**Разработка алгоритма и его программная реализация для нахождения кратчайшего пути в неориентированном графе**

# Аннотация

В предложенной работе рассматривается задача нахождения кратчайшего пути в неориентированном графе. Для решения выбран алгоритм Дейкстры. Разработана программа с пользовательским интерфейсом, решающая поставленную задачу. Данная программа может быть доработана и расширена для применения в различных практических областях.

Содержание

[Аннотация 2](#_Toc2783410)

[ВВЕДЕНИЕ 4](#_Toc2783411)

[1. Анализ задачи и разработка требований 5](#_Toc2783412)

[2. Выбор алгоритма для реализации системы 6](#_Toc2783413)

[3. Проектирование приложения 8](#_Toc2783414)

[3.1. Проектирование интерфейса пользователя 8](#_Toc2783415)

[3.2. Проектирование расчётного модуля 9](#_Toc2783416)

[3.3. Проектирование промежуточного модуля 9](#_Toc2783417)

[4. Реализация программной системы 10](#_Toc2783418)

[4.1. Пользовательский интерфейс 10](#_Toc2783419)

[4.2. Модуль подготовки данных 13](#_Toc2783420)

[4.3. Модуль реализации алгоритма 14](#_Toc2783421)

[5. Тестирование и отладка программной системы 15](#_Toc2783422)

[6. Руководство пользователя 18](#_Toc2783423)

[Заключение 21](#_Toc2783424)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 22](#_Toc2783425)

[Приложение 1 23](#_Toc2783426)

# ВВЕДЕНИЕ

В работе рассматривается задача нахождения кратчайшего пути между двумя вершинами произвольного неориентированного графа.

**Задачи исследования:**

1. Разработать архитектуру системы, решающей поставленную задачу.
2. Проанализировать имеющиеся алгоритмы решения задачи и выбрать оптимальный.
3. Спроектировать структуру программного обеспечения, разделив её на модули и установив их взаимосвязь.
4. Реализовать программное обеспечение разработанной системы.
5. Выполнить серию экспериментов с различными тестовыми данными.

# Анализ задачи и разработка требований

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. Данная задача известна и под другими названиями: задача о минимальном пути, задача о дилижансе. Постановка задачи: в графе, где каждому ребру приписано некоторое числовое значение (вес, длина, стоимость) найти путь между двумя заданными вершинами с минимальной суммой весов. В общем случае рассматривается неориентированный граф, хотя есть задачи для ориентированных графов, для графов с изменяющейся структурой и т.п. На сегодняшний день известно множество алгоритмов для решения классической задачи с весьма хорошими оценками вычислительной сложности (примерно ≈1.5–2n3 операций для графа с числом вершин равным n). Наиболее известными алгоритмами решения классической задачи о кратчайшем пути являются алгоритмы Дейкстры, Флойда и Данцига.[5,6] В данной работе для реализации был взят алгоритм Дейкстры, как имеющий наилучшую оценку сложности.[7]

В связи с тем, что задача достаточно хорошо и давно известна, а сам алгоритм прост и надёжен, количество программ, реализующих расчёты по данному методу, тоже велико. Существуют как консольные приложения, так и программы с интерфейсом пользователя. Все их можно разделить на две части по способу задания исходного графа

* задание матрицей весов
* задание перечислением дуг

В первом случае вводится матрица размерностью N\*N, где N – количество вершин. Во втором вводятся тройки чисел – номер начальной вершины, номер конечной и вес дуги. В качестве ответа всегда приводится последовательность прохождения вершин от заданных старта к финишу.

# Выбор алгоритма для реализации системы

Рассматриваемая нами система является классической задачей нахождения минимального пути в графе. Из оценок сложности было решено взять алгоритм Дейкстры. В оригинале алгоритм выглядит следующим образом.

Алгоритм Дейкстры для нахождения пути минимального веса в графе. Основное требование алгоритма – отсутствие петель и циклов отрицательного веса. Последнее требование можно усилить и требовать отсутствия дуг с отрицательными весами, что означает, что матрица весов должна иметь неотрицательные элементы.

Обозначим начальную вершину, из которой будем искать пути, через a, а множество всех вершин через V. Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a и номер предшественника – номер вершины, из которой достигается оптимальное значение. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация. Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояние от a до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

Шаг алгоритма. Если все вершины посещены, алгоритм завершается. В противном случае, из ещё не посещённых вершин выбирается вершина u, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут рёбра из u, назовём соседями этой вершины. Для каждого соседа вершины u, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины, а номер предшественника – на номер этого соседа. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим шаг алгоритма.

Построение самого маршрута для данного алгоритма осуществляется обратным ходом, от конца к началу. Рассмотрим массив P номеров соседей, которые мы формировали в ходе выполнения шагов алгоритма. Если b – это вершина, в которую мы хотим попасть из начальной a, то P[b] – это номер вершины, из которой мы приходим в b оптимальным маршрутом. Потом рассматриваем P[P[b]] и так далее, пока на каком-то шаге не придём в начальную вершину.

# Проектирование приложения

Исходя из принципов объектно-ориентированного программирования и общей тенденции разработки программного обеспечения, в разрабатываемой программе целесообразно разделить интерфейс пользователя от основных расчётов – отделить визуальную часть от невизуальной. Такой подход позволит сделать модуль расчёта по алгоритму Дейкстры автономным и использовать его в случае необходимости в других задачах.

Будет использован способ задания графа перечнем рёбер с весовыми коэффициентами, а сами вершины пользователь будет наносить с помощью мыши на специально выделенном пространстве. Нумерация вершин будет вестись по порядку их появления на экране.

## Проектирование интерфейса пользователя

Интерфейс пользователя условно можно разделить на три части

* Зона задания списка рёбер
* Зона задания вершин графа и прорисовка его всего
* Зона ввода данных и управления выполнением

Процедуру ввода информации о рёбрах можно реализовать для одного и потом просто зациклить. Причём количество рёбер не обязательно задавать заранее, пользователь может ввести некоторое их количество, просчитать задачу и добавить ещё. В итоге мы будем иметь некоторый массив из объектов одного типа. Визуально это будет представлять собой некоторую таблицу из трёх полей – номера начальной и конечной вершин и вес ребра.

Задание вершин графа тоже реализуется циклически, для каждой вершины действия однотипны и на выходе мы тоже имеем набор из объектов одного типа, хранящих информацию о номере вершины, её координатах на плоскости (это потребуется в дальнейшем для прорисовки графа).

Зона ввода данных и управления должна предоставить возможность задать номера начальной и конечной вершин пути, запустить алгоритм расчёта.

## Проектирование расчётного модуля

Классический алгоритм Дейкстры получает в качестве начальных данных матрицу весов графа, номер начальной вершины и количество всех вершин. В качестве выходных данных он возвращает массив расстояний от начальной до всех вершин и массив номеров вершин «предыдущих», который используется для построения пути.

## Проектирование промежуточного модуля

Также нужен промежуточный модуль преобразования данных от пользовательского интерфейса к алгоритму и обратно, что позволит сделать каждый объект автономным и в случае необходимости переделки какой-либо части остальные не будут затрагиваться, а это позволит избежать дополнительных ошибок и повысит качество разработки ПО. Он получает от пользовательского интерфейса массив объектов типа ребро и преобразует их в матрицу весов. Вызывает расчётный модуль, получает от него результаты и получает из массива «предшественников» путь от начальной вершины к конечной. Возвращает значение веса всего пути и последовательность вершин.

# Реализация программной системы

Программа написана на языке C# в среде разработки MS Visual Studio 2015 в виде WinForms-приложения. При разработке использовались стандартные компоненты, дополнительные библиотеки не подключались.

Вся программа, реализующая данный алгоритм, разделяется на три части –

1. Пользовательский интерфейс (ПИ) позволяет визуально увидеть моделируемую систему, задать начальные данные для расчётов, выбрать начальную и конечную вершины. После расчётов показывает оптимальный маршрут.
2. Преобразование полученных данных (ПД) для получения матрицы весов и передача её в алгоритм расчёта. Получение результата и возврат преобразованных данных обратно в ПИ.
3. Стандартный алгоритм Дейкстры(АД). Получает на вход матрицу весов и номер начальной вершины. В качестве выходных данных получаем массив значений весов маршрутов из начальной вершины во все остальные и массив номеров оптимальных предшественников.

Для хранения информации о рёбрах в программе был создан класс

public class Edge

{

public int vBeg { get; set; } // Номер начальной вершины

public int vEnd { get; set; } // Номер конечной вершины

public double Val { get; set; } // Вес ребра

}

апапап

## Пользовательский интерфейс

Реализован в виде WinForms-приложения с одной основной формой (рис. 4.1).

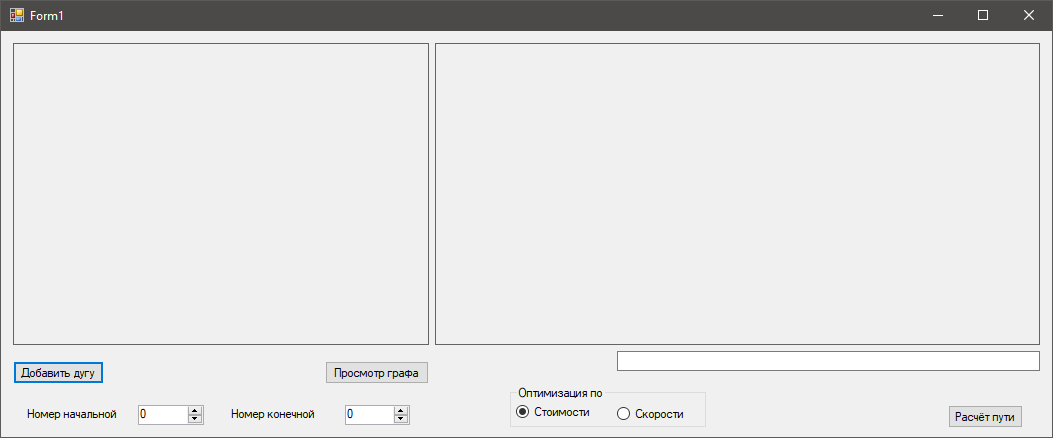


Рисунок 4.1. Интерфейс программы

Задание вершин графа производится в правой части формы нажатием левой кнопки мыши. Нумерация точек начинается с 0.

В левой части формы формируются список рёбер графа, соответствующий связям между точками. Порядок полей для заполнения следующий: номер начальной вершины, номер конечной вершины, значение длины(веса) для данного ребра.

В проекте основная форма находится в модуле MainForm.cs, в модуле PointContainer.cs содержится класс PointContainer, использующийся для работы с рёбрами графа. Класс создан как потомок от визуального GroupBox и представляет из себя объект с тремя элементами – два для ввода номеров вершин типа NumericUpDown и одно поле типа TextBox для ввода числового параметра – веса ребра.

public class PointContainer : GroupBox

{

public CheckBox newCheckBox;

public NumericUpDown newEdgeBeg;

public NumericUpDown newEdgeEnd;

public TextBox newEdgeWeight;

public int number;

В классе также предусмотрен элемент типа CheckBox, чтобы с его помощью изменять конфигурацию графа, временно отключая рёбра из рассмотрения и подключая их обратно, но в данной программе этот функционал не реализован.

Для показа графа разработана процедура ShowPicture.

private void ShowPicture()

{

Graphics g = panelShow.CreateGraphics();

panelShow.Refresh(); // Очистка изображения

//Clear(Color.Transparent); //clear all

// Create pen.

Pen blackPen = new Pen(Color.Black, 3); // Перо - чёрное, толстое

// Create size of ellipse.

int width = 2; int height = 2;

Brush br = new SolidBrush(Color.Green);

g.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias;

int i = 0;

foreach (var p in PointList)

{

// Draw ellipse to screen.

g.DrawEllipse(blackPen, p.X, p.Y, width, height); // Вершины графа

// Номера вершин

g.DrawString(i.ToString(), Font, br, p.X+1, p.Y+1); // Координаты размещения текста

i++;

}

blackPen = new Pen(Color.Black, 1); // Перо - чёрное, тонкое

foreach (var p in PointContainerList)

{

int pb = (int)p.newEdgeBeg.Value; // Точка начальная

int pe = (int)p.newEdgeEnd.Value; // Точка конечная

// Ребро между ними

g.DrawLine(blackPen, PointList[pb].X, PointList[pb].Y, PointList[pe].X, PointList[pe].Y);

}

}

Для хранения информации о вершинах используется стандартный тип Point.

Для показа рассчитанного оптимального пути применяется процедура

private void ShowPath(List<int> Path)

{

Graphics g = panelShow.CreateGraphics();

// Create pen.

Pen blackPen = new Pen(Color.Red, 2); // Перо - красное, потолще

var startNode = Path.First(); // Начальная точка

Path.Remove(startNode); // Убираем стартовую из маршрута

// Прорисовка нового маршрута

foreach (var node in Path)

{

g.DrawLine(blackPen, PointList[startNode].X, PointList[startNode].Y, PointList[node].X, PointList[node].Y);

startNode = node;

}

}

Основные действия, связанные с нажатием на кнопку «Расчёт пути», реализованы в процедуре PerfomCalc.

private void PerfomCalc()

{

List<Edge> lEdge = new List<Edge>();

string stringMess = "Оптимальное значение ";

// Создаём новый список, преобразуя в значения нужных типов

lEdge = PointContainerList.Select(p => new Edge

{ vBeg = (int)p.newEdgeBeg.Value, vEnd = (int)p.newEdgeEnd.Value, Val = double.Parse(p.newEdgeWeight.Text) }).ToList();

Calc calc = new Calc(PointList.Count);

calc.Start = (int)numericUpDownStart.Value; // - 1;

calc.End = (int)numericUpDownEnd.Value; // - 1;

double[] val = calc.GetData(lEdge);

double retVal = val[calc.End];

if (retVal <= 0)

{

stringMess = "Нет пути";

textBoxResult.Text = stringMess;

MessageBox.Show(stringMess);

return;

}

var Path = calc.Path;

// Показ графа - заново

ShowPicture();

// Прорисовка пути

ShowPath(Path);

// Надпись о результате

textBoxResult.Text = stringMess + retVal.ToString("F5");

}

## Модуль подготовки данных

Модуль принимает от интерфейса пользователя информацию о параметрах рёбер, формирует из них матрицу весов для графа и передаёт для расчётов в модуль стандартного или модифицированного алгоритма в зависимости от выбора пользователя. Предполагается, что граф неориентированный и матрица симметричная, т.е. вес ребра от вершины i до вершины j совпадает с весом ребра от вершины j до вершины i. После получения результатов расчётов находит оптимальный путь из начальной вершины в конечную и передаёт данные о пути и его числовой параметр обратно в ИП. Текст модуля находится в файле Calc.cs.

## Модуль реализации алгоритма

Реализует стандартный алгоритм Дейкстры для полученной матрицы, находит оптимальный маршрут из начальной для всех вершин и возвращает в качестве результата два массива – оптимальные значения для веса маршрута и массив достижимости вершин. Стандартный алгоритм реализован в файле DStand.cs.

# Тестирование и отладка программной системы

* 1. **Нахождение оптимального пути без изменения топологии сети**

В качестве примера было рассмотрено следующее расположение вершин, между которыми надо было кратчайший путь 0 до точки 9 (рис. 5.1).

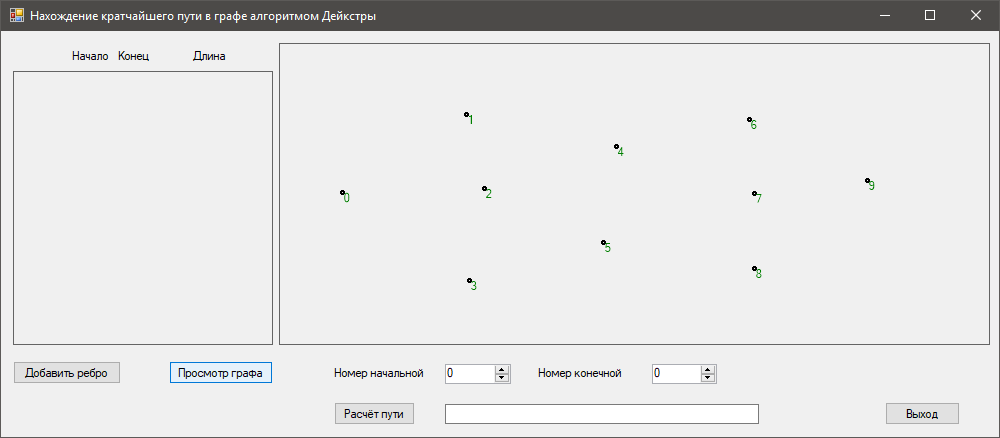


Рис. 5.1. Начальная конфигурация вершин

Далее для этой конфигурации были заданы значения для рёбер, соединяющих некоторые вершины.

|  |  |  |
| --- | --- | --- |
| Номер начала | Номер конца | Длина |
| 0 | 1 | 4 |
| 0 | 2 | 5 |
| 0 | 3 | 4 |
| 1 | 4 | 7 |
| 2 | 5 | 6 |
| 3 | 5 | 5 |
| 4 | 6 | 6 |
| 5 | 7 | 6 |
| 5 | 8 | 7 |
| 6 | 9 | 9 |
| 7 | 9 | 7 |
| 8 | 9 | 7 |

Граф для данной конфигурации выглядит следующим образом (рис.5.2)

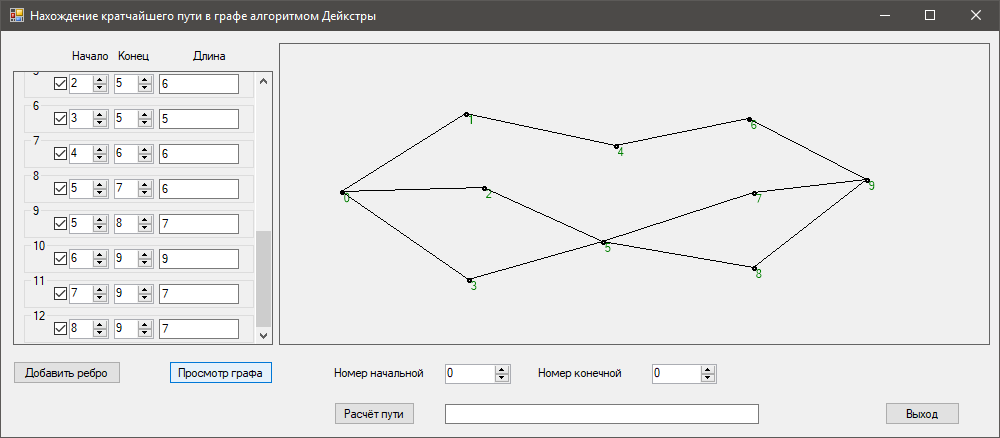


Рис. 5.2. Граф с заданными параметрами

Для этой конфигурации был найден оптимальный путь (рис. 5.3).

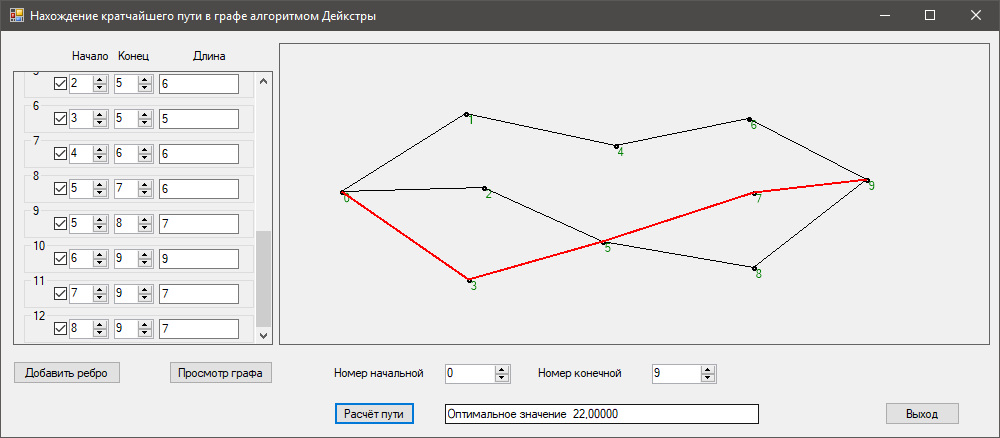


Рис. 5.3. Оптимальный путь

* 1. **Нахождение оптимального пути с изменением топологии сети**

В реальных задачах, которые используют данный алгоритм для решения практических проблем, иногда возникает ситуация в исходном графе с полученным результатом добавить несколько рёбер к уже имеющимся и просчитать полученный заново. Данная задача без проблем решается разработанной программой. Рассмотрим пример графа, разобранный в п.5.1. В качестве дополнительного ребра в графе было добавлено (2,6) с числовым параметром 7. В результате получено следующее решение для изменённой конфигурации (рис. 5.4).

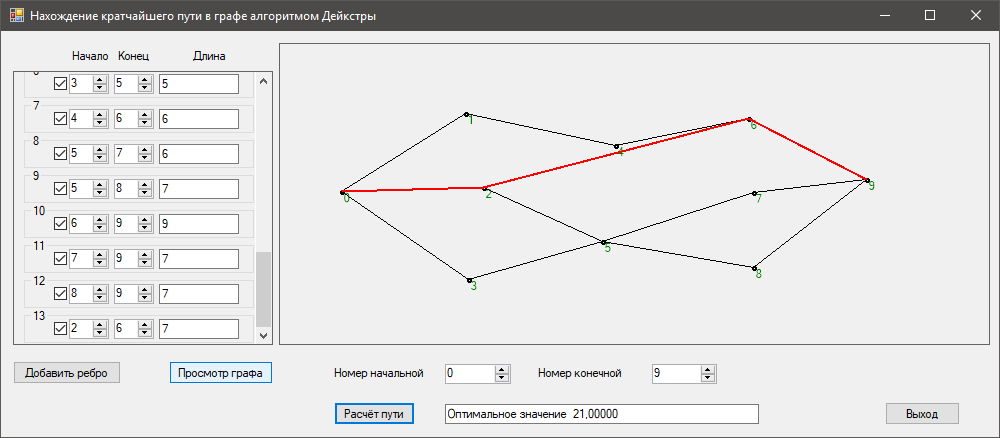


Рис. 5.4. Оптимальный путь изменённой системы

# Руководство пользователя

После первоначальной загрузки программы появляется окно с расположенным на нём элементами управления. Визуально всё окно можно разделить на три части (см. рис. 6.1)

1. Зона задания списка рёбер
2. Зона задания вершин графа и прорисовка его всего
3. Зона ввода данных и управления выполнением

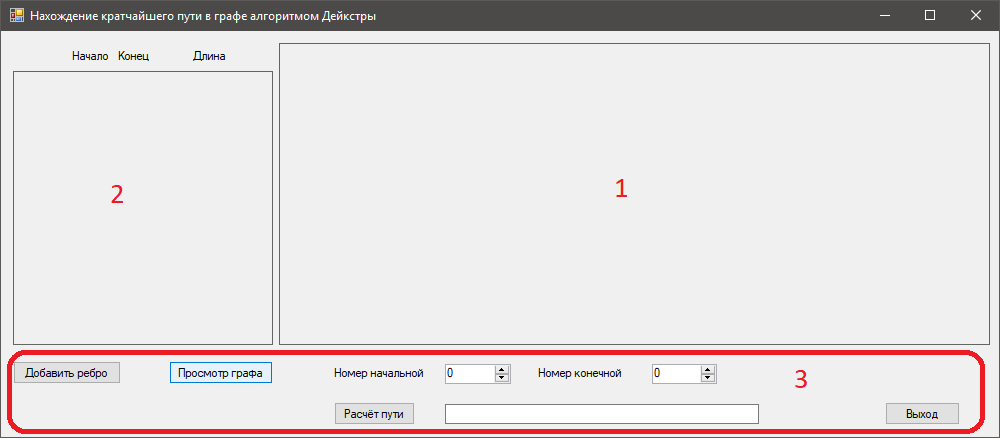


Рисунок 6.1. Схема расположения зон

Первая зона служит для начального задания вершин графа и потом отображения самого графа и найденного оптимального пути. Во второй зоне осуществляется ввод рёбер графа, по принципу номер начальной вершины, номер конечной вершины, длина (вес) ребра. В третьей зоне расположены кнопки управления, поля ввода номеров начальной и конечной вершин пути, поле вывода результата.

С помощью мыши в правой части расставляются точки графа. Далее итерационно вводятся значения для ребер графа, используя кнопку «Добавить ребро». В любой момент времени можно посмотреть заданную конфигурацию, она отображается в правой части формы (рис.6.2).

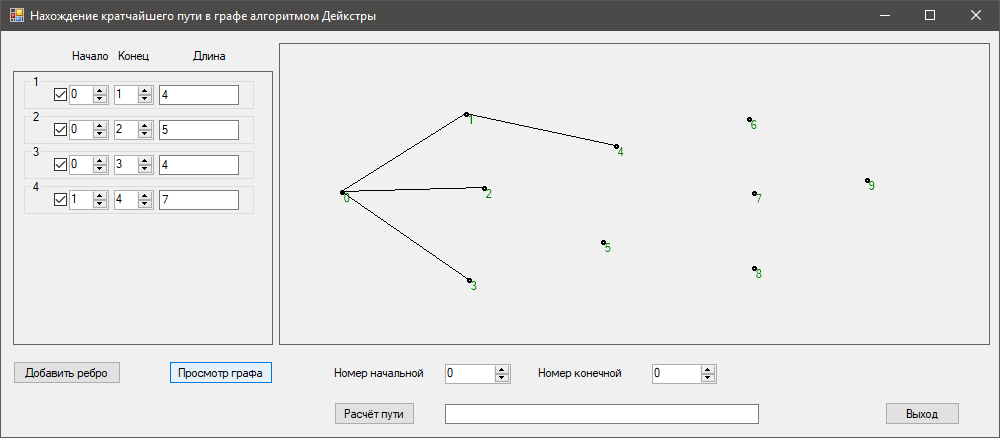


Рисунок 6.2. Граф промежуточного состояния

После ввода всех данных возможен просмотр полного графа. Для этого надо нажать на соответствующую кнопку (см. рис. 6.3).

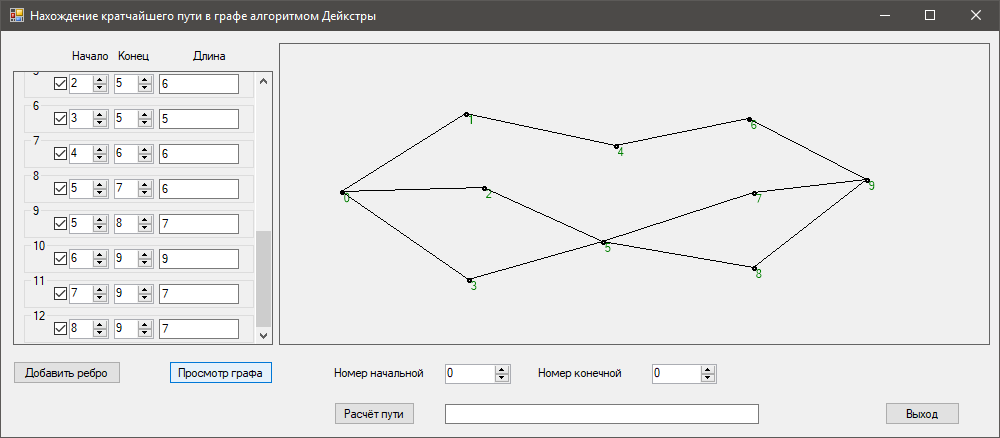


Рисунок 6.3. Граф конечного состояния

После задания номеров начальной и конечной вершин, нажатием на кнопку «Расчёт пути» можно сделать расчёт и получить графическое изображение результатов расчёта и значение оптимального пути (рис. 6.4)

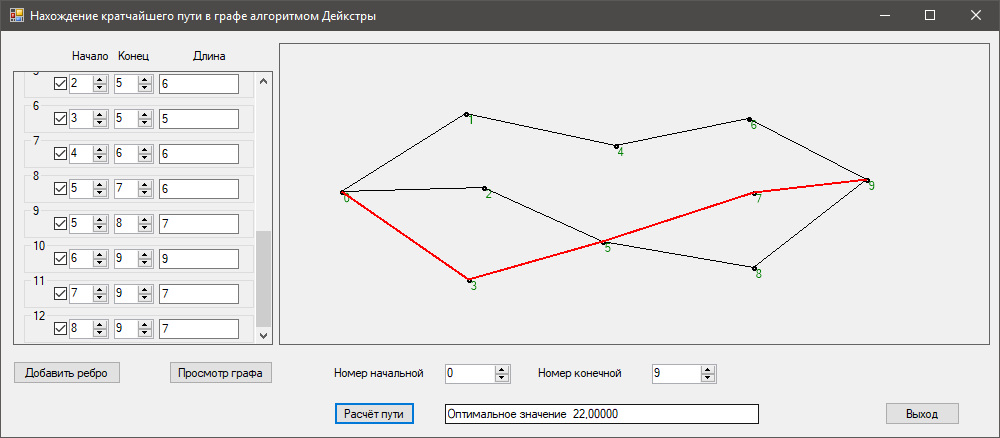


Рисунок 6.4. Граф конечного состояния с оптимальным путём

Интерфейс позволяет внести коррективы в уже обработанные данные и заново сделать расчёты. Для примера было добавлено дополнительное ребро (2,6) с числовым параметром 7. В результате получено следующее решение для изменённой конфигурации (рис. 6.5).

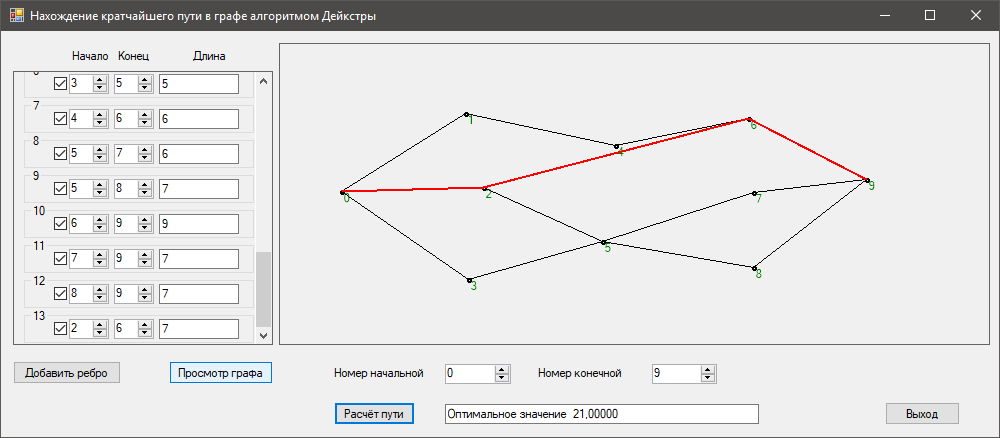


Рисунок 6.5. Оптимальный путь для изменённых данных

# Заключение

В представленной работе были изучена задача проектирования сетей передачи данных. Для поставленной задачи разработана методика и написана диалоговая программа выбора оптимального маршрута соединения двух объектов. Программа написана с применением принципов модульного программирования и может быть впоследствии доработана отдельно визуальная часть и расчётная для реализации других алгоритмов оптимизации сети.

Для реализации программы была использована среда разработки MS Visual Studio 2015 и язык программирования C#, применён принцип модульного и объектно-ориентированного программирования.

Стоит отметить, что разработанная программа может быть использована для решения прикладных задач. Если вместо пустой зоны, на которой располагаются точки графа, разместить географическую карту, атлас дорог или схему печатной платы, то можно будет решать конкретные практические задачи оптимизации.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Бишоп, Дж. С# в кратком изложении / Дж. Бишоп, Н. Хорспул. - М.: Бином. Лаборатория знаний, 2011. - 472 c.
2. Васильев, Алексей C#. Объектно-ориентированное программирование / Алексей Васильев. - М.: Питер, 2012. - 320 c.
3. Троелсен, Эндрю Язык программирования C# 5.0 и платформа .NET 4.5 / Эндрю Троелсен. - М.: Вильямс, 2015. - 633 c.
4. Иванов, Б.Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория базовых знаний, 2002. – 288 с.
5. Кристофидес, Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.
6. Майника, Э. Алгоритмы оптимизации на сетях и графах. – М.: Мир, 1981. – 325 с.
7. Новиков, Ф.А. Дискретная математика для программистов. – СПб.: Питер, 2001. – 304 с.
8. Уилсон, Р. Введение в теорию графов. – М.: Мир, 1977. – 207 с.
9. Харари, Ф. Теория графов. – М.: Мир, 1973. – 300 с.
10. <https://msdn.microsoft.com/>

# Приложение 1

Тексты программ

Модуль Program.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace NetPlan

{

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new MainForm());

}

}

}

Модуль MainForm.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace NetPlan

{

public partial class MainForm : Form

{

private List<Point> PointList = new List<Point>();

private List<PointContainer> PointContainerList = new List<PointContainer>();

private int dim;

public MainForm()

{

InitializeComponent();

}

private void panelShow\_MouseDown(object sender, MouseEventArgs e)

{

Point p = new Point(e.X, e.Y);

AddPoint(p); // Добавление новой точки

ShowPoints(); // Показ всех точек

dim = PointList.Count;

}

/// <summary>

/// Добавление новой точки в список

/// </summary>

/// <param name="p">Переменная типа Point</param>

private void AddPoint(Point p)

{

PointList.Add(p);

}

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

{

AddPointInfo();

}

/// <summary>

/// Добавление информации

/// </summary>

private void AddPointInfo()

{

PointContainer pointContainer = new PointContainer();

PointContainerList.Add(pointContainer);

pointContainer.Parent = panelPoints;

pointContainer.newEdgeBeg.Minimum = 0;

pointContainer.newEdgeBeg.Maximum = dim;

pointContainer.newEdgeEnd.Minimum = 0;

pointContainer.newEdgeEnd.Maximum = dim;

ShowPointsInfo();

}

/// <summary>

/// Показ информации о дугах

/// Номер начала, конца, длина

/// </summary>

private void ShowPointsInfo()

{

int num = 1;

foreach (var p in PointContainerList)

{

p.Text = num.ToString();

p.Visible = true;

p.Location = new Point(10, 3 + (p.Height) \* (num - 1));

num++;

}

}

/// <summary>

/// Показ всех точек на графике

/// </summary>

private void ShowPoints()

{

Graphics g = panelShow.CreateGraphics();

// Create pen.

Pen blackPen = new Pen(Color.Black, 3);

// Create size of ellipse.

int width = 2;

int height = 2;

foreach (var p in PointList)

{

// Draw ellipse to screen.

g.DrawEllipse(blackPen, p.X, p.Y, width, height);

}

}

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

{

ShowPicture();

buttonAdd.Enabled = true;

}

/// <summary>

/// Показ всего графа - точки и дуги

/// </summary>

private void ShowPicture()

{

Graphics g = panelShow.CreateGraphics();

panelShow.Refresh(); // Очистка изображения

//Clear(Color.Transparent); //clear all

// Create pen.

Pen blackPen = new Pen(Color.Black, 3); // Перо - чёрное, толстое

// Create size of ellipse.

int width = 2; int height = 2;

Brush br = new SolidBrush(Color.Green);

g.TextRenderingHint = System.Drawing.Text.TextRenderingHint.AntiAlias;

int i = 0;

foreach (var p in PointList)

{

// Draw ellipse to screen.

g.DrawEllipse(blackPen, p.X, p.Y, width, height); // Вершины графа

// Номера вершин

g.DrawString(i.ToString(), Font, br, p.X+1, p.Y+1); // Координаты размещения текста

i++;

}

blackPen = new Pen(Color.Black, 1); // Перо - чёрное, тонкое

foreach (var p in PointContainerList)

{

int pb = (int)p.newEdgeBeg.Value; // Точка начальная

int pe = (int)p.newEdgeEnd.Value; // Точка конечная

// Ребро между ними

g.DrawLine(blackPen, PointList[pb].X, PointList[pb].Y, PointList[pe].X, PointList[pe].Y);

}

}

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

{

PerfomCalc();

}

/// <summary>

/// Расчёты и показ результатов

/// </summary>

private void PerfomCalc()

{

List<Edge> lEdge = new List<Edge>();

string stringMess = "Оптимальное значение ";

// Создаём новый список, преобразуя в значения нужных типов

lEdge = PointContainerList.Select(p => new Edge

{ vBeg = (int)p.newEdgeBeg.Value, vEnd = (int)p.newEdgeEnd.Value, Val = double.Parse(p.newEdgeWeight.Text) }).ToList();

Calc calc = new Calc(PointList.Count);

calc.Start = (int)numericUpDownStart.Value; // - 1;

calc.End = (int)numericUpDownEnd.Value; // - 1;

double[] val = calc.GetData(lEdge);

double retVal = val[calc.End];

if (retVal <= 0)

{

stringMess = "Нет пути";

textBoxResult.Text = stringMess;

MessageBox.Show(stringMess);

return;

}

var Path = calc.Path;

// Показ графа - заново

ShowPicture();

// Прорисовка пути

ShowPath(Path);

// Надпись о результате

textBoxResult.Text = stringMess + retVal.ToString("F5");

}

/// <summary>

/// Показ оптимального маршрута

/// </summary>

/// <param name="Path"></param>

private void ShowPath(List<int> Path)

{

Graphics g = panelShow.CreateGraphics();

// Create pen.

Pen blackPen = new Pen(Color.Red, 2); // Перо - красное, потолще

var startNode = Path.First(); // Начальная точка

Path.Remove(startNode); // Убираем стартовую из маршрута

// Прорисовка нового маршрута

foreach (var node in Path)

{

g.DrawLine(blackPen, PointList[startNode].X, PointList[startNode].Y, PointList[node].X, PointList[node].Y);

startNode = node;

}

}

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

{

Close();

}

}

}

Модуль DStand.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace NetPlan

{

/// <summary>

/// Расчёт пути минимального веса алгоритмом Дейкстры

/// </summary>

public class DStand

{

private const double X\_MAX = double.MaxValue;

private int MinimumDistance(double[] distance, bool[] shortestPathTreeSet, int verticesCount)

{

double min = X\_MAX;

int minIndex = 0;

for (int v = 0; v < verticesCount; ++v)

{

if (shortestPathTreeSet[v] == false && distance[v] <= min)

{

min = distance[v];

minIndex = v;

}

}

return minIndex;

}

public double[] Dijkstra(double[,] graph, int source, int verticesCount, int[] prev)

{

double[] distance = new double[verticesCount];

bool[] shortestPathTreeSet = new bool[verticesCount];

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

{

distance[i] = X\_MAX;

shortestPathTreeSet[i] = false;

prev[i] = source;

}

distance[source] = 0;

for (int count = 0; count < verticesCount - 1; ++count)

{

int u = MinimumDistance(distance, shortestPathTreeSet, verticesCount);

shortestPathTreeSet[u] = true;

for (int v = 0; v < verticesCount; ++v)

if (!shortestPathTreeSet[v]

&& (graph[u, v] > 0)

&& (distance[u] != X\_MAX)

&& (distance[u] + graph[u, v] < distance[v]))

{

distance[v] = distance[u] + graph[u, v];

prev[v] = u;

}

}

return distance;

}

}

}

Модуль Calc.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace NetPlan

{

public class Edge

{

public int vBeg { get; set; } // Номер начальной вершины

public int vEnd { get; set; } // Номер конечной вершины

public double Val { get; set; } // Вес ребра

}

public class Calc

{

private double[,] A; // Матрица весов графа

private int[] prev; // Массив предшественников

public List<int> Path { get; set; } = new List<int>(); // Путь от начала к концу

public int Start { get; set; } // Начальная вершина пути

public int End { get; set; } // Конечная вершина пути

public int dimA { get; set; } // Количество вершин

public Calc(int dim)

{

dimA = dim;

A = new double[dimA, dimA];

prev = new int[dimA];

}

/// <summary>

/// Получение параметров сети, формирование матрицы, расчёт

/// </summary>

/// <param name="Coeff">Параметры графа</param>

public double[] GetData(List<Edge> Coeff)

{

double d = 0; // double.MaxValue / dimA;

//for (int i = 0; i < dimA; ++i)

//{

// A[i, i] = 0; // Вес дуги от вершины к ней же

//}

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

{

for (int j = 0; i < dimA; ++i)

{

A[i, j] = d; // Все отсутствующие рёбра - с нулевым --максимальным-- весом

}

}

// Заполнение матрицы коэффициентов

foreach (var c in Coeff)

{

A[c.vBeg, c.vEnd] = c.Val;

A[c.vEnd, c.vBeg] = c.Val; // Предполагаем симметричность матрицы

}

//SaveA("ap.csv"); // Промежуточное сохранение матрицы коэффициентов - проверка

// Создание объекта для решения задачи

double[] rett = new double[dimA];

DStand dStand = new DStand();

// Получаем пути ко всем вершинам графа

rett = dStand.Dijkstra(A, Start, dimA, prev);

// Получаем путь из начала в конец

GetPath();

return rett;

}

/// <summary>

/// Построение пути из начальной в конечную вершину

/// </summary>

void GetPath()

{

Path.Clear();

int j = End; // Номер вершины, в которую попадаем

Path.Add(End);

// Идём от конца к началу

do

{

j = prev[j];

Path.Add(j);

} while (j != Start);

Path.Reverse(); // Переворот - от начала к концу

}

/// <summary>

/// Сохранение матрицы коэффициентов в текстовый файл

/// </summary>

/// <param name="Fname">Имя файла</param>

void SaveA(string Fname)

{

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

string s = "";

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

{

s = "";

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

{

s += A[i, j].ToString() + " ; ";

}

list.Add(s);

}

System.IO.File.WriteAllLines(Fname, list);

}

}

}

Модуль PointContainer.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Drawing;

namespace NetPlan

{

public class PointContainer : GroupBox

{

public CheckBox newCheckBox;

public NumericUpDown newEdgeBeg;

public NumericUpDown newEdgeEnd;

public TextBox newEdgeWeight;

public int number;

public PointContainer(int num)

{

number = num;

newCheckBox = new CheckBox();

newEdgeBeg = new NumericUpDown();

newEdgeEnd = new NumericUpDown();

newEdgeWeight = new TextBox();

newEdgeBeg.Parent = this;

newEdgeEnd.Parent = this;

newEdgeWeight.Parent = this;

newCheckBox.Parent = this;

newEdgeWeight.Text = "0";

SetSize();

}

public PointContainer() : this(-1)

{

}

void SetSize()

{

int width = 230;

int height = 35;

int spacing = 5;

int curr = 0;

int wwidth = 40;

Size = new Size(width, height);

newCheckBox.Location = new Point(30, 10);

newCheckBox.Checked = true;

newCheckBox.Size = new Size(wwidth, 20);

curr += newCheckBox.Width;

newEdgeBeg.Size = new Size(wwidth, height);

newEdgeBeg.Location = new Point(spacing + curr, 10);

curr += spacing + newEdgeBeg.Width;

newEdgeEnd.Size = new Size(wwidth, height);

newEdgeEnd.Location = new Point(spacing + curr, 10);

curr += spacing + newEdgeEnd.Width;

newEdgeWeight.Size = new Size(2\*wwidth, height);

newEdgeWeight.Location = new Point(spacing + curr, 10);

curr += spacing + newEdgeWeight.Width;

}

}

}