

**Содержание**

[**Реферат** 3](#_Toc184656913)

[**Введение** 4](#_Toc184656914)

[**1. Постановка задачи** 5](#_Toc184656915)

[**2. Теоретическая часть задания** 6](#_Toc184656916)

[**3. Описание алгоритма программы** 9](#_Toc184656917)

[**4. Описание программы** 12](#_Toc184656918)

[**5. Тестирование** 23](#_Toc184656919)

[**6. Ручной расчет задачи** 27](#_Toc184656920)

[**Заключение** 28](#_Toc184656921)

[**Список литературы** 29](#_Toc184656922)

[**Приложение A** 30](#_Toc184656923)

# **Реферат**

Отчет 34 стр, 19 рисунков.

ГРАФ, ТЕОРИЯ ГРАФОВ, РАСКРАСКА ГРАФА, ХРОМАТИЧЕСКОЕ ЧИСЛО ГРАФА.

Цель исследования – разработка программы, реализующей алгоритм раскрашивания графов.

В работе рассмотрен жадный алгоритм для нахождения хроматического числа графа. Установлено, что с помощью данного алгоритма можно найти хроматическое число, согласно которому удастся раскрасить граф в соответствии с правилами раскрашивания графа.

# **Введение**

Одной из самых трудоемких задач теории графов является задача нахождения хроматического числа графа, то есть минимального числа цветов, необходимых для раскраски вершин графа. Предложены различные алгоритмы решения данной задачи, однако поиск эффективного алгоритма продолжается. Раскраска вершин позволяет моделировать многие проблемы планирования. В частности, при помощи алгоритма раскраски графа может быть решена задача составления расписания.

В рамках данной работы рассматривается вершинная раскраска графа, осуществляется анализ алгоритма вершинной раскраски графа, известный как «жадный».

В качестве среды разработки мною была выбрана среда Microsoft Visual Studio 2019, язык программирования – С#. Целью данной курсовой работы является разработка программы на языке С#, который является широко используемым. Именно с его помощью в данном курсовом проекте реализуется «жадный» алгоритм.

# **1. Постановка задачи**

Требуется разработать программу, которая реализует алгоритм раскрашивания графов с учетом всех правил раскрашивания графов.

Исходный граф в программе должен задаваться матрицей смежности, причём при генерации данных должны быть предусмотрены граничные условия. Программа должна давать пользователю возможность указать количество вершин, а также заполнить матрицу автоматически, вручную или из текстового файла, с учетом выбранного количества вершин. Также программа должна иметь возможность сохранять получившийся результат в качестве изображения. После обработки этих данных на экран должен выводиться граф, с вершинами, раскрашенными в соответствии с хроматическим числом. Необходимо предусмотреть различные исходы алгоритмов, чтобы программа не выдавала ошибок и работала правильно. Устройство ввода – клавиатура и мышь.

# **2. Теоретическая часть задания**

Здесь будет рассмотрена задача вершинной раскраски обыкновенного графа. Перед тем как приступать к рассмотрению задач напомним определение графа, которое будет использовано далее, без углубления в теорию графов. Под «графом» в данной работе будет пониматься обыкновенный неориентированный граф (рисунок 1). Стоит заметить, что большинство определений теории графов даются по-разному в различных источниках, но они просто демонстрируют разные подходы и в большинстве своём не противоречат друг другу.

* Граф – упорядоченная пара множеств (V, E), где V– непустое множество вершин, а E – множество неупорядоченных пар вида (vi, vj), называемых рёбрами, где vi и vj принадлежат множеству V.
* Вершины vi и vj графа G = (V, E) называются смежными, если они соединены ребром, то есть если в E существует ребро (vi, vj).
* Одним из способов задания графа является матрица смежности–матрица A размером [NxN], где N – количество вершин, в которой элемент aij =1, если в графе вершины i и j смежные, и 0 в противном случае.

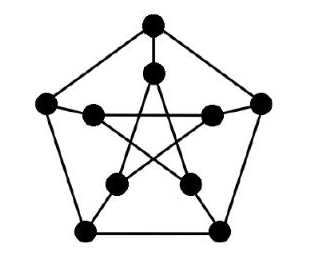


Рисунок 1 - Пример графа

Теперь рассмотрим подробнее задачу раскраски графа. Данная задача является одной из основных в теории графов, потому что входит в класс NP-полных и к ней сводятся многие другие задачи этой области. Раскраска графа может быть вершинной, рёберной и тотальной, однако все три варианта задачи можно свести друг к другу, поэтому рассмотрим только вершинную раскраску графа. «Раскраской графа называется такое приписывание цветов его вершинам, что никакие две смежные вершины не получают одинакового цвета.»

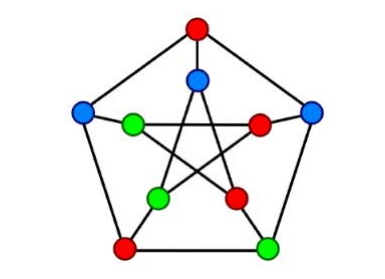


Рисунок 2- Вариант раскраски графа на рисунке 1.

Задача же раскраски состоит в нахождении минимального количества цветов для раскраски графа. Такое решение далее будем называть оптимальным. На рисунке 2 показана оптимальная раскраска графа, приведённого на рисунке 1. Задача не имеет известного единого алгоритма для поиска решения за полиномиальное время. Существуют варианты таких решений только для определённых типов графов. Единственным подходом к нахождению оптимального решения является перебор вариантов. Однако предлагаются разные подходы и алгоритмы для поиска неточного, приближённого решения, которое найти гораздо проще оптимального. Приведем кратко принцип работы некоторых из них.

* Алгоритмы последовательного перебора. Такие алгоритмы последовательно отделяют группы цветов. На каждом этапе создаётся группа вершин, в которую пытаются включить нераспределённые вершины, пока все они не будут распределены по цветам.
* Жадные алгоритмы. Жадные алгоритмы упорядочивают вершины по какому-либо правилу (чаще всего встречаются примеры, где используют степени вершин) и последовательно присваивают им цвета, в которые не были окрашены смежные с ними вершины.
* Алгоритм с использованием битовых операций. Алгоритм похож на первый, но работает с матрицами, используя битовые операции. Такой подход позволяет сократить время работы в несколько раз.

# **3. Описание алгоритма программы**

В данной работе, я использовал «жадный» алгоритм для поиска хроматического числа графа, следовательно, его нужно подробно изучить.

* 1. **Жадный алгоритм раскраски графа.**

Далее рассмотрим алгоритм раскраски графа, построенный как жадный алгоритм. В большинстве случаев, как уже отмечено ранее, жадные алгоритмы упорядочивают вершины по их степени (или относительной степени, то есть степени без учёта окрашенных вершин). В этом есть смысл, потому что окрашивание таких вершин приносит наибольшее количество информации в систему: с окраской таких вершин мы не можем окрасить большее количество цветов в цвет, выбранный для этой вершины. Для разработанного алгоритма было решено использовать степень вершин только как второстепенный фактор. Первичным же фактором для сравнения было выбрано количество цветов, в которые вершину окрасить уже нельзя. Во многих случаях такой подход формирует полный пул цветов, потому что рассматривает прежде всего вершины, для которых приходится добавлять цвет. Кроме того, будем упорядочивать не только вершины, но и цвета, чтобы сделать распределение вершин, а соответственно и нагрузки, более равномерными.

В алгоритме можно выделить следующие этапы работы:

1. Считать матрицу смежности.
2. Инициализировать вектор цветов (изначально пустой), содержащий вектора вершин, окрашенных в соответствующий цвет. Сюда добавляется окрашенная вершина.
3. Сформировать вектор вершин. Для хранения информации о вершинах создан класс Vertex, содержащий номер вершины в графе, её степень и вектор цветов, в которые её нельзя окрасить.
4. Упорядочить вектор вершин по приоритетам. Приоритет вершины больше, если её нельзя окрасить в большее количество цветов, иначе приоритет по степени.
5. Подобрать цвет для вершины, проверяя с наименее распространённых. Все доступные цвета хранятся в векторе, упорядоченном по количеству вершин, окрашенных в данный цвет. Проверяется вхождение рассматриваемого цвета в вектор цветов, в которые нельзя покрасить вершину.
   1. Если цвет найден
      1. Окрасить вершину.
      2. Пометить цвет как запрещённый для всех соседей. Для этого по соответствующей строке матрицы смежности находятся все соседние вершины и в вектор запрещённых для них цветов добавляется текущий, если его ещё нет.
      3. Пересортировать вершины и цвета. Нет необходимости рассматривать соответствующие векторы целиком.
         1. После окрашивания вершины увеличилось количество вершин данного цвета, поэтому проверяется соседний с ним в векторе цветов и, пока необходимо, цвета меняются местами.
         2. Каждая соседняя вершина получает дополнительный запрещённый цвет– её приоритет увеличивается, поэтому проверяются соседние вершины в списке вершин и, пока необходимо, вершины меняются местами.
      4. Исключить вершину. В матрице смежности обнуляются соответствующие строка и столбец, чтобы эта вершина не рассматривалась как соседняя, и вершина удаляется из вектора вершин.
   2. Если цвет не найден, добавить цвет и выполнить п. 5.1.
6. Повторять п. 5 пока не будут окрашены все вершины (пока вектор вершин не опустеет).

Полный исходный код алгоритма можно найти в приложении A (Файл Form1.cs).

# **4. Описание программы**

Для написания данной программы использован язык программирования С# - универсальный язык программирования, который завоевал особую популярность у программистов, благодаря сочетанию возможностей языков программирования высокого и низкого уровней.

Проект был создан в виде приложения Windows Forms. Данная программа является многомодульной, поскольку состоит из множества функций, а также из 2 файлов кода (Form1.c, ManualGraphInputForm.cs).

Работа программы начинается с начального экрана, где необходимо ввести количество вершин в графе. Затем пользователю предоставлена возможность выбрать метод генерации графа, при нажатии соответствующей кнопки пользователь может сгенерировать случайный граф, заполнить его вручную в всплывающем окне или загрузить текстовый файл с уже заполненной матрицей. Если же пользователь выбрал считать готовую матрицу из файла, то сначала, она считывает количество вершин в графе, а затем и матрицу смежности. Ниже можно увидеть оформление начального экрана и все вышеописанные функции. (Рисунки 3, 4, 5, 6, 7, 8, 9, 10, 11, 12).

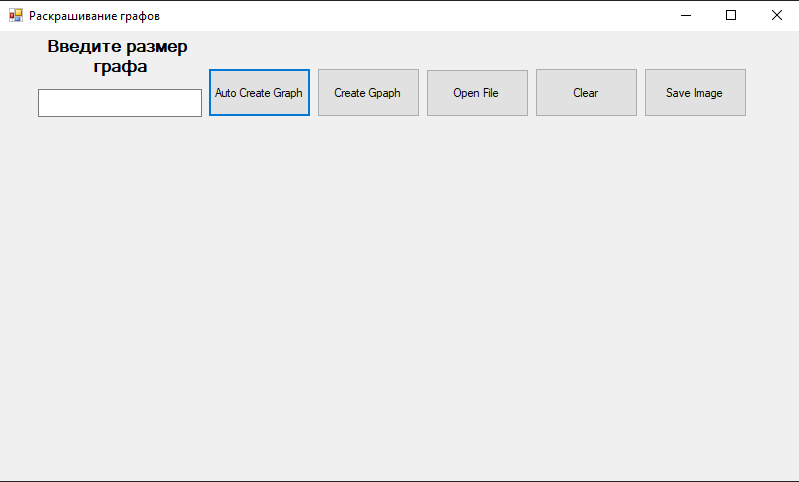


Рисунок 3 – начальный экран программы

На данном экране выполняются все основные операции. Нажатием на соответствующую кнопку пользователь может выбрать действие, которое он хочет выполнить.

В данной программе предусмотрены следующие функции:

1. В пустом окне необходимо ввести желаемое количество вершин.
2. Кнопка «Auto Create Graph» - отвечает за автоматическое создание матрицы смежности, с учетом указанного количества его вершин.
3. Кнопка «Create Graph» - вызывает новое окно, в котором пользователю предоставляется возможность заполнить матрицу смежности вручную.
4. Кнопка «Open File» - вызывает диалоговое окно, в котором пользователю предоставляется возможность выбрать текстовый файл, из которого будет считана матрица смежности.
5. Кнопка «Clear» - при нажатии удаляет сгенерированный граф.
6. Кнопка «Save Image» - вызывает диалоговое окно, в котором пользователь может указать имя файла и расположение на компьютере, в котором он желает его сохранить.

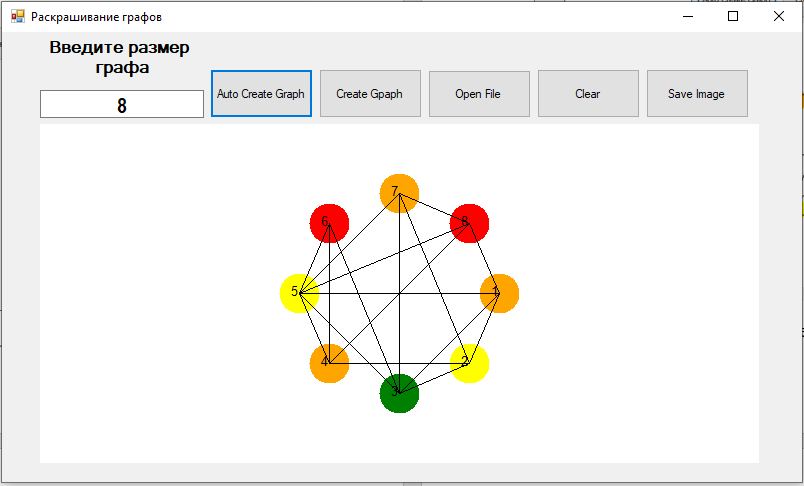


Рисунок 4 - Кнопка «Auto Create Graph» (автоматическая генерация графа)

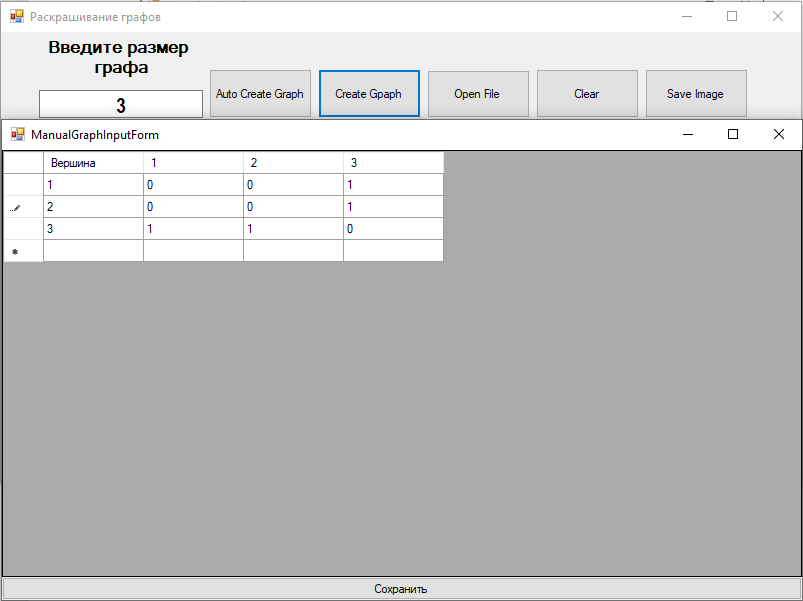


Рисунок 5 - Кнопка «Create Graph» (ручное заполнение матрицы смежности)

При нажатии данной кнопки открывается окно с таблицей, в которой пользователь может заполнить матрицу смежности, вводя значения в ее ячейки. Когда пользователь заполнит таблицу, необходимо нажать на кнопку «Сохранить», после чего окно с таблицей закроется и на экране появится граф, сгенерированный согласно данной таблице.

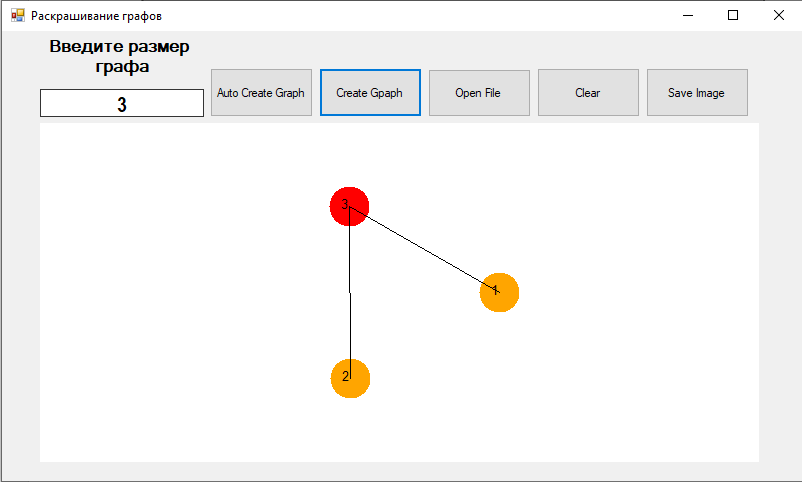


Рисунок 6 – Граф созданный в результате ручного ввода

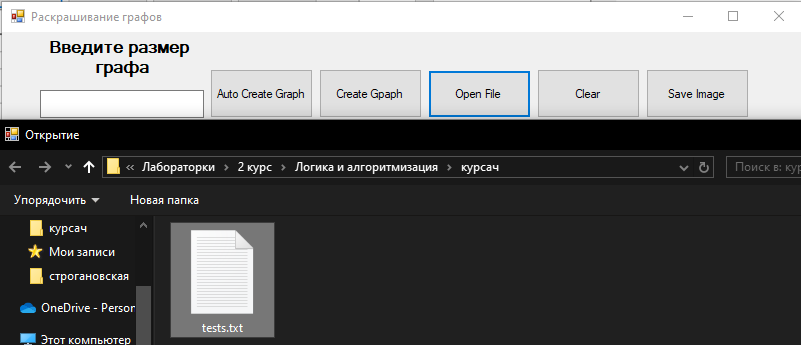


Рисунок 7 - Кнопка «Open File» (граф создается согласно матрице, заданной в выбранном файле)

При создании графа таким образом и количество вершин, и матрица смежности считываются из выбранного файла. Первая строка в файле определяет количество его вершин, затем следует матрица смежности.

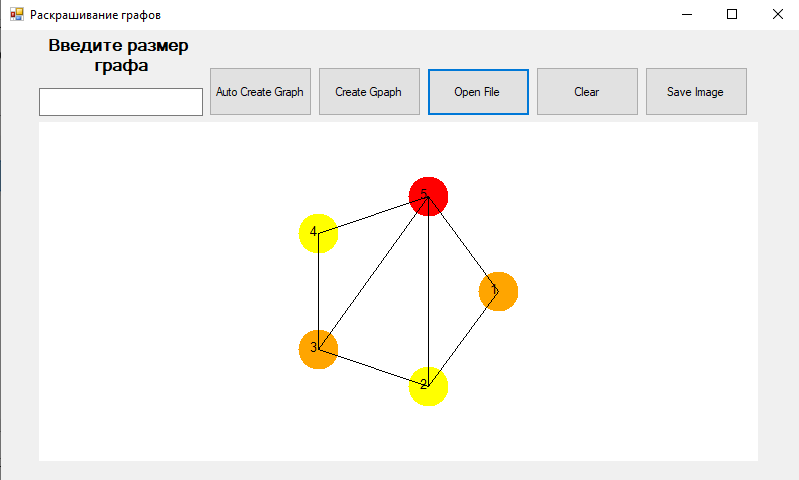


Рисунок 8 – Граф созданный в результате считывания из файла

Для проверки достоверности генерации графа ниже приведен загруженный файл. (Рисунок 9).

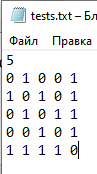


Рисунок 9 – Файл, загруженный для генерации графа

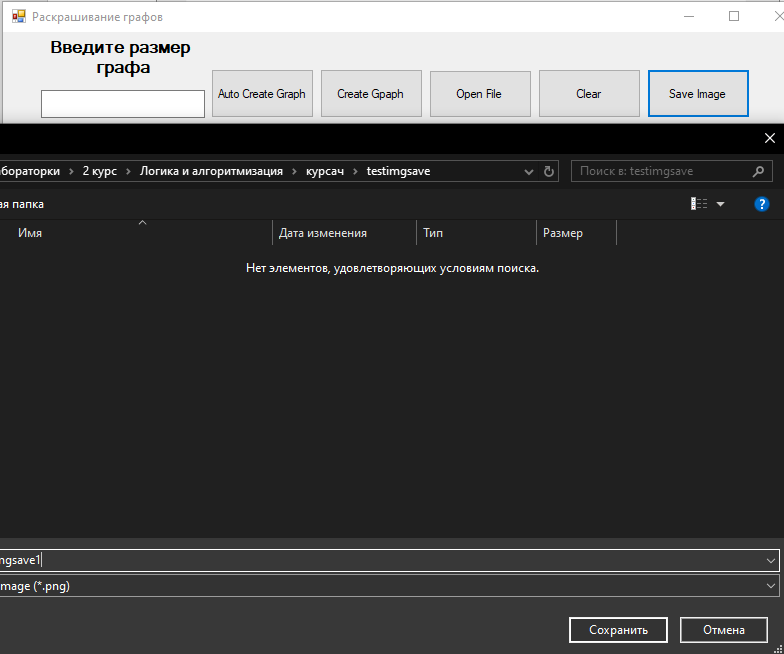


Рисунок 10 – Кнопка «Save Image» (позволяет сохранить получившийся граф)

После сохранение в указанном месте появится изображение с ранее сгенерированным графом. (Рисунок 11).

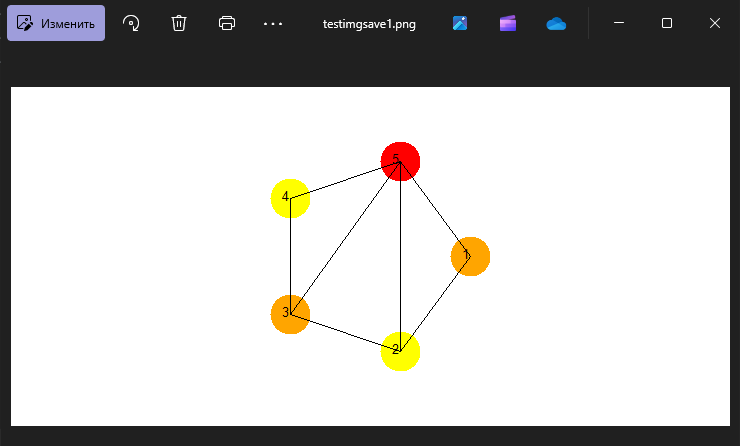


Рисунок 11 – Результат сохранения графа

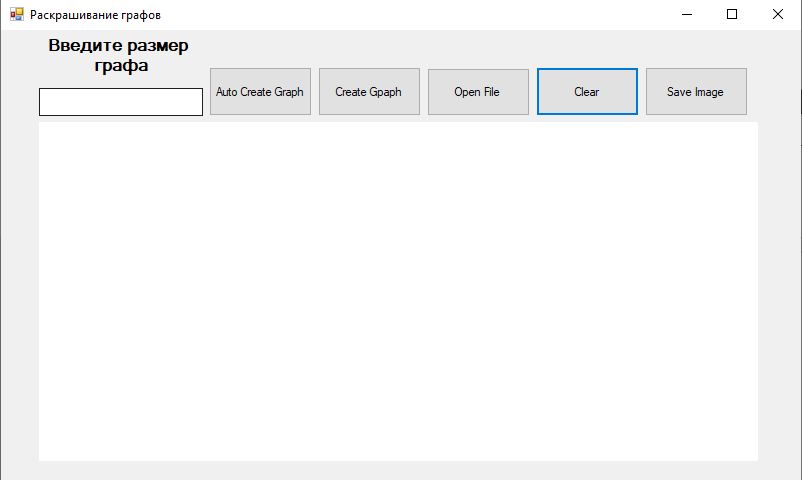


Рисунок 12 – Кнопка «Clear» (удаляет сгенерированный граф)

Далее более подробно о самой программе. В главном файле Form1.cs записаны все функции, отвечающие за корректное функционирование программы. Пользователь сам выбирает нужное ему действие, затем, в зависимости от его выбора, программа разделяется на несколько вариантов развития.

Работа каждой кнопки описана в отдельной функции. Так генерация случайного графа выглядит следующим образом (Файл Form1.cs):

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

{

// Генерация случайного графа

graphSize = int.Parse(txtGraphSize.Text);

InitializeGraph();

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

{

for (int j = i + 1; j < graphSize; j++)

{

if (random.Next(2) == 1)

{

graph[i].Add(j);

graph[j].Add(i);

}

}

}

colors = new int[graphSize]; // Сброс раскраски

ColorGraph();

}

Для реализации функции ручного заполнения графа был создан отдельный файл, в котором описана логика работы самой функции и алгоритм создания соответствующей таблицы, для упрощения пользовательского ввода. (Файл ManualGraphInputForm.cs).

Таким образом, формирование таблицы реализовано в данном виде:

private void InitializeDataGridView()

{

dgv = new DataGridView

{

ColumnCount = graphSize + 1, // Первая колонка для номеров вершин

RowCount = graphSize, // Столько же строк, сколько и вершин

Dock = DockStyle.Fill

};

// Создаем заголовки столбцов

dgv.Columns[0].Name = "Вершина";

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

{

dgv.Columns[i].Name = i.ToString();

}

// Заполняем таблицу нулями (по умолчанию нет соединений)

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

{

int ii = i + 1;

dgv.Rows[i].Cells[0].Value = ii.ToString();

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

{

dgv.Rows[i].Cells[j].Value = 0; // 0 - значит, нет ребра

}

}

// Добавляем кнопку сохранения

Button btnSave = new Button

{

Text = "Сохранить",

Dock = DockStyle.Bottom

};

btnSave.Click += BtnSave\_Click;

this.Controls.Add(dgv);

this.Controls.Add(btnSave);

}

А чтение пользовательского ввода реализовано в данной функции:

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

{

List<List<int>> graph = new List<List<int>>();

// Чтение данных из таблицы и формирование списка смежности

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

{

var neighbors = new List<int>();

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

{

if (Convert.ToInt32(dgv.Rows[i].Cells[j].Value) == 1)

{

neighbors.Add(j - 1); // Сосед по индексу

}

}

graph.Add(neighbors);

}

// Возвращаем граф в основную форму

this.DialogResult = DialogResult.OK;

this.Tag = graph;

this.Close();

}

Для возможности использования пользовательского ввода в файле Form1.cs была предусмотрена следующая функция:

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

{

graphSize = int.Parse(txtGraphSize.Text);

var manualInputForm = new ManualGraphInputForm(graphSize);

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

{

graph = manualInputForm.GetGraph();

colors = new int[graphSize]; // Сброс раскраски

ColorGraph();

}

}

Сохранение созданного графа в виде изображения представлено следующим образом (файл Form1.cs):

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

{

SaveFileDialog saveDialog = new SaveFileDialog();

saveDialog.Filter = "PNG Image|\*.png|JPEG Image|\*.jpg|Bitmap Image|\*.bmp";

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

{

try

{

// Сохраняем изображение

pictureBox1.Image.Save(saveDialog.FileName);

MessageBox.Show("Изображение успешно сохранено!", "Сохранение", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при сохранении изображения: " + ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

В свою очередь, считывание данных из файла целиком описано в файле Form1.cs и реализовано следующим образом:

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

{

OpenFileDialog openFileDialog = new OpenFileDialog();

openFileDialog.Filter = "Text Files|\*.txt";

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

{

try

{

string[] lines = File.ReadAllLines(openFileDialog.FileName);

graphSize = int.Parse(lines[0]); // Размер графа

InitializeGraph();

colors = new int[graphSize]; // Сброс раскраски

// Читаем матрицу смежности

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

{

string[] row = lines[i + 1].Split(' ');

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

{

if (row[j] == "1") // Если связь существует

{

graph[i].Add(j);

}

}

}

ColorGraph();

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при чтении файла: " + ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

Далее подробно разберем основной алгоритм работы программы – «жадный» алгоритм раскраски графа, описанный в файле Form1.cs. Вначале создаётся массив availableColors, который будет хранить информацию о том, какие цвета доступны для текущего узла. Каждый индекс массива соответствует цвету, а значение false означает, что этот цвет доступен, и true — что он занят:

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

{

bool[] availableColors = new bool[graphSize];

Затем для текущего узла i перебираются все его соседи. Если соседний узел уже имеет цвет (значение не равно -1), то этот цвет считается занятым для текущего узла. Если сосед имеет цвет, то в массив передается значение true, соответственно этот цвет становится недоступным для текущего узла:

foreach (var neighbor in graph[i])

{

if (colors[neighbor] != -1)

{

availableColors[colors[neighbor]] = true;

}

}

В конце происходит проверка доступных цветов, где вершине присваивается первый найденный доступный цвет:

int color = -1;

for (int j = 0; j < availableColors.Length; j++)

{

if (!availableColors[j])

{

color = j;

break;

}

}

colors[i] = color;

После раскрашивания всех вершин в назначенные им цвета применяется алгоритм, который выводит получившийся граф на экран. Данный алгоритм целиком описан в файле Form1.cs:

private void DrawGraph()

{

// Очистить экран

pictureBox1.Image = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics g = Graphics.FromImage(pictureBox1.Image);

g.Clear(Color.White);

// Нарисовать вершины

int radius = 20;

int centerX = pictureBox1.Width / 2;

int centerY = pictureBox1.Height / 2;

int offsetX = 100, offsetY = 100;

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

{

int angle = i \* (360 / graphSize);

int x = centerX + (int)(offsetX \* Math.Cos(Math.PI \* angle / 180));

int y = centerY + (int)(offsetY \* Math.Sin(Math.PI \* angle / 180));

Brush vertexColor = new SolidBrush(GetColorForVertex(i));

int ii = i + 1;

g.FillEllipse(vertexColor, x - radius, y - radius, radius \* 2, radius \* 2);

g.DrawString(ii.ToString(), new Font("Arial", 10), Brushes.Black, x - radius / 2, y - radius / 2);

}

// Нарисовать ребра

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

{

foreach (var neighbor in graph[i])

{

int x1 = centerX + (int)(offsetX \* Math.Cos(Math.PI \* i \* (360 / graphSize) / 180));

int y1 = centerY + (int)(offsetY \* Math.Sin(Math.PI \* i \* (360 / graphSize) / 180));

int x2 = centerX + (int)(offsetX \* Math.Cos(Math.PI \* neighbor \* (360 / graphSize) / 180));

int y2 = centerY + (int)(offsetY \* Math.Sin(Math.PI \* neighbor \* (360 / graphSize) / 180));

g.DrawLine(Pens.Black, x1, y1, x2, y2);

}

}

pictureBox1.Invalidate();

}

Цвета для окрашивания берутся из функции, которая возвращает их согласно выполненным вычислениям:

private Color GetColorForVertex(int vertex)

{

// Возвращаем цвет для вершины в зависимости от ее раскраски

switch (colors[vertex])

{

case 0: return Color.Red;

case 1: return Color.Orange;

case 2: return Color.Yellow;

case 3: return Color.Green;

case 4: return Color.Blue;

case 5: return Color.Purple;

default: return Color.Gray;

}

}

# **5. Тестирование**

Среда разработки Microsoft Visual Studio 2019 предоставляет все средства, необходимые при разработке и отладке многомодульной программы. Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна программы, алгоритмом программы, взаимодействием функций. Ниже продемонстрирован результат (Рисунки 13, 14, 15, 16, 17, 18) тестирования программы при вводе пользователем различного количества вершин и использовании различного функционала программы.

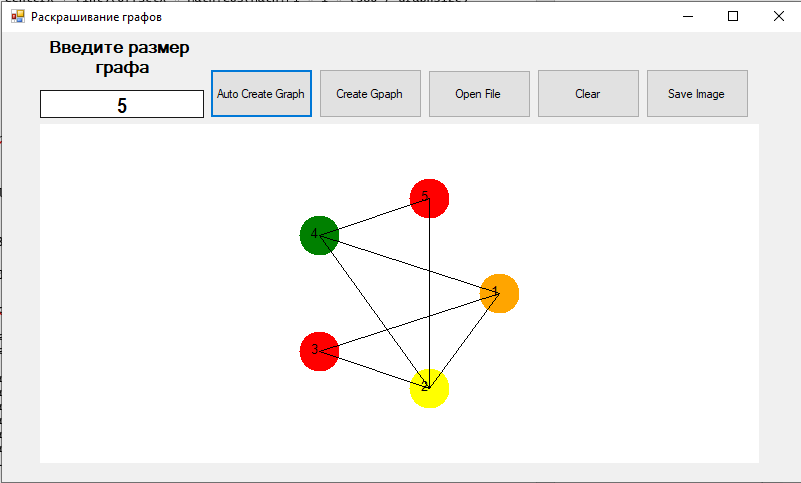


Рисунок 13 – Автоматическая генерация графа из 5 вершин

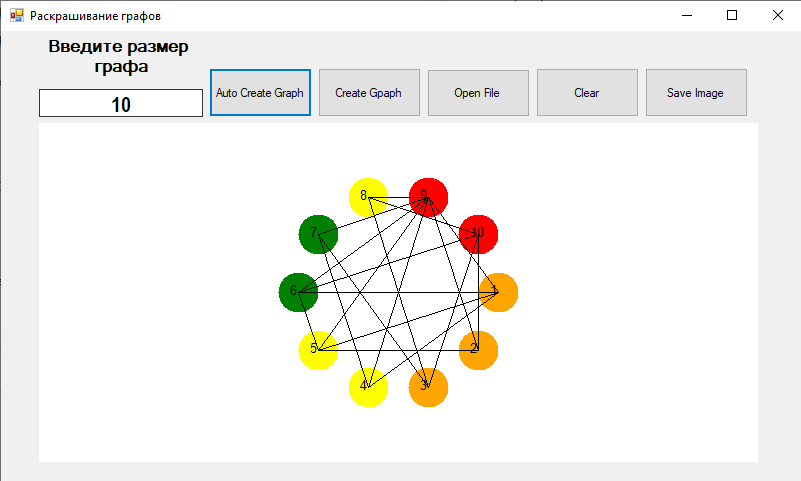


Рисунок 14 – Автоматическая генерация графа из 10 вершин

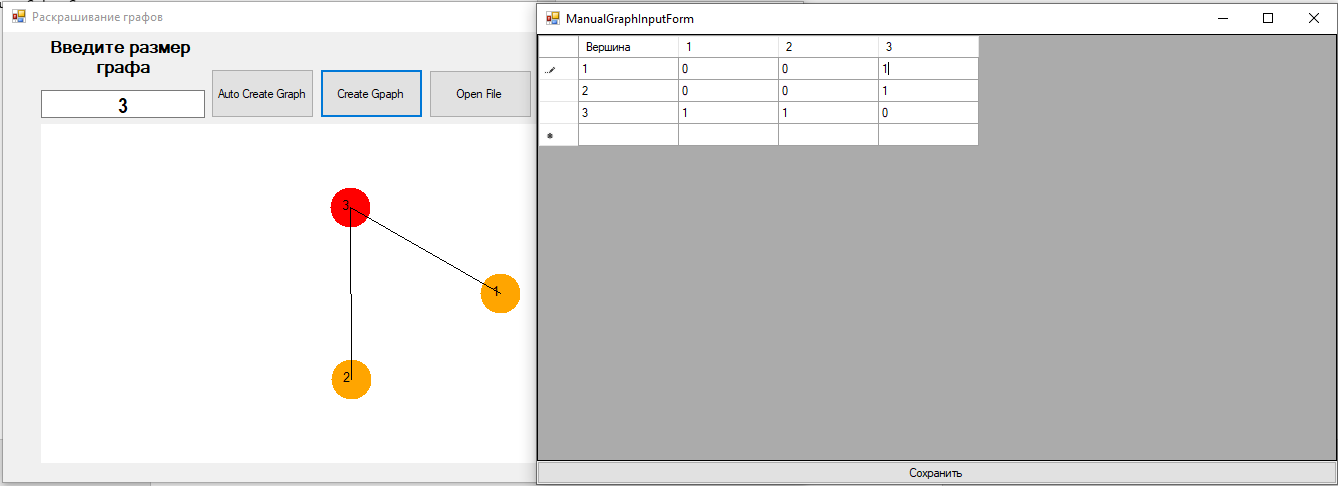


Рисунок 15 – Генерация графа при заполнении пользователем

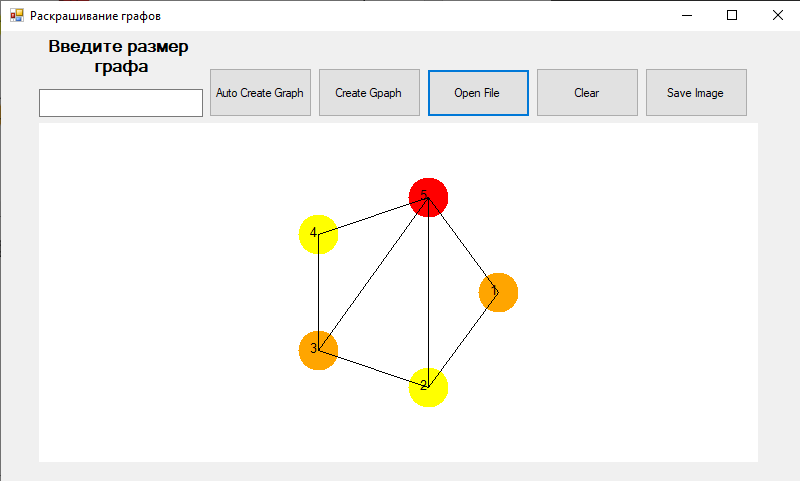


Рисунок 16 – Генерация графа при считывании из файла (Рисунок 17)

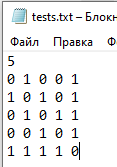


Рисунок 17 – Файл, использованный для генерации графа

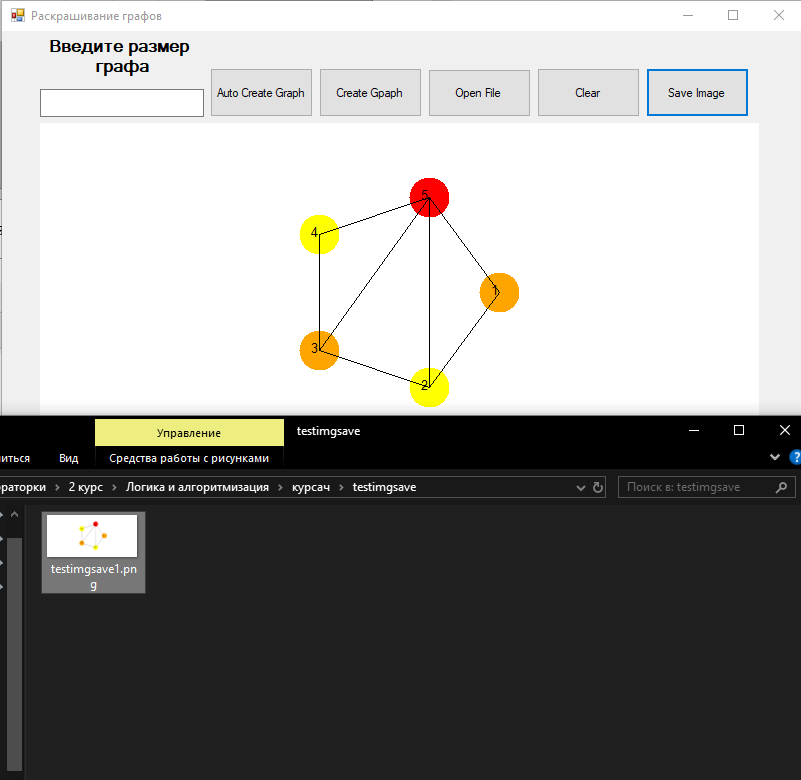


Рисунок 18 – Сохранение сгенерированного графа в виде изображения

Таблица 1 – Описание поведения программы при тестировании

|  |  |  |
| --- | --- | --- |
| Описание теста | Ожидаемый результат | Полученный результат |
| Запуск программы | Открытие окна с полем для ввода количества вершин и кнопками для взаимодействия с функционалом программы | Верно |
| Автоматическая генерация графа | При вводе в поле количества вершин графа и нажатии соответствующей кнопки на экране должен появится случайно сгенерированный граф с заданным количеством вершин | Верно |
| Генерация графа с использованием пользовательского ввода | При нажатии соответствующей кнопки должно открыться окно с таблицей, в которой пользователь сможет заполнить матрицу смежности | Верно |
| Генерация графа посредствам чтения из файла | При выборе необходимого файла на экране должна появиться матрица, сгенерированная согласно описанным в нем данным | Верно |
| Сохранение сгенерированного графа | Программа должна сохранить изображение сгенерированного графа | Верно |

В результате тестирования было выявлено, что программа успешно проверяет данные на соответствие необходимым требованиям.

# **6. Ручной расчет задачи**

Проведем проверку программы посредством ручных вычислений на примере графа с 5 вершинами на рисунке 19.

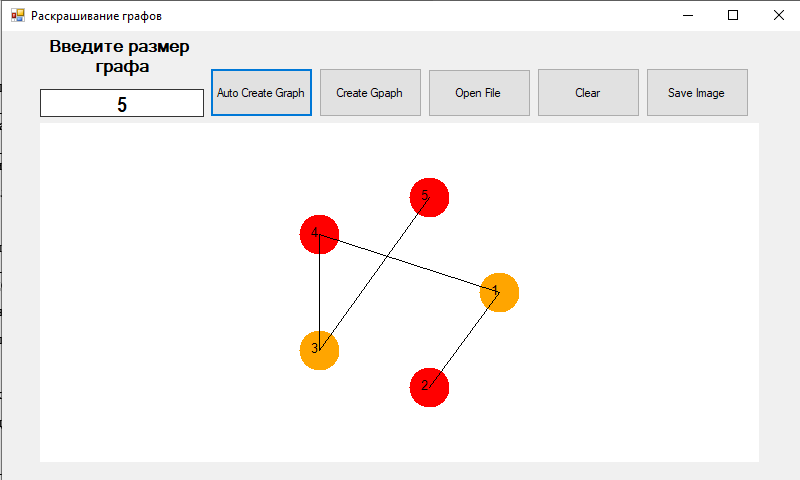


Рисунок 19 – Результат работы программы

Для начала покрасим вершину 1 в оранжевый цвет. В таком случае, смежные ей вершины 4 и 2 мы можем покрасить в красный.

Так как вершина вершина 3 смежна вершине 4, но не смежна вершине 1, можно покрасить ее в оранжевый цвет.

В свою очередь, вершина 5 смежна только вершине 3, соответственно ее можно покрасить в красный цвет.

В результате можно сделать вывод, что программа работает полностью исправно.

# **Заключение**

Таким образом, в процессе создания данного проекта разработана программа, реализующая алгоритм раскраски графов в Microsoft Visual Studio 2019.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания матриц смежностей, основанных на теории графов. Приобретены навыки по реализации «жадного» алгоритма. Углублены знания языка программирования C#.

Недостатком разработанной программы является графическое отображение сгенерированного графа. При текущем подходе, при большом количестве вершин изображение будет уменьшаться, что затруднит ознакомление с результатом.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

Программа попадает в категорию NP-полных, поэтому результат может отличаться от ручных расчётов.

# **Список литературы**

1. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и анализ- М.: МЦНМО,2001.- 960 с
2. Кристофидес Н. «Теория графов. Алгоритмический подход»- Мир, 1978
3. Герберт Шилдт «Полный справочник по C++»- Вильямс, 2006
4. З. Оре О. Графы и их применение: Пер. с англ. 1965. 176 с
5. Кандидатской диссертации Красновой А. Ю. «Матрицы инциденций и раскраски графа»
6. Харви Дейтел, Пол Дейтел. Как программировать на C/C++. 2009 г.
7. [www.wikipedia.com](http://www.wikipedia.com)
8. Дискретная математика: Методические указания к лабораторным работам / Рязанская государственная радиотехническая академия; Сост. А.М. Гостин. Рязань, 2004. 24 с.

# **Приложение A**

**Листинг программы**

Файл Form1.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace курсач\_\_WF\_

{

public partial class Form1 : Form

{

private int graphSize = 5; // Размер графа

private List<List<int>> graph; // Список смежности

private int[] colors; // Массив цветов для вершин

private Random random = new Random();

public Form1()

{

InitializeComponent();

graph = new List<List<int>>();

colors = new int[graphSize];

InitializeGraph();

}

private void InitializeGraph()

{

// Инициализация пустого графа

graph.Clear();

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

{

graph.Add(new List<int>());

}

}

private void DrawGraph()

{

// Очистить экран

pictureBox1.Image = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics g = Graphics.FromImage(pictureBox1.Image);

g.Clear(Color.White);

// Нарисовать вершины

int radius = 20;

int centerX = pictureBox1.Width / 2;

int centerY = pictureBox1.Height / 2;

int offsetX = 100, offsetY = 100;

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

{

int angle = i \* (360 / graphSize);

int x = centerX + (int)(offsetX \* Math.Cos(Math.PI \* angle / 180));

int y = centerY + (int)(offsetY \* Math.Sin(Math.PI \* angle / 180));

Brush vertexColor = new SolidBrush(GetColorForVertex(i));

int ii = i + 1;

g.FillEllipse(vertexColor, x - radius, y - radius, radius \* 2, radius \* 2);

g.DrawString(ii.ToString(), new Font("Arial", 10), Brushes.Black, x - radius / 2, y - radius / 2);

}

// Нарисовать ребра

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

{

foreach (var neighbor in graph[i])

{

int x1 = centerX + (int)(offsetX \* Math.Cos(Math.PI \* i \* (360 / graphSize) / 180));

int y1 = centerY + (int)(offsetY \* Math.Sin(Math.PI \* i \* (360 / graphSize) / 180));

int x2 = centerX + (int)(offsetX \* Math.Cos(Math.PI \* neighbor \* (360 / graphSize) / 180));

int y2 = centerY + (int)(offsetY \* Math.Sin(Math.PI \* neighbor \* (360 / graphSize) / 180));

g.DrawLine(Pens.Black, x1, y1, x2, y2);

}

}

pictureBox1.Invalidate();

}

private Color GetColorForVertex(int vertex)

{

// Возвращаем цвет для вершины в зависимости от ее раскраски

switch (colors[vertex])

{

case 0: return Color.Red;

case 1: return Color.Orange;

case 2: return Color.Yellow;

case 3: return Color.Green;

case 4: return Color.Blue;

case 5: return Color.Purple;

default: return Color.Gray;

}

}

private void ColorGraph()

{

// Применяем жадный алгоритм раскраски

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

{

bool[] availableColors = new bool[graphSize];

// Проверяем соседей

foreach (var neighbor in graph[i])

{

if (colors[neighbor] != -1)

{

availableColors[colors[neighbor]] = true;

}

}

// Находим первый доступный цвет

int color = -1;

for (int j = 0; j < availableColors.Length; j++)

{

if (!availableColors[j])

{

color = j;

break;

}

}

colors[i] = color;

}

DrawGraph();

}

private void Form1\_Load(object sender, EventArgs e)

{

}

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

{

// Генерация случайного графа

graphSize = int.Parse(txtGraphSize.Text);

InitializeGraph();

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

{

for (int j = i + 1; j < graphSize; j++)

{

if (random.Next(2) == 1)

{

graph[i].Add(j);

graph[j].Add(i);

}

}

}

colors = new int[graphSize]; // Сброс раскраски

ColorGraph();

}

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

{

pictureBox1.Image = new Bitmap(pictureBox1.Width, pictureBox1.Height);

Graphics g = Graphics.FromImage(pictureBox1.Image);

g.Clear(Color.White);

}

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

{

graphSize = int.Parse(txtGraphSize.Text);

var manualInputForm = new ManualGraphInputForm(graphSize);

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

{

graph = manualInputForm.GetGraph();

colors = new int[graphSize]; // Сброс раскраски

ColorGraph();

}

}

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

{

SaveFileDialog saveDialog = new SaveFileDialog();

saveDialog.Filter = "PNG Image|\*.png|JPEG Image|\*.jpg|Bitmap Image|\*.bmp";

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

{

try

{

// Сохраняем изображение

pictureBox1.Image.Save(saveDialog.FileName);

MessageBox.Show("Изображение успешно сохранено!", "Сохранение", MessageBoxButtons.OK, MessageBoxIcon.Information);

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при сохранении изображения: " + ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

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

{

OpenFileDialog openFileDialog = new OpenFileDialog();

openFileDialog.Filter = "Text Files|\*.txt";

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

{

try

{

string[] lines = File.ReadAllLines(openFileDialog.FileName);

graphSize = int.Parse(lines[0]); // Размер графа

InitializeGraph();

colors = new int[graphSize]; // Сброс раскраски

// Читаем матрицу смежности

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

{

string[] row = lines[i + 1].Split(' ');

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

{

if (row[j] == "1") // Если связь существует

{

graph[i].Add(j);

}

}

}

ColorGraph();

}

catch (Exception ex)

{

MessageBox.Show("Ошибка при чтении файла: " + ex.Message, "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

}

}

Файл ManualGraphInputForm.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 курсач\_\_WF\_

{

public partial class ManualGraphInputForm : Form

{

private int graphSize;

private DataGridView dgv;

public ManualGraphInputForm(int size)

{

InitializeComponent();

graphSize = size;

InitializeDataGridView();

}

private void InitializeDataGridView()

{

dgv = new DataGridView

{

ColumnCount = graphSize + 1, // Первая колонка для номеров вершин

RowCount = graphSize, // Столько же строк, сколько и вершин

Dock = DockStyle.Fill

};

// Создаем заголовки столбцов

dgv.Columns[0].Name = "Вершина";

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

{

dgv.Columns[i].Name = i.ToString();

}

// Заполняем таблицу нулями (по умолчанию нет соединений)

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

{

int ii = i + 1;

dgv.Rows[i].Cells[0].Value = ii.ToString();

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

{

dgv.Rows[i].Cells[j].Value = 0; // 0 - значит, нет ребра

}

}

// Добавляем кнопку сохранения

Button btnSave = new Button

{

Text = "Сохранить",

Dock = DockStyle.Bottom

};

btnSave.Click += BtnSave\_Click;

this.Controls.Add(dgv);

this.Controls.Add(btnSave);

}

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

{

List<List<int>> graph = new List<List<int>>();

// Чтение данных из таблицы и формирование списка смежности

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

{

var neighbors = new List<int>();

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

{

if (Convert.ToInt32(dgv.Rows[i].Cells[j].Value) == 1)

{

neighbors.Add(j - 1); // Сосед по индексу

}

}

graph.Add(neighbors);

}

// Возвращаем граф в основную форму

this.DialogResult = DialogResult.OK;

this.Tag = graph;

this.Close();

}

public List<List<int>> GetGraph()

{

return (List<List<int>>)this.Tag;

}

private void ManualGraphInputForm\_Load(object sender, EventArgs e)

{

}

}

}