

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по учебной практике**  
**Тема: Кратчайшие пути в графе. Алгоритм А\*.**

Студент гр. 1384	_____	Галенко А.С.
Студент гр. 1384	_____	Феопентов А.Ю.
Студент гр. 1384	_____	Алиев Д.А.
Руководитель	_____	Шестопалов Р.П.

Санкт-Петербург  
2023

**ЗАДАНИЕ**  
**НА «НАИМЕНОВАНИЕ ПРАКТИКИ» ПРАКТИКУ**

Студент Галенко А.С. группа 1384

Студент Феопентов А.Ю. группа 1384

Студент Алиев Д.А. группа 1384

Тема практики: Кратчайшие пути в графе. Алгоритм А\*.

Задание на практику: командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

кратко указываются исходные данные (задание на практику)

Сроки прохождения практики: 30.07.2023 – 13.07.2023

Дата сдачи отчета: 13.07.2023

Дата защиты отчета: 13.07.2023

Студент		Галенко А.С.
Студент		Феопентов А.Ю.
Студент		Алиев Д.А.
Руководитель		Шестопалов Р.П.

## **АННОТАЦИЯ**

Создание программы с поддержкой графического интерфейса для нахождения кратчайшего пути в графе с помощью алгоритма A\*.

## СОДЕРЖАНИЕ

	Введение	4
1.	Требования к программе	6
1.1.	Исходные данные	6
1.1.1.	Ввод исходных данных	6
1.1.2.	Визуализация	6
1.2.	Шаблон архитектура	8
2.	План разработки и распределение ролей	11
2.1.	Общий план	11
2.2.	Распределение ролей	11
3.	Реализация	12
3.1	Архитектура приложения	12
3.2	Классы структур данных	12
3.3	Классы модели	14
3.4	Классы визуализации	17
3.5	Классы контроллера	18
4.	Тестирование	19
4.1	Описание работы программы	19
4.1.1	Демонстрация работы пошагового режима на маленьком графе	23
4.2	Разбор исключительных ситуаций	25
4.2.1	Тестирование алгоритма A*	32
	Заключение	37
	Список использованных источников	39

## **ВВЕДЕНИЕ**

В данном проекте требуется реализовать алгоритм  $A^*$  и графический интерфейс к нему.

Визуализация работы алгоритма будет дополнена возможностью редактирования графа, к которому будет применен алгоритм поиска пути, а также возможность сохранения созданного графа в файл и чтение из него.

Для большей наглядности работы алгоритма будет введен пошаговый режим, что поможет пользователю в понимании работы алгоритма изнутри.

## 1. ТРЕБОВАНИЯ К ПРОГРАММЕ

### 1.1. Исходные данные.

#### 1.1.1. Ввод исходных данных.

Алгоритм должен получать на вход взвешенный ориентированный граф, который представим на плоскости в виде сетки и имеет положительные веса ребер. Данные могут задаваться либо с помощью файла, либо через графический интерфейс. Определим ввод параметров через файл:

- первая строка - высота поля(координата Y).
- вторая строка - ширина поля (координата X).
- Ни же само поле в котором находятся следующие символы:
  - от '1' до '5' - проходимость клетки,
  - '0' - стена («камень»),
  - 'S' - старт,
  - 'F' - финиш,

старт и финиш должны быть обязательно и определены однозначно.

Пример:

```
4
5
12345
55S54
00001
5543F
```

#### 1.1.2. Визуализация.

Интерфейс программы должен содержать окно со следующими областями: отображение поля с графом и визуализация на нем работы алгоритма, поле вывода о статусе программы, рабочей области. Общий вид представлен на рисунке 1.

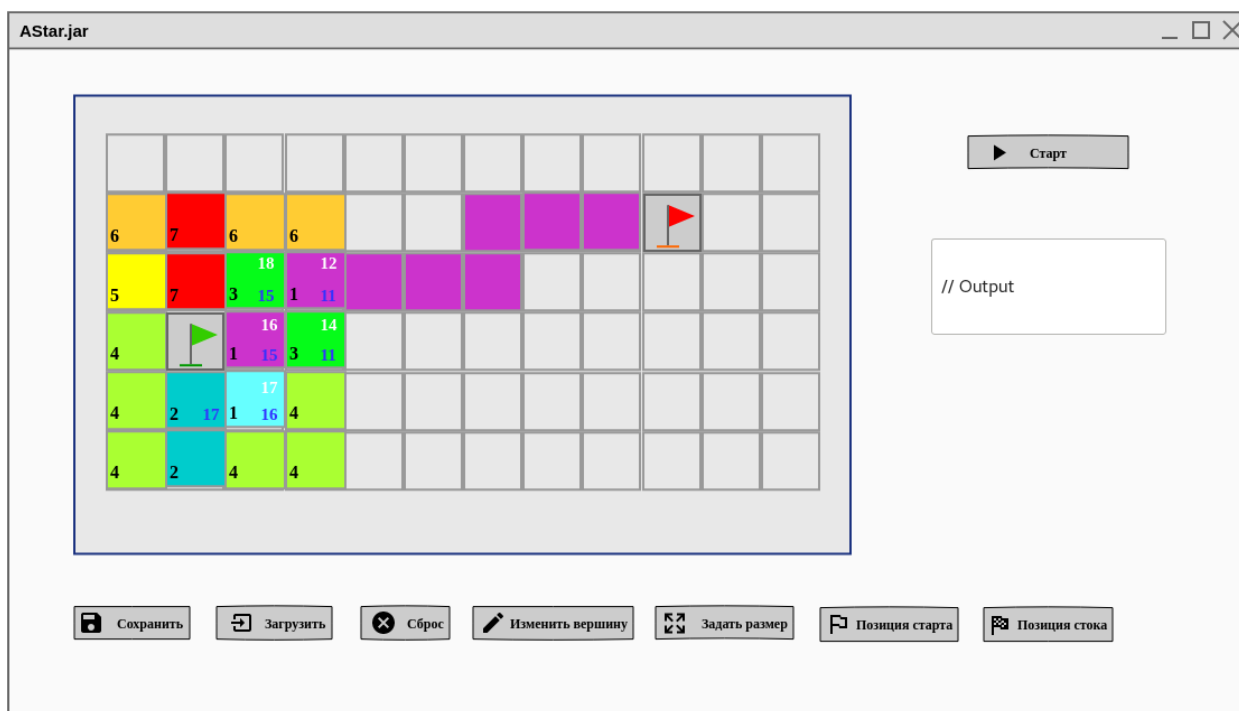


Рисунок 1 — Общий концепт интерфейса

Описание каждой области:

- Область отображения графа — сетчатое поле, где каждая клетка — вершина. Цвет клетки означает конкретную стоимость пути к данной вершине (вес). Соответствие числового значения и цвета будет приведено далее. Если вершина была рассмотрена алгоритмом, то в клетке отображается три числа:  $g(v)$  — вес (левый нижний угол),  $h(v)$  — значение эвристической функции (правый нижний угол), и  $f(v) = g(v) + h(v)$  — оценка стоимости пути (правый верхний угол). После того, как алгоритм отработает, найденный путь будет выделен фиолетовым цветом.
- Поле вывода алгоритма — необходимо для описания результата работы  $A^*$ , т. е. итоговой стоимости пути.
- Рабочая область — это набор кнопок, предназначенных для управления программой.

Функционал рабочей области:

- Кнопка «Изменить вершину» - позволяет изменить значение веса от старта до данной вершины, выдает список возможных «весов» для вершины — рис. 2.

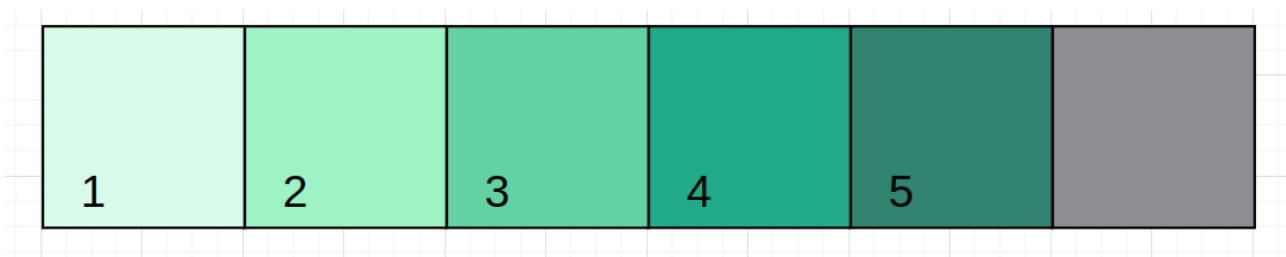


Рисунок 2 — Вершины по значению весов (от непроходимой к самой проходимой)

- Кнопка «Задать размер сетки» - дает возможность пользователю задать размер сетки, в которой нарисован граф. При нажатии кнопки открывается нужное диалоговое окно — рис. 3.

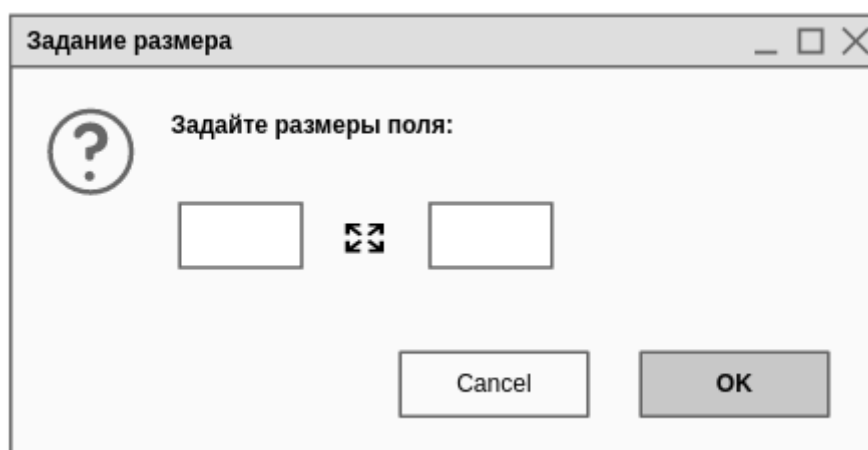


Рисунок 3 - Окно задания размеров поля

- Кнопка «Сброс» - очищает вершины.
- Кнопка «Позиция старта» - позволяет задать позицию вершины-источника.
- Кнопка «Позиция стока» - позволяет задать позицию вершины-стока (финиша).

## 1.2. Архитектура программы.

Для реализации программы была выбрана структура Model-View-Controller. То есть вся реализация будет разделена на три компонента: контроллер, модель, графика. Схема данной архитектуры представлена на рис. 4.



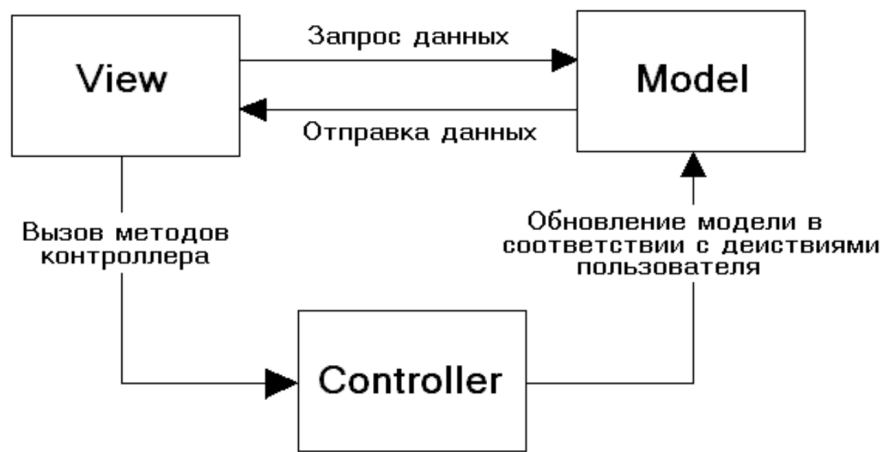


Рисунок 4 — Схема MVC

MVC разделяет приложение на три основных компонента: Модель (Model): Модель представляет данные и бизнес-логику приложения. Она отвечает за хранение, обработку и передачу данных между различными компонентами архитектуры. Модель не зависит от других компонентов и может быть использована в различных контекстах. Представление (View): Представление отвечает за отображение данных модели пользователю. Оно предоставляет пользовательский интерфейс, через который пользователи могут взаимодействовать с приложением. Представление получает данные из модели и отображает их в удобной форме для пользователя. Контроллер (Controller): Контроллер является посредником между моделью и представлением. Он отвечает за обработку пользовательских действий, таких как нажатие кнопок, ввод данных и других событий, и взаимодействует с моделью для обновления данных и с представлением для их отображения. Контроллер также может обрабатывать логику валидации данных и принимать решения о дальнейших действиях в приложении. Преимущества архитектуры MVC включают: Разделение ответственности: MVC позволяет разделить логику приложения на три независимых компонента, что упрощает разработку, тестирование и сопровождение кода. Повторное использование кода: Благодаря разделению на модель, представление и контроллер, каждый из компонентов может быть использован повторно в различных контекстах или приложениях. Улучшенная поддержка параллельной разработки: Команда разработчиков может работать

над различными компонентами независимо друг от друга, что улучшает эффективность и позволяет распараллелить процесс разработки. Легкая замена компонентов: При необходимости можно легко заменить или модифицировать один из компонентов (например, заменить представление без изменения модели или контроллера).

## **2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ.**

### **2.1. План разработки.**

1. Разработка прототипа до 5-7 июля:

Создание окна визуализации. Реализация классов-обработчиков событий в компоненте Controller (ввод/вывод, инициализация начальных данных), структуры архитектуры.

2. Разработка 1-ой версии до 7-10 июля:

Реализация алгоритма А\*. Обеспечение взаимодействия с пользователем с помощью графического интерфейса, вывод результата. Исправление замечаний.

3. Разработка 2-ой версии до 10-12 июля

Добавление тестирования. Исправление замечаний

4. Сдача финальной версии 12-13 июля

Исправление замечаний

### **2.2. Распределение ролей.**

1. Галенко Алексей - написание компонента Model.
2. Алиев Дмитрий - написание компонента Controller.
3. Феопентов Аким - написание компонента View.
4. Совместно - составление архитектуры, проработка дизайна, написание отчёта.

### 3.РЕАЛИЗАЦИЯ.

### 3.1. Архитектура приложения.

Архитектура приложения основана на наборе следующих компонентов:

Графика, модель и контроллер. Общая схема приложения представлена на рис. 5

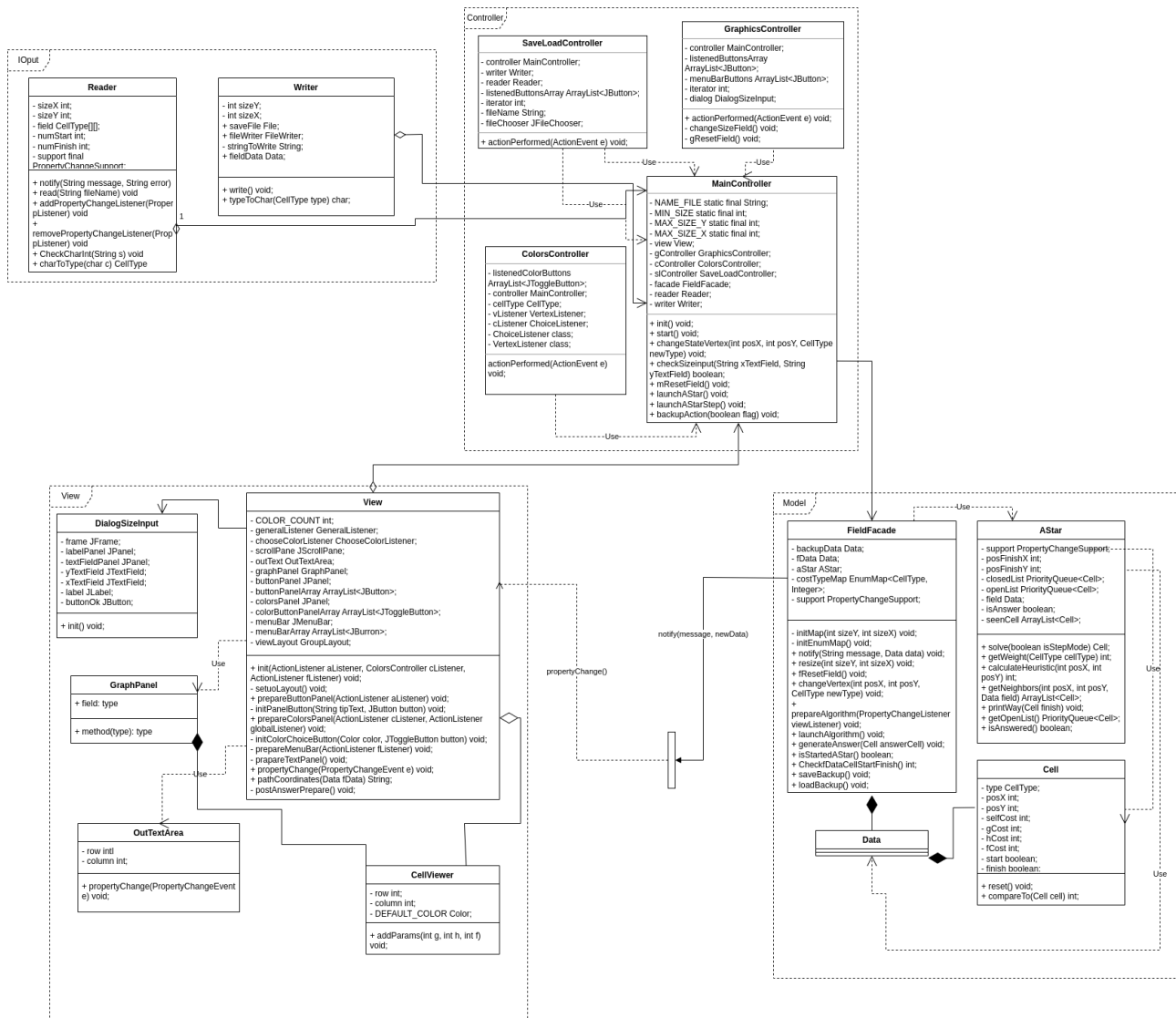


Рисунок 5 - UML диаграмма

### 3.2. Классы структур данных.

Основным классом в данной категории является **Data**. Он содержит в себе описание поля вершин, хранит ссылки на старт, финиш, найденный путь.

Поля класса:

*private int sizeY, sizeX;* - размеры поля графа

*private Cell[][] field;* - матрица вершин

*private Cell startCell;* - ссылка на стартовую вершину

*private Cell finishCell;* - ссылка на финишную вершину

*private Cell updatedCell;* - ссылка на недавно измененную пользователем вершину

*private Cell curCell;* - ссылка на рассматриваемую в конкретный момент алгоритмом вершину-кандидата

*private ArrayList<Cell> path;* - список, в котором хранится путь алгоритмом

*private int pathCost;* - стоимость пути

*private ArrayList<Cell> openList;* - общий список обработанных алгоритмом вершин на конкретной итерации

Класс **Cell**. Непосредственно описывает состояние каждой вершины графаю

Поля класса:

*Cell parentCell;* - ссылка на клетку, из которой можно прийти в данную, чтобы путь из старта финиша был минимальным (записывается алгоритмом)

*private CellType type;* - тип клетки

*private int posX;* - позиция по X

*private int posY;* - позиция по Y

*private int selfCost;* - стоимость ребра

*private int gCost;* - стоимость пути из старта в данную клетку

*private int hCost;* - значение эвристической функции для данной клетки

*private int fCost;* - значение эвристической оценки ( $gCost + hCost$ )

*private boolean start;* - флаг для стартовой вершины

*private boolean finish;* - флаг для финишной вершины

Класс **CellViewer**. Представляет собой описание клетки с точки зрения графики. Это наследник JButton, который рисует клетку в окне.

Поля класса:

*private int row;* - позиция в строке

`private int column;` - позиция в столбце

`static final Color DEFAULT_COLOR;` - цвет клетки по умолчанию

### 3.3. Классы модели.

Рассмотрим каждый компонент по отдельности.

Модель включает в себя следующие классы: `FieldFacade`, `Cell`, `AStar` и `Data`. Для взаимодействия контроллера и модели был создан главный класс в данном компоненте - `FieldFacade`. В нем реализован весь основной функционал для изменения модели. Фасад хранит в себе `fData` (экземпляр класса `Data`), который описывает состояние графа в какой-то момент времени. Так как граф представлен на двумерной плоскости, `fData` содержит в себе двумерный массив экземпляров класса `Cell`. `Cell` описывает конкретную вершину на плоскости: ее позицию, тип, флаг старта и финиша, стоимость перехода в нее, эвристика и эвристическая оценка пути в нее.

Класс `AStar` нужен для реализации алгоритма `AStar`, за это отвечает функция `solver`. В поля входят: Позиции финиша (координаты `x`, `y`), две очереди с приоритетом (`openList` - список вершин для просмотра, `closedList` - список просмотренных вершин), `Data field`, подробнее в описании класса `Data`. Флаг готовности ответа `boolean isAnswer`. И список клеток для просмотра `seenCell`. И поле для отправки сигналов `PropertyChangeSupport`.

В конструктор передается `data` из фасада и слушатель визуализации. Из `data` извлекаются позиции финишной клетки. Выделяется память под две очереди с приоритетом. В стартовой клетки значение эвристической функции `h` присваиваются к максимальному значению `integer`. В `openList` добавляется клетка старта. Выделяется память под список клеток для просмотра. В сигнал добавляется слушатель.

Эвристическая функция высчитывается здесь как максимум из разности координат `x` и `y` между текущей и финишной клеткой.

Метод solve. Он получает на вход флаг boolean isStepMode. В зависимости от этого алгоритм будет посылать сигналы после каждого шага или нет. Далее идет цикл пока список вершин для просмотра не пустой. Затем достается из этого списка клетку с самой маленькой  $f$  (путь до этой клетки( $g$ ) + эвристическая функция  $f$ ). Далее идет проверка на то, является ли эта вершина финишной. Если да, то мы достаем из неё  $g$  и присваиваем длине пути. Также isAnswer присваиваем true и возвращаем финишную клетку. Иначе, создается список соседей и присваиваем туда соседей клетки, которую получает из очереди. Далее проходит всех соседей. Если сосед не лежит ни в одной из очереди с приоритетом, то присваивается родитель соседа, считаются три функции  $f$ ,  $g$ ,  $h$  и клетка помещается в список вершин для просмотра. Иначе, если у соседа длина пройденного пути меньше, чем длина пути у той же клетки, что в каком-нибудь списке, то мы делаем тоже самое, что выше. И если вершина является просмотренной, то достаем её из этого списка и помещаем в список вершин для просмотра. После просмотра всех соседей текущую клетку достается из списка вершин для просмотра и помещается в список вершин просмотренных. Если алгоритм работает в пошаговом режиме, то передается список просмотренных вершин и field сигналом. После выхода из цикла while очищаем всю память и возвращаем null. так как пути нет.

Класс Cell имеет следующие поля. Тип клетки type (Элемент класса перечисления enum), позиции клетки (x,y), Значение клетки от 1 до 5, 0 - камень. Также start, finish типа boolean - нужны для того, чтобы понять является ли клетка стартовой или финишной.  $g$  - вес пути от стартовой до текущей, определяется в алгоритме AStar. по умолчанию равен весу клетки.  $h$  - эвристическая функция, тоже определяется в алгоритме AStar по умолчанию равна нулю.  $f$  - сумма  $h$  и  $g$ , тоже определяется в алгоритме AStar по умолчанию равна нулю. Из методов getter и setter всех полей. Функция reset - очищение клетки, а также compareTo - переопределение оператора сравнения, который используется для очереди с приоритетом в AStar.

Класс `CellType` - enum. Содержит типы клеток от 1 до 5: `FIRST_TYPE`, `SECOND_TYPE`, `THIRD_TYPE`, `FOURTH_TYPE`, `FIFTH_TYPE`, также камень(стена): `BLOCK_TYPE`, тип старта и финиша: `SOURCE_TYPE`, `STOCK_TYPE`, соответственно.

Класс `Data` содержит: размеры поля `sizeX`, `sizeY`. `field` - само поле двумерный массив типа `Cell`. Клетку старта и финиша. Список клеток входящих в путь и вес пути. Из методов `getter` и `setter` всех полей, конструкторы и метод очищения поля.

Класс `FieldFacade` содержит: параметры поля `Data fData`, элемент класса `Astar astar`. Поле для работы наблюдателями `PropertyChangeSupport support`;

Фасад оповещает визуализацию через `PropertyChangeSupport` и `PropertyChangeListener` в следующих случаях:

- поле было прочитано из файла или загружено по умолчанию
- была нажата кнопка “Сброс” (поле было пересоздано)
- был изменен размер поля
- был завершен алгоритм без пошагового режима
- алгоритм нашел или не нашел решение
- при запуске алгоритма не были заданы старт и(или) финиш
- при очистке решения
- была изменена одна вершина(обычная, старт или финиш)

Данные сообщения получают классы, входящие в компонент Визуализации, в зависимости от строки события, которая прилагается к отправке вместе с новым состоянием типа `Data`, графика перерисовывает в нужном `JPanel` конкретный объект, который был изменен в модели.



### 3.4. Классы визуализации.

Включает в себя классы: View, GraphPanel, CellViewer и OutTextArea. Основным является View, он наследуется от JFrame и содержит в себе остальные компоненты. Также View является наблюдателем для FieldFacade и AStar, то есть он обрабатывает сообщения, поступающие при изменении модели рис. 6.

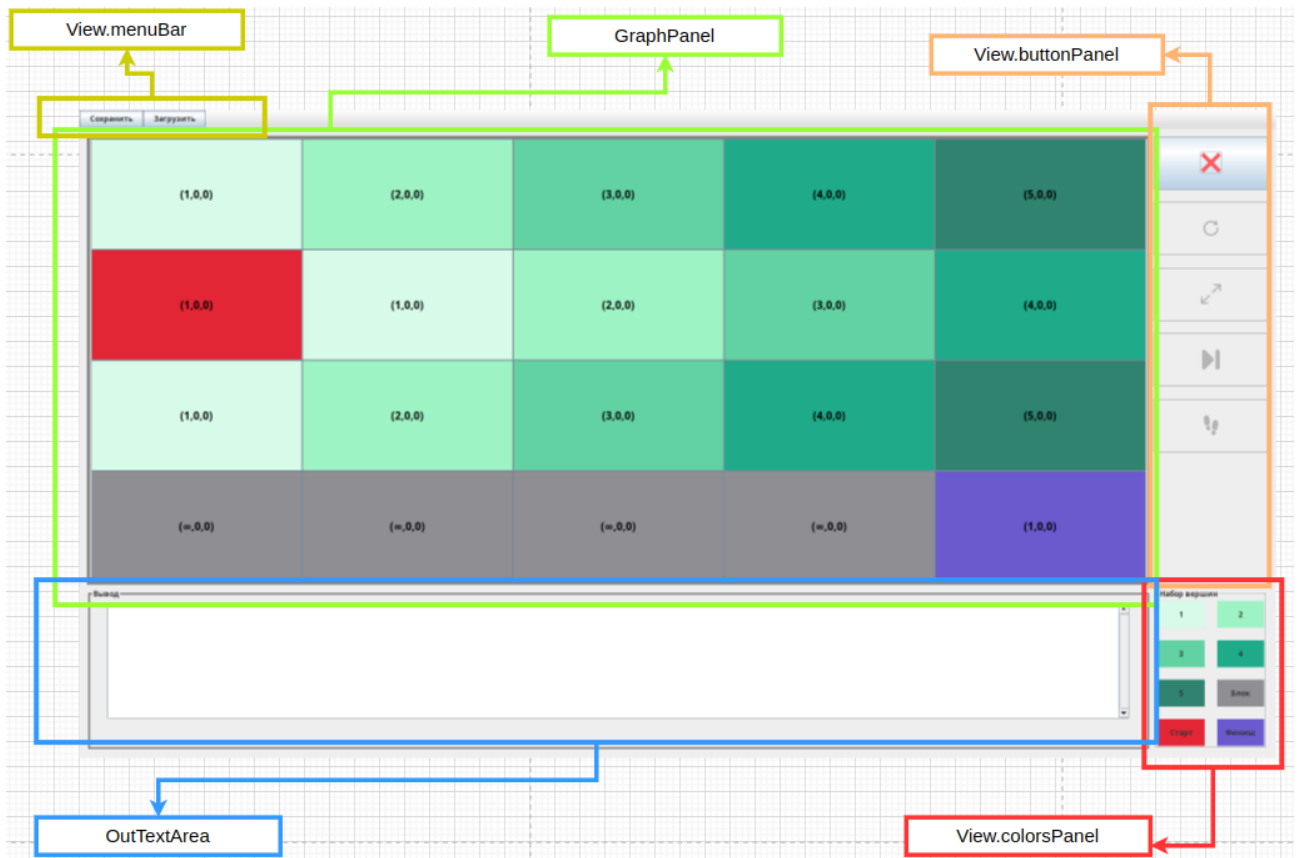


Рисунок 6 - Составные части JFrame в View

GraphPanel - наследник JPanel, реализует наблюдателя также, как и View. Его задача - содержать в себе кнопки (каждая вершина в визуализации является кнопкой) и контролировать их состояние, поведение в зависимости от модели.

### **3.5. Классы контроллера.**

В нем реализованы классы, такие как Write. Read в папке IOput. В этих двух классах реализовано взаимодействие с файлами. В классе Read реализована проверка на правильность и корректность вводимого файла. Рассмотрены все краевые случаи. GraphicsController - здесь реализован слушатель и взаимодействие с кнопками на главном экране.

ColorsController - реализован слушатель и взаимодействие с кнопками в панели цветов, которая открывается после нажатия кнопки редактировать из GraphicsController.

SaveLoadController - реализован слушатель и взаимодействие с верхней панелью сохранения и загрузки.

MainController - за обработку и вызов всех контроллеров, перечисленных выше, вызов пошагового или обычного алгоритма AStar. Также за изменение состояний вершин поля и его размера.

## 4.ТЕСТИРОВАНИЕ.

### 4.1. Демонстрация работы приложения.

Запуск программы (рис. 7):



Рисунок 7 - Первичный запуск программы

При первичном запуске панель графа уже содержит все необходимые компоненты и служит примером графа, который можно сделать в программе.

Теперь функционал. Поле можно “сбросить”, то есть очистить его от всех компонентов (рис. 8).

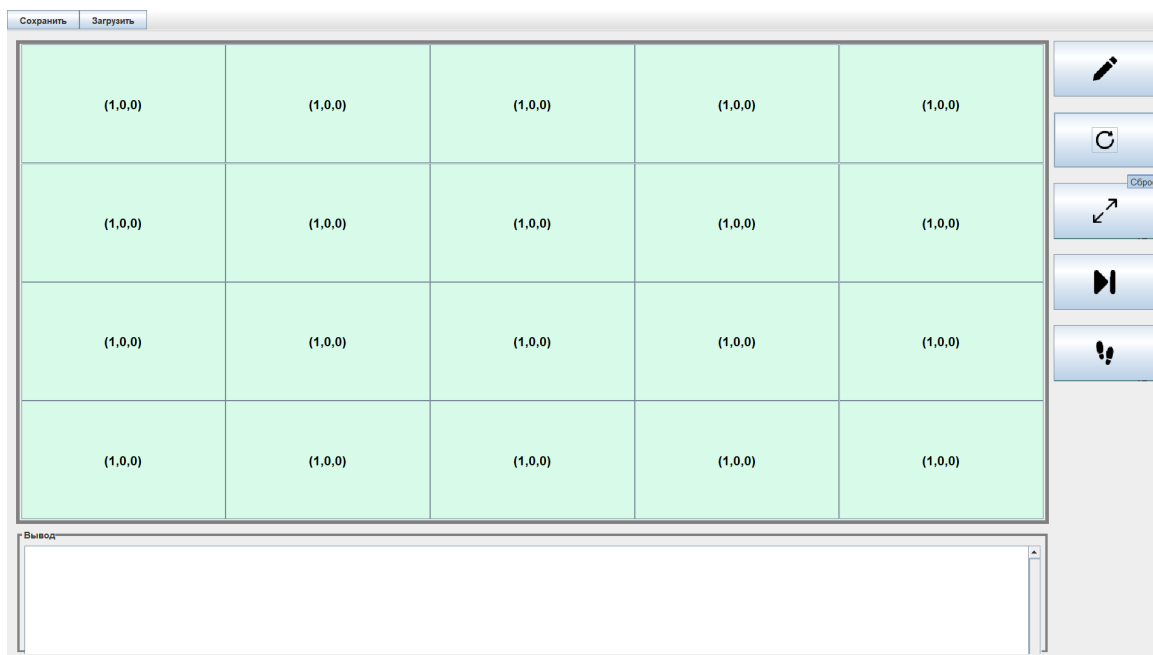


Рисунок 8 - Сброс текущего поля

Программа позволяет задать размер графа. В текстовых полях ввода следует указать высоту(Y) и ширину(X) сетки графа (рис. 9).

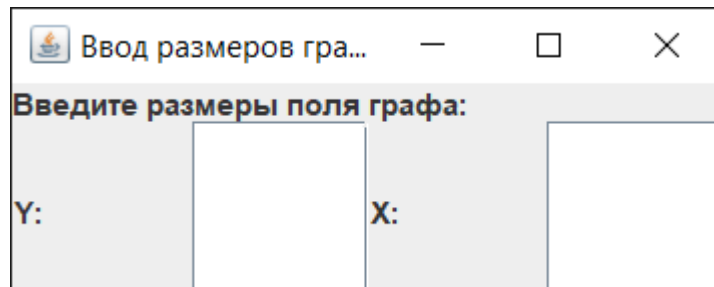


Рисунок 9 - Окно ввода размеров графа

Предположим, что в окне ввода размеров мы ввели 10 и 10. Результатом такого ввода будет граф на рисунке 10.

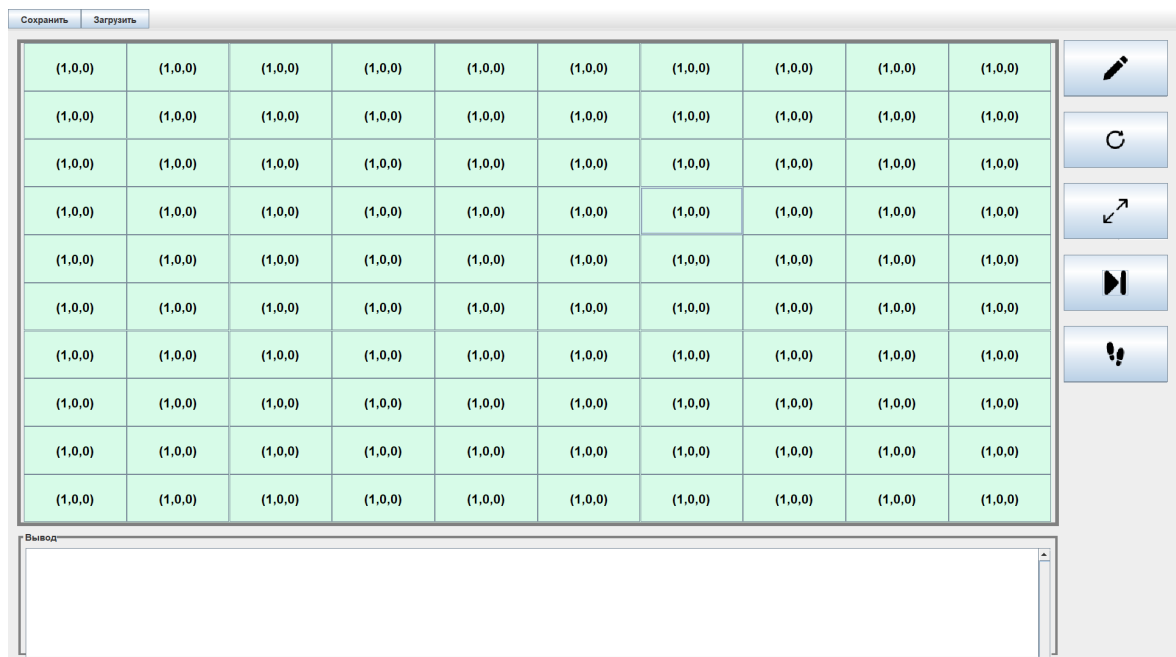


Рисунок 10 - Граф с новым размером

Теперь зададим старт, финиш и вес клеток. Для этого нужно нажать на кнопку “редактировать” с иконкой карандаша. Нажатие на неё переводит программу в режим редактирования графа (рис. 11).

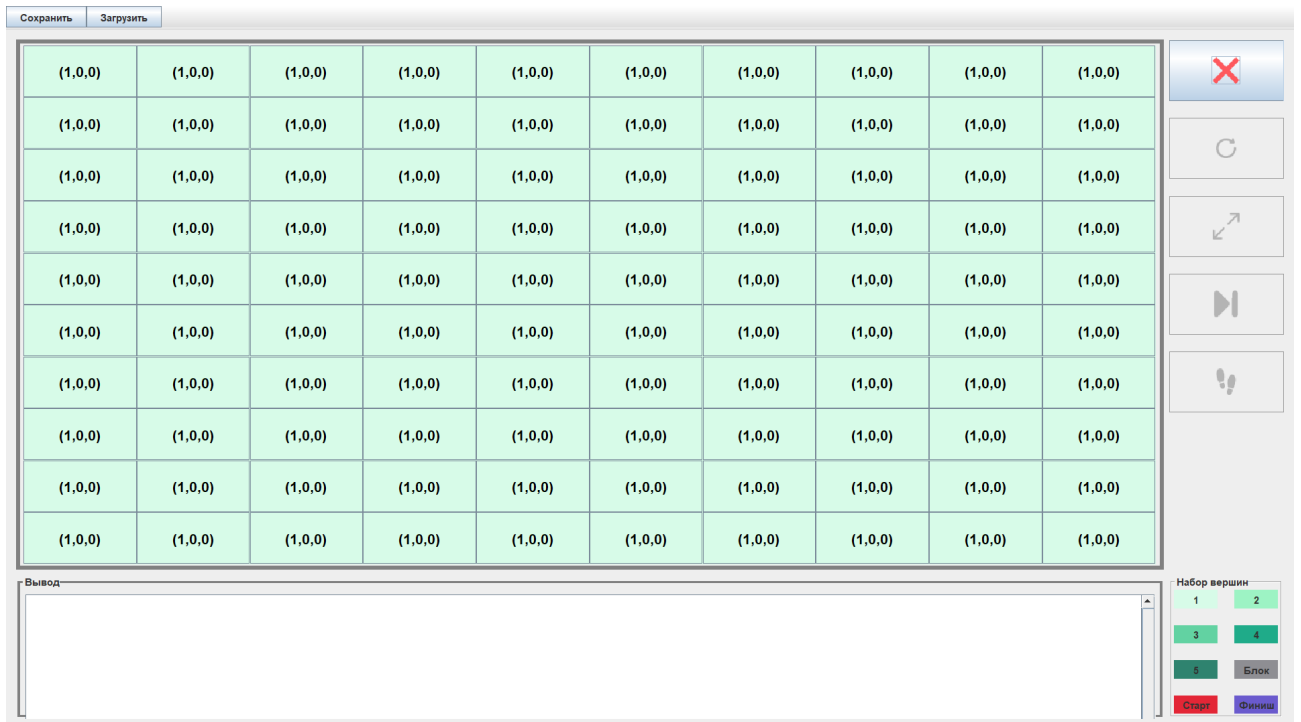


Рисунок 11 - Панель задания цвета, старта и финиша в графе  
Зададим произвольный граф (рис. 12).

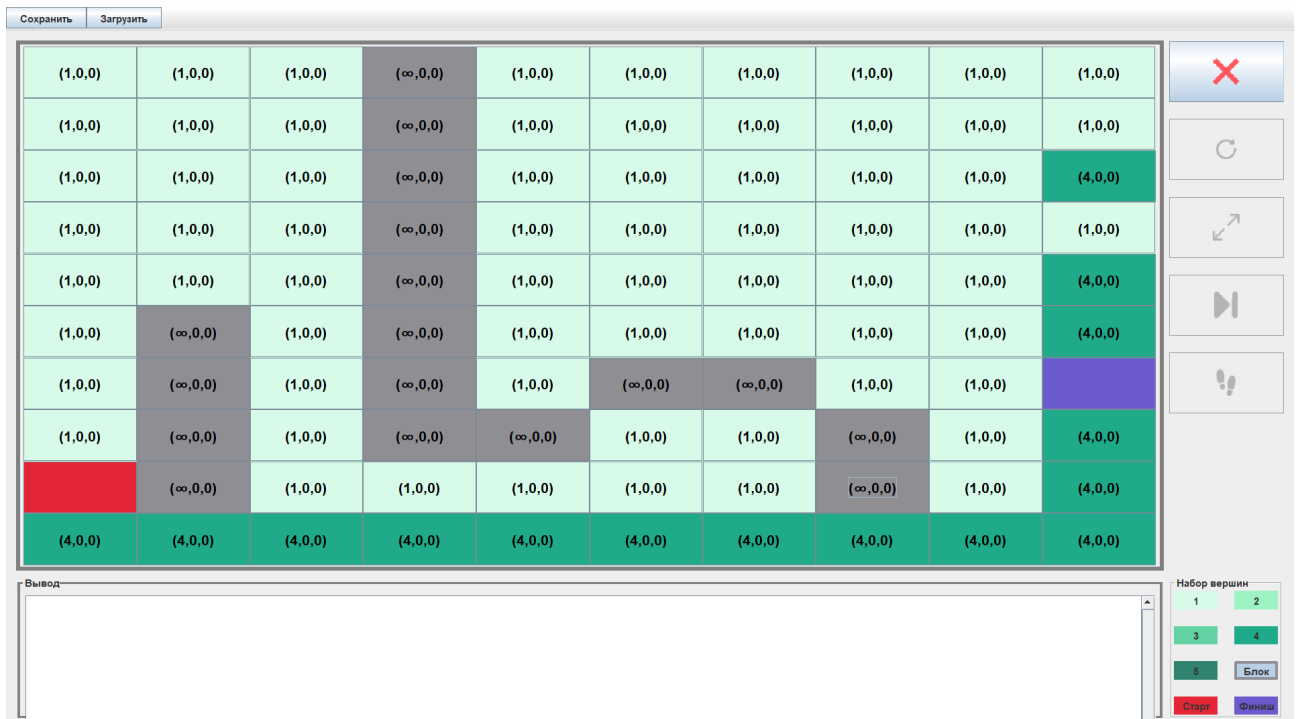


Рисунок 12 - Произвольный граф (рис. 12).

После выхода из режима редактирования(по кнопке “крест”), можем запустить программу. Алгоритм  $A^*$  находит кратчайший путь, если таковой есть. Ответ выводится в текстовую панель под графом. В данном случае путь есть (рис. 13).

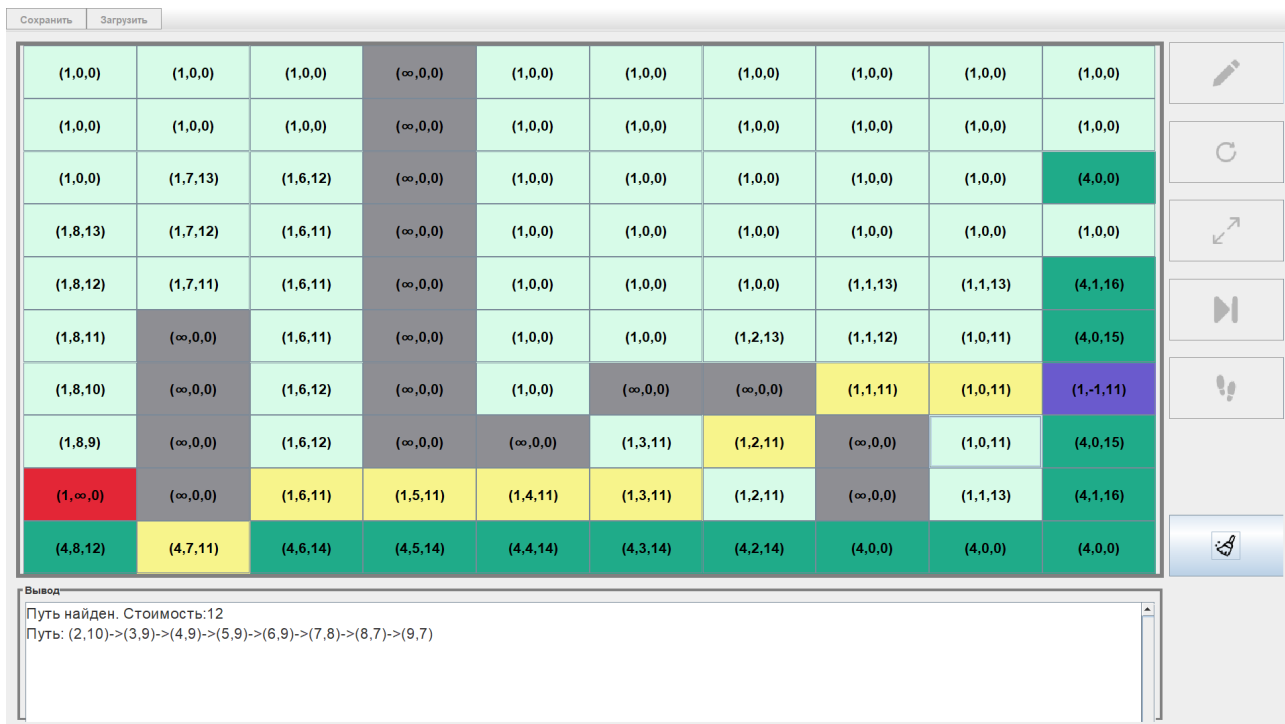


Рисунок 13 - Визуализация выполнения алгоритма

#### 4.1.1 Демонстрация работы пошагового режима на маленьком графе.

Граф (рис. 14).



Рисунок 14 - Граф для демонстрации пошагового режима

На первом шаге светло зеленым обведены вершины, которые попали в список вершин на просмотр. Желтым обведена вершина, которая имеет наименьшее значение  $f$  - (вес пути до текущей вершины + значение эвристической функции) (рис. 15).

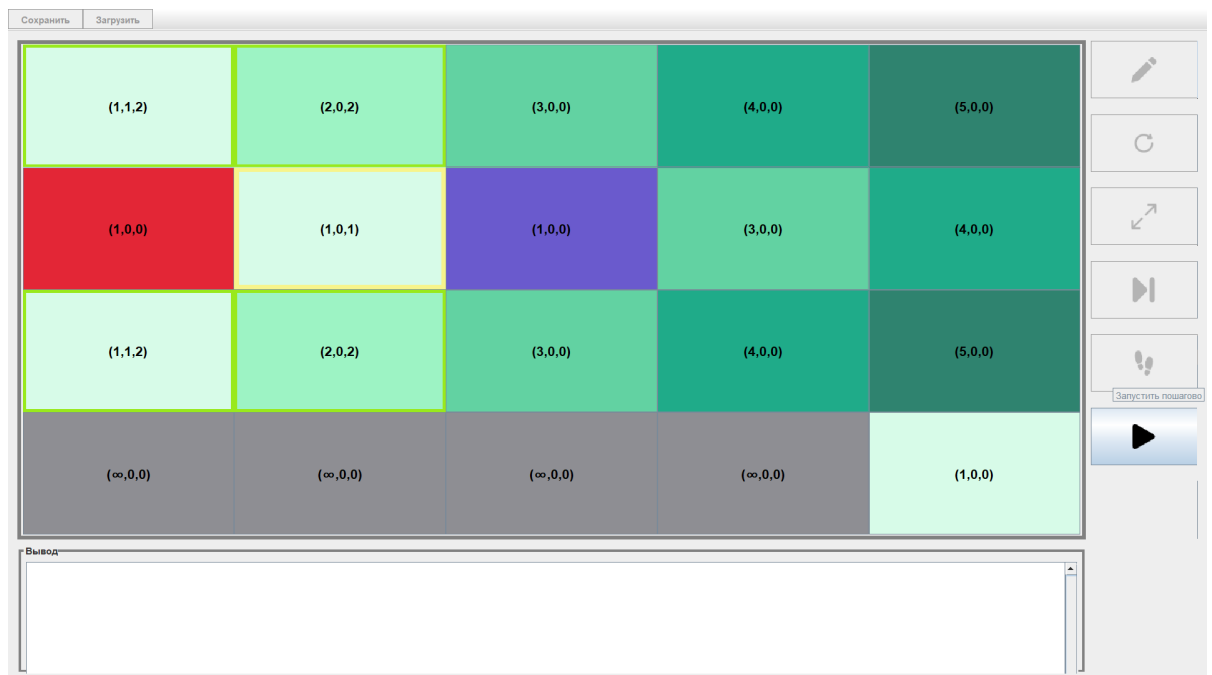


Рисунок 15 - Первый шаг

Второй шаг (рис. 16):



Рисунок 16 - Второй шаг

Вывод пути (рис. 17):



Рисунок 17 - Вывод алгоритма

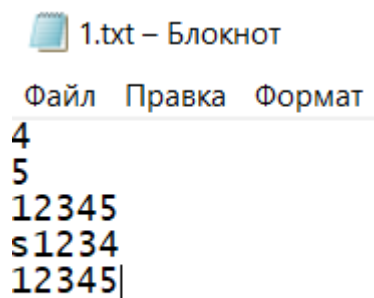
После очищения граф возвращается в исходное состояние.



## 4.2. Разбор исключительных ситуаций.

### Ввод файла.

Попробуем ввести неправильно заданный файл (рис. 18):



```
1.txt – Блокнот
Файл  Правка  Формат
4
5
12345
s1234
12345|
```

Рисунок 18 - Файл с нехваткой одной строки

Обработаем пример (рис. 19).



Рисунок 19 - Обработка примера из рисунка 18

Рассмотрим файл (рис. 20).

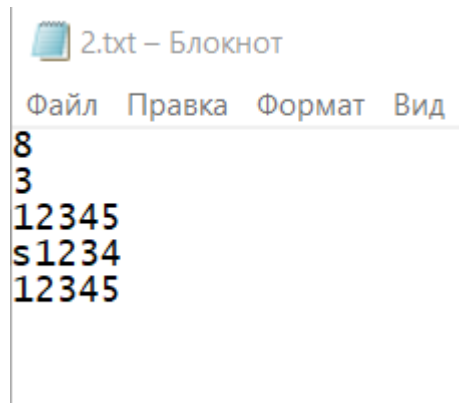


Рисунок 20 - Несоответствие поля его размерам

Обработаем пример (рис. 21).



Рисунок 21 - Отработка примера из рисунка 20

Рассмотрим файл (рис. 22).

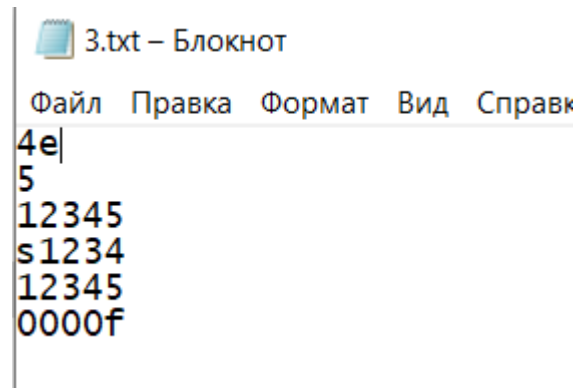


Рисунок 22 - Неправильный ввод размера поля

Обработаем пример (рис. 23).

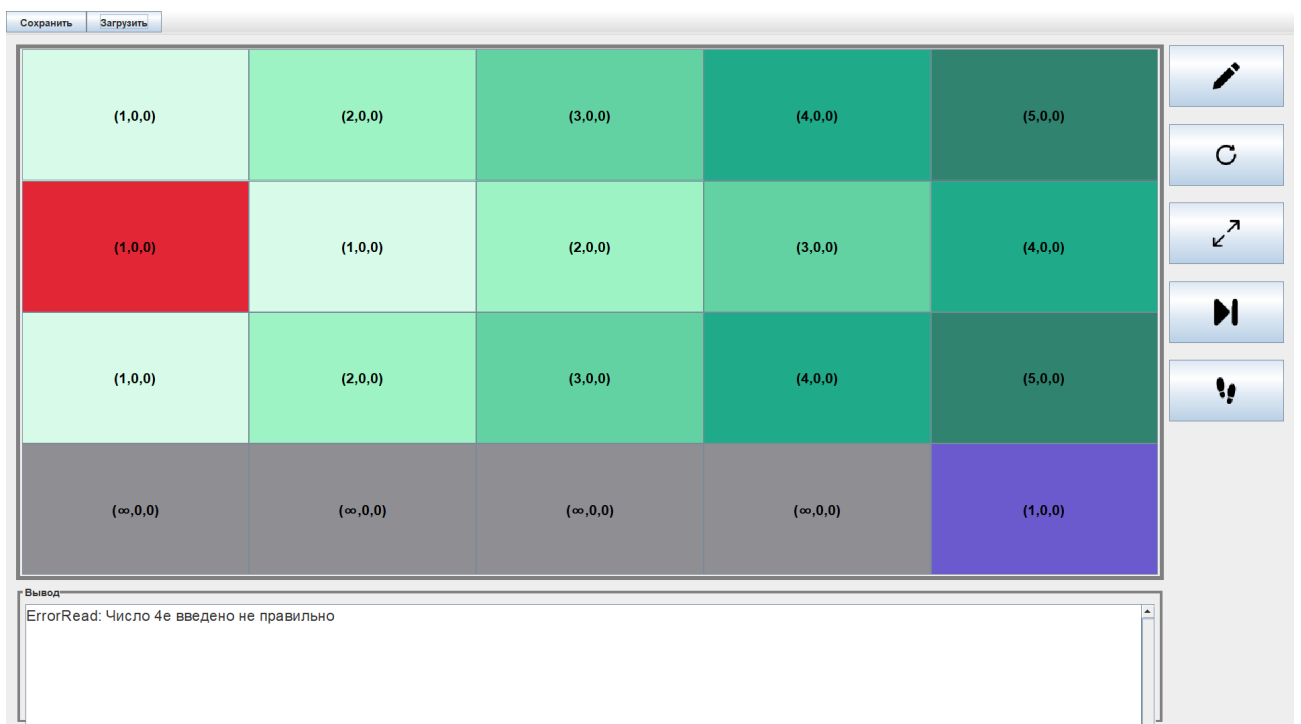


Рисунок 23 - Отработка примера из рисунка 22

Рассмотрим файл (рис. 24).

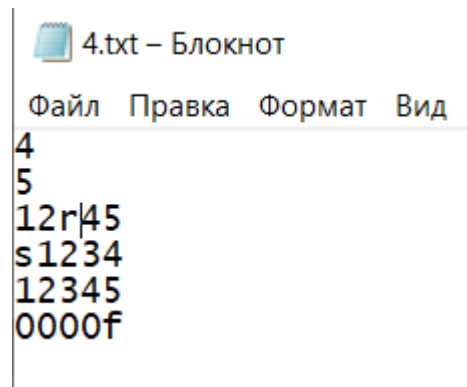


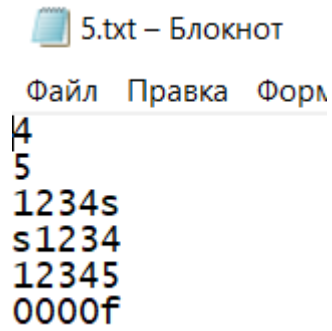
Рисунок 24 - Несертифицированный символ r

Обработаем пример (рис. 25).



Рисунок 25 - Обработка примера из рисунка 24

Рассмотрим файл (рис. 26).



```
4
5
1234s
s1234
12345
0000f
```

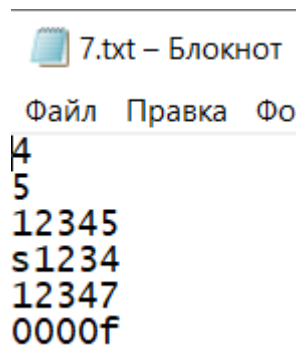
Рисунок 26 - Два старта

Обработаем пример (рис. 27).



Рисунок 27 - Обработка примера в рисунке 26

Рассмотрим файл (рис. 28).



```
4
5
12345
s1234
12347
0000f
```

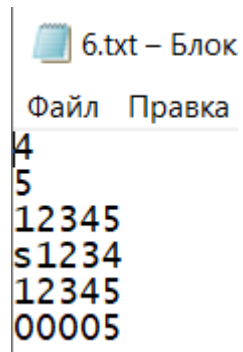
Рисунок 28 - Неправильный символ

Обработаем пример (рис. 29).



Рисунок 29 - Обработка примера в рисунке 28

Рассмотрим файл (рис. 30).



```
4
5
12345
s1234
12345
00005
```

Рисунок 30 - Нет финиша

Обработаем пример (рис. 31).



Рисунок 31 - Обработка примера из рисунка 30

### 4.2.1 Тестирование алгоритма А\*.

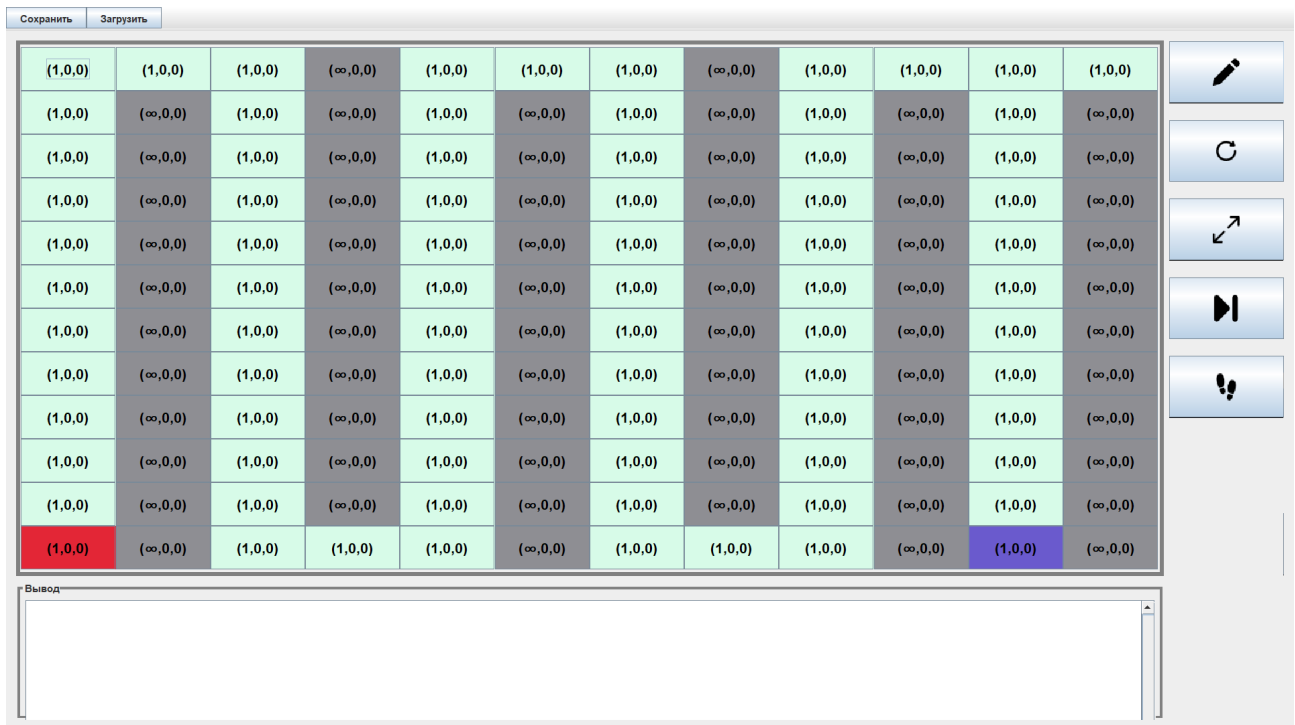


Рисунок 32 - Тест 1

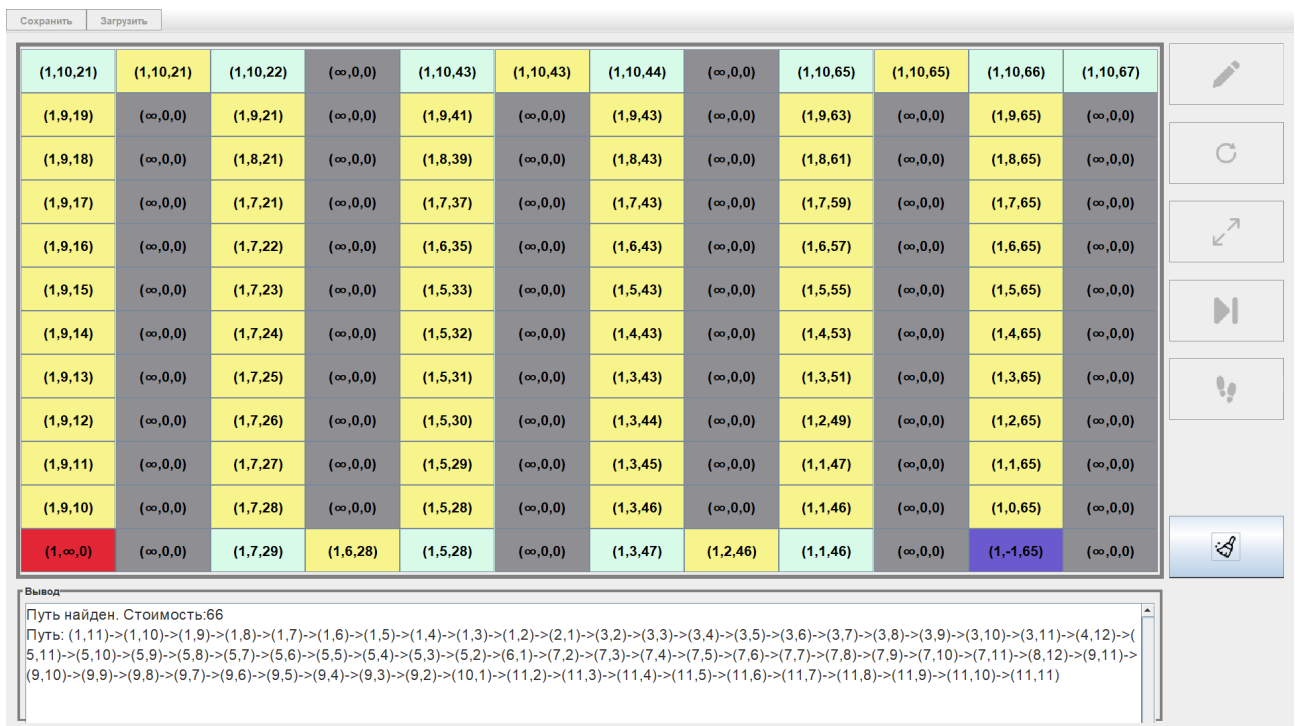


Рисунок 33 - Выполнение первого теста









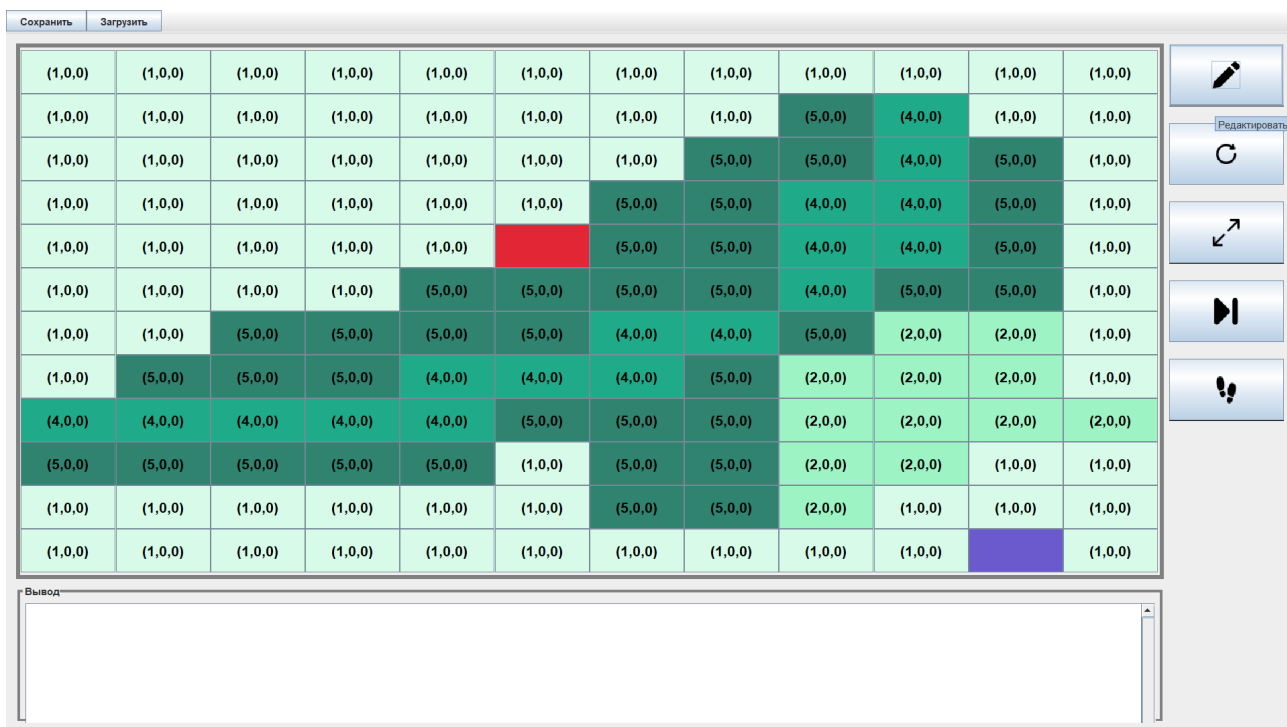


Рисунок 40 - Тест 5

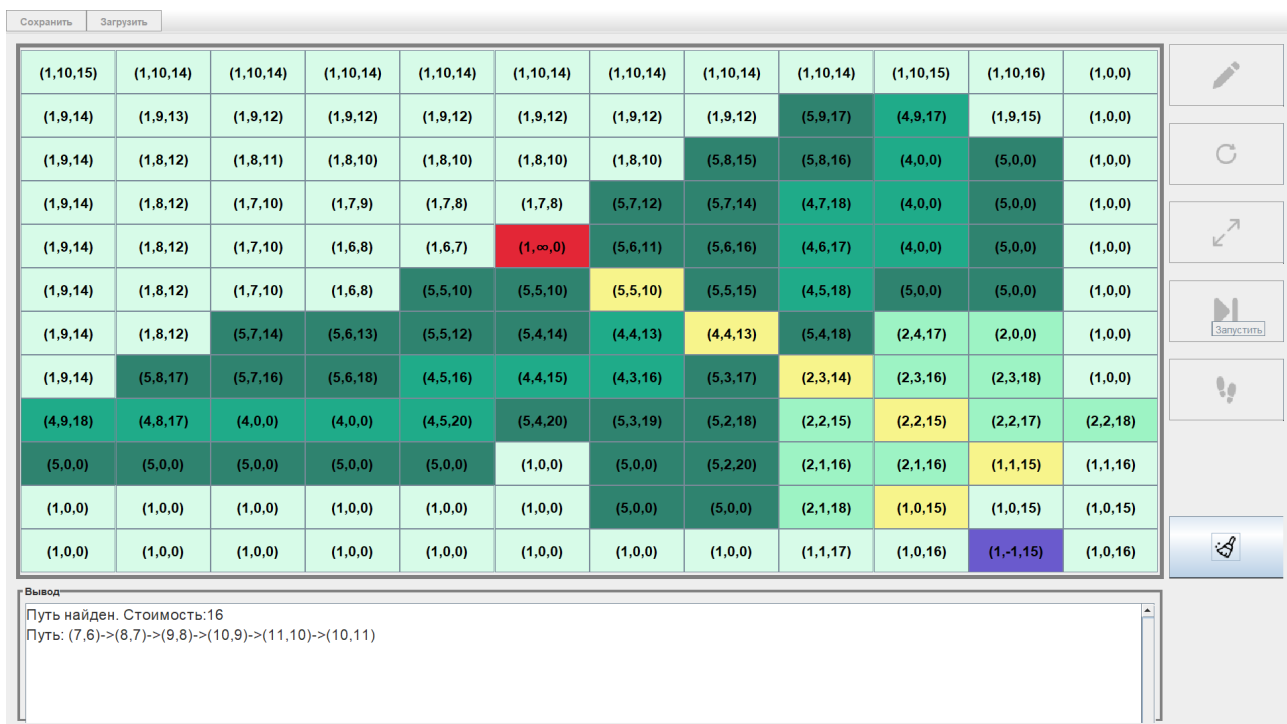


Рисунок 41 - Выполнение пятого теста

## ЗАКЛЮЧЕНИЕ

В заключение учебной практики, которая включала разработку визуализации алгоритма  $A^*$  на языке Java с использованием JavaSwing, можно отметить значительные достижения и положительные результаты, полученные бригадой из трёх человек.

В ходе практики нами была успешно реализована визуализация алгоритма  $A^*$  с использованием JavaSwing. Мы сосредоточились на создании интуитивно понятного пользовательского интерфейса, который позволяет визуально представить работу алгоритма  $A^*$  на двумерной сетке. Благодаря этому, пользователи могут наглядно наблюдать, как алгоритм находит оптимальный путь между двумя заданными точками, используя эвристику и оценку стоимости.

В ходе разработки проекта мы активно сотрудничали в команде, делаясь идеями, знаниями и навыками. Каждый участник команды внес значимый вклад в общий результат. Мы эффективно распределяли задачи, учитывая наши индивидуальные сильные стороны, что позволило нам оптимально использовать время и ресурсы.

Кроме того, мы придерживались лучших практик разработки программного обеспечения, следуя четкому плану работы, устанавливая цели и контролируя свои промежуточные достижения. Мы также активно использовали систему контроля версий, чтобы отслеживать изменения, вносимые каждым членом команды, и обеспечить целостность кодовой базы.

В результате нашей работы мы получили полностью функциональную и эффективную визуализацию алгоритма  $A^*$ , которая соответствует требованиям проекта. Наше приложение обладает понятным пользовательским интерфейсом,

алгоритм  $A^*$  работает корректно и находит оптимальные пути в заданных условиях.

В процессе практики мы также набрались ценного опыта в разработке на языке Java и использовании JavaSwing для создания графического интерфейса. Этот опыт будет нам полезен в будущем при разработке других проектов.

Заключая, мы с гордостью отмечаем, что наша команда успешно справилась с заданием и достигла поставленных целей. Эта практика позволила нам не только расширить наши знания и навыки в разработке программного обеспечения, но и сформировать эффективный коллективный подход к работе в команде. Мы готовы применить эти навыки и опыт на будущих проектах и с нетерпением ждем новых вызовов и возможностей для развития.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Java. Базовый курс // Stepik.org  
URL: <https://stepik.org/course/187/syllabus>
2. JavaSwing Tutorial // JavaTPoint.com  
URL: <https://www.javatpoint.com/java-swing>
3. Алгоритм A\* // wikipedia.org  
URL: [https://ru.wikipedia.org/wiki/A\\*](https://ru.wikipedia.org/wiki/A*)
4. Репозиторий бригады // github.com  
URL: [https://github.com/Dlexeyn/SummerPractice\\_Project](https://github.com/Dlexeyn/SummerPractice_Project)
5. Java PropertyChangeListener tutorial(шаблон наблюдателя) // demo2s.com  
URL:  
<https://www.demo2s.com/java/java-propertychangelistener-tutorial-with-examples.html>