s - суммарная длина всех образцов, e - общее количество всех совпадений. Доказательство: обработка текста за $O(n)$ + обработка всех образцов за $O(n)$ + переход по сжатым ссылкам за $O(n)$.

По памяти

Так как для хранения дочерних вершин используется хэш-таблица, то сложность по памяти составляет $O(s)$, где s - суммарная длина всех образцов(по которым строится бор).

Описание функций и структур данных

В реализованной программе (см. исх. код в приложении А)

Используемые классы и структуры

- *struct PatternOccurrence* - структура для вывода результата

  public int CharPosition { get; set; } - позиция вхождения в строке

  public int PatternNumber { get; set; } - номер образца

- *class Node* - вершина в боре / состояние в автомате

  private readonly Node parent - родительская вершина

  private readonly Node rootNode - корневая вершина

  private Node suffixLink - суффиксная ссылка

  private Node supressedSuffixLink - сжатая суффиксная ссылка

  private readonly Dictionary<char, Node> children - хэш-таблица дочерних элементов

  private readonly Dictionary<char, Node> transitions - хэш-таблица переходов

  private readonly List patternInfos - список структур, для хранения информации об образце (для терминальных вершин)

  public static char? Joker { get; set; } - символ-джокер

  public static char? JokerException { get; set; } - символ-исключение для джокера

  private char Name { get; set; } = 'Z' - символ, соответствующий вершине

  private struct PatternInfo - структура для хранения информации о шаблоне

  - *public int PatternNumber { get; set; }* - номер шаблона
  - *public int PatternLength { get; set; }* - длина шаблона

Используемые методы

- *public void AddString(string str, int numberOfPattern, int index = -1)* - используется для добавления строки

string str - строка для добавления

int numberOfPattern - номер шаблона

int index - номер индекса в строке

- *public IEnumerable FindPatternsInText(string text)* - ищет вхождения шаблонов в тексте

  string text - текст для поиска

- *private bool IsRoot* - определяет, является ли вершина корневой
- *private bool IsTerminal* - определяет, является ли вершина терминальной
- *private IEnumerable ComputePatternOccurrences(int index)* - используется для вычисления результата - индекса начала вхождения образца

  int index - индекс конца конца вхождения шаблона

- *private Node GetSuffixLink(Node node)* - используется для нахождения суффиксной ссылки

  Node node - вершина, для которой необходимо найти суффиксную ссылку

- *private Node GetTransition(Node node, char c)* - используется для вычисления перехода

  Node node - вершина, для который необходимо вычислить переход

  char c - символ, по которому необходимо совершить переход

- *private Node GetSuppressedSuffixLink(Node node)* - используется для вычисления сжатой суффиксной ссылки

  Node node - вершина, для которой необходимо вычислить сжатую суффиксную ссылку

Тестирование

Точный поиск образцов

Тест 1:

Тест 2:

Тест 3:

Тест 4:

Поиск образцов с джокером

Тест 1:

Тест 2:

Тест 3:

Тест 4:

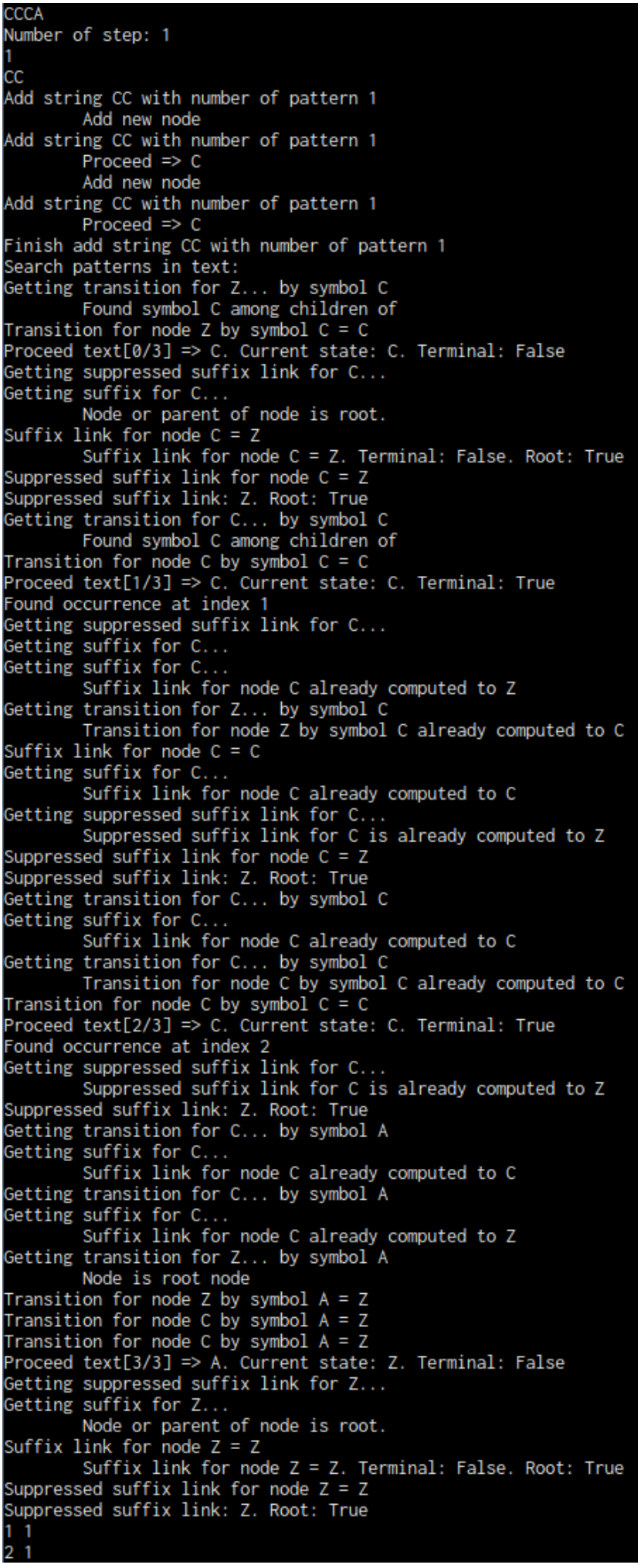

Figure 3: image-20200420171939764

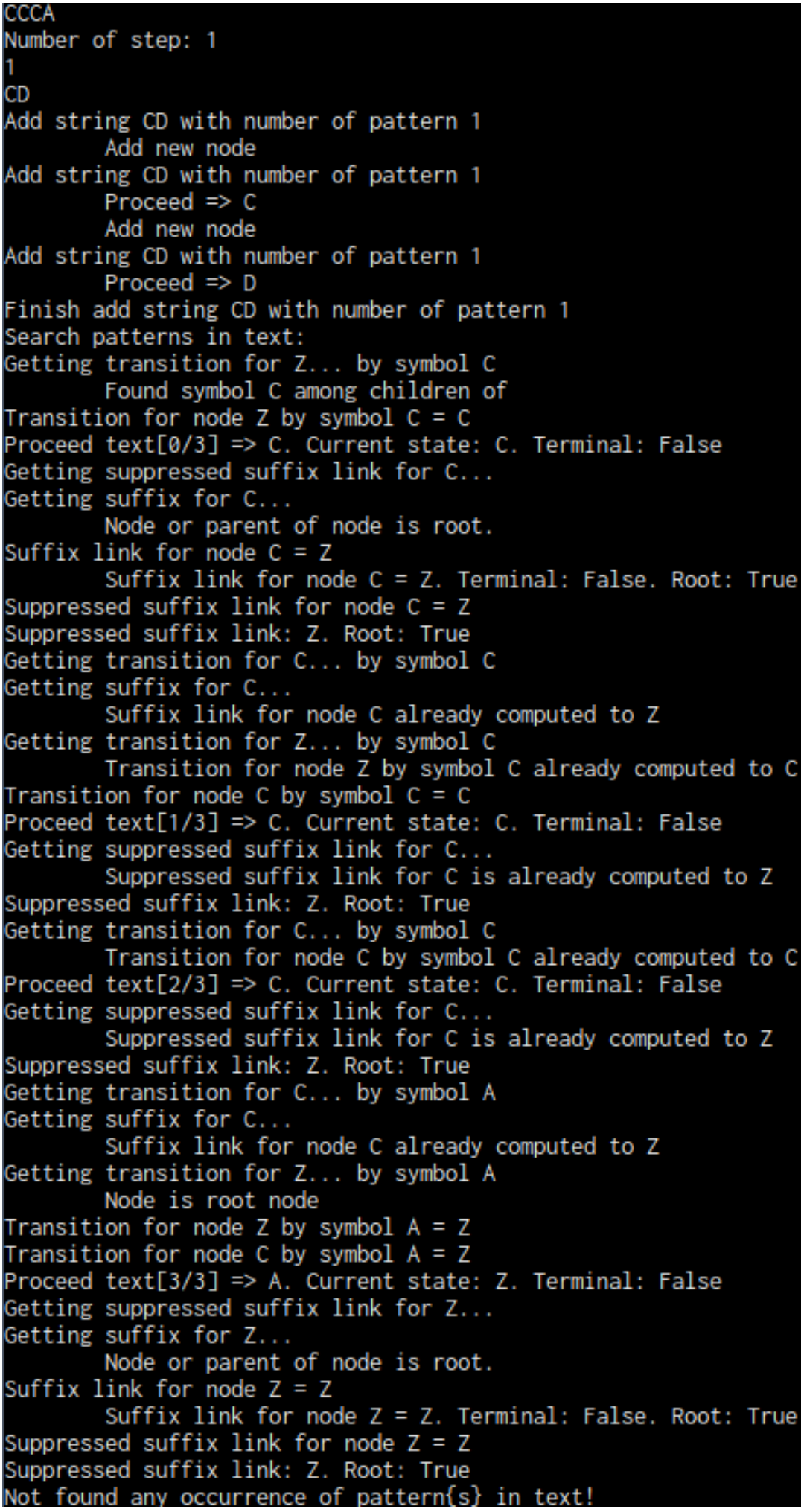

Figure 4: image-20200420172028277

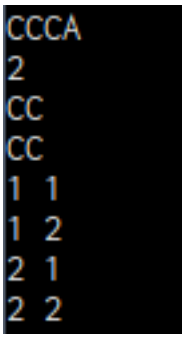

Figure 5: image-20200420172102791

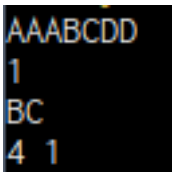

Figure 6: image-20200420172141007

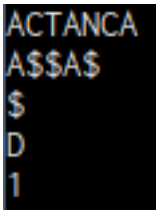

Figure 7: image-20200420172221077

```
ACTANCA
Number of step: 2
A$$A$
Add string A$$A$ with number of pattern 1
        Add new node
Add string A$$A$ with number of pattern 1
        Proceed => A
        Add new node
Add string A$$A$ with number of pattern 1
        Proceed => $
        Add new node
Add string A$$A$ with number of pattern 1
        Proceed => $
        Add new node
Add string A$$A$ with number of pattern 1
        Proceed => A
        Add new node
Add string A$$A$ with number of pattern 1
        Proceed => $
Finish add string A$$A$ with number of pattern 1
$
C
Search patterns in text:
Getting transition for Z... by symbol A
        Symbol A is acceptable for joker : True
        Found symbol A among children of
Transition for node Z by symbol A = A
Proceed text[0/6] => A. Current state: A. Terminal: False
Getting suppressed suffix link for A...
Getting suffix for A...
        Node or parent of node is root.
Suffix link for node A = Z
        Suffix link for node A = Z. Terminal: False. Root: True
Suppressed suffix link for node A = Z
Suppressed suffix link: Z. Root: True
Getting transition for A... by symbol C
        Symbol C is acceptable for joker : False
Getting suffix for A...
        Suffix link for node A already computed to Z
Getting transition for Z... by symbol C
        Symbol C is acceptable for joker : False
        Node is root node
Transition for node Z by symbol C = Z
Transition for node A by symbol C = Z
Proceed text[1/6] => C. Current state: Z. Terminal: False
Getting suppressed suffix link for Z...
Getting suffix for Z...
        Node or parent of node is root.
Suffix link for node Z = Z
        Suffix link for node Z = Z. Terminal: False. Root: True
Suppressed suffix link for node Z = Z
Suppressed suffix link: Z. Root: True
Getting transition for Z... by symbol T
        Symbol T is acceptable for joker : True
        Node is root node
Transition for node Z by symbol T = Z
Proceed text[2/6] => T. Current state: Z. Terminal: False
Getting suppressed suffix link for Z...
        Suppressed suffix link for Z is already computed to Z
Suppressed suffix link: Z. Root: True
Getting transition for Z... by symbol A
        Transition for node Z by symbol A already computed to A
Proceed text[3/6] => A. Current state: A. Terminal: False
Getting suppressed suffix link for A...
        Suppressed suffix link for A is already computed to Z
Suppressed suffix link: Z. Root: True
Getting transition for A... by symbol N
        Symbol N is acceptable for joker : True
        Found symbol N among children of
Transition for node A by symbol N = $
Proceed text[4/6] => N. Current state: $. Terminal: False
Getting suppressed suffix link for $...
Getting suffix for $...
Getting suffix for A...
        Suffix link for node A already computed to Z
Getting transition for Z... by symbol $
        Symbol $ is acceptable for joker : True
        Node is root node
Transition for node Z by symbol $ = Z
Suffix link for node $ = Z
        Suffix link for node $ = Z. Terminal: False. Root: True
Suppressed suffix link for node $ = Z
Suppressed suffix link: Z. Root: True
Getting transition for $... by symbol C
        Symbol C is acceptable for joker : False
Getting suffix for $...
        Suffix link for node $ already computed to Z
Getting transition for Z... by symbol C
        Transition for node Z by symbol C already computed to Z
Transition for node $ by symbol C = Z
Proceed text[5/6] => C. Current state: Z. Terminal: False
Getting suppressed suffix link for Z...
        Suppressed suffix link for Z is already computed to Z
```

Figure 8: image-20200420172413510

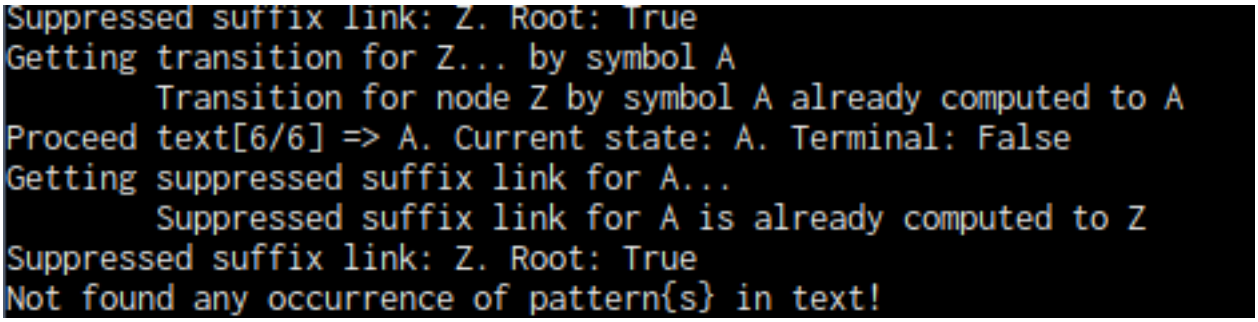

Figure 9: image-20200420172431186

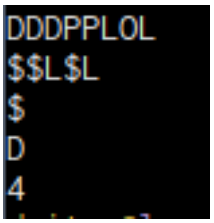

Figure 10: image-20200420172501700

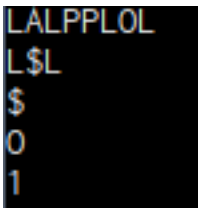

Figure 11: image-20200420172525620

Вывод

В ходе выполнения лабораторной работы были получены практические навыки применения алгоритма “Ахо-Корасик” и были написаны две программы: для нахождения точно вхождения множества образцов в тексте и для нахождения шаблона с маской(джокером) в тексте.

Приложение А. Исходный код

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace lab5
{
    class Logger
    {
        public static LogLevelEnum LogLevel { get; set; }

        /// <summary>
        /// Log message with specify log levelEnum
        /// </summary>
        /// <param name="text"></param>
        /// <param name="levelEnum"></param>
        public static void Log(object text,
            LogLevelEnum levelEnum = LogLevelEnum.Info)
        {
            if (levelEnum >= LogLevel)
            {
                Console.WriteLine(text.ToString());
            }
        }

        public enum LogLevelEnum
        {
            Debug,
            Info
        }
    }

    struct PatternOccurrence
    {
```

```
    public int CharPosition { get; set; }

    public int PatternNumber { get; set; }
}

class Node
{
    private readonly Node parent;
    private readonly Node rootNode;
    private Node suffixLink;
    private Node supressedSuffixLink;
    private readonly Dictionary<char, Node> children
        = new Dictionary<char, Node>();
    private readonly Dictionary<char, Node> transitions
        = new Dictionary<char, Node>();
    private readonly List<PatternInfo> patternInfos
        = new List<PatternInfo>();

    public Node()
    {
        rootNode = this;
    }

    private Node(Node rootNode, Node parent)
    {
        this.parent = parent;
        this.rootNode = rootNode;
    }

    public static char? Joker { get; set; }

    public static char? JokerException { get; set; }

    public void AddString(string str, int numberOfPattern, int index = -1)
    {
```

```
Logger.Log($"Add string {str} with number of pattern "
    "{numberOfPattern+1}", Logger.LogLevelEnum.Debug);
// If not root node
if (index != -1)
{
    Logger.Log($"\tProceed => {str[index]}",
        Logger.LogLevelEnum.Debug);
    Name = str[index];
}

if (index == str.Length - 1)
{
    Logger.Log($"Finish add string {str} with number of pattern"
     "{numberOfPattern+1}", Logger.LogLevelEnum.Debug);

    // Save info about pattern
    patternInfos.Add(new PatternInfo
    {
        PatternNumber = numberOfPattern + 1,
        PatternLength = str.Length
    });
    return;
}


var nextChar = str[index + 1];
Node existedNode;
if (!children.TryGetValue(nextChar, out existedNode))
{
    Logger.Log($"\tAdd new node", Logger.LogLevelEnum.Debug);
    existedNode = new Node(rootNode, this);
    children.Add(nextChar, existedNode);
}
else
{
```

```
            Logger.Log($"\tFound existing node",
                Logger.LogLevelEnum.Debug);
        }

        existedNode.AddString(str, numberOfPattern, ++index);
    }

    public IEnumerable<PatternOccurrence> FindPatternsInText(string text)
    {
        Logger.Log($"Search patterns in text:",
        Logger.LogLevelEnum.Debug);
        var retList = new List<PatternOccurrence>();
        var current = rootNode;
        for (var i = 0; i < text.Length; ++i)
        {
            current = GetTransition(current, text[i]);
            Logger.Log($"Proceed text[{i}/{text.Length-1}] =>"
            " {text[i]}. Current state: {current.Name}. "
            "Terminal: {current.IsTerminal}",
                Logger.LogLevelEnum.Debug);
            if (current.IsTerminal)
            {
                Logger.Log($"Found occurrence at index {i}",
                    Logger.LogLevelEnum.Debug);
                retList.AddRange(current.ComputePatternOccurrences(i));
            }
            var currentSuppressedSuffixLink =
                 GetSuppressedSuffixLink(current);
            Logger.Log($"Suppressed suffix link:"
            " {currentSuppressedSuffixLink.Name}."
            " Root: {currentSuppressedSuffixLink.IsRoot}",
                Logger.LogLevelEnum.Debug);

            // Find all others terminals
            while (!currentSuppressedSuffixLink.IsRoot)
```

```
            {
                retList.AddRange(currentSuppressedSuffixLink
                    .ComputePatternOccurrences(i));
                currentSuppressedSuffixLink
                    = GetSuppressedSuffixLink(currentSuppressedSuffixLink);
                Logger.Log($"Suppressed suffix link:"
                " {currentSuppressedSuffixLink.Name}."
                " Root: {currentSuppressedSuffixLink.IsRoot}",
                    Logger.LogLevelEnum.Debug);
            }
        }

        return retList;
    }

    private bool IsRoot
    {
        get { return parent == null; }
    }

    private bool IsTerminal
    {
        get { return patternInfos.Count != 0; }
    }

    private char Name { get; set; } = 'Z'; // Stub name for root

    private IEnumerable<PatternOccurrence>
        ComputePatternOccurrences(int index)
    {
        return patternInfos.Select(x => new PatternOccurrence
            {
                // Find start index of occurrence
                CharPosition = index - x.PatternLength + 2,
                PatternNumber = x.PatternNumber
```

```
        });
    }

    private Node GetSuffixLink(Node node)
    {
        Logger.Log($"Getting suffix for {node.Name}...",
            Logger.LogLevelEnum.Debug);
        if (node.suffixLink != null)
        {
            Logger.Log($"\tSuffix link for node {node.Name}"
            " already computed to {node.suffixLink.Name}",
                Logger.LogLevelEnum.Debug);
            return node.suffixLink;
        }

        if (node.IsRoot || node.parent.IsRoot)
        {
            Logger.Log($"\tNode or parent of node is root.",
                Logger.LogLevelEnum.Debug);
            node.suffixLink = rootNode;
        }
        else
        {
            // Try to find at transition by parent's suffix link
            node.suffixLink
            = GetTransition(
                GetSuffixLink(node
                .parent),
                node.Name);
        }

        Logger.Log($"Suffix link for node {node.Name}"
            " = {node.suffixLink.Name}",
            Logger.LogLevelEnum.Debug);
        return node.suffixLink;
```

```
}

private Node GetTransition(Node node, char c)
{
    Logger.Log($"Getting transition for {node.Name}... by symbol {c}",
        Logger.LogLevelEnum.Debug);
    Node retNode;
    if (node.transitions.TryGetValue(c, out retNode))
    {
        Logger.Log($"\tTransition for node {node.Name} by symbol {c}"
            " already computed to {retNode.Name}",
            Logger.LogLevelEnum.Debug);
        return retNode;
    }

    var isAcceptableForJoker = JokerException == null
                || c != JokerException;

    if (JokerException != null)
    {
        Logger.Log($"\tSymbol {c} is acceptable for joker :"
            " {isAcceptableForJoker}",
            Logger.LogLevelEnum.Debug);
    }

    if (node.children.TryGetValue(c, out retNode)
        || isAcceptableForJoker
        && Joker != null
        && node.children.TryGetValue((char)Joker, out retNode))
    {
        Logger.Log($"\tFound symbol {c} among children of",
            Logger.LogLevelEnum.Debug);
        node.transitions.Add(c, retNode);
    }
    else if (node == rootNode)
```

```
        {
            Logger.Log($"\tNode is root node",
                Logger.LogLevelEnum.Debug);
            node.transitions.Add(c, rootNode);
        }
        else
        {
            // Fallback to suffix link
            node.transitions.Add(c, GetTransition(GetSuffixLink(node), c));
        }

        node.transitions.TryGetValue(c, out retNode);

        Logger.Log($"Transition for node {node.Name} by symbol {c} ="
            " {retNode?.Name}",
            Logger.LogLevelEnum.Debug);
        return retNode;
    }

    private Node GetSuppressedSuffixLink(Node node)
    {
        Logger.Log($"Getting suppressed suffix link for {node.Name}...",
            Logger.LogLevelEnum.Debug);
        if (node.supressedSuffixLink != null)
        {
            Logger.Log($"\tSuppressed suffix link for {node.Name} is"
            " already computed to {node.supressedSuffixLink.Name}",
                Logger.LogLevelEnum.Debug);
            return node.supressedSuffixLink;
        }

        var computedSuffixLink = GetSuffixLink(node);
        if (computedSuffixLink.IsTerminal || computedSuffixLink == rootNode)
        {
            Logger.Log($"\tSuffix link for node {node.Name}"
```

```
                " = {computedSuffixLink.Name}. "
                "Terminal: {computedSuffixLink.IsTerminal}."
                " Root: {computedSuffixLink.IsRoot}",
                    Logger.LogLevelEnum.Debug);
                node.supressedSuffixLink = computedSuffixLink;
            }
            else
            {
                // Fallback to suffix link
                node.supressedSuffixLink =
                    GetSuppressedSuffixLink(
                        GetSuffixLink(node));
            }

            Logger.Log($"Suppressed suffix link for node {node.Name}"
                " = {node.supressedSuffixLink.Name}",
                Logger.LogLevelEnum.Debug);
            return node.supressedSuffixLink;
        }

        private struct PatternInfo
        {
            public int PatternNumber { get; set; }

            public int PatternLength { get; set; }
        }
    }


    class Program
    {
        static void Main(string[] args)
        {
            var logLevelFromEnv = Environment
                .GetEnvironmentVariable("LAB5_LOG_LEVEL");
```

```
Logger.LogLevel = string.IsNullOrWhiteSpace(logLevelFromEnv)
    ? Logger.LogLevelEnum.Info
    : (Logger.LogLevelEnum)Enum.Parse(typeof(Logger.LogLevelEnum),
        logLevelFromEnv);
const string firstStepNumber = "1";

var text = Console.ReadLine();
var root = new Node();

// Use dotnet run {stepNumber} for choice step number
// stepNumber == 1 by default
var isFirstStep = args.Length == 0 || args[0] == firstStepNumber;

if (isFirstStep)
{
    Logger.Log("Number of step: 1", Logger.LogLevelEnum.Debug);
    var countsOfStrings = int.Parse(Console.ReadLine());
    for (var i = 0; i < countsOfStrings; ++i)
    {
        root.AddString(Console.ReadLine(), i);
    }
}
else
{
    Logger.Log("Number of step: 2", Logger.LogLevelEnum.Debug);
    root.AddString(Console.ReadLine(), 0);
    Node.Joker = Console.ReadLine()[0];
    Node.JokerException = Console.ReadLine()[0];
}
var result = root.FindPatternsInText(text).ToArray();
if (!result.Any())
{
    Logger.Log("Not found any occurrence of pattern{s} in text!");
}
```

```
                foreach (var occurrence in result
                    .OrderBy(x => x.CharPosition)
                    .ThenBy(x => x.PatternNumber))
                {
                    var resultString = !isFirstStep
                        ? $"{occurrence.CharPosition}"
                        : $"{occurrence.CharPosition} {occurrence.PatternNumber}";

                    Logger.Log(resultString);
                }
            }
        }
    }
```