Минобрнауки России
ФГБОУ ВПО «НИУ МЭИ» АВТИ
Кафедра математического и компьютерного моделирования

Лабораторная работа №6

Технологии программирования

"Разработка программы для работы с графом на языке C#"

Работу выполнил:
Солонин Егор А-14-19
Вариант 14
Работу принял:
Князев А. В.

Москва 2021

Задание:

Разработать абстрактный класс для представления неориентированного графа и поиска всех циклов или путей между двумя заданными вершинами.
Класс должен включать объект класса "Стек" для представления стека.
Класс должен включать как минимум следующие методы:
- конструктор;
- метод для построения всех путей или циклов, удовлетворяющих некоторому условию;
- абстрактный метод для поиска ещё не обработанной смежной вершины;
- абстрактный метод для проверки соответствия построенного пути или цикла заданным условиям.
Используя описанный класс как базовый разработать производный класс для решения задачи о построении всех путей или циклов, удовлетворяющих конкретным условиям задания.
Производный класс должен содержать описание графа в формате, указанном в задании.
В производном классе должны быть переопределены виртуальные методы базового класса с учётом конкретного задания.
Необходимо разработать класс "Стек" для представления стека.
Нельзя использовать никакие библиотеки, кроме встроенных в язык C#. Не должны использоваться коллекции.
Разработать программу, работающую в среде Visual Studio на основе Windows Forms и реализующую конкретное задание.
Программа должна обеспечивать ввод описания графа из файла и с клавиатуры.
Программа должна обеспечивать редактирование и сохранение описания графа.
Программа должна обеспечивать представление исходного графа и результатов в графическом виде.
Программа должна сначала найти и запомнить все пути или циклы, а затем показывать их по запросу.

В описаниях заданий используются следующие обозначения:
- стек_спис – стек на основе связанного списка;
- граф_матр – для описания графа используется матрица смежности;

14	Солонин Е.В.	Найти все пути между двумя вершинами, проходящие хотя бы через одну периферийную вершину (стек_спис, граф_матр).

Описание работы программы:

При открытии приложения пользователь имеет возможность:

1. Задать вершины графа заглавными буквами английского алфавита через пробел или запятую
2. Задать ребра графа парой заглавных букв английского алфавита через пробел или запятую
3. Задать стартовую вершину, для поиска путей в графе
4. Задать конечную вершину, для поиска путей в графе
5. Сохранить описание графа в файл
6. Загрузить описание графа из файла
7. Получить визуализацию заданного графа
8. Осуществить поиск всех путей между двумя вершинами, проходящие хотя бы через одну периферийную вершину
9. Получив таблицу всех путей, отобразить один из путей, выбрав соответствующую ячейку в таблице dataGridView

Желтый цвет вершины – периферийная вершина
Бежевый цвет вершины – непериферийная вершина
Красный цвет ребра – ребро из пути
Фиолетовый цвет ребра – обычное ребро из графа

Алгоритмы операций на псевдокоде:

Абстрактный класс `Graph` имеет:

- `Stack<char> currentPath_stack` – стек вершин
- `protected const int N` – максимальное число вершин в графе
- Конструктор
- `virtual string[] FindAllPaths(char start, char finish, int[,] AdjacencyMatrix)` – виртуальный метод поиска всех путей в графе
    - `start` – начальная вершина
    - `finish` – конечная вершина
    - `AdjacencyMatrix` – матрица смежности
- `string GetPath(Stack<char> stack, char finish)` – построение пути по стеку вершин
    - `finish` – конечная вершина
    - `stack` – стек вершин
- `abstract char FindConnectable(char curr, bool[] visits, int startSearch)` – абстрактный метод для поиска связной еще не посещённой вершины
    - `curr` – текущая вершина
    - `visits` – массив посещений вершин
    - `startSearch` – индекс стартовой вершина для обхода
- `abstract string[] FindValidPaths(string[] allPaths, char[] PerVertexes)` – абстрактный метод для поиска всех путей удовлетворяющих некоторому условию
    - `allPaths` – все пути в графе
    - `PerVertexes` – периферийные вершины
- `abstract void FromString(string data)` - абстрактный метод для заполнения матрицы смежности
    - `data` – вершины и ребра в одной строке

Дочерний класс `GraphMatrix : Graph` имеет:

- `public int[,] AdjacencyMatrix` – матрица смежности
- `char[] perVertexes` – периферийные вершины
- Конструктор
- `void Clear()` – очистить матрицу смежности
- `private int[,] GetDistMatrix()` – метод, позволяющий получить матрицу расстояний
- `private int[] GetEccentricity()` – метод, позволяющий получить массив эксцентриситетов графа
- `private char[] GetPeriphericVertexes()` – метод, позволяющий получить массив периферийных вершин
- `override void FromString()` – перегрузка метода из родительского класса
- `override string[] FindAllPaths()` - перегрузка метода из родительского класса
- `override char FindConnectable()` - перегрузка метода из родительского класса
- `override string[] FindValidPaths()` - перегрузка метода из родительского класса

Алгоритмы:

`FindAllPaths()` - метод поиска всех путей в графе:

1. `Stack<string>` *`allPaths`* – заводим стек куда будем добавлять все найденный пути
2. `bool[]` *`visits`* – заводим массив посещений вершин и отмечаем все вершины как не посещенные
3. Добавляем в стек *`currentPath_stack`* точку старта пути *`start`*
4. Отмечаем вершину start посещенной
5. *`startSearch`* = *`0`* - индекс начала поиска вершины
6. Пока *`currentPath_stack`* не пуст
    a. Получаем текущую вершину *`curr`* с вершины стека *`currentPath_stack`*
    b. Получаем следующую не посещённую смежную вершину next с помощью метода *`FindConnectable`*

c. Если *next* - вершина конца пути, добавляем путь в стек *allPaths*
d. Иначе если следующая вершина найдена
    i. Добавляем в стек *next*
    ii. Отмечаем *next* посещенной
    iii. Сбрасываем индекс начала поиска вершины *startSearch*
e. Если следующая вершина не найдена или вершина совпадает с концом пути
    i. Увеличиваем индекс начала поиска вершины *startSearch*
    ii. Отмечаем *curr* посещенной
    iii. Удаляем из *currentPath_stack* вершину

7. Конец цикла, возврат *allPaths* в виде массива

FindConnectable() - поиск связной еще не посещённой вершины

1. *curr* – текущая вершина
2. *visits* – массив посещений вершин
3. *startSearch* – индекс стартовой вершина для обхода
4. Цикл по *i* со стартовой вершины до N
    1. Если нет ребра между вершиной *curr* и *i*, переходим к следующей вершине
    2. Если вершина *i* посещенная, переходим к следующей вершине
    3. Иначе возвращаем вершину
5. Возвращаем ‘!’ – не нашли подходящих вершин

GetDistMatrix() - метод, позволяющий получить матрицу расстояний

1. Заводим матрицу расстояний *DistMatrix*, копируя ее из матрицы смежности
2. Заменяем все нулевые недиагональные элементы на числа, которые точно превосходят максимальное расстояние в графе, т.е. на N+1(максимальное число вершин в графе)
3. По алгоритму Флойда–Уоршелла заполняем матрицу расстояний
4. Возврат *DistMatrix*

GetEccentricity() – метод, позволяющий получить массив эксцентриситетов графа

1. Заводим массив эксцентриситетов
2. Находим максимальный элемент в каждой строке матрицы расстояний и добавляем его в массив эксцентриситетов

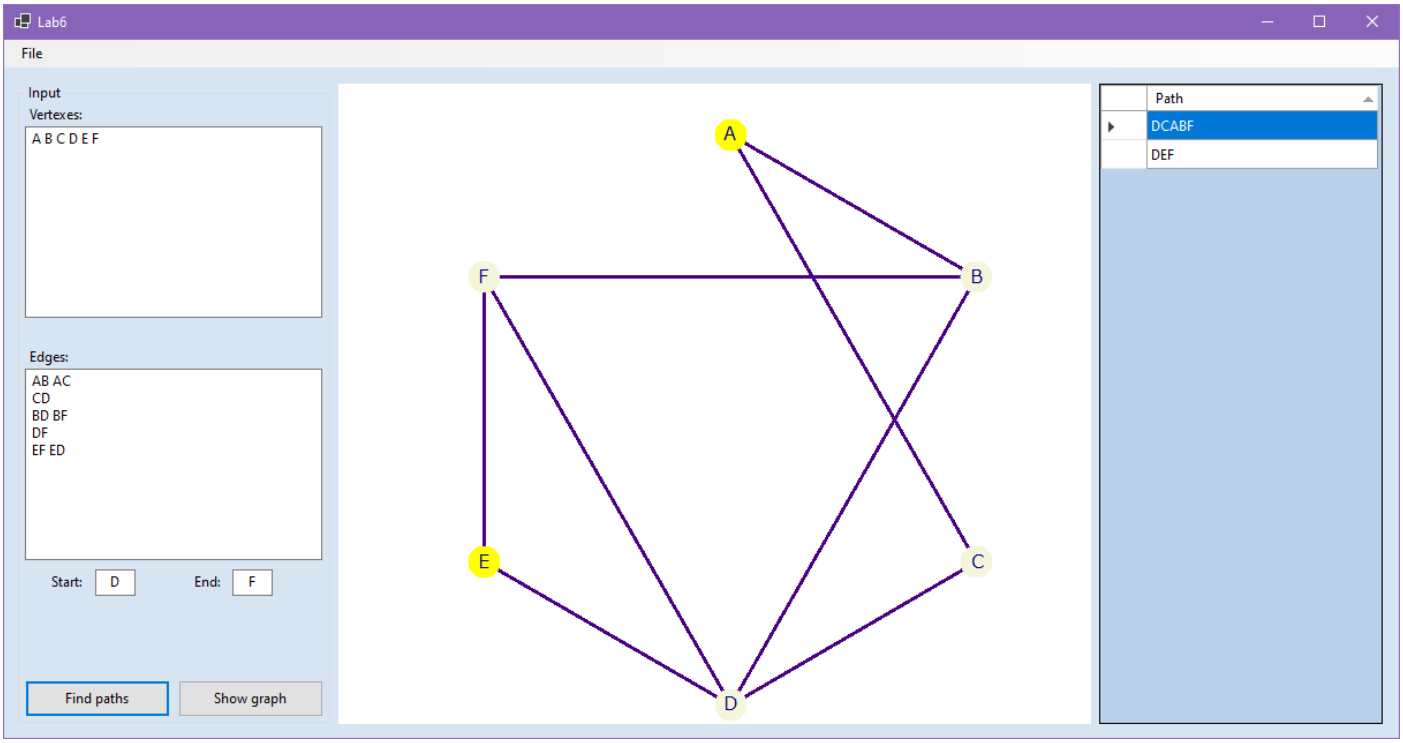
GetPeriphericVertexes() – метод, позволяющий получить массив периферийных вершин

1. Заводим массив периферийных вершин
2. Добавляем туда все вершины с максимальным эксцентриситетом

FindValidPaths() - поиска всех путей, которые содержат хотя бы одну периферийную вершину

1. *allPaths* – массив всех путей
2. *PerVertexes* – массив периферийных вершин
3. Заводим стек *PathsWithCondition* – пути, содержащие периферийную вершину
4. Для каждого *path* в *allPaths*
    1. Для каждой *perVertex* в *PerVertexes*
        i. Если *path* содержит *perVertex*
            1. Добавляем путь в *PathsWithCondition*
            2. Переходим к сдедующему *path*
5. Возврат *PathsWithCondition* в виде массива

Тесты работы программы:



Возьмем граф, изображенный на рисунке.

Требуется найти все пути из вершины D в вершину F, проходящие хотя бы через одну из периферийных вершин.

Периферийные вершины: A, E

Будут найдены пути:

D-C-A-B-F

D-E-F

Входное условие	Правильные классы эквивалентности	Неправильные классы эквивалентности
Число периферийных вершин в пути	>0 *(1)*	Ни одной *(2)*

X1	X2	F
0	0	0
0	1	1
1	0	1
1	1	1

X1 – путь проходит через вершину A

X2 – путь проходит через вершину E

Таким образом, здесь должен быть один тест, где истинно хотя бы одно из условий, и один тест, в котором ложны оба условия.

1. Путь проходит через A или E, и программа должна находить пути (причём их должно быть как минимум два), удовлетворяющие этим условиям

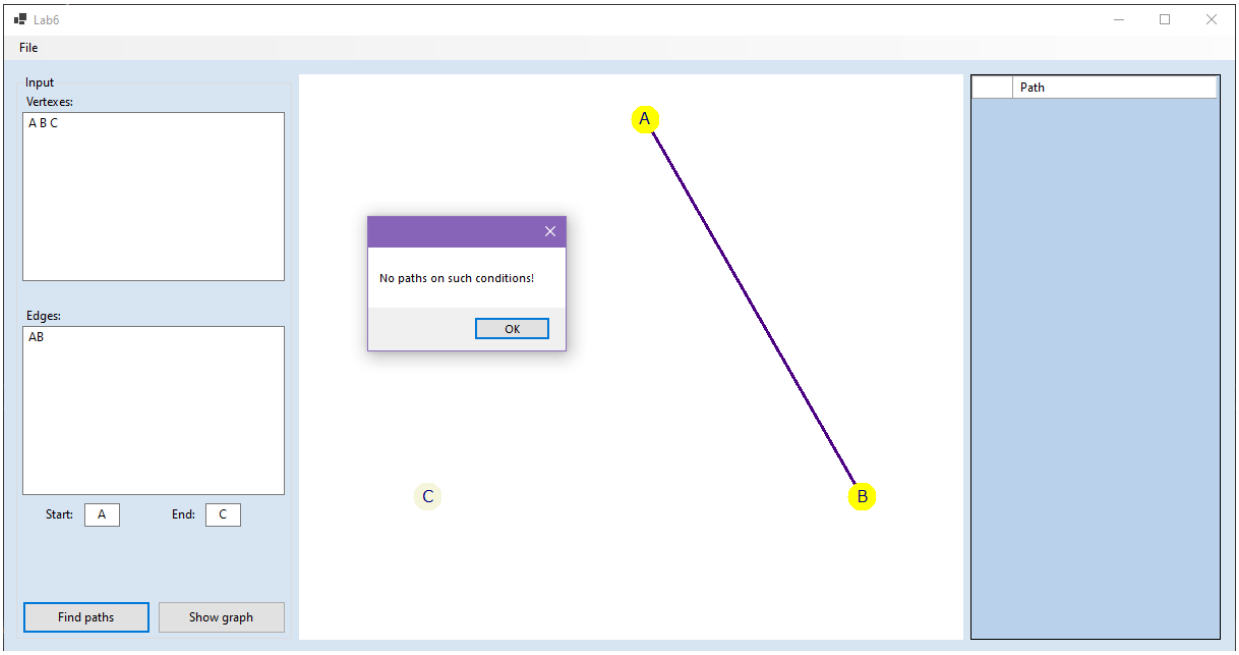
D-C-A-B-F
D-E-F

2. Существует путь, для которых одновременно не выполняется условие X1 и X2, и программа отвергает этот путь.

Например путь: D-F или D-B-F

Таким образом, данный граф с указанным заданием является полным тестом.

<u>Тест, когда между заданными вершинами не существует пути:</u>



Листинг программы:

```
class Node<T>
    {
        public Node(T data)
        {
            Data = data;
        }
        public T Data { get; set; }
        public Node<T> Next { get; set; }
    }
class Stack<T>
    {
        Node<T> head;
        int count;
        private InvalidOperationException err411 = new InvalidOperationException("Error 411: Stack
is empty");
        public bool IsEmpty() => count == 0;
        public int Count() => count;
        public void Push(T item)
        {
            Node<T> node = new Node<T>(item);
            node.Next = head;
            head = node;
            count++;
        }
        public T Pop()
        {
            if (IsEmpty())
                throw err411;
            Node<T> temp = head;
            head = head.Next;
            count--;
            return temp.Data;
        }
        public T Peek() => IsEmpty() ? throw err411 : head.Data;
        public Stack<T> Reverse()
        {
            Stack<T> copy = new Stack<T>();
            Node<T> node = head;
            for (; node != null; node = node.Next)
                copy.Push(node.Data);
            return copy;
        }
        public T[] ToArray()
        {
            T[] arr = new T[count];
            int i = 0;
            Node<T> node = head;
            for(; node != null; node = node.Next, i++)
                arr[i] = node.Data;
            return arr;
        }
    }
abstract class Graph
       {
             Stack<char> currentPath_stack = new Stack<char>();
             protected const int N = 26;
             protected const int ASCII_SHIFT = 65;
             public Graph(string verts, string edges)
             {
                    FromString(verts + ';' + edges);
             }

             public virtual string[] FindAllPaths(char start, char finish, int[,] AdjacencyMatrix)
       {
                    Stack<string> allPaths = new Stack<string>();
                    bool[] visits = new bool[N];
```

```
                for (int i = 0; i < N; i++) visits[i] = false;

                currentPath_stack.Push(start);
                visits[start - ASCII_SHIFT] = true;
                int startSearch = 0;

                while (!currentPath_stack.IsEmpty())
                {
                        char curr = (char)(currentPath_stack.Peek() - ASCII_SHIFT);
                        char next = FindConnectable(curr, visits, startSearch);

                        if (next == finish)
                        {
                                string path = GetPath(currentPath_stack, finish);
                                allPaths.Push(path);
                        }
                        else if (next != '!')
                        {
                                currentPath_stack.Push(next);
                                visits[next - ASCII_SHIFT] = true;
                                startSearch = 0;
                        }

                        if (next == '!' || next == finish)
                        {
                                startSearch = curr + 1;
                                visits[curr] = false;
                                currentPath_stack.Pop();
                        }
                }
                return allPaths.ToArray();
        }

        private string GetPath(Stack<char> stack, char finish)
        {
                string path = "";
                Stack<char> temp = stack.Reverse();
                for (int i = 0; !temp.IsEmpty(); i++)
                        path += temp.Pop();
                path += finish;
                return path;
        }

        public abstract char FindConnectable(char curr, bool[] visits, int startSearch);

        public abstract string[] FindValidPaths(string[] allPaths, char[]
AppropriateVertexes);

        public abstract void FromString(string data);
    }

    class GraphMatrix : Graph
    {
        public int[,] AdjacencyMatrix = new int[N, N];
        char[] perVertexes;
        public GraphMatrix(string verts, string edges) : base(verts, edges) {
                perVertexes = GetPeripheralVertexes();
        }

        public char[] PerVertexes
        {
                get { return perVertexes; }
        }
        public void Clear()
        {
                for (int i = 0; i < N; i++)
                        for (int j = 0; j < N; j++)
                                AdjacencyMatrix[i, j] = -1;
        }
```

```
		public override void FromString(string data)
		{
			Clear();
			string[] temp = data.Split(';');
			string vertexesData = temp[0];
			string edgesData = temp[1];

			char[] separators = new char[] { ' ', '\r', '\n', ',' };
			string[] vertexes = vertexesData.Split(separators,
StringSplitOptions.RemoveEmptyEntries);
			string[] edges = edgesData.Split(separators,
StringSplitOptions.RemoveEmptyEntries);


			for (int i = 0; i < vertexes.Length; i++)
				for (int j = 0; j < vertexes.Length; j++)
				{
					int row = vertexes[i][0] - ASCII_SHIFT;
					int col = vertexes[j][0] - ASCII_SHIFT;

					AdjacencyMatrix[row, col] = 0;
					AdjacencyMatrix[col, row] = 0;
				}

			foreach (var edge in edges)
			{
				int start = edge[0] - ASCII_SHIFT;
				int finish = edge[1] - ASCII_SHIFT;

				AdjacencyMatrix[start, finish] = 1;
				AdjacencyMatrix[finish, start] = 1;
			}
		}

		public override string[] FindAllPaths(char start, char finish, int[,] adj = null)
		{
			string[] allPaths = base.FindAllPaths(start, finish, AdjacencyMatrix);
			return FindValidPaths(allPaths, perVertexes);
			//return allPaths;
		}

		private int[,] GetDistMatrix()
	{
			int[,] DistMatrix = new int[N, N];
			Array.Copy(AdjacencyMatrix, DistMatrix, N * N);

			for (int k = 0; k < N; ++k)
				for (int i = 0; i < N; ++i)
					if (DistMatrix[k, i] == 0 && i != k)
						DistMatrix[k, i] = N + 1;

			for (int k = 0; k < N; ++k)
				for (int i = 0; i < N; ++i)
					for (int j = 0; j < N; ++j)
						if (DistMatrix[i, k] != -1 && DistMatrix[k, j] != -1 &&
DistMatrix[i, j] != -1)
							DistMatrix[i, j] = Math.Min(DistMatrix[i, j],

	DistMatrix[i, k] + DistMatrix[k, j]);

			return DistMatrix;
		}

		private int[] GetEccentricity()
	 {
			int[,] DistMatrix = GetDistMatrix();
			int[] ecc = new int[N];
```

```
            for (int i = 0; i < N; ++i)
        {
                int max = 0;
                for (int j = 0; j < N; ++j)
                    if (DistMatrix[i, j] > max && DistMatrix[i, j] <= N)
                        max = DistMatrix[i, j];
                ecc[i] = max;
            }

            return ecc;

        }

        private char[] GetPeriphericVertexes()
        {
            int[] ecc = GetEccentricity();
            int max = 0;
            int PerVertexesCount = 1;
            for (int j = 0; j < N; ++j)
            {
                if (ecc[j] > max)
                {
                    max = ecc[j];
                    PerVertexesCount = 1;
                }
                else if (ecc[j] == max)
                    PerVertexesCount++;
            }

            char[] PerVertexes = new char[PerVertexesCount];
            int index = 0;

            for (int j = 0; j < N; ++j)
                if (ecc[j] == max)
                {
                    PerVertexes[index] = (char)(j + ASCII_SHIFT);
                    index++;
                }
            return PerVertexes;
        }

        public override char FindConnectable(char curr, bool[] visits, int startSearch)
        {
            for (int i = startSearch; i < N; i++)
            {
                if (AdjacencyMatrix[curr, i] != 1) continue;
                if (visits[i]) continue;
                return (char)(i + ASCII_SHIFT);
            }
            return '!';
        }

        public override string[] FindValidPaths(string[] allPaths, char[] PerVertexes)
        {
            Stack<string> PathsWithCondition = new Stack<string>();

            foreach (string path in allPaths)
                foreach (char perVertex in PerVertexes)
                    if (path.Contains(perVertex))
                    {
                        PathsWithCondition.Push(path);
                        break;
                    }

            return PathsWithCondition.ToArray();
        }
}
```

```
public partial class Form1 : Form
    {
        Visualization vis;
        GraphMatrix graph;


        public Form1()
        {
            InitializeComponent();
        }

        private void openToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                string filename = openFileDialog1.FileName;
                string fileText = System.IO.File.ReadAllText(filename);

                string[] data = fileText.Split(';');

                this.vertexesData.Text = data[0];
                this.edgesData.Text = data[1];
            }
        }

        private void saveToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (saveFileDialog1.ShowDialog() == DialogResult.OK)
            {
                string filename = saveFileDialog1.FileName;
                string data = vertexesData.Text + ';' + edgesData.Text;
                System.IO.File.WriteAllText(filename, data);
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (FieldsValid())
            {
                PathsTable.Rows.Clear();
                string verts = vertexesData.Text;
                string edges = edgesData.Text;
                char start = startVertex.Text[0];
                char end = endVertex.Text[0];

                graph = new GraphMatrix(verts, edges);

                string[] Paths = graph.FindAllPaths(start, end);

                if (Paths.Length == 0) MessageBox.Show("No paths on such conditions!");

                foreach (string path in Paths)
                    PathsTable.Rows.Add(path);
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            if (FieldsValid())
            {
                graph = new GraphMatrix(vertexesData.Text, edgesData.Text);
                vis = new Visualization(this.panel1, vertexesData.Text, edgesData.Text, graph !=
null ? graph.PerVertexes : null);
            }
        }

        private void PathsTable_CellClick(object sender, DataGridViewCellEventArgs e)
        {
            if (e.RowIndex != -1 && vis != null)
```

```
            {
                string path = PathsTable.Rows[e.RowIndex].Cells["Path"].Value.ToString();
                vis.HighlightPath(path);
            }

        }

        private bool FieldsValid()
        {
            string verts = vertexesData.Text;
            string edges = edgesData.Text;
            char start = startVertex.Text[0];
            char end = endVertex.Text[0];

            char[] separators = new char[] { ' ', '\r', '\n', ',' };

            try
            {
                foreach (char vert in verts)
                    if (!isSeparator(separators, vert) && (vert < 'A' || vert > 'Z'))
                        throw new InvalidOperationException("Invalid vertexes construction");

                foreach (char edgeSymb in edges)
                    if (!isSeparator(separators, edgeSymb))
                        if(!verts.Contains(edgeSymb))
                            throw new InvalidOperationException("There is no vertex for the one of
the edges");

                if (!verts.Contains(start))
                    throw new InvalidOperationException("There is no such start vertex");

                if (!verts.Contains(end))
                    throw new InvalidOperationException("There is no such end vertex");

                string[] edges_arr = edges.Split(separators, StringSplitOptions.RemoveEmptyEntries);
                foreach (var edge in edges_arr)
                    if (edge.Length != 2)
                        throw new InvalidOperationException("Inappropriate edges' construction");

            }
            catch (InvalidOperationException msg)
            {
                MessageBox.Show(msg.Message);
                return false;
            }
            return true;
        }

        private bool isSeparator(char[] separators, char item)
        {
            foreach (char lex in separators)
                if (lex == item) return true;
            return false;
        }

    }
class Point
    {
        double x;
        double y;
        string name;
        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public string Name
        {
```

```
            get { return name; }
            set { name = value; }
        }

        public double X
        {
            get { return x; }
        }

        public double Y
        {
            get { return y; }
        }


    }

    class Edge
    {
        Point start;
        Point end;

        public Edge(Point start, Point end)
        {
            this.start = start;
            this.end = end;
        }

        public Point Start
        {
            get { return start; }
        }

        public Point End
        {
            get { return end; }
        }
    }

    class Visualization
    {
        private Graphics g;
        private Point Center;
        private int radius;
        private int VertexR = 14;
        private Point[] coordsV;
        private Edge[] coordsE;
        private char[] perVerts;

        public Visualization(System.Windows.Forms.Panel panel1, string verts, string edges, char[]
perVerts)
        {
            this.perVerts = perVerts;
            g = panel1.CreateGraphics();
            g.Clear(Color.White);
            SetDimentions(panel1);
            coordsV = GetVertsCoords(verts);
            coordsE = GetEdgesCoords(edges, coordsV);
            DrawEdges(coordsE, Color.Indigo);
            DrawVerts(coordsV);
        }

        private void SetDimentions(System.Windows.Forms.Panel panel1)
        {
            Center = new Point(panel1.Width / 2, panel1.Height / 2);
            radius = panel1.Height / 2 - 30;
        }

        private Point[] GetVertsCoords(string verts)
```

```
        {
            char[] separators = new char[] { ' ', '\r', '\n', ',' };
            string[] vertexes = verts.Split(separators, StringSplitOptions.RemoveEmptyEntries);

            int verts_count = vertexes.Length;
            double angle = 2 * Math.PI / verts_count;
            Point[] coords = new Point[verts_count];

            for (int i = 0; i < verts_count; ++i)
            {
                coords[i] = new Point(Center.X + radius * Math.Sin(angle * i), Center.Y - radius *
Math.Cos(angle * i));
                coords[i].Name = vertexes[i];
            }
            return coords;

        }

        private Edge[] GetEdgesCoords(string edgesStr, Point[] vertsCoords)
        {
            char[] separators = new char[] { ' ', '\r', '\n', ',' };
            string[] edges = edgesStr.Split(separators, StringSplitOptions.RemoveEmptyEntries);

            int edges_count = edges.Length;
            Edge[] coords = new Edge[edges_count];

            for (int i = 0; i < edges_count; ++i)
            {
                Point start = GetCoordsByName(vertsCoords, edges[i][0].ToString());
                Point end = GetCoordsByName(vertsCoords, edges[i][1].ToString());
                coords[i] = new Edge(start, end);
            }
            return coords;

        }

        private Point GetCoordsByName(Point[] coords, string Name)
        {
            foreach(Point coord in coords)
                if (coord.Name == Name) return coord;
            return null;
        }

        private void DrawVerts(Point[] coords)
        {
            Brush vertexColor;
            Brush fontColor = new SolidBrush(Color.Navy);
            Font font = new Font("Verdana", 12);
            StringFormat stringFormat = new StringFormat() { Alignment = StringAlignment.Center,
LineAlignment = StringAlignment.Center };

            foreach (Point coord in coords)
            {
                if(perVerts!=null && Array.IndexOf(perVerts, coord.Name[0]) != -1)
                    vertexColor = new SolidBrush(Color.Yellow);
                else vertexColor = new SolidBrush(Color.Beige);
                g.FillEllipse(vertexColor, (int)coord.X, (int)coord.Y, VertexR * 2, VertexR * 2);
                g.DrawString(coord.Name, font, fontColor, new RectangleF((float)coord.X,
(float)coord.Y, VertexR * 2, VertexR * 2), stringFormat);
            }

        }

        private void DrawEdges(Edge[] coords, Color clr)
        {
            Pen EdgeColor = new Pen(clr, 3);

            foreach (Edge coord in coords)
```

```
                g.DrawLine(EdgeColor, (int)coord.Start.X + VertexR, (int)coord.Start.Y + VertexR,
(int)coord.End.X + VertexR, (int)coord.End.Y + VertexR);
        }

        public void HighlightPath(string path)
        {
            Edge[] highlightedEdges = new Edge[path.Length - 1];
            for (int i = 0; i < path.Length - 1; ++i)
            {
                Point start = GetCoordsByName(coordsV, path[i].ToString());
                Point end = GetCoordsByName(coordsV, path[i + 1].ToString());
                highlightedEdges[i] = new Edge(start, end);
            }
            DrawEdges(coordsE, Color.Indigo);
            DrawEdges(highlightedEdges, Color.Red);
            DrawVerts(coordsV);
        }
    }
```