**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Департамент программной инженерии

|  |  |  |
| --- | --- | --- |
| СОГЛАСОВАНО  Руководитель,  доцент департамента  программной инженерии факультета компьютерных наук,  кан. тех. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р.З.Ахметсафина  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г. |  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия», профессор департамента программной инженерии, кан. тех. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Шилов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020г. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** | RU.17701729.04.01-01 81 01-1-ЛУ | | **ИГРА «ПЕРЕМЫЧКИ»**  **Пояснительная записка**  **ЛИСТ УТВЕРЖДЕНИЯ**  RU.17701729.04.01-01 81 01-1-ЛУ | | |
|  |  | |
| Исполнитель  студент группы БПИ197  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Джапаров Э.М. /  «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г. | |
|  | | |
|  | |  |

**Москва 2020**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| УТВЕРЖДЕН  RU.17701729.04.01-01 81 01-1-ЛУ |  | |  | |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** | RU.17701729.04.01-01 81 01-1-ЛУ | | **ИГРА «ПЕРЕМЫЧКИ»**  **Пояснительная записка**  **RU.17701729.04.01-01 81 01-1**  **Листов 55** | | | | |
|  | |  | | |
|  | | |
|  | | | | |
|  | | | |  |

**Москва 2020**

**АННОТАЦИЯ**

Данный документ является пояснительной запиской к программному продукту – «Игра «Перемычки»» («Game «Bridges»»).

Настоящая пояснительная записка к программе «Игра «Перемычки»» содержит следующие разделы: «Введение», «Назначение разработки и область применения», «Технические характеристики», «Ожидаемые технические и экономические показатели», «Источники, использованные при разработке».

В разделе «Введение» указано наименование программы, краткое наименование программы и документы, на основании которых ведется разработка.

В разделе «Назначение разработки и область применения» указано функциональное назначение программы, эксплуатационное назначение программы и краткая характеристика области применения программы.

В разделе «Технические характеристики» содержатся следующие подразделы:

− постановка задачи на разработку программы;

− описание алгоритма и функционирования программы;

− описание метода организации входных и выходных данных;

− описание и обоснование выбора состава технических и программных средств.

В разделе «Ожидаемые технические и экономические показатели» указана предполагаемая потребность и экономические преимущества разработки по сравнению с отечественными и зарубежными образцами или аналогами.

Настоящий документ разработан в соответствии с требованиями:

1) ГОСТ 19.101-77 Виды программ и программных документов;

2) ГОСТ 19.103-77 Обозначения программ и программных документов;

3) ГОСТ 19.104-78 Основные надписи;

4) ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению.

Изменения к Пояснительной записке оформляются согласно ГОСТ 19.603-78, ГОСТ 19.604-78.

Перед прочтением данного документа рекомендуется ознакомиться с терминологией, приведенной в Приложении 1 настоящей пояснительной записки.

ОГЛАВЛЕНИЕ

[1. ВВЕДЕНИЕ 5](#_Toc41337409)

[1.1. Наименование программы 5](#_Toc41337410)

[1.2. Документы, на основании которых ведется разработка 5](#_Toc41337411)

[2. НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ 6](#_Toc41337412)

[2.1. Назначение программы 6](#_Toc41337413)

[2.1.1. Функциональное назначение 6](#_Toc41337414)

[2.1.2. Эксплуатационное назначение 6](#_Toc41337415)

[2.2. Краткая характеристика области применения 6](#_Toc41337416)

[3. ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ 7](#_Toc41337417)

[3.1. Постановка задачи на разработку программы: 7](#_Toc41337418)

[3.2. Описание применяемых математических методов 7](#_Toc41337419)

[3.3. Описание алгоритма и функционирования программы 10](#_Toc41337420)

[3.3.1. Алгоритм генерации игрового поля 10](#_Toc41337421)

[3.3.2. Алгоритм решения сгенерированной задачи 16](#_Toc41337422)

[3.3.3. Алгоритм записи/чтения данных о сгенерированных заданиях и решениях в файл 18](#_Toc41337423)

[3.3.4. Автосохранение текущего состояния игры на текущем уровне 19](#_Toc41337424)

[3.3.5. Алгоритм генерации подсказки пользователю 21](#_Toc41337425)

[3.3.6. Алгоритм перехода к новому полю 22](#_Toc41337426)

[3.3.7. Алгоритм перехода к решенным/просмотренным заданиям 23](#_Toc41337427)

[3.3.8. Алгоритм взаимодействия пользователя с игровыми вершинами (кнопками) 23](#_Toc41337428)

[3.3.9. Алгоритм проверки решенного задания 24](#_Toc41337429)

[3.3.10. Алгоритм сохранения результата и дальнейшего его выведения в статистике 25](#_Toc41337430)

[3.4. Описание и обоснование выбора метода организации входных и выходных данных 26](#_Toc41337431)

[3.4.1. Описание метода организации входных и выходных данных 26](#_Toc41337432)

[3.4.2. Обоснование выбора методов организации входных и выходных данных 26](#_Toc41337433)

[3.5. Описание и обоснование выбора состава технических и программных средств 26](#_Toc41337434)

[4. ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ 27](#_Toc41337435)

[4.1. Прогнозируемая потребность 27](#_Toc41337436)

[4.2. Экономические преимущества разработанного продукта перед отечественными и зарубежными образцами или аналогами 27](#_Toc41337437)

[5. ПРИЛОЖЕНИЯ К ДОКУМЕНТУ 28](#_Toc41337438)

[ПРИЛОЖЕНИЕ 1 28](#_Toc41337439)

[ТЕРМИНОЛОГИЯ 28](#_Toc41337440)

[ПРИЛОЖЕНИЕ 2 30](#_Toc41337441)

[ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ КЛАССОВ 30](#_Toc41337442)

[ПРИЛОЖЕНИЕ 3 31](#_Toc41337443)

[ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПОЛЕЙ, МЕТОДОВ И СВОЙСТВ 31](#_Toc41337444)

# ВВЕДЕНИЕ

## Наименование программы

Программа называется «Игра «Перемычки»»(«Game «Bridges»»).

Краткое название программы для пользователя – «BridgesPuzzle».

## Документы, на основании которых ведется разработка

Приказ декана факультета компьютерных наук Национального исследовательского университета «Высшая школа экономики» №2.3-02/1212-01 от 12.12.2017 «Об утверждении тем, руководителей курсовых работ студентов образовательной программы Программная инженерия факультета компьютерных наук».

# НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ

## Назначение программы

### Функциональное назначение

Функциональным назначением программы является предоставление пользователю игровых полей трех видов в зависимости от уровня с гарантированным решением.

Программа проверяет решение пользователя и сообщает ему, правильно ли решено поле.

Программа предоставляет пользователю возможность получить подсказку на легком и среднем уровнях.

Программа решает предоставленные пользователю игровые поля.

Программа дает возможность пользователю сохранить свой результат решения.

Программа сохраняет текущий процесс игры при выходе из окна игрового поля.

Программа предоставляет пользователю изучение правил.

**Правила игры:**

Пользователь должен решать игровые поля так, чтобы от каждого кружочка было проведено ровно столько перемычек к соседним кружочкам, какое значение указано внутри данного кружочка, таким образом, чтобы в одном секторе была максимум одна связь между диагональными соседями.

### Эксплуатационное назначение

Игра является непростой головоломкой, которая решается посредством логических рассуждений, хорошей памяти и базовых математических навыков. Данной игрой могут пользоваться люди любого возраста для развития своих когнитивных способностей и для улучшения уровня своего логического мышления.

## Краткая характеристика области применения

В образовательных целях игра может быть использована студентами и школьниками в курсе теории графов. Учащийся, эксплуатируя приложение, может наглядно представить себе, как устроены некоторые бинарные отношения на упорядоченных/неупорядоченных множествах. На основании этой игры, а точнее на основании полей из этой игры, в учебных учреждениях студенту могут объясняться базовые понятия теории графов, такие как: пара множество элементов и иррефлексивное, симметричное бинарное отношение смежности на этом множестве, порядок и размер графа, степень вершины и ее окрестность, сложные и простые пути, циклы, компоненты связности и так далее.

# ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ

## Постановка задачи на разработку программы:

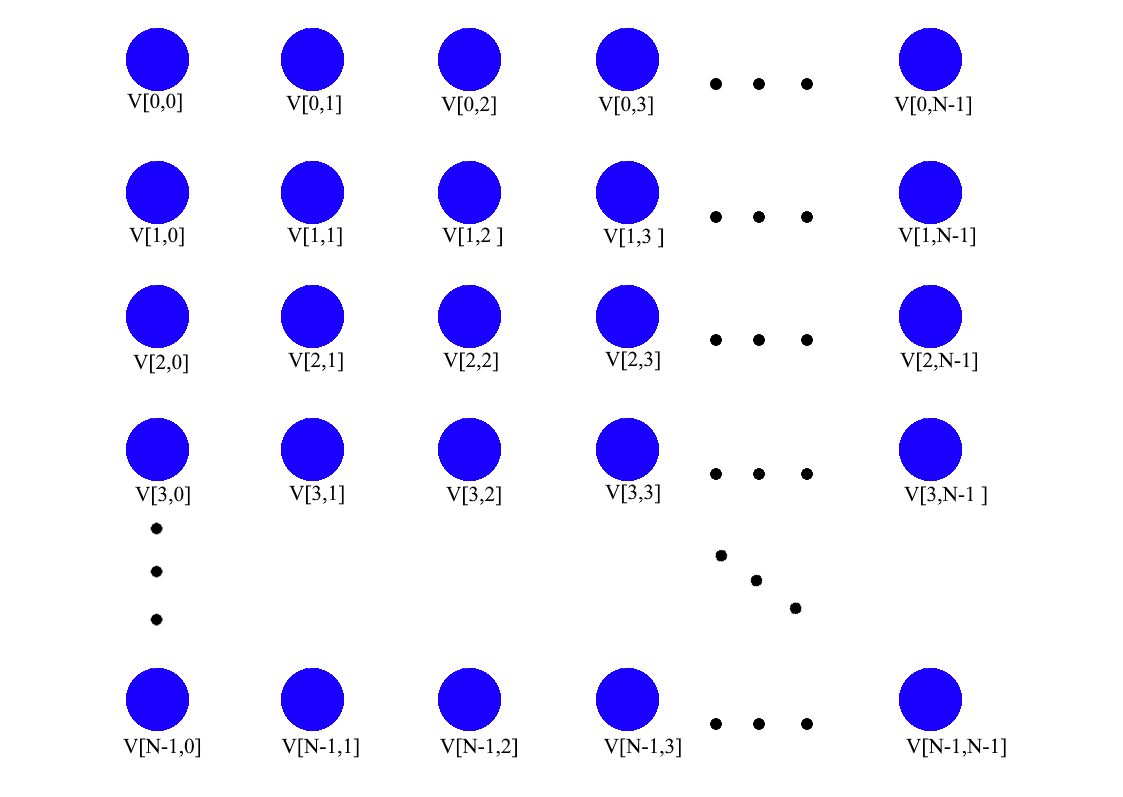
Цели текущего проекта:

1. Программа должна генерировать и выдавать пользователю неограниченное количество игровых поле с гарантированным решением;
2. Поля должны быть трех видов по размеру в зависимости от выбранного пользователем уровня;
3. Должно быть реализовано автосохранение текущего процесса игры на каждом уровне и при старте новой игры пользователю должен выдаваться последний сохраненный процесс, если он есть на текущем уровне;
4. Пользователю на легком уровне каждое поле должно генерироваться с готовыми первоначальными связями, то есть пользователю надо только дорешать поле.
5. На среднем уровне на каждом поле, когда оно пустое, пользователю должна даваться возможность получить подсказку. Но при желании пользователь может обойтись без подсказки в начале игры. Поэтому на среднем уровне поле выдается пользователю без готовых связей-подсказок.
6. Должна быть реализована проверка на то, правильно ли решено данное игровое поле.
7. Пользователю на каждом поле должна предоставляться возможность посмотреть правильное решение к текущему полю.
8. Должен быть реализован список всех результатов, куда записываются имена пользователей с их временем решения поля.
9. Нужно реализовать справочник по правилам игры, по тому, как решать головоломку, и по специфике управления.
10. Выделять вершины определенными цветами при совершении определенных действий над вершинами для того, чтобы процесс игры был удобнее.

## Описание применяемых математических методов

Каждое игровое поле можно считать упорядоченным множеством вершин V мощности N x N, где каждый элемент множества ассоциируется с соответствующей ему «игровой» вершиной (кружочком) и у каждого элемента множества есть своя характеристика, именуемая степенью или весом, которая генерируется внутри соответствующего «игрового» элемента на данном игровом поле. Множество вершин может быть представлено в виде квадратной матрицы (рис. 1):

*Рисунок 1*. Пустое игровое поле размера N x N.



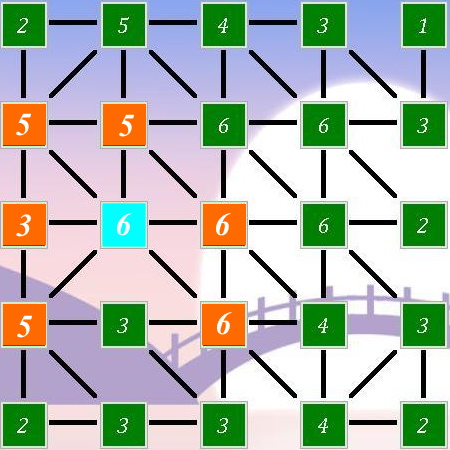
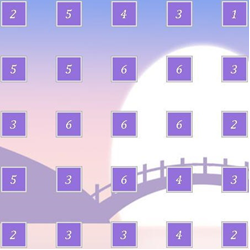
Решением подобного игрового поля является построение некоторого иррефлексивного и симметричного бинарного отношения S на этом упорядоченном множестве вершин V. Каждый элемент этого отношения (a,b) и соответствующую ему пару симметричную (b,a) будем называть одним словом – связь. Это отношение S такое, что:

* На данном упорядоченном наборе вершина , где 0≤i<N, 0≤j<N в общем случае может состоять в построенном бинарном отношении S только с вершинами , где 0≤m<N, 0≤n<N, m ϵ {i-1, i, i+1}, n ϵ {j-1, j, j+1}, m!=n. Эти вершины называются соседними. Если есть отношение между соседними вершинами, то они называются смежными.
* Количество пар бинарного отношения S, в которые входит некоторый элемент , должно равняться 2\*d, где d – мощность окрестности данной вершины или, просто, количество соседних вершин, смежных с данной вершиной. Иначе вершина должна состоять в бинарном отношении с d соседними вершинами.
* Важно, что если есть отношение между одними (главными или побочными) диагональными вершинами в одном секторе, то не может быть другой диагональной связи (побочной или главной, соответственно) в этом же секторе. То есть графически это означает, что не должно быть пересечений диагональных связей. Таким образом в правильно построенном графе от каждой вершины должно быть проведено к соседним вершинам ровно столько ребер, сколько указано в весе или степени вершины.
* Нет вершин с нулевой степенью. Минимальная степень вершин равна 1. Максимальная зависит от позиции.
* Если первый индекс вершины либо 0, либо N-1, и второй индекс либо 0, либо N-1, то вершина может иметь максимум 3 связи в силу своего расположения.
* Если первый индекс вершины в диапазоне [1..N-2] и второй индекс либо 0, либо N-1 (то есть вершина в первом столбце, либо в последнем столбце), то вершина может иметь максимум 5 связей.
* Если второй индекс вершины в диапазоне [1..N-2] и первый индекс либо 0, либо N-1 (то есть вершина в первой строке, либо в последней строке), то вершина может иметь максимум 5 связей.
* В остальных случаях, если первый индекс в диапазоне [1,N-2] и второй индекс в диапазоне [1,N-2], то вершина может иметь максимум 8 связей.
* Максимально возможное количество элементов в бинарном отношении:

.

* Построенное бинарное отношение на множестве и само множество образуют неориентированный граф. В данной игре этот граф может состоять из нескольких КС (компонент связностей).

*Рисунок 2*. Некоторое множество размера 5 х 5.



*Рисунок 3*. Пример бинарного отношения на множестве 5 х 5.

Рассмотрим пример бинарного отношения, который является решением на даннном множестве 5 на 5(рис. 3)

## Описание алгоритма и функционирования программы

### Алгоритм генерации игрового поля

В игре пользователю предоставляется возможность выбрать три вида полей размерности *N x N* в зависимости от уровня. Мы рассмотрим весь процесс генерации посредством данного алгоритма на примере поля конкретного размера 4 x 4, то есть N = 4 – размер поля.

Каким же образом пользователю предоставляются поля и в чем же состоит алгоритм генерации поля? Стоит сказать, что каждая вершина (она же узел) в игровом поле ассоциируется с элементом типа Node, то есть все, что происходит с вершиной, параллельно так же отражается и на соответствующем объекте Node. У каждого объекта Node есть поля: его позиции в массиве (они же позиции вершины в игровом поле), степень (она же степень вершины в поле), список (List<Node>) смежных элементов(соседние элементы, с которыми соединена вершина). Всегда, когда затрагивается вершина, по умолчанию затрагивается и объект типа Node, который ассоциируется с этой вершиной. Таким образом, игровое поле – это наглядное представление двумерного массива, то есть все операции, проводящиеся с массивом, проводятся и с игровым полем, и наоборот. Под полем всегда также подразумевается и соответствующий двумерный массив узлов (объектов Node), и наоборот.

**public** **class** Node

{

*// Позиции узла в двумерном массиве.*

*// Они же позиции узла в игровом поле.*

**int** x;

**int** y;

*// Степень вершины(узла).*

**int** weight;

*// Список смежных вершин.*

    List<Node> connectedNodes;

}

Суть алгоритма в том, что сначала создается двумерный массив узлов(элементов типа Node) без связей между узлами и без степеней узлов. По этому двумерному массиву затем будет генерироваться квадратное поле, но наглядно на данном этапе игровое поле, которое соответствует двумерному массиву, выглядит следующим образом(рис. 4):

**public** **class** Form

{

*// Размер поля зависит от уровня: 4-легкий, 5-средний, 6-сложный.*

**int** fieldSize = level == 0 ?4:level ==1?5:6;

*// Двумерный массив из узлов(вершин) игрового поля.*

    Node[,] nodesForGen;

*/// Инициализация массива nodes и инициализация его элементов.*

**public** **void** FirstPartOFFieldGeneration()

    {

        nodesForGen = new Node[fieldSize,fieldSize];

**for**(**int** i = 0; i< nodesForGen.GetLength(0);i++)

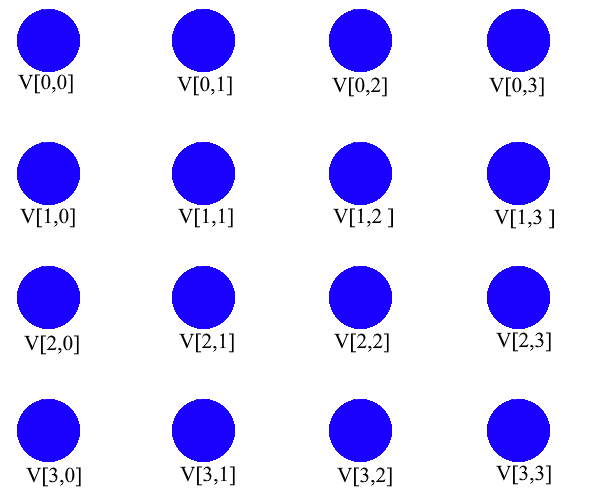
**for**(**int** j = 0; j<nodesForGen.GetLength(1);j++)

            {

                nodesForGen[i,j] = new Node(i,j);

            }

    }



}

*Рисунок 4*. 1 шаг алгоритма генерации

На первом шаге создается пустое (без связей между узлами и без степеней узлов) поле.

Второй шаг. Затем циклически осуществляется осуществляется обход этого двумерного массива по строкам. На каждой итерации цикла фиксируется текущий элемент(вершина) и заполняется его список из соседей по принципу «Горизонтально вправо» и «Вертикально вниз» и («Левый верхний-Правый Нижний» или «Правый Верхний-Левый Нижний».

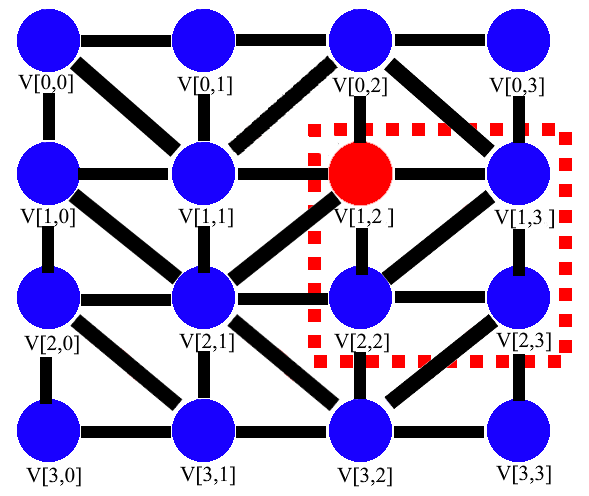
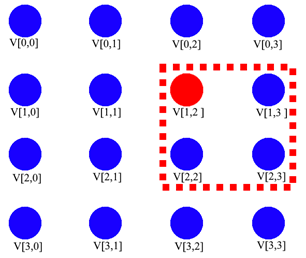
Что же это за принцип заполнения? Это значит, что от каждого зафиксированного элемента на каждой итерации цикла проводится обязательно: горизонтальная связь справа с соседом по горизонтали(она же Г-связь), если он есть; вертикальная связь с соседом снизу по вертикали(она же В-связь), если он есть; главная диагональная связь(она же ГД-связь), в секторе, который соответствует текущей вершине, с соседом по диагонали справа-снизу в секторе, если он есть, либо проводится побочная диагональная связь(ПД-связь) между соседями данного элемента, расположенными по горизонтали(сосед справа) и по вертикали(сосед снизу), если они есть. Не может быть проведена главная и побочная диагональные связи одновременно в одном секторе, то есть нет пересечений. Провести связь между двумя соседними объектами типа Node и (они же узлы, вершины) – это одно и то же, что добавить друг друга в свой список смежных соседей:

V[i,j].AddToConnectedNodesList(V[m,n]);

V[m,n].AddToConnectedNodesList(V[i,j]);

Сектор вершины (рис. 5-6) есть часть игрового поля из четырех, либо из двух вершин, либо из одной вершины (сама же вершина ) и в него помимо вершины входят вершины , либо (если второй индекс N-1, то есть вершина в последнем столбце) или (если первый индекс N-1, то есть вершина в последней строчке), либо других вершин нет (если оба индекса N-1):

*Рисунок 5*. Сектор вершины *Рисунок 6*. 2 шаг алгоритма генерации



Рассмотрим поле 4 на 4 и вершину (рис. 5-6). Сектор для вершины (выделена красным цветом) обведен красным пунктиром. То есть при обходе соответствующей этому полю матрицы на какой – то итерации фиксируется вершина и от нее в обязательном порядке проводятся связи к и и с вероятностью 0.5 проводится связь к , либо с такой же вероятностью проводится связь между вершинами и . В данном варианте заполнения (рис. 6) поля в секторе вершины проведено три связи: две от к вертикальному и горизонтальному соседям и одна связь между вертикальным и горизонтальным соседями данной вершины, то есть на этой итерации к степень вершины увеличилась на 2.

Таким образом у элементов двумерного массива nodes полностью заполняется смежными соседями список nodesConnected.

Количество связей вычисляется по формуле: .

Стоит также заметить, что у каждого элемента в полностью заполненном связями поле имеется минимум 2 связи гарантированно. У элементов с обоими индексами в диапазоне [1,N-2] имеется минимум 4 связи гарантированно. Перед тем, как провести связи от текущей вершины на данной итерации по принципу Г,В,ГД,ПД – связи, стоит заметить, что если вершина имеет оба индекса в диапазоне [1,N-1], то у этой вершины уже перед проведением связей имеется как минимум две связи. Если вершина в первой строке или первом столбце, то у нее гарантированно имеется одна связь, если оба индекса не равны 0. Код второго шага алгоритма:

**public** **void** SecondPartOfFieldGeneration()

    {

**for** (**int** i = 0; i < nodesForGen.GetLength(0); i++)

**for** (**int** j = 0; j < nodesForGen.GetLength(1); j++)

                {

*// Если вершина не в правом нижнем углу.*

**if** (i != fieldSize - 1 || j != fieldSize - 1)

                    {

*// Если текущий узел не в последнем столбце и не в последней строке.*

**if** (0 <= i && i < fieldSize-1 && 0 <= j && j < fieldSize - 1)

                        {

*// Соединение с вертикальным соседом.*

                            AddToNodesList(nodesForGen[i, j], nodesForGen[i + 1, j]);

*// Соединение с горизонтальным соседом.*

                            AddToNodesList(nodesForGen[i, j], nodesForGen[i, j + 1]);

*// Соединение с соседом по-диагонали.*

**if** (rnd.Next(0, 2) == 0)

                                AddToNodesList(nodesForGen[i, j], nodesForGen[i + 1, j + 1]);

*// Соединение горизонтального и вертикального соседей данного // элемента nodesForGen[i,j]*

**else**

                                AddToNodesList(nodesForGen[i + 1, j], nodesForGen[i, j + 1]);

                        }

*// Текущий узел в последнем столбце или в последней строке.*

**else**

                        {

*// Узел в последней строке.*

**if** (i == nodesForGen.GetLength(1) - 1)

                                AddToNodesList(nodesForGen[i, j], nodesForGen[i, j + 1]);

**else**

                                AddToNodesList(nodesForGen[i, j], nodesForGen[i + 1, j]);

                        }

                    }

                }

    }

Третий шаг алгоритма заключается в том, что у каждого элемента двумерного массива удаляются рандомные сочетания элементов в списке *nodesConnected*, но никак не может быть такого, что у элемента удалились все связи – процесс рандома происходит в интервале от 1, то есть одна связь гарантированно остается. Если подробнее, то происходит обход массива циклом по каждому элемента этого массива, и в начале каждой итерации в зависимости от того, на какой позиции находится элемент и какое на начало итерации количество связей осталось после удаления на предшествующих итерациях, у элемента на текущей итерации удаляются рандомные сочетания элементов.

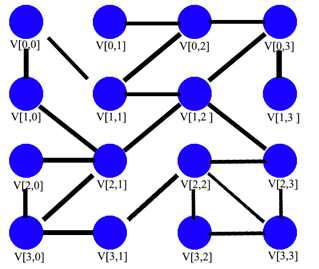
Если элемент имеет индексы в диапазоне [1,N-2] и до итерации у него удалилось m связей, которые были к нему проведены в предшествующих итерациях, то на текущей итерации на возможность удаления рассматриваются связи которые проведены в секторе данной вершины:

* Если на начало итерации у элемента количество связей больше двух и имеется в секторе элемента ПД-связь, то с вероятностью 0.5 удаляется Г-связь данного элемента, и с такой же вероятностью удаляем В-связь, с такой же вероятностью удаляем ПД-связь в секторе.
* Если на начало итерации у элемента количество связей равно двум и имеется в секторе элемента ПД-связь, то с вероятностью 0.25 удаляем Г-связь, если В-связь не удалилась, и с вероятностью 0.25 удаляем В-связь, если Г-связь не удалилась, с вероятностью 0.5 удаляем ПД-связь.
* Если на начало итерации у элемента количество связей больше трех и в секторе элемента имеется ГД-связь, то с вероятностью 0.5 удаляем Г-связь, с вероятностью 0.5 удаляем В-связь, с вероятностью 0.5 удаляем ГД-связь.
* Если на начало итерации у элемента количество связей равно трем и в секторе элемента имеется ГД-связь, то мы удаляем либо сочетания из каких-то двух связей(сочетаний будет 3: Г,В-связи, Г-ГД-связи, В-ГД-связи) с вероятностями 1/3, либо одну связь с вероятностью 1/3.

Если элемент в последнем столбце или последней строке:

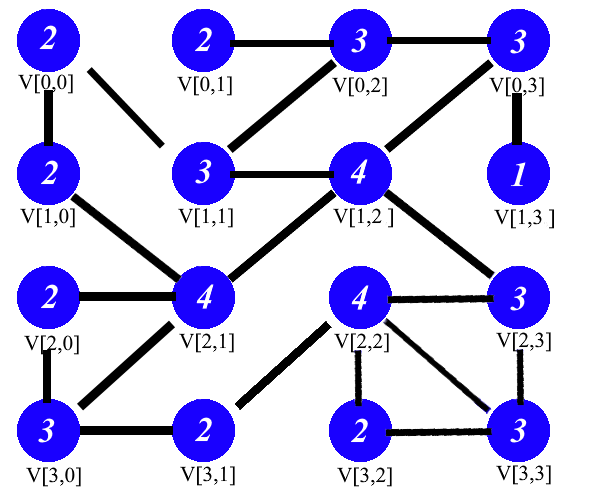
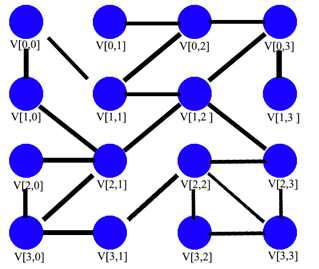
* Если элемент в последнем столбце, то есть его второй индекс N-1, и на начало итерации у элемента больше одной связи, то с вероятностью 0.5 удаляем В-связь.
* Если элемент в последней строке, то есть его первый индекс N-1, и он не предпоследний в строке и у него больше одной связи, то с вероятностью 0.5 удаляем Г-связь.
* Если текущий элемент предпоследний в последней строке, то есть его первый и второй индексы равны N-1 и N-2, соответственно, и если у него больше одной связи и больше одной связи также у самого последнего элемента, то есть у элемента, у которого оба индекса равны N-1, то с вероятностью 0.5 удаляем Г связь текущего элемента, иначе не удаляем.

В конце третьего шага алгоритма, мы получим двумерный массив элементов с непустыми списками смежных соседей. Наглядно этот массив представим на рис. 7:



*Рисунок 7*. Массив элементов с проведенными связями.

Далее осуществляется обход массива еще раз и на основании количества уже проведенных связей устанавливаются степени вершин (рис. 8). Для напоминания: степень вершины - это мощность окрестности вершины, то есть количество вершин, смежных с данной вершиной, или, просто, количество связей, исходящих от данной вершины.



*Рисунок 8*. Расстановка степеней вершин на основании проведенных связей.

Таким образом в конце выполнения третьего шага алгоритма на выходе мы получим двумерный массив, элементы которого корректно соединены с соседними элементами и у этих элементов корректные значения степеней. В итоге все для генерации задания с гарантированным решением готово.

Четвертый шаг. Для того, чтобы записать информацию о поле для дальнейшего предоставлению пользователю этого задания, в цикле осуществляется обход массива и в переменную *info* типа *String* записывается информация о степенях каждого узла в нужном порядке: сначала последовательно записываются степени элементов *k*-ой строки и затем происходит переход к *k+1*-ой строке и по ней также происходит обход и записываются степени элементов уже этой строки и так далее до последней строки массива. В приведенном выше примере(рис. 8) поля 4 на 4 переменная *info* имеет вид:

*info = “2 2 3 3;2 3 4 1;2 4 4 3;3 2 2 3”*

**int** weight;

**string** info = "";

*// Заполнение поля Weight у каждого элемента массива nodesForGen*

*// Запись информации о поле в переменную info для генерации задания.*

**for** (**int** i = 0; i < nodesForGen.GetLength(0); i++)

{

**for** (**int** j = 0; j < nodesForGen.GetLength(1); j++)

    {

      weight = nodesForGen[i,j].Weight.ToString();

        info += j != fieldSize - 1 ? weight + " " : weight;

    }

    info += i != nodesForGen.GetLength(1) - 1 ? "," : "";

}

Пятый шаг. Переменная *info* записывается в текстовый файл с расширением *.khan*, который соответствует текущему уровню и в котором хранится информация обо всех последовательно сгенерированных на данном уровне заданиях. При считывании из файла информации о задании считывается также и номер строки в файле, где расположено данное задание. Этот номер будет идентификатором данного задания, чтобы пользователь при желании мог вернуться к этому заданию по идентификатору данного задания.

Пример идентификатора id = 232:

* Уровень средний, так как первая цифра - 2. Номер задания(номер строки-1) в соответствующем уровню файле - 32.

При старте пользователем новой игры или при переходе к новым полям срабатывает весь вышеприведенный алгоритм, считывается информация из файла о сгенерированном задании и пользователь получает задание с гарантированным решением:

Node[,] nodes = new Nodes[fieldSize, fieldSize];

**string**[] taskArr = new **string**[countOfTasks];

**string** task = taskArr[numOfField];

**int** tempWeight;

**string**[] taskSplitted = task.Split(',');

**string**[] row = new **string**[fieldSize];

**for** (**int** i = 0; i < fieldSize; i++)

  {

    row = taskSplitted[i].Split(' ');

**for** (**int** j = 0; j < fieldSize; j++)

      {

**int**.TryParse(row[j],**out** tempWeight);

          nodes[i,j] = new Node(i,j,tempWeight);

      }

  }

По данному массиву *nodes* затем достраивается графическая часть задания в *WinForms*. Приводить специфику построения графической части алгоритма необходимости нет.

### Алгоритм решения сгенерированной задачи

Данный алгоритм сильно завязан на алгоритм генерации задачи. В третьем шаге приведенного алгоритма генерации задания информация, содержащаяся в массиве *nodesForGen*, представляет из себя уже в то же время корректное решение задания, которое соответствует полю, которое будет строиться по данному массиву. То есть после удаления случайных связей и последующей расстановки степеней по оставшимся связям мы имеем корректное задание, как упоминалось ранее, и его правильное решение, поскольку именно по нему и генерируется задание.

Информация о решении записывается, как информация о связях. Поскольку наше поле полностью соответствует графу, который строится на основании упорядоченного множества вершин(а это вершины поля) и иррефлексивного, симметрического бинарного отношения(а это, по сути дела, удвоенное множество связей, так как связь между вершинами a и b соответствует двум элементам(парам) бинарного отношения: (a,b) и (b,a)), то мы можем так же рассмотреть и множество связей – элементов(пар вершин) бинарного отношения.

Информация о связях готового поля с решением записывается в двумерный массив с размерностями первого и второго измерения, равными 2\*N – 1, что достаточно очевидно. Что же представляет из себя этот массив? Это двумерный массив *connections* типа *string[,]*, элементы которого это строки вида “G”,”V”,”PVLN”,”LVPN”,”P” – это, на самом деле, строчное представление принципа полного заполнения поля: Г,В,ПД,ГД-связи. То есть, Г-связь это есть “G”, В-связь это есть “V”, ПД-связь – “PVLN”, ГД-связь – “LVPN”. Каждый элемент этого массива содержит в себе информацию, о том, какие два узла он соединяет. То есть по индексам и значению некоторого элемента этого массива можно узнать, какой индекс имеют узлы в массиве Node, которые соединяются данным элементом – связью из массива *connections*, и, наоборот, по индексам элементов массива Node, можно узнать индексы связей, которыми соединяются смежные вершины. Например, если вершина соединена с соседней вершиной , то индексы связи, которая соединяет две эти вершины, в массиве *connections* равны соответственно . И, наоборот, если рассмотреть некоторый элемент из массива *connections*, допустим, со значением “PVLN” и индексами , соответственно, то по его значению очевидно, что это побочная диагональная связь в секторе некоторого узла, и по индексам этой связи можно восстановить индексы элементов *Node* в массиве *nodesForGen* которые соединяются этой связью: .

Для элемента массива *connections* со значением “LVPN” та же самая аналогия: данная связь соединяет элементы уже по главной диагонали сектора, и индексы смежных по данной связи элементов равны, и , соответственно.

Элемент массива *connections* со значением “G” – это Г-связь, то есть связь между ведущим узлом сектора, где расположена связь, с его горизонтальным соседом в этом секторе, и индексы смежных по данной связи элементов равны , и , соответственно.

Элемент массива *connections* со значением “V” – это В-связь, то есть связь между ведущим узлом сектора, где расположена связь, с его вертикальным соседом в этом секторе, и индексы этих вершин равны: , и , соответственно.

Элемент массива *connections* со значением “P” – это значит, что связи на данной позиции в поле и в массиве *connections* нет.

В алгоритме мы проходимся по двумерному массиву *connections,* и проверяем каждый текущий элемент итерации, на какой позиции он стоит. Затем по этой позиции и по наличию связи между объектами *Node* в массиве *nodesForGen,* индексы которых получаем из индексов связи, как приведено выше. Затем мы инициализируем текущий элемент solution соответствующим значением:

* Если у элементов оба индекса четные, то этот элемент массива *connections* не проводится по этой позиции и ему присваивается значение “P”, так как он совпадет с элементом массива *nodesForGen* на игровом поле. Наглядно, можно сказать, что массив связей накладывается на массив узлов. Есть элементы массива связей, которые в точности перекрываются узлами в поле, это и есть элементы с четными индексами.
* Если у элемента оба индекса нечетные, то, значит, что это может быть только одна из двух диагональных связей. Мы проверяем сначала на то, есть ли вообще эта связь между узлами, проверяем на наличие ГД-связи или ПД-связи. Если одно из двух истинно то, данная диагональная связь является соответствующей и данному элементу массива присваивается значение “LVPN”, либо “PVLN”, соответственно.
* Если первый индекс четный, а второй нечетный, то предполагаемая связь является Г-связью. Мы проверяем есть ли эта связь между узлами на соответствующих данной позиции связи индексах. Если истинно, то элементу массива связей присваивается значение “G”.
* Если второй индексы четный, а первый нечетный, то предполагаемся связь является В-связью. Мы проверяем, есть ли эта связь между узлами на соответствующих данной позиции связи индексах. Если истинно, то элементу массива связей присваивается значение “V”.
* Если в пунктах 2-4 условие ложно, то элементу массива *connections* на данной позиции присваивается значение “P”.

Таким образом, мы получили массив *connections* (он же является массивом связей), который содержит в себе информацию о решении к данному сгенерированному заданию.

Эта информация об этом массиве дублируется в строковую переменную *solution* точно по тому же алгоритму, как было приведено в четвертом шаге алгоритма генерации задания.

**string** solution = "";

*// Заполнение поля Weight у каждого элемента массива nodesForGen*

*// Запись информации о поле в переменную info для генерации задания.*

**for** (**int** i = 0; i < size; i++)

{

**for** (**int** j = 0; j < size; j++)

    {

        solution += j != size - 1 ? connections[i,j]+ " " : connections[i,j];

    }

    solution += i != size- 1 ? "," : "";

}

То есть для приведенного выше(рис. 8) задания для поля 4 на 4, переменная *solution* выглядит следующим образом:

*solution = “P P P G P G P,V LVPN P PVLN P PVLN V,P P P G P P P,…,P G P P P G P”*

Затем эта переменная с решением задания записывается в файл с расширением *.khan* в ту же строку файла через разделитель ‘;’ , в которой находится соответствующее задание.

При нажатии пользователем на кнопку «Показать решение», считывается необходимая информация из файла о решении для текущего задания по идентификатору данного задания, то есть по его номеру строки в файле. Достраивается графическая часть и пользователь получает корректное решение данного игрового поля.

### Алгоритм записи/чтения данных о сгенерированных заданиях и решениях в файл

Как же выглядят вышесказанные шаги записи в файл информаций о заданиях и решениях? Как было сказано, все данные записываются в текстовые файлы с расширениями .*khan.* Всего с таким расширением три файла, так как всего три уровня сложности в игре. В каждом файле содержится информация о сгенерированных заданиях на соответствующем уровне и о решениях к этим заданиям. Данные о заданиях и решениях разделяются символом ‘;’. То есть в этом файле всего два столбца: информация о заданиях и информация о решениях к этим заданиям. Задание и его решение располагаются в одной строке и разделяются символом ‘;’. При работе с файлами в вышеприведенных алгоритмах вызываются методы, класса *KhanParser* в созданной мной библиотеке *ParserLib*.

Метод добавления информации о задании/решении в файл:

*// Уже содержит в себе информацию о строках файла.*

**string**[][] dataDouble;

/// path – директория файла для добавления информации, data –задание/решение, column – /// номер столбца файла, в который записывается информация: 0 – задание, 1 - решение

**public** **static** **void** AddField(**string** path,**string** data,**int** column)

{

    Array.Resize(**ref** dataDouble, dataDouble.Length + 1);

    dataDouble[dataDouble.Length - 1] = new **string**[2];

    dataDouble[dataDouble.Length - 1][column] = data;

*// Массив строк с информациями о строках файла.*

**string**[] dataSingle = new **string**[dataDouble.Length];

**for** (**int** i = 0; i < dataDouble.Length; i++)

    {

**for** (**int** j = 0; j < dataDouble[i].Length; j++)

        {

            dataSingle[i] += j!=dataDouble[i].Length-1?dataDouble[i][j] + ";":dataDouble[i][j];

        }

    }

        File.WriteAllLines(path,dataSingle,Encoding.UTF8);

        ReadFile(path);

}

По идентификатору текущего задания можно получить информацию о номере задания в файле, соответствующем текущему уровню. Методы чтения необходимой информации из файла – задания/решения по нужному идентификатору:

*/// Чтение информации из файла.*

**public** **static** **void** ReadFile(**string** path)

{

*// Массив, в который записываются все строки файла, то есть все решения и задания.*

    allLines = File.ReadAllLines(path);

*// Этот массив содержит ту же информацию, что и allLines, но в другом формате.*

    dataDouble = new **string**[allLines.Length][];

*// Инициализация элементов массива массивов dataDouble информацией из allLines*

    ConvertFromSingleToDouble(allLines, **ref** dataDouble);

}

*/// Возвращает ровно ту информацию, которой соответствует параметры.*

*/// field - номер строки в файле(номер задания/решения)*

*/// column - что именно прочитать - задание или решение к этому заданию.*

**public** **static** **string** GetField(**int** field, **int** column)

{

**return** dataDouble[field][column];

}

Алгоритмы генерации решения/задания, как было сказано, сильно завязаны друг на друга, так как они выполняются вместе последовательно друг за другом: сначала генерируется задание и по нему генерируется решение. В конце каждого из этих двух алгоритмов генерации вызывается метод записи в файл следующим образом:

*/// Запись в файл сгенерированного задания. 0 – запись задания.*

Parser.AddField(path, task, 0);

*/// Запись в файл сгенерированного решения. 1 – запись решения к заданию.*

Parser.AddField(path, solution, 1);

### Автосохранение текущего состояния игры на текущем уровне

В программе реализован алгоритм автосохранения текущего состояния игры на каждом уровне. Данный алгоритм реализован при помощи методов XML – сериализации. Благодаря сериализации сохраняется состояние игрового поля: вершины со значением степени, все проведенные связи, состояние каждого узла-объекта типа Node и необходимых полей узлов (степень, список соседей в виде массива, позиции и так далее), идентификатор поля, логического значение, о том, решено ли поле или нет, размер поля.

Для информации в виде многомерного массива, которую нужно сохранить, используются дополнительно е эквивалентное представление в виде зубчатого массива, то есть массива массивов.

Стоит отметить, что для сериализации используется «транзитный» класс InfoForSave, объекты которого мы создаем через конструкторы с необходимыми параметрами и в эти параметры мы передаем необходимую для сохранения информацию. Самая важная информация для сохранения – это, конечно, информация об узлах и проведенных связях, так как вся графическая часть достраивается именно по этой информации. Какая же информация об узлах сериализуется? Это, в первую очередь список смежных соседей для каждого элемента Nodes, представленный в виде двумерного массива *string[][]*, который заполняется в следующем методе:

**string**[][] connectedNodesCopy;

**void** FillConnections()

{

    connectedNodesCopy = new **string**[nodes.GetLength(0)][];

**for** (**int** i = 0; i < nodes.GetLength(0); i++)

    {

        connectedNodesCopy[i] = new **string**[nodes.GetLength(1)];

**for** (**int** j = 0; j < nodes.GetLength(1); j++)

        {

            connectedNodesCopy[i][j] = nodes[i, j].ShowInfoForSave();

        }

    }

}

Элементы данного массива массивов записываются через метод экземпляра класса Node:

**string** info;

**public** **string** ShowInfoForSave()

{

    info = **null**;

**int** counter = 0;

**foreach** (Node elem **in** connectedNodesList)

    {

        counter++;

**if** (counter != connectedNodesList.Count)

            info += elem.X + " " + elem.Y + ",";

**else**

            info += elem.X + " " + elem.Y;

    }

**return** info;

}

Также не менее важная информация - это проведенные связи - которая нужна будет для графического построения сохраненного процесса. Эта информация записывается в массив массивов логического типа *connectionsIsSet*(в программе это массив *pb -* сокращенно PictureBox, это встроенный элемент управления в *WinForms*, графически все связи в поле – это экземпляры PictureBox), этот массив по структуре является квадратным, то есть длина внешнего массива равна длине каждого вложенного в него массива. И размерность этого массива, как было сказано ранее, равна 2\*N-1, где N – это размер игрового поля. Массив заполняется по циклу по правилу: если есть связь(то есть если элемент не *null*) в массиве связей *connections*(основной массив в программе, содержащий информацию о проведенных связях) на текущей позиции, то элемент массива *connectionsIsSet* на текущей итерации цикла получает значение *true*, иначе *false.* Другими словами, если есть связь на текущей позиции в поле, то элемент в массиве *connectionsIsSet* получает значение *true,* иначе *false:*

*// Массив для того, чтобы потом при сериализации можно было спокойно расставить объекты picture box.*

**bool**[][] connectionsIsSetCopy;

*/// <summary>*

*/// Заполнение массива connectionsIsSetCopy. True - если на текущей позиции имеется связь.*

*/// </summary>*

**void** FillConnectionsIsSetCopy()

{

    connectionsIsSetCopy = new **bool**[connections.GetLength(0)][];

**for** (**int** i = 0; i < connections.GetLength(0); i++)

    {

        connectionsIsSetCopy[i] = new **bool**[connections.GetLength(1)];

**for** (**int** j = 0; j < connections.GetLength(1); j++)

        {

**if** (connections[i, j] != **null**)

            {

                connectionsIsSetCopy[i][j] = **true**;

            }

        }

    }

}

Алгоритм сериализации и десериализации объектов представлен в виде следующего куска кода:

*/// Сериализации объекта типа obj.*

**private** **void** AutoSave(InfoForSave obj, **string** path)

{

    XmlSerializer serGameField = new XmlSerializer(typeof(InfoForSave));

**using** (FileStream fs = new FileStream(path, FileMode.Create))

    {

        serGameField.Serialize(fs, obj);

    }

}

Алгоритм автосохранения срабатывает, когда пользователь покидают игру, точнее в обработчике закрытия окна игрового поля .

/// Десериализация объекта типа obj.

**private** InfoForSave LoadSavedGame(**string** path)

{

    InfoForSave obj = **null**;

    XmlSerializer deserGameField = new XmlSerializer(typeof(InfoForSave));

**using** (FileStream fs = new FileStream(path, FileMode.Open))

    {

        obj = (InfoForSave)deserGameField.Deserialize(fs);

    }

**return** obj;

}

Процесс загрузки сохраненного процесса игры срабатывает в случае, если существует сам *xml* файл по определенной директории. В результате десериализации мы получим объект типа *InfoForSave*, поля которого содержат необходимую для загрузки сохраненного процесса информацию. Чтение этой информации данного объекта тривиальный кусок кода, который приводить необходимости нет.

### Алгоритм генерации подсказки пользователю

Игра создана так, что пользователь получает подсказку по умолчанию в начале загрузки каждого поля на легком уровне. Также у него имеется возможность самому получить подсказку на среднем уровне, нажав на кнопку «Подсказка». На сложном уровне получить пользователем подсказку возможности нет. Важно, что подсказка генерируется только тогда, когда в поле никаких проведенных связей нет. То есть на легком уровне пользователь не может второй раз получить подсказку. Как же генерируются подсказки на легком и среднем уровне? Алгоритм предельно прост:

* Сначала считывается решение по идентификатору текущего поля из файла с данными о заданиях и решениях к ним. Эта информация записывается в строку, информация с которой потом дублируется в двумерный массив *solution*.
* Генерируется случайное число в диапазоне [4..7]. Это число обозначает количество связей, которые проведутся для подсказки.
* В цикле заполняются значениями элементы двумерного массива *connectionsIsSet* логического типа. Цикл срабатывает от i = 0 до сгенерированного числа связей, внутри которого есть другой цикл, который срабатывает до тех пор, пока не инициализируется валидный элемент данного массива *connectionsIsSet.* Валидный элемент – это элемент, на позициях которого может быть проведена связь в поле по информации из массива *solution*. То есть в этом цикле проверяется, не равен ли “P” элемент из массива *solution.* Если нет, то данным индексам элементу из массива *connectionsIsSet* присваивается значение *true.*
* Затем в цикле проходимся по массиву *connectionsIsSet* и проверяем на *true* каждый элемент текущей итерации, если он *true*, то проверяем какое значение по этим индексам хранится в элементе массива *solution* и проводим соответствующую связь в поле и соединяем соответствующие элементы в массиве *node,,* то есть добавляем эти узлы в список своих смежных соседей. В этом же цикле осуществляется графическая отрисовка всех проведенных связей.

Другими словами, мы уже считываем готовое решение к данному заданию по его идентификатору из файла. По этому решению создаем двумерный массив связей и инициализируем элемента этого массива ровно по информации из решения. Выбираем случайным образом из этого массива только некоторое количество связей, которые являются частью правильного решения данного задания и проводим выбранные связи. Таким образом генерируется подсказка для каждого поля на легком и среднем уровнях.

рисунок

### Алгоритм перехода к новому полю

Этот алгоритм включает в себя алгоритм генерации задания. То есть по нажатии пользователем на кнопку «Новое поле», отрабатывает алгоритм генерации, затем уже в данном алгоритме мы получаем информацию из файла по последней строке, так как она соответствует новому сгенерировавшемуся заданию, она же передается в двумерный массив *nodes* и после этого отрабатывает свою работу просто графическая часть программы:

**void** NewField()

{

*// leveLength –это количество заданий после генерации нового задания.*

**int** numOfField = levelLength;

    fieldSize =Parser.GetFieldLength(numOfField, 0);

**string**[] dataRows;

    dataRows = Parser.GetField(numOfField, 0).Split(',');

**string**[] row = new **string**[fieldSize];

    nodes = new Node[fieldSize, fieldSize];

**for** (**int** i = 0; i < nodes.GetLength(0); i++)

    {

*// Массив из элементов заданной строки.*

        row = dataRows[i].Split(' ');

**for** (**int** j = 0; j < nodes.GetLength(1); j++)

        {

            nodes[i, j] = new Node(**int**.Parse(row), i, j);

        }

    }

}

рисунок

### Алгоритм перехода к решенным/просмотренным заданиям

Каждый раз, когда осуществляется переход к новому заданию, пользователю предоставляется возможность вернуться к уже решенным или просто просмотренным заданиям на данном уровне. Это просто список номеров заданий из файла. Каждый раз когда пользователь выбирает какой-то номер M, который соответствует заданию с номером M из файла, отрабатывает алгоритм перехода к заданию:

**void** ToFieldAt(**int** number)

{

**int** numOfField = number;

    fieldSize =Parser.GetFieldLength(numOfField, 0);

**string**[] dataRows;

    dataRows =Parser.GetField(numOfField, 0).Split(',');

**string**[] row = new **string**[fieldSize];

    nodes = new Node[fieldSize, fieldSize];

**for** (**int** i = 0; i < nodes.GetLength(0); i++)

    {

*// Массив из элементов заданной строки.*

        row = dataRows[i].Split(' ');

**for** (**int** j = 0; j < nodes.GetLength(1); j++)

        {

            nodes[i, j] = new Node(**int**.Parse(row), i, j);

        }

    }

}

рисунок

### Алгоритм взаимодействия пользователя с игровыми вершинами (кнопками)

Логика проведения связей основана на том, что каждый раз пользователь связывает последовательно только две вершины, поэтому в коде программы создаются две переменные, в которые записываются выбранные вершины и информация об этих вершинах, как позиции в поле. В программе каждая вершина внешне это элемент типа Button. Все действия вершины синхронизируются с соответствующими объектами–узлами типа Node, которые в свою очередь содержат уже более полную информацию о вершинах, такую как: вес(степень) вершины, количество проведенных связей, список соседей, с которыми соединен данный узел. Осуществляется проверка на то, соединяет ли пользователь именно соседние вершины, и на то, были ли уже соединены данные вершины:

Если пользователь пытается соединить две несоседние вершины, то фокус с кнопок сбрасывается и все значения, которые соответствуют данным кнопкам умалчиваются, включая ссылки на нажатые кнопки и связь не проводится.

Если пользователь пытается провести одну из диагональных связей в то время, как другая диагональная связь в том же секторе уже проведена, то первая не проводится, так как не может быть пересечения диагональных связей и фокус с кнопок сбрасывается и все значения, которые соответствуют данным кнопкам умалчиваются, включая ссылки на нажатые кнопки и связь не проводится.

Если связь проводится между соседними вершинами, которые еще не были соединены, то связь графически проводится, и узлы, которые соответствуют данным вершинам добавляются в список смежных соседей друг друга. Затем фокус с кнопок сбрасывается и все значения, которые соответствуют данным кнопкам умалчиваются, включая ссылки на нажатые кнопки.

Если пользователь соединяет две соседние кнопки, которые уже были соединены, то связь, наоборот, уже убирается, и так же эти узлы, соответствующие этим вершинам(кнопкам) удаляются друг у друга в своих списках соединенных соседей.

*// Взаимное удаление узлов в своих списках смежных соседей.*

RemoveFromNodesList(firstNode, secondNode);

Для комфортного и удобного процесса игры реализован алгоритм выставления нужных цветов вершин при совершении определенных действий пользователем:

Если две временные переменные, куда записываются данные о нажатых вершинах(игровых кнопках), пусты, то при нажатии любой кнопки игрового поля она подсвечивается голубым цветом. Затем при наведении на другие игровые кнопки поля валидные кнопки для проведения связи так же подсвечиваются голубым цветом, а невалидные – оранжевым. Если нажать на уже нажатую кнопку, то ее подсветка сбрасывается. И информация об этой кнопке стирается.

Если количество проведенных связей от вершины в диапазоне [1..d-1], где d – степень вершины, то кнопка принимает желтый окрас.

Если количество связей равно d, то кнопка принимает зеленый окрас – это говорит о том, что максимальное количество связей для данной кнопки проведено.

Если количество связей больше d, то кнопка принимает красный окрас – это говорит о том, что от кнопки проведено уже больше связей, чем нужно.

Если d = 0, то кнопка имеет первоначальный нейтральный окрас.

рисунок

### Алгоритм проверки решенного задания

Алгоритм срабатывают по нажатии на кнопку «Проверить». Алгоритм проверки достаточно прост, что следует из самого главного правила этой игры: связей от каждого элемента должно быть проведено ровно столько, сколько указано в степени этой вершины. То есть алгоритм заключается в том, что осуществляется циклический обход массива *nodes* и на каждой итерации цикла проверяется, проведено ли от текущего элемента необходимое количество связей. Как только встречается элемент от которого проведено не то количество связей, что указано в степени этого элемента, то пользователю сообщается, что задание решено неправильно и переменная логического типа *finished* принимает значение *false*, иначе пользователю сообщается, что поле решено правильно и *finished = true.*

/// Алгоритм проверки правильно ли решено поле.

**foreach** (**var** memb **in** nodes)

{

**if** (memb.Weight != memb.ConnectedNodesCount())

    {

        finished = **false**;

    }

}

finished = **true**;

рисунок

### Алгоритм сохранения результата и дальнейшего его выведения в статистике

Алгоритм предусматривает сохранение результата прохождения текущего задания. То есть как только пользователь правильно решит задание, ему дается возможность сохранить свой результат, который содержит имя, время, *id* поля. Затем этот результат сохраняется в файле расширения *.khan* , соответствующем текущему уровню, посредством метода записи в файл класса *File* посредством созданной библиотеки *ParserLib*. Пользователи могут посмотреть результаты в главном меню по нажатии на кнопку «Статистика». В окне выведется элемент *DataGridView*, в котором в зависимости от выбранного уровня, будет показываться вся статистика по всем решенным полям на данном уровне. Показ осуществляется путем циклического обхода двумерного массива, который содержит в себе информацию о строках файла статистики, и присваивания значений элементов этого массива ячейкам *DataGridView*. Пользователь может взаимодействовать с элементом *DataGridView,* удаляя некоторые результаты или полностью очищая все результаты, что, собственно, отразится и на файлах, которые хранят статистику, данные из этих файлов так же удалятся.

Рисунок

## Описание и обоснование выбора метода организации входных и выходных данных

### Описание метода организации входных и выходных данных

В данной программе входными данными является строковая информация из файлов автосохранения, статистики и сгенерированных заданий/решений. Основным разделителем информации в файлах статистики является символ ‘;’, например, некоторая строка из файла *EasyLevelStatistics.khan* имеет вид *“ID\_121;00:25:21;Emirkhan”.* В качестве разделителей информации в файле сгенерированных заданий и решений выступают символы одинарный пробел, запятая и точка с запятой. Пробел разделяет информацию о степенях вершин в строке поля, если это задание, или о проведенных связях в поле, если это решение, запятая разделяет эти строки, точка с запятой разделяет информацию о заданиях и информацию о решениях. Выше в алгоритмах генерации задания и решения были приведены примеры того, как выглядят эти строки.

В данной программе выходными данными является реагирование тех или иных игровых компонент на действия пользователя, так же при закрытии окна с игровым полем происходит автосохранение текущего состояния игры в XML-файл посредством технологии сериализации. Также при удалении некоторой информации из *DataGridView* эти же данные удаляются из файлов статистики. При генерации нового поля, происходит сначала запись информации об этом задании и его решении в файлы с заданиями и решениями.

### Обоснование выбора методов организации входных и выходных данных

Технологии сериализации и десериализации выбраны, потому что благодаря этому можно достаточно быстро сохранить текущее состояние любого объекта и в любой момент можно восстановить сохраненное состояние объекта. Вывод информации в *DataGridView* выбран, потому что элементы *DataGridView* могут ясно отобразить всю необходимую информацию из текстовых файлов. С помощью введенных разделителей можно спокойно разбивать информацию из файла и брать ровно ту часть информации, которая нужна.

## Описание и обоснование выбора состава технических и программных средств

Программа была полностью разработано на компьютере со следующими характеристиками технических средств:

- Intel(R) Pentium(R) CPU B960 Pentium;

- Частота процессора 2.20 ГГц.

- 4 гб оперативной памяти типа DDR3;

- свободное место на жестком диске: 100 гб, но для корректного функционирования программы достаточно иметь не менее 256 мб свободной памяти на диске;

- так же для возможности играть надо иметь клавиатуру и мышь/точпэд.

Характеристики программных средств, которые использовались при разработке:

- ОС windows 10;

- Microsoft .NET Framework 4.7.2.

Поскольку на данном спектре характеристик компьютер справлялся с разработкой и процессом игры, то всем потенциальным пользователям рекомендуется иметь технические и программные средства, схожие с приведенными выше, или с лучшими характеристиками, чем приведенные выше, для комфортного функционирования программы.

# ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

## Прогнозируемая потребность

Игра «Перемычки» будет пользоваться спросом среди любых возрастов, так как головоломки подобного рода одинаково интересны и востребованы абсолютно всеми. Игра предоставляет возможность полезно провести свободное время. Игра очень помогает улучшить уровень когнитивных способностей, внимательность, концентрацию и способность думать на несколько шагов вперед.

Как было приведено выше, на основании этой игры в учебных учреждениях можно достаточно доступно и понятно объяснить учащимся некоторые базовые понятия теории графов.

## Экономические преимущества разработанного продукта перед отечественными и зарубежными образцами или аналогами

Результат поиска аналогов данной программы в интернете и Google Play Market говорит, что:

* Аналогов данной игры с такими же заложенными правилами нет в Google Play Market. Поиск в этом магазине приложений выдал примитивные варианты этой игры, где есть только ограниченное количество заданий. И в правилах к этим играм отсутствует возможность проведения диагональных связей, то есть пользователь может проводить только горизонтальные, либо вертикальные связи.
* В интернете были найдены сайты с данной игрой с абсолютно теми же правилами, что и в играх из Google Play Market.

Стоит отметить явные преимущества данной игры:

1. Бесплатный продукт и не требуется никаких затрат со стороны пользователя в процессе игры;
2. Отсутствие рекламы в процессе игры;
3. Игра имеет еще очень много возможностей для развития своего функционала, но уже на данный момент является очень интересной и увлекательной;
4. Игра имеет встроенный генератор заданий, а это значит, что пользователю предоставляется неограниченное количество интересных задач;

# 

# ПРИЛОЖЕНИЯ К ДОКУМЕНТУ

# ПРИЛОЖЕНИЕ 1

## ТЕРМИНОЛОГИЯ

**Граф** – это пара из некоторого множества элементов *V* и иррефлексивного, симметричного бинарного отношения *S* на этом множестве, называемое отношением смежности.

**Бинарное отношение** на множестве *V*– это подмножество декартового произведения *V* x *V*.

**Иррефлексивное бинарное отношение** – это такое отношение на множестве *V*, у которого нет пар вида (a,a), где a – элемент V.

**Симметричное бинарное отношение** – это такое отношение на множестве *V*, в котором из наличия пары вида (a,b) следует наличие пары (b,a), где a,b – элементы *V*.

**Смежные вершины –** это вершины, между которыми есть связь в поле, или, на языке теории графов, это элементы a,b из множества *V*, образующие пары вида (a,b) и (b,a), которые содержатся в бинарном отношении *S*.

**Узел** – объект типа Node.

**Вершина –** синоним термина **узел.**

**Игровая кнопка –** то, что графически представляет из себя вершина(узел) в окне *WinForms,* то есть объект *Button*. Порой «игровая» опускается, но никаких вопросов не должно возникнуть так как другие кнопки программы называются «кнопки управления».

В данной работе эти термины равнозаменяемы, так как действия над одним объектом влекут те же самые действия над другим объектом. То есть в этой программе и в настоящем документе «**вершина**»⬄ «**узел**» ⬄ «**кнопка**» ⬄ «**кружок**» - синонимы в силу контекста.

**Игровое поле** – это совокупность только игровых кнопок и связей между ними. Остальные элементы управления не входят в пространство игрового поля.

**Степень** вершины **–** количество связей, которые должны быть проведены от данной вершины к соседним.

**Степень** вершины ⬄ **вес** вершины

**Связь** – это линия между кпопками (вершинами).

С каждой связью ассоциируется параллельно и ребро графа, который соответствует игровому полю, ведь, в начале было сказано, что решение игры – это построение некоторого иррефлексивного, симметричного бинарного отношения на данном упорядоченном множестве. А пара (множество, бинарное отношение на этом множестве) – есть неориентированный граф.

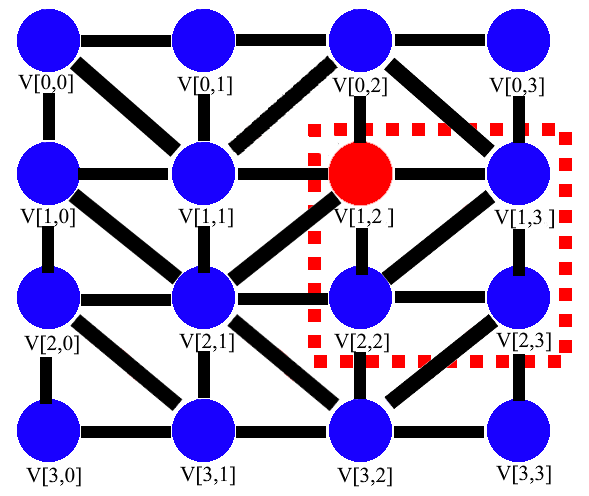
«**Связь»**⬄ «**ребро»**⬄ «**соединение»** ⬄ «**перемычка»** ⬄ «**мост»** – это равнозаменяемые термины в силу контекста.

**Окрестность** вершины – это множество смежных элементов.

Рассмотрим некоторое упорядоченное множество и бинарное отношение *S*, которое является корректным с точки зрения решения полей данной игры.

На данном упорядоченном наборе вершина , где 0≤i<N, 0≤j<N в общем случае может состоять в построенном бинарном отношении S только с вершинами , где 0≤m<N, 0≤n<N, m ϵ {i-1, i, i+1}, n ϵ {j-1, j, j+1}, m!=n. Эти вершины называются **соседними**. Если есть отношение между соседними вершинами, то они называются смежными.

**Сектор вершины** (рис. П1) есть часть игрового поля из четырех, либо из двух вершин, либо из одной вершины (сама же вершина ) и в него помимо вершины входят вершины , либо (если второй индекс N-1, то есть вершина в последнем столбце) или (если первый индекс N-1, то есть вершина в последней строчке), либо других вершин нет (если оба индекса N-1):



*Рисунок П1.* Пример сектора вершины.

Рассмотрим вершину на приведенном рисунке.

Сектор вершины обведен по периметру красным пунктиром. Данный сектор является **сектором – квадратом**.

Секторы могут быть трех видов: **сектор-квадрат** (часто «квадрат» опускается), **сектор-столбец**, **сектор-строка**.

**Главная диагональная связь** – это связь в секторе-квадрате, которая соединяет ведущий элемент сектора (выделен красным на рисунке) с его соседом по диагонали, то есть в приведенном секторе для вершины диагональная связь могла бы быть между .

**Побочная диагональная связь** – это связь в секторе-квадрате, которая соединяет горизонтального (на рисунке -горизонтальный сосед для ) и вертикального (на рисунке -вертикальный сосед для ) соседей ведущего элемента. Она проведена на рисунке между и

**Диагональные связи** – это главная и побочная диагональные связи.

# ПРИЛОЖЕНИЕ 2

## ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ КЛАССОВ

Таблица 2.1

Описание и функциональное назначение классов

|  |  |
| --- | --- |
| Класс | Назначение |
| MainMenuForm | Главное меню в программе. Здесь пользователь может узнать правила игры, посмотреть статистику о решенных полях и прочитать информацию об игре. |
| GameField | Вторая форма в программе. Весь основной процесс игры реализован в этом классе. Именно в этом классе пользователю предоставляется игровое поле для решения. |
| PauzeForm | Третья форма в программе. Окно паузы. Вызывается из второй формы. В этой форме можно выйти в главное меню, можно прочитать правила игры. |
| InfoForSave | Класс предназначен для сохранения информации о текущем состоянии игры. |
| SaveRecordForm | Четвертая форма в программе. Предназначена для получения информации о текущем пользователе. |
| Node | Основной класс для работы и взаимодействия с узлами в программе. |
| Parser | Класс предназначен для выполнения разного рода работы с файлами. |
| BridgesException | Класс исключений, специализированных под эту игру. |

# ПРИЛОЖЕНИЕ 3

## ОПИСАНИЕ И ФУНКЦИОНАЛЬНОЕ НАЗНАЧЕНИЕ ПОЛЕЙ, МЕТОДОВ И СВОЙСТВ

Таблица 3.1

Описание полей и функциональных членов класса GameField.cs

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | |  | |  | | |  |
| Имя | | Модификатор доступа | | Тип | | | Назначение |
| pathCurrentLevel | | private | | string | | | Директория на файл с заданиями и решениями для текущего уровня. |
| pathEasyLevel | | private | | string | | | Константа, содержит в себе адрес файла с заданиями и решениями для легкого уровня. |
| pathMediumLevel | | private | | string | | | Константа, содержит в себе адрес файла с заданиями и решениями для среднего уровня. |
| pathHardLevel | | private | | string | | | Константа, содержит в себе адрес файла с заданиями и решениями для сложного уровня. |
| pathEasyLevelStatistics | | private | | string | | | Константа, содержит в себе адрес файла со статистикой пользователей для легкого уровня. |
| pathMediumLevelStatistics | | private | | string | | | Константа, содержит в себе адрес файла со статистикой пользователей для среднего уровня. |
| pathHardLevelStatistics | | private | | string | | | Константа, содержит в себе адрес файла со статистикой пользователей для сложного уровня. |
| rnd | | private | | Random | | | Для генерации случайных чисел. |
| level | | private | | int | | | Уровень игры. |
| nodes | | private | | Node[,]  Продолжение таблицы 3.1 | | | Двумерный массив из узлов Node. |
| buttons | | private | | Buttons[,] | | | Двумерный массив из кнопок игрового поля. |
| fieldSize | | private | | int | | | Размер поля текущего уровня. |
| numOfField | | private | | int | | | Порядковый номер поля в своем уровне. |
| previousColor | | private | | Color | | | Предыдущее значение цвета кнопки. |
| tempButton1 | | private | | Button | | | Информация о первой нажатой игровой кнопки. |
| pos1 | | private | | string[] | | | Массив из значений позиций первой нажатой кнопки. |
| row1 | | private | | int | | | Строка игрового поля, в которой расположена первая кнопка. |
| col1 | | private | | int | | | Столбец игрового поля, в котором расположена первая кнопка. |
| tempButton2 | | private | | Button | | | Информация о первой нажатой игровой кнопки. |
| pos2 | | private | | string[] | | | Массив из значений позиций второй нажатой кнопки. |
| row2 | | private | | string | | | Строка игрового поля, в которой расположена вторая кнопка. |
| col2 | | private | | string | | | Столбец игрового поля, в котором расположена вторая кнопка. |
| pb | | private | | PictureBox[,] | | | Двумерный массив, содержащий элементы PictureBox – они же связи в игре. |
| hours | | private | | int | | | Информация о времени решения поля: часы |
| mins | | private | | int | | | Информация о времени решения поля: минуты  Продолжение таблицы 3.1 |
| secs | | private | | int | | | Информация о времени решения поля: секунды |
| pathSavedGameFieldEasyLevel | | private | | string | | | Константа, содержит в себе адрес файла с сохраненным процессом игры для легкого уровня. |
| pathSavedGameFieldMediumLevel | | private | | string | | | Константа, содержит в себе адрес файла с сохраненным процессом игры для среднего уровня. |
| pathSavedGameFieldHardLevel | | private | | string | | | Константа, содержит в себе адрес файла с сохраненным процессом игры для сложного уровня. |
| id | | private | | int | | | Уникальный идентификатор текущего игрового поля. |
| isFinished | | private | | bool | | | Информация о том, решено ли текущее поле. |
| pbIsSet | | private | | bool[,] | | | Двумерный массив с данными о проведенных/непроведенных связях на соответствующих элементам массива позициях. Для десериализации. |
| pbIsSetCopy | | private | | bool[][] | | | Представление массива pbIsSet в виде массива массивов с теми же данными. Для сериализации. |
| connectedNodes | | private | | string[,] | | | Двумерный массив, который, содержит список позиций соседей в виде строки для соответствующего узла Node. Для десериализации.  Продолжение таблицы 3.1 |
| connectedNodesCopy | | private | | string[][] | | | Представление массива connectedNodes в виде зубчатого массива. Для сериализации. |
| nodesCopy | | private | | Nodes[][] | | | Представление массива nodes в виде зубчатого массива. Для сериализации. |
| **Методы** | |  | |  | | |  |
| Имя | Модификатор доступа | | Тип | | Аргументы | Назначение | |
| GameField | public | | конструктор | | int level | Конструктор класса, в котором инициализируется значение текущего уровня. | |
| GameFieldLoad | private | | void | | object sender, EventArgs e | Обработчик загрузки формы. В нем происходит инициализация всех необходимых для построения полей и собственно графическое построение поля. | |
| RunSavedGame | private | | void | | InfoForSave ifr | Запускает последний сохраненный игровой процесс, если тот есть и достраивает его графическую и программную часть. | |
| FieldLoad | private | | void | | - | Запускается самое первое игровое поле на уровне, если нет сохраненного процесса. | |
| SetFirstHelp | private | | void | | - | Расставляет связи на пустом поле в качестве подсказки. | |
| FromStringToDoubleArr | private | | string[,] | | string info | Из строки данных получает эти же данные в виде двумерного массива и возвращает это двумерное представление данных. | |
| GenerateNewField | private | | void | | string path | Генерирует задание с гарантированным решением и само решение и записывает эту информацию в соответствующий файл.  Продолжение таблицы 3.1 | |
| ToNewFieldButtonClick | private | | void | | object sender, EventArgs e | Обработчик нажатия на кнопку <Новое поле>. В нем осуществляется переход к новому полю. | |
| NumOfField | private | | int | | - | Возвращает порядковый номер текущего поля в файле с заданиями по его идентификатору. | |
| ToNewField | private | | void | | - | Осуществляет переход к новому полю, которое еще не просматривалось пользователем. Строится графическая часть поля и инициализируются данные о новом поле. | |
| ClearAllGameField | private | | void | | - | Очищается все игровое пространство. | |
| DrawSavedLines | private | | void | | int i, int j, string type | Графическое проведение связей сохраненного процесса игры. | |
| SetPictureBoxes | private | | void | | - | Расстановка связей сохраненного процесса игры на игровом поле. | |
| FillConnectedNodesList | private | | void | | Node node, string[] neighbours | Заполнение списка смежных соседей элемента node. | |
| ButtonMouseLeave | private | | void | | object sender, EventArgs e | Обработчик покидания курсором кнопки.  Работает алгоритм графической подсветки. | |
| ButtonMouseEnter | private | | void | | object sender, EventArgs e | Обработчик вхождения курсора в игровую кнопку.  Работает алгоритм графической подсветки. | |
| SetButtonColor | private | | void | | Node node | Установка цвета кнопки в зависимости от количества проведенных к нему связей. | |
| ConnectionsExist | private | | bool | | - | Возвращает false -если связей вообще нет во всем поле.  true - если хотя бы одна связь проведена в поле. | |
| PictureBoxRemovingByClick | private | | void | | object sender, EventArgs e  Продолжение таблицы 3.1 | Удаление связи по нажатию на нее и удаление элементов из списков соседей друг друга, которые были соединены данной связью. | |
| GameButtonClick | private | | void | | object sender, EventArgs e | обработчик события нажатия на игровую кнопку поля. Здесь проводятся валидные связи между двумя соседними кнопками. | |
| AddToNodesList | private | | void | | Node elem1, Node elem2 | Метод, который добавляет в поле nodesConnected двух заданных в параметрах элементов друг друга. | |
| AddToNodesList | private | | void | | ref Node elem1, ref Node elem2 | Метод, который добавляет в поле nodesConnected двух заданных в параметрах элементов, переданных по ссылке, друг друга. | |
| RemoveFromNodesList | private | | void | | Node elem1, Node elem2 | Взаимное удаление из своих списков смежных соседей друг друга. | |
| RemoveFromNodesList | private | | void | | ref Node elem1, ref Node elem2 | Взаимное удаление узлов, переданных по ссылке, из своих списков смежных соседей друг друга. | |
| DrawLines | private | | void | | int x, int y, string type | Графическое проведение связи (линии) между двумя нажатыми кнопками. | |
| DeleteLines | private | | void | | ref PictureBox pb | Графическое удаление связи, которая передается в параметры метода по ссылке. | |
| SetDefaultValues | private | | void | | - | Сбрасывает все значения, которые относятся к кнопкам tempButton1 и tempButton2, до умалчиваемых.  Также отводит фокус на label. | |
| SaveResultButtonClick | private | | void | | object sender, EventArgs e | Сохранение результата решенной задачи пользователем. | |
| Checker | private | | bool | | - | Проверка правильно ли решено текущее поле.  Продолжение таблицы 3.1 | |
| ResetTimer | private | | void | | - | Метод, который обнуляет показания таймера. | |
| TimerTick | private | | void | | object sender, EventArgs e | Обработчик события, когда проходит интервал в 1 секунду. | |
| FillPBIsSetCopy | private | | void | | - | Метод, в котором дублируется информация о массиве pbIsSet в массив массивов pbIsSetCopy для дальнейшей сериализации. | |
| FillConnectedNodesCopy | private | | void | | - | Заполнение зубчатого массива connectedNodesCopy информацией о соседях каждого элемента в массиве nodes.  Для дальнейшей сериализации. | |
| FillNodesCopyArr | private | | void | | - | Заполняет зубчатый массив nodesCopy информацией из nodes. Для дальнейшей сериализации. | |
| AutoSave | private | | void | | InfoForSave obj, string typeOfFile | Механизм сериализации данных.  Сохраняет информацию, записанную в объект для сохранения InfoForSave, по директории указанной в typeOfFile. | |
| LoadSavedGame | private | | InfoForSave | | string typeOFFile | Механизм десериализации данных.  Загрузить информацию из файла с сохраненным процессом игры. | |
| GameFieldFormClosing | private | | void | | object sender, FormClosingEventArgs e | Обработчик закрытия окна с игровым полем. Внутри срабатывает механизм сериализация для сохранения текущего процесса игры. | |
| FillPBIsSet | private | | void | | - | Заполняет двумерный массив PbISSet, в котором каждый элемент содержит информацию о том, есть ли на данной позиции связь или нет.  true - есть . false - нет связи.  Продолжение таблицы 3.1 | |
| TryAgainButtonClick | private | | void | | object sender, EventArgs e | Обработчик события нажатия на кнопку "Решить заново".  Метод, который отвечает за то, чтобы пользователь заново начал решать текущее поле.  Очищается все игровое пространство.  Возможность выбрать один из двух вариантов: полностью очищается и таймер сбрасывается или не сбрасывается. | |
| ClearField | private | | void | | object sender, EventArgs e | Очистка всех связей в игровом поле и очистка списка соседей у каждого элемента Node. Вся графическая часть так же сбрасывается до умолчания. | |
| PauzeButtonClick | private | | void | | object sender, EventArgs e | Обработчик события нажатия на кнопку "Пауза". Открывается диалоговое окно паузы, где пользователь может продолжить или покинуть игру,  а также посмотреть правила. | |
| TurnToFieldButtonClick | private | | void | | object sender, EventArgs e | Обработчик события нажатия на кнопку "Перейти к полю". Метод предоставляет пользователю перейти к ранее решенным или просмотренным полям. | |
| ToFieldAt | private | | void | | number | Загрузит информацию об игровом поле по указанному порядковому номеру в файле с заданиями. | |
| SolutionButtonClick | private | | void | | object sender, EventArgs e | Обработчик события нажатия по кнопке "Показать решение".  Метод, в котором считывается информация о решении к текущему игровому полю.  Продолжение таблицы 3.1 | |
| HelpButtonClick | private | | void | | object sender, EventArgs e | Обработчик нажатия кнопки "Подсказка".  Считывает информацию из файла с заданиями о решении к текущему заданию, и предоставляет пользователю только этого решения,  что и является только подсказкой. | |

Таблица 3.2

Описание полей и функциональных членов класса Node.cs

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | |  | |  | |  | |
| Имя | | Модификатор доступа | | Тип | | Назначение | |
| weight | | private | | int | | Степень узла. | |
| connectedNodesList | | private | | List<Node> | | Список соединенных (смежных) соседей узла. | |
| connectedNodesInfo | | private | | string | | Информация о соседних узлах данного объекта в виде строки. | |
| isFull | | private | | bool | | Информация, о том проведено ли необходимое количество связей. | |
| x | | private | | int | | Номер строки, в которой располагается данный объект. | |
| y | | private | | int | | Номер столбца, в котором располагается данный объект. | |
| **Методы** | |  | |  | |  | |
| Имя | Модификатор доступа | | Тип | | Аргументы | | Назначение |
| Node | public | | конструктор | | - | | Конструктор умолчания данного класса.  Продолжение таблицы 3.2 |
| Node | public | | конструктор | | int x, int y | | Конструктор, который инициализирует номер строки и номер столбца создаваемого объекта. |
| Node | public | | конструктор | | int weight, int x, int y | | Конструктор, который инициализирует номер строки и номер столбца создаваемого объекта, а также его вес. |
| GetElemAt | public | | Node | | int i | | Метод, который возвращает элемент из списка соседей с индексом i. |
| RemoveAllList | public | | void | | - | | Очистка всего списка соседей. |
| ElemContains | public | | bool | | Node node | | Показывает, содержится ли элемент node из параметра в массиве connectedNodesList у элемента, для которого вызван этот метод. |
| ShowInfo | public | | string | | - | | Возвращает информацию о соединенных соседях в виде их позиций в виде строки. |
| ShowInfoForSave | public | | string | | - | | Возвращает информацию о соединенных соседях данного элемента в виде строки для дальнейшего сохранения и восстановления  этого списка соседей.  Продолжение таблицы 3.2 |
| ElemCount | public | | int | | Node node | | Считает, сколько раз элемент node встречается в списке. |
| AddToNodesList | public | | int | | Node elem | | Добавление элемента elem в список соседей данного объекта. |
| RemoveFromNodesListAt | public | | void | | int i | | Удаление соседа из списка соседей по его индексу в этом списке. |
| RemoveFromNodesList | public | | void | | Node elem | | Удаление соседа elem из списка соседей. |
| ConnectedNodesCount | public | | int | | - | | Количество связей или количество элементов, с которыми соединен данный объект.  Или, грубо говоря, количество соседей. |
| **Свойства** | |  | |  | |  | |
| Имя | Модификатор доступа | | Тип | | Доступ | | Назначение |
| Weight | public | | int | | get, set | | Свойство для доступа к private полю weight, которое содержит информацию о весе вершины. |
| X | public | | int | | get, set | | Свойство для доступа к private полю номера строки, в которой располагается данный объект. |
| Y | public | | int | | get, set | | Свойство для доступа к private полю номера столбца, в котором располагается данный объект.  Продолжение таблицы 3.2 |
| IsFull | public | | bool | | get, set | | Свойства с доступом к private полю isFull.  Возвращает информацию о том, заполнен ли список соседей или нет. |

Таблица 3.3

Описание полей и функциональных членов класса Parser.cs

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | | | |  | | |  | | |  | | | | |
| Имя | | | | Модификатор доступа | | | Тип | | | Назначение | | | | |
| allLines | | | | private | | | string[] | | | Все строки из файла с заданиями и решениями. | | | | |
| dataDouble | | | | private | | | string[][] | | | Ровно та же информация, что и в allLines, но в виде зубчатого массива. | | | | |
| **Методы** | | | |  | | |  | | |  | | | | |
| Имя | | Модификатор доступа | | | Тип | | | Аргументы | | | | | Назначение | |
| FileExists | | public | | | void | | | string path | | | | | Возвращает информацию о том, существует ли файл по данной директории path. | |
| ReadAll | | public | | | string[] | | | string path | | | | | Метод, который считывает массив строк из файла по директории path. | |
| WriteText | | public | | | void | | | string path, string text | | | | | Метод, который перезаписывает файл по директории из параметра, записывает в него информацию из параметра text. | |
| AppendText | | public | | | void | | | string path, string text | | | | | Метод, который добавляет в файл по директории path информацию, содержащуюся в text.  Продолжение таблицы 3.3 | |
| WriteLines | | public | | | void | | | string path, string[] text | | | | | Метод, который записывает массив строк text в файл по директории path. Перезаписывает файл, если он уже создан. | |
| DeleteFile | | public | | |  | | | string path | | | | | Удаляет файл по директории path. | |
| LevelColumnLength | | public | | | int | | | int column | | | | | Возвращает длину столбца с заданиями из файла, который соответствует данному уровню. | |
| ReadFile | | public | | |  | | | string path | | | | | Метод, который считывает информацию из файла с заданиями и решениями.  В методе инициализируется ассоциированный с файлом массив allLines. | |
| GetField | | public | | | string | | | int field, int column | | | | | Метод, который возвращает необходимое поле (ячейку из базы данных). | |
| GetFieldLength | | public | | | int | | | int field, int column | | | | | Возвращает размер поля, который будет построен по информации из файла. | |
| ConvertFromSingleToDouble | | public | | | void | | | string[] singleData, ref string[][] doubleData | | | | | Записывает всю информацию из singleData[] в dataDouble[][]. | |
| AddField | | public | | | void | | | string path, string currentFieldInfo, int column | | | | | Запись сгенерированных данных в файл с заданиями и решениями в зависимости от column. 0 - задания, 1- решения.  Продолжение таблицы 3.3 | |
| **Свойства** | | |  | | |  | | | | |  | | |
| Имя | Модификатор доступа | | | | Тип | | | | Доступ | | | Назначение | |
| Length | public | | | | int | | | | get | | | Возвращает длину массива из строк файла. | | |

Таблица 3.4

Описание полей и функциональных членов класса InfoForSave.cs

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | | | |  | | | |  | | | | |  |
| Имя | | | | Модификатор доступа | | | | Тип | | | | | Назначение |
| finished | | | | private | | | | bool | | | | | Информация, о том, решено ли поле или нет. |
| nodes | | | | private | | | | Node[][] | | | | | Зубчатый массив из узлов типа Node. |
| id | | | | private | | | | int | | | | | Идентификатор сохраненного поля. |
| level | | | | private | | | | int | | | | | Уровень сохраненного поля. |
| hours | | | | private | | | | int | | | | | Показания таймера часы. |
| mins | | | | private | | | | int | | | | | Показания таймера: минуты. |
| secs | | | | private | | | | int | | | | | Показания таймера: секунды. |
| pbIsSet | | | | private | | | | bool[][] | | | | | Зубчатый массив с информацией о наличии связей на позициях поля. |
| fieldSize | | | | private | | | | int | | | | | Размер сохраненного поля. |
| helpButtonVisible | | | | private | | | | bool | | | | | Содержит информацию о том, скрыта ли кнопка подсказки. |
| connections | | | | private | | | | string[][]  Продолжение таблицы 3.4 | | | | | Зубчатый массив. Содержит информацию о соединенных узлах для каждого элемента из nodes.  т.е. элемент connections[i][j] содержит позиции смежных соседей для элемента nodes[i][j]. |
| **Методы** | | | |  | | | |  | | | | |  |
| Имя | | | Модификатор доступа | | | Тип | | | | Аргументы | | Назначение | |
| InfoForSave | | | public | | | конструктор | | | | - | | Конструктор класса по умолчанию. Нужен для десериализации данных. | |
| InfoForSave | | | public | | | конструктор | | | | int id, int level, int hours, int mins, int secs,bool finished, Node[ ][ ] nodes, bool[ ][ ] pbIsSet, string[ ][ ] connections,int fieldSize, bool helpButtonVisible | | Конструктор класса, в котором выполняется инициализация необходимых для сохранения полей. | |
| FromNodeOfNodeToDoubleNode | | | public | | | Node[,] | | | | - | | Метод который из массива массивов nodes возвращает двумерный массив doubNodes. | |
| FromPbIsSetOfPbIsSetToDoublePbIsSet | | | public | | | bool[,] | | | | - | | Метод, который всю информацию из зубчатого массива pbIsSet дублирует в двумерный массив doubIsSet.  Продолжение таблицы 3.4 | |
| FromStrOfStrToDoubleStr | | | public | | | string[,] | | | | - | | Метод, который из зубчатого массива connections дублирует всю информацию в двумерный массив doubConnections. | |
| **Свойства** | |  | | | | |  | | | |  | | |
| Имя | Модификатор доступа | | | | Тип | | | | Доступ | | | Назначение | |
| Id | public | | | | int | | | | get, set | | | Свойство с доступом к полю id. Возвращает идентификатор сохраненного поля. | |
| Level | public | | | | int | | | | get,set | | | Свойство с доступом к полю level. Возвращает уровень сохраненного поля. | |
| Hours | public | | | | int | | | | get, set | | | Свойство с доступом к полю hours. Возвращает сохраненное время в часах. | |
| Mins | public | | | | int | | | | get,set | | | Свойство с доступом к полю mins. Возвращает сохраненное время в минутах. | |
| Secs | public | | | | int | | | | get, set | | | Свойство с доступом к полю secs. Возвращает сохраненное время в секундах. | |
| Nodes | public | | | | Node[ ][ ] | | | | get,set | | | Свойство с доступом к массиву nodes. Возвращает зубчатый массив с элементами - узлами. | |
| PbIsSet | public | | | | bool[ ][ ] | | | | get, set | | | Свойство с доступом к полю pbIsSet. Возвращает зубчатый массив, элементы которого  содержат информацию о проведенных связях в позициях поля.  Продолжение таблицы 3.4 | |
| Connections | public | | | | string[ ][ ] | | | | get,set | | | Свойство с доступом к полю connections. Возвращает зубчатый массив, элементы которого  содержат информацию о соединенных соседях для каждого элемента из nodes. | |
| Finished | public | | | | bool | | | | get, set | | | Свойство с доступом к полю finished. Возвращает информацию, решено ли текущее поле или нет. | |
| FieldSize | public | | | | int | | | | get,set | | | Свойство с доступом к полю fieldSize. Возвращает размер сохраненного поля. | |
| HelpButtonVisible | public | | | | bool | | | | get, set | | | Свойство с доступом к полю helpButtonVisible. Возвращает информацию, скрыта ли подсказка или нет. | |

Таблица 3.5

Описание полей и функциональных членов класса MainMenuForm.cs

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | |  | | |  |  | |
| Имя | | Модификатор доступа | | | Тип | Назначение | |
| pathEasyLevelStatistics | | private | | | string | Константа, содержит в себе адрес файла со статистикой пользователей для легкого уровня. | |
| pathMediumLevelStatistics | | private  Продолжение таблицы 3.5 | | | string | Константа, содержит в себе адрес файла со статистикой пользователей для среднего уровня. | |
| pathHardLevelStatistics | | private | | | string | Константа, содержит в себе адрес файла со статистикой пользователей для сложного уровня. | |
| choisedLevel | | private | | | string | Выбранный пользователем уровень. | |
| dataFromStatisticsFile | | private | | | string[ ] | Массив строк из файла статистики о прорешанных пользователями заданиях. | |
| dataFromDataGrid | | private | | | string[ ] | Массив данных строк из элемента DataGridView. | |
| **Методы** | |  | | |  |  | |
| Имя | Модификатор доступа | | Тип | Аргументы | | | Назначение |
| MainForm | public | | конструктор | - | | | Конструктор умолчания для данного класса. |
| MainFormLoad | private | | void | object sender, EventArgs e | | | Вызывается в начале работы скрипта. |
| StartGameButtonClick | private | | void | object sender, EventArgs e | | | Открывает и закрывает информацию о ракете. |
| ExitButtonClick | private | | void | object sender, EventArgs e | | | Обработчик события нажатия на кнопку "Выход". |
| DiffLevelComboBoxSelectedIndexChanged | private | | void | object sender, EventArgs e  Продолжение таблицы 3.5 | | | Обработчик выбора уровня для начала игры.  Инициализируется переменная со значением выбранного уровня. |
| RulesButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия на кнопку "Правила".  Пользователю предлагается изучить правила и пройти обучение. |
| HideRulesButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия кнопки "Скрыть", когда открыта часть окна с Правилами. |
| StatisticsButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия на кнопку "Статистика".  Пользователю открывается статистика о всех прорешанных заданиях на выбранном уровне. |
| HideGameInfoButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия кнопки "Скрыть", когда открыта часть окна описания продукта. |
| HideStatisticsButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия кнопки "Скрыть", когда открыта часть окна о статистике.  Продолжение таблицы 3.5 |
| GameInfoButtonClick | private | | void | object sender, EventArgs e | | | Обработчик события нажатия кнопки "Об игре".  Пользователю откроется описание продукта. |
| DeleteRecordsButtonClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия на кнопку "Удалить результат" в окне статистики.  Удаляются выбранные пользователем строки и из таблицы, и из файла. |
| ClearAllRecordsClick | private | | void | object sender, EventArgs e | | | Обработчик нажатия на кнопку "Очистить статистику" в окне статистики.  Удаляется вся информация и из таблицы, и из файла. |
| СhooseLevelComboBoxSelectedIndexChanged | private | | void | object sender, EventArgs e | | | Обработчик события выбора уровня для выведения статистики.  При срабатывании события в DataGridView выводится статистика  о решенных пользователями полях на выбранном уровне.  Продолжение таблицы 3.5 |

Таблица 3.6

Описание полей и функциональных членов класса SaveResultForm.cs

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Поля** | | | |  | | | | |  | | | | | | |  | |
| Имя | | | | Модификатор доступа | | | | | Тип | | | | | | | Назначение | |
| nameForStatistics | | | | private | | | | | string | | | | | | | Имя пользователя для сохранения результата. | |
| **Методы** | | | | |  | | | | |  | | | | |  | | |
| Имя | | | Модификатор доступа | | | | Тип | | | | | Аргументы | | | | | Назначение |
| InputNameTextBoxTextChanged | | | private | | | | void | | | | | object sender, EventArgs e | | | | | Обработчик события изменения текста в TextBox. |
| NameIsCorrect | | | private | | | | bool | | | | | string str | | | | | Метод проверки на то, вводится ли корректное значение для сохранения результата. |
| InputButtonClick | | | private | | | | void | | | | | object sender, EventArgs e | | | | | Обработчик события нажатия на кнопку "Готово".  Введенное имя сохраняется в файл статистики и затем выводится в DataGridView. |
| InputNameTextBoxKeyDown | | | private | | | | void | | | | | object sender, EventArgs e | | | | | Обработчик события нажатия на клавишу Enter.  Введенное имя сохраняется в файл статистики и затем выводится в DataGridView. |
| CancelButtonClick | | | private | | | | void | | | | | object sender, EventArgs e | | | | | Обработчик события нажатия на кнопку "Отмена".  Отменяется сохранение данных.  Продолжение таблицы 3.6 |
| **Свойства** | |  | | | | | |  | | | | |  | | | | |
| Имя | Модификатор доступа | | | | | Тип | | | | | Доступ | | | Назначение | | | |
| NameForStatistics | public | | | | | string | | | | | get, set | | | Свойство для доступа к полю nameForStatistics. Возвращает имя пользователя. | | | |
| CancelClicked | public | | | | | bool | | | | | get, set | | | Автосвойство, содержащее информацию, была ли нажата кнопка «Отмена». | | | |

Таблица 3.7

Описание полей и функциональных членов класса PauzeForm.cs

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Поля |  | | | | |  | | | |  | |
| Имя | Модификатор доступа | | | | | Тип | | | | Назначение | |
| continueClicked | private | | | | | bool | | | | Информация о том, была ли нажата кнопка "продолжить". | |
| helpClicked | private | | | | | bool | | | | Информация о том, была ли нажата кнопка "Как играть ?". | |
| exitClicked | private | | | | | bool | | | | Информация о том, была ли нажата кнопка "В главное меню". | |
| Методы |  | | | | |  | | | |  | |
| Имя | | Модификатор доступа | | Тип | | | | Аргументы | | | Назначение |
| PauzeForm | | public | | конструктор | | | | - | | | Конструктор умолчания данного класса. |
| СontinueButtonClick | | private | | void | | | | object sender, EventArgs e | | | Обработчик события нажатия на кнопку "Продолжить".  Продолжается процесс игры. |
| ExitButtonClick | | private | | void | | | | object sender, EventArgs e  Продолжение таблицы 3.7 | | | Обработчик события нажатия на кнопку "В главное меню".  Закрывается окно с игровым полем и открывается главное меню. |
| HelpButtonClick | | private | | void | | | | object sender, EventArgs e | | | Обработчик события нажатия на кнопку "Как играть?".  Открывается краткий список правил. |
| **Свойства** | | |  | | | |  | |  | | |
| Имя | | Модификатор доступа | | | Тип | | | Доступ | | Назначение | |
| ExitClicked | | public | | | bool | | | get, set | | Свойство с доступом к полю exitClicked. | |
| HelpClicked | | public | | | bool | | | get, set | | Свойство с доступом к полю helpClicked. | |
| ContinueClicked | | public | | | bool | | | get, set | | Свойство с доступом к полю continueClicked. | |

Таблица 3.8

Описание полей и функциональных членов класса BridgesException.cs

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Методы** | |  | |  | |  | |
| Имя | Модификатор доступа | | Тип | | Аргументы | | Назначение |
| BridgesException | public | | конструктор | | - | | Конструктор умолчания. |
| BridgesException | private | | конструктор | | string message | | Конструктор с одним параметром о сообщении об ошибке. |
| BridgesException | private | | конструктор | | string message, Exception inner | | Конструктор с двумя параметрами с информациями об ошибке. |
| BridgesException | private | | конструктор | | System.Runtime.Serialization.SerializationInfo info,  System.Runtime.Serialization.StreamingContext context | | Конструктор с информацией об ошибке. |

**ЛИСТ РЕГИСТРАЦИИ ИЗМЕНЕНИЙ**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Лист регистрации изменений | | | | | | | | | |
| Номера листов (страниц) | | | | | Всего листов (страниц в докум.) | № документа | Входящий № сопроводительного докум. и дата | Подп. | Дата |
| Изм. | Измененных | Замененных | Новых | Аннулированных |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |