**Московский государственный технический**

**университет им. Н.Э. Баумана.**

Факультет «Радиотехнический»

Кафедра ИУ5. Курс «Базовые компоненты интернет-технологий»

Отчет по домашнему заданию

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы РТ5-11 |  | преподаватель каф.ИУ5 |
| Стадник Елена |  | Гапанюк Ю. Е. |
|  |  |  |

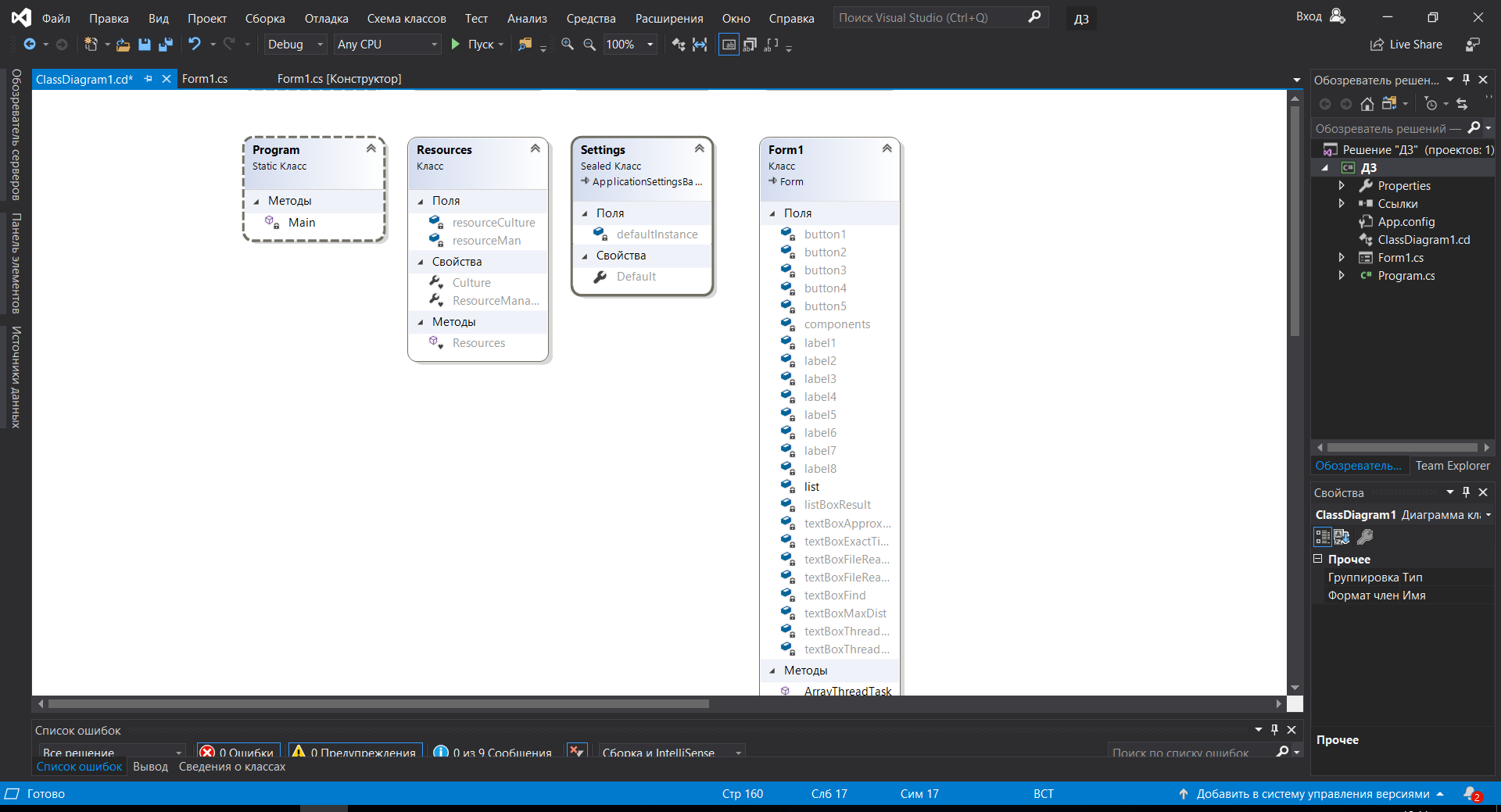
Москва, 2020 г.

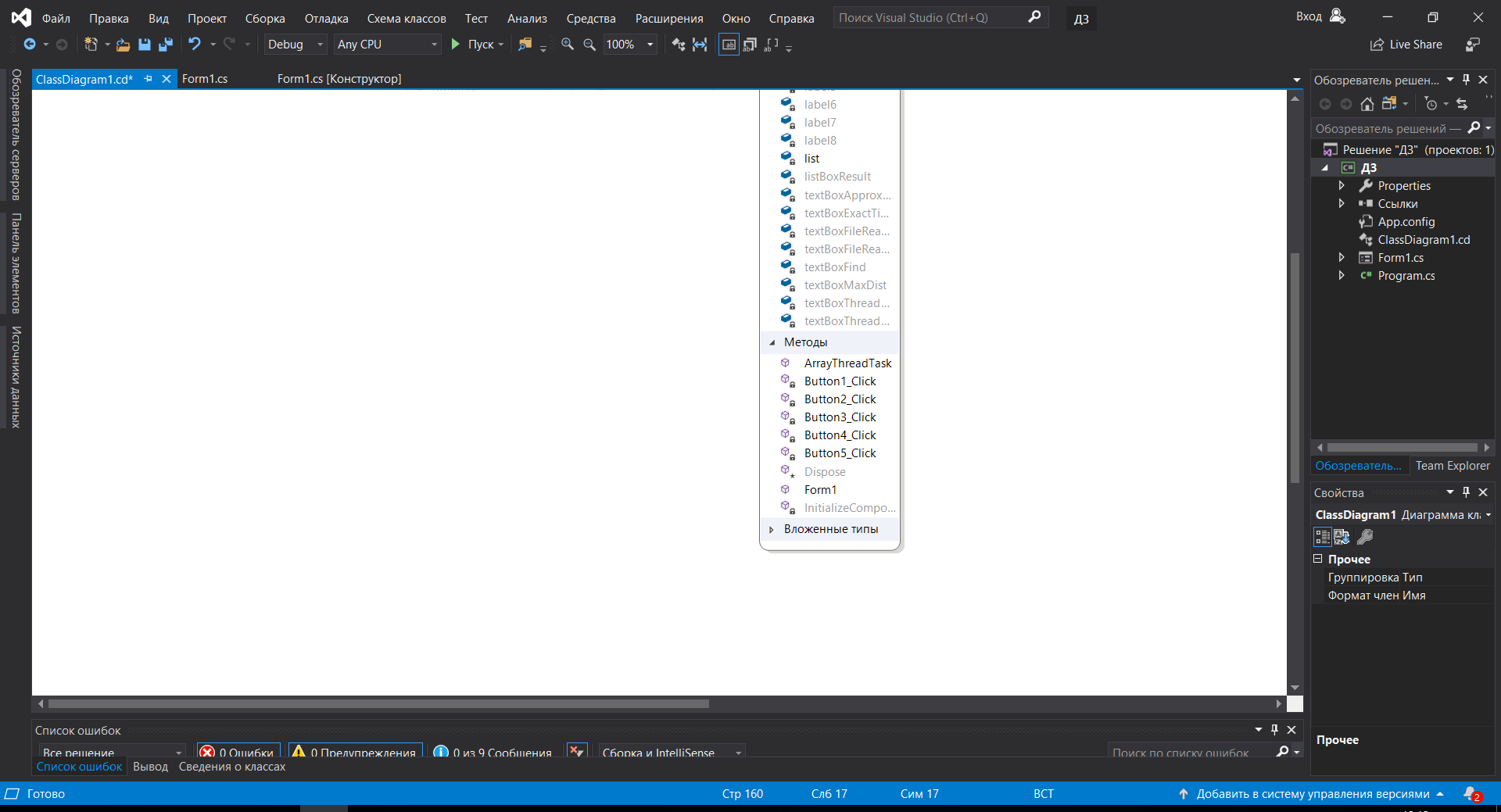
**Домашнее задание**

Разработать программу, реализующую многопоточный поиск в файле.

1. Программа должна быть разработана в виде приложения Windows Forms на языке C#. По желанию вместо Windows Forms возможно использование WPF;
2. В качестве основы используется макет, разработанный в лабораторных работах №4 и №5;
3. Реализуйте функцию поиска с использованием расстояния Левенштейна в многопоточном варианте. Количество потоков для запуска функции поиска вводится на форме в поле ввода (TextBox). В качестве примера используйте проект «Parallel» из примера «Введение в C#».
4. Реализуйте функцию записи результатов поиска в файл отчета. Файл отчета создается в формате .txt или .html. В качестве примера используйте проект «WindowsFormsFiles» (обработчик события кнопки «Сохранение отчета») из примера «Введение в C#».

**Диаграмма классов**





**Текст программы**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using System.IO;

using System.Diagnostics;

using System.Threading.Tasks;

namespace ДЗ

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

List<string> list = new List<string>();

private void Button5\_Click(object sender, EventArgs e)

{

this.Close();

}

private void Button1\_Click(object sender, EventArgs e)

{

OpenFileDialog openFileDialog = new OpenFileDialog()

{

Multiselect = false,

Filter = "текстовые файлы|\*.txt"

};

var stopwatch = new Stopwatch();

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

stopwatch.Start();

string text = File.ReadAllText(openFileDialog.FileName);

char[] separators = new char[] { ' ', '.', ',', '!', '?', '/', '\t', '\n' };

foreach (var strTemp in text.Split(separators))

{

string str = strTemp.Trim();

if (!list.Contains(str))

{

list.Add(str);

}

}

stopwatch.Stop();

this.textBoxFileReadTime.Text = stopwatch.Elapsed.ToString() + " ms";

this.textBoxFileReadCount.Text = list.Count.ToString();

}

else

{

MessageBox.Show("Необходимо выбрать файл");

}

}

private void Button2\_Click(object sender, EventArgs e)

{

//Слово для поиска

string word = this.textBoxFind.Text.Trim();

//Если слово для поиска не пусто

if (!string.IsNullOrWhiteSpace(word) && list.Count > 0)

{

//Слово для поиска в верхнем регистре

string wordUpper = word.ToUpper();

//Временные результаты поиска

List<string> tempList = new List<string>();

Stopwatch t = new Stopwatch(); t.Start();

foreach (string str in list)

{

if (str.ToUpper().Contains(wordUpper))

{

tempList.Add(str);

}

}

t.Stop();

this.textBoxExactTime.Text = t.Elapsed.ToString();

this.listBoxResult.BeginUpdate(); //Очистка списка

this.listBoxResult.Items.Clear();

//Вывод результатов поиска

foreach (string str in tempList)

{

this.listBoxResult.Items.Add(str);

}

this.listBoxResult.EndUpdate();

}

else

{

MessageBox.Show("Необходимо выбрать файл и ввести слово для поиска");

}

}

public static List<ParallelSearchResult> ArrayThreadTask(object paramObj)

{

ParallelSearchThreadParam param = (ParallelSearchThreadParam)paramObj;

//Слово для поиска в верхнем регистре

string wordUpper = param.wordPattern.Trim().ToUpper(); //Результатыпоиска в одном потоке

List<ParallelSearchResult> Result = new List<ParallelSearchResult>();

//Перебор всех слов во временном списке данного потока

foreach (string str in param.tempList)

{

//Вычисление расстояния Дамерау-Левенштейна

int dist = EditDistance.Distance(str.ToUpper(), wordUpper);

//Если расстояние меньше порогового, то слово добавляется в результат

if (dist <= param.maxDist)

{

ParallelSearchResult temp = new ParallelSearchResult()

{

word = str,

dist = dist,

ThreadNum = param.ThreadNum

};

Result.Add(temp);

}

}

return Result;

}

private void Button3\_Click(object sender, EventArgs e)

{

//Слово для поиска

string word = this.textBoxFind.Text.Trim(); //Если слово для поиска не пусто

if (!string.IsNullOrWhiteSpace(word) && list.Count > 0)

{

int maxDist;

if (!int.TryParse(this.textBoxMaxDist.Text.Trim(), out maxDist))

{

MessageBox.Show("Необходимо указать максимальное расстояние"); return;

}

if (maxDist < 1 || maxDist > 5)

{

MessageBox.Show("Максимальное расстояние должно быть в диапазоне от 1 до 5");

return;

}

int ThreadCount;

if (!int.TryParse(this.textBoxThreadCount.Text.Trim(),

out ThreadCount))

{

MessageBox.Show("Необходимо указать количество потоков"); //потоки, на которые разделяется массив слов исходногo файла

return;

}

Stopwatch timer = new Stopwatch();

timer.Start();//Начало параллельного поиска

List<ParallelSearchResult> Result = new List<ParallelSearchResult>();//Результирующий список

//Деление списка на фрагменты для параллельного запуска в потоках

List<MinMax> arrayDivList = SubArrays.DivideSubArrays(0, list.Count, ThreadCount);

int count = arrayDivList.Count;

//Количество потоков соответствует количеству фрагментов массива

//Task - класс, используюшийся для параллельного поиска(задача)

Task<List<ParallelSearchResult>>[] tasks = new Task<List<ParallelSearchResult>>[count];

//Запуск потоков

for (int i = 0; i < count; i++)

{

//Создание временного списка, чтобы потоки не работали параллельно с одной коллекцией

List<string> tempTaskList = list.GetRange(arrayDivList[i].Min,

arrayDivList[i].Max - arrayDivList[i].Min);

tasks[i] = new Task<List<ParallelSearchResult>>

(ArrayThreadTask, new ParallelSearchThreadParam()

{

tempList = tempTaskList,

maxDist = maxDist,

ThreadNum = i,

wordPattern = word

});

//Запуск потока

tasks[i].Start();

}

//ожидание завершения работы всех потоков, чтобы получить результаты поиска

Task.WaitAll(tasks);

timer.Stop();

//Объединение результатов

for (int i = 0; i < count; i++)

{

Result.AddRange(tasks[i].Result);

}

timer.Stop();

//Вывод результатов

//Время поиска

this.textBoxApproxTime.Text = timer.Elapsed.ToString();

this.textBoxThreadCountAll.Text = count.ToString(); //Вычисленное количество потоков

this.listBoxResult.BeginUpdate();//Начало обновления списка результатов

this.listBoxResult.Items.Clear(); //Очистка списка

foreach (var x in Result) //Вывод результатов поиска

{

string temp = x.word + "(расстояние=" + x.dist.ToString() + " поток=" + x.ThreadNum.ToString() + ")";

this.listBoxResult.Items.Add(temp);

}

this.listBoxResult.EndUpdate();//Окончание обновления спискарезультатов

}

else

{

MessageBox.Show("Необходимо выбрать файл и ввести слово для поиска");

}

}

private void Button4\_Click(object sender, EventArgs e)

{

//Имя файла отчета

string TempReportFileName = "Report\_" +

DateTime.Now.ToString("dd\_MM\_yyyy\_hhmmss")

; //Диалог сохранения файла отчета

SaveFileDialog fd = new SaveFileDialog();

fd.FileName = TempReportFileName; fd.DefaultExt

= ".html";

fd.Filter = "HTML Reports|\*.html";

if (fd.ShowDialog() == DialogResult.OK)

{

string ReportFileName = fd.FileName;

//Формирование отчета

StringBuilder b = new StringBuilder();

b.AppendLine("<html>");

b.AppendLine("<head>");

b.AppendLine("<meta http-equiv='Content-Type' content = 'text/html; charset = UTF - 8' /> ");

b.AppendLine("<title>" + "Отчет: " + ReportFileName + "</title>");

b.AppendLine("</head>");

b.AppendLine("<body>");

b.AppendLine("<h1>" + "Отчет: " + ReportFileName + "</h1>");

b.AppendLine("<table border='1'>"); b.AppendLine("<tr>");

b.AppendLine("<td>Время чтения из файла</td>");

b.AppendLine("<td>" + this.textBoxFileReadTime.Text + "</td>");

b.AppendLine("</tr>");

b.AppendLine("<tr>");

b.AppendLine("<td>Количество уникальных слов в файле </ td > ");

b.AppendLine("<td>" + this.textBoxFileReadCount.Text + "</td>");

b.AppendLine("</tr>");

b.AppendLine("<tr>");

b.AppendLine("<td>Слово для поиска</td>");

b.AppendLine("<td>" + this.textBoxFind.Text + "</td>");

b.AppendLine("</tr>");

b.AppendLine("<tr>");

b.AppendLine("<td>Максимальное расстояние для нечеткого поиска </ td > ");

b.AppendLine("<td>" + this.textBoxMaxDist.Text +

"</td>"); b.AppendLine("</tr>");

b.AppendLine("<tr>");

b.AppendLine("<td>Время четкого поиска</td>");

b.AppendLine("<td>" + this.textBoxExactTime.Text + "</td>");

b.AppendLine("</tr>");

b.AppendLine("<tr>");

b.AppendLine("<td>Время нечеткого поиска</td>");

b.AppendLine("<td>" + this.textBoxApproxTime.Text + "</td>");

b.AppendLine("</tr>");

b.AppendLine("<tr valign='top'>");

b.AppendLine("<td>Результаты поиска</td>");

b.AppendLine("<td>");

b.AppendLine("<ul>");

foreach (var x in this.listBoxResult.Items)

{

b.AppendLine("<li>" + x.ToString() + "</li>");

}

b.AppendLine("</ul>");

b.AppendLine("</td>");

b.AppendLine("</tr>");

b.AppendLine("</table>");

b.AppendLine("</body>");

b.AppendLine("</html>");

//Сохранение файла

File.AppendAllText(ReportFileName, b.ToString());

MessageBox.Show("Отчет сформирован. Файл: " +

ReportFileName);

}

}

public static class EditDistance

{

/// Вычисление расстояния Дамерау-Левенштейна

public static int Distance(string str1Param, string str2Param)

{

if ((str1Param == null) || (str2Param == null)) return -1;

int str1Len = str1Param.Length; int str2Len =

str2Param.Length;

//Если хотя бы одна строка пустая, возвращается длина другой строки

if ((str1Len == 0) && (str2Len == 0))

{

return 0;

}

if (str1Len == 0)

{

return str2Len;

}

if (str2Len == 0)

{

return str1Len;

}

//Приведение строк к верхнему регистру

string str1 = str1Param.ToUpper();

string str2 = str2Param.ToUpper(); //Объявление матрицы

int[,] matrix = new int[str1Len + 1, str2Len + 1];

//Инициализация нулевой строки и нулевого столбца матрицы

for (int i = 0; i <= str1Len; i++)

{

matrix[i, 0] = i;

}

for (int j = 0; j <= str2Len; j++)

{

matrix[0, j] = j;

}

//Вычисление расстояния Дамерау-Левенштейна

for (int i = 1; i <= str1Len; i++)

{

for (int j = 1; j <= str2Len; j++)

{

//Эквивалентность символов, переменная symbEqual соответствует m(s1[i], s2[j])

int symbEqual = ((str1.Substring(i - 1, 1) == str2.Substring(j - 1, 1)) ? 0 : 1);

int ins = matrix[i, j - 1] + 1;

//Добавление

int del = matrix[i - 1, j] + 1; //Удаление

int subst = matrix[i - 1, j - 1] + symbEqual; //Замена

//Элементматрицы вычисляется как минимальный из трех случаев

matrix[i, j] = Math.Min(Math.Min(ins, del), subst);

//Дополнение Дамерау по перестановке соседних символов

if ((i > 1) && (j > 1) &&

(str1.Substring(i - 1, 1) == str2.Substring(j - 2, 1)) &&

(str1.Substring(i - 2, 1) == str2.Substring(j - 1, 1)))

{

matrix[i, j] = Math.Min(matrix[i, j], matrix[i - 2, j -

2] + symbEqual);

}

}

}

//Возвращается нижний правый элемент матрицы

return matrix[str1Len, str2Len];

}

}

/// Результаты параллельного поиска

//содержит входной массив слов и слово для поиска, максимальное расстояние для нечеткого поиска и номер потока

public class ParallelSearchResult

{

/// Найденное слово

public string word { get; set; }

/// Расстояние

public int dist { get; set; }

/// Номер потока

public int ThreadNum { get; set; }

}

/// Параметры которые передаются в поток для параллельного поиска

class ParallelSearchThreadParam

{

/// Массив для поиска

public List<string> tempList { get; set; }

/// Слово для поиска

public string wordPattern { get; set; }

/// Максимальное расстояние для нечеткого поиска

public int maxDist { get; set; }

/// Номер потока

public int ThreadNum { get; set; }

}

/// Хранение минимального и максимального значений диапазона

public class MinMax

{

public int Min { get; set; }

public int Max { get; set; }

public MinMax(int pmin, int pmax)

{

this.Min = pmin;

this.Max = pmax;

}

}

//Для деления массива на подмассивы

public static class SubArrays

{

/// Деление массива на последовательности(подмассивы)

/// <param name="beginIndex">Начальный индекс массива</param>

/// <param name="endIndex">Конечный индекс массива</param>

/// <param name="subArraysCount">Требуемое количество подмассивов</param>

/// <returns>Список пар с индексами подмассивов</returns>

public static List<MinMax>

DivideSubArrays(int beginIndex, int endIndex, int subArraysCount)

{

//Результирующий список пар с индексами подмассивов

List<MinMax> result = new List<MinMax>();

//Если число элементов в массиве слишком мало для деления, то возвращается массив целиком

if ((endIndex - beginIndex) <= subArraysCount)

{

result.Add(new MinMax(0, (endIndex - beginIndex)));

}

else

{

//Размер подмассива

int delta = (endIndex - beginIndex) / subArraysCount;

//Начало отсчета

int currentBegin = beginIndex;

//Пока размер подмассива укладывается в оставшуюся последовательность

while ((endIndex - currentBegin) >= 2 \* delta)

{

//Формируем подмассив на основе начала последовательности

result.Add(new MinMax(currentBegin, currentBegin + delta));

//Сдвигаем начало последовательности вперед на размер подмассива

currentBegin += delta;

}

//Оставшийся фрагмент массива

result.Add(new MinMax(currentBegin, endIndex));

}

//Возврат списка результатов

return result;

}

}

}

}

