

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

ОТЧЕТ
по учебной практике
Тема: Разработка визуализатора алгоритма A* на языке Java с
графическим интерфейсом.

Студент гр. 0382	_____	Афанасьев Н.С.
Студент гр. 0382	_____	Крючков А.М.
Студентка гр. 0382	_____	Рубежова Н.А.
Руководитель	_____	Фирсов М.А.

Санкт-Петербург
2022

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Афанасьев Н.С. группы 0382

Студент Крючков А.М. группы 0382

Студентка Рубежова Н.А. группы 0382

Тема практики: Разработка визуализатора алгоритма A* на языке Java с графическим интерфейсом.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: A* (нахождение кратчайшего пути в графе).

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 08.07.2020

Дата защиты отчета: 08.07.2020

Студент	_____	Афанасьев Н.С.
Студент	_____	Крючков А.М.
Студентка	_____	Рубежова Н.А.
Руководитель	_____	Фирсов М.А.

АННОТАЦИЯ

Цель практики заключается в командной итеративной разработке визуализатора алгоритма A^* (A-star) с графическим интерфейсом. Работа подразумевает выполнение задач в команде для достижения поставленной цели. Результатом работы должна стать программа, которая визуализирует алгоритм A^* на графе, введенном пользователем. Также целью практики является изучение нового языка программирования Java и отработка полученных знаний на практике.

SUMMARY

The purpose of the practice is the team iterative development of the A^* algorithm visualizer (A-star) with a graphical interface. Work involves performing tasks in a team to achieve a goal. The result of the work should be a program that visualizes the A^* algorithm on a graph entered by the user. Also, the purpose of the practice is to learn a new Java programming language and practice the acquired knowledge in practice.

СОДЕРЖАНИЕ

	Введение	5
1.	Требования к программе	6
1.1.	Исходные требования к программе	6
2.	План разработки и распределение ролей в бригаде	8
2.1.	План разработки	8
2.2.	Распределение ролей в бригаде	9
3.	Особенности реализации	10
3.1.	Представление графа	10
3.2.	Реализация алгоритма	10
3.3	Графический интерфейс	11
3.3	Визуализация алгоритма	13
4.	Тестирование	15
4.1	Тестирование структуры представления графа	15
4.2	Тестирование кода алгоритма А*	16
	Заключение	19
	Список использованных источников	20
	Приложение А. UML-диаграмма	21

ВВЕДЕНИЕ

Цель практики: итеративно разработать визуализатор алгоритма A^* на языке Java с графическим интерфейсом.

Задачи практики:

1. Изучить новый язык программирования *Java* и его основные средства.
2. Научиться итеративной разработке в команде с использованием системы контроля версий *Git*.
3. Реализовать выбранный алгоритм на языке *Java* с визуализацией и графическим интерфейсом.
4. Защитить разработанный проект.

Реализуемый алгоритм:

Алгоритм A^* . Позволяет найти кратчайший путь в ориентированном графе от начальной точки до конечной. Вместо равномерного исследования всех возможных путей он отдаёт предпочтение путям с низкой стоимостью.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение эвристики $f(x)$, после чего этот узел раскрывается.

Примечание: $f(x) = g(x) + h(x)$, где $g(x)$ — функция стоимости достижения рассматриваемой вершины x из начальной, $h(x)$ — функция эвристической оценки расстояния от рассматриваемой вершины к конечной.

На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в очереди с приоритетом. Приоритет пути определяется по значению $f(x)$. Алгоритм продолжает свою работу до тех пор, пока значение $f(x)$ целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1 Исходные Требования к программе

1.1.1 Общие Исходные Требования

а. Приложение должно быть с графическим интерфейсом.

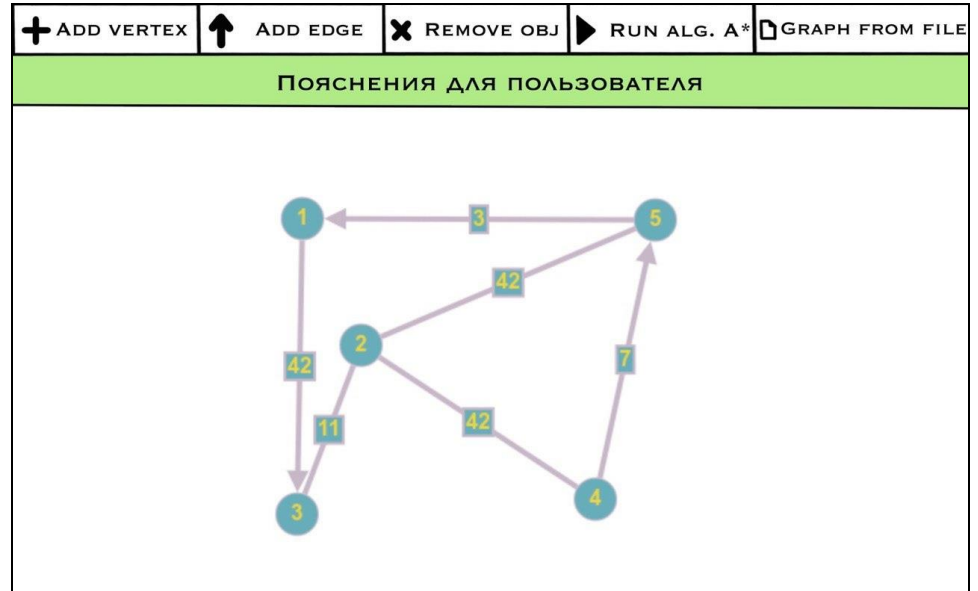


Рисунок 1 – Эскиз графического интерфейса

б. Приложение должно быть ясным и удобным для пользователя.

1.1.2 Исходные Требования к вводу исходных данных

Пользователь должен иметь возможность задать граф двумя способами: создавать/удалять вершины и ребра щелчком мыши или с помощью текстового файла (указывая матрицу весов и координаты вершин).

1.1.2. Формат выходных данных

- Визуализация процесса работы алгоритма (пройденные ребра, рассмотренные вершины с их эвристиками).
- Графическое отображение найденного кратчайшего пути, а также вывод его длины.

1.1.3 Исходные требования к визуализации алгоритма

- a. Помимо визуализации алгоритма, должны выводиться текстовые пояснения происходящего для пользователя.
- b. Визуализация алгоритма должна быть пошаговой, шаги не должны быть крупными, полное описание пошаговой визуализации алгоритма см. в разделе 3.4.
- c. Пользователь должен иметь возможность перейти к предыдущему или следующему шагу и поставить на паузу процесс выполнения алгоритма.
- d. Соблюдать единый стиль приложения.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

1. Изучение нового ЯП Java на платформе Stepik – **до 30 июня**;
2. Согласование спецификации и плана разработки – **30 июня**;
3. Разработка прототипа – **до 1 июля**:
 - a. Подготовка среды разработки, создание репозитория на GitHub;
 - b. Создание макета графического интерфейса без основных функций
 - c. Реализация ввода входных данных: через файл и через интеракцию с окном - и корректного их отображения в виде графа.
4. Утверждение / сдача прототипа – **1 июля**.
5. Разработка 1ой версии приложения – **до 6 июля**:
 - a. Реализация алгоритма A*
 - b. Обеспечение взаимодействия с пользователем и вывод результатов алгоритма – найденный кратчайший путь и процесс его нахождения
 - c. Обработка исключений
 - d. Создание тестов
6. Утверждение / сдача 1 версии приложения – **6 июля**
7. Разработка 2ой версии приложения – **до 8 июля**:
 - a. Добавление стилей в графический интерфейс, работа над внешним видом приложения
 - b. Доработка программы и тестов
8. Сдача / защита финальной версии – **8 июля**

2.2. Распределение ролей в бригаде

Написание отчёта – общие, основные пункты – Рубежова / UML-диаграмма – Афанасьев / в особенностях реализации – каждый описывает свою часть.

Разработка прототипа:

Крючков – Создание макета графического интерфейса без основных функций.

Афанасьев – Реализация ввода входных данных: через файл и через интеракцию с окном - и корректного их отображения в виде графа.

Рубежова – написание текста для стартового окна About, подготовка файла README.md.

Разработка 1ой версии приложения:

Рубежова - Реализация алгоритма A^* и создание тестов

Крючков – Обеспечение взаимодействия пользователя с интерфейсом, сохранение графа в файл, добавление возможности вывода результатов непрерывно по таймеру.

Афанасьев – Вывод результатов алгоритма – найденный кратчайший путь и процесс его нахождения по шагам, а также обработка исключений, вывод сообщений об ошибках.

Разработка 2ой версии приложения:

Командная работа – Добавление стилей в графический интерфейс, работа над внешним видом приложения, финальные исправления.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Представление графа

В качестве основной структуры данных, используемой в программе, выступает ориентированный взвешенный граф, который описан в классе *Graph*. Граф описывается двумя полями. Первое поле – словарь вершин с порядковым номером вершины в качестве ключа и позицией вершины на плоскости в качестве значения. Второе поле – матрица весов – словарь, где ключ – это начальная вершина ребра, а значение – коллекция пар, состоящих из конечной вершины и соответствующего ребру веса. Для построения графа реализованы методы для добавления в граф или удаления ребра или вершины. Помимо этого, были реализованы геттеры как для обоих словарей в целом, так и для позиции конкретной вершины или веса конкретного ребра, а также созданы методы для проверки графа на наличие определённых рёбер или вершин.

3.2. Реализация алгоритма

При реализации алгоритма A^* необходимо было учитывать тот факт, что для реализации этого алгоритма необходимо иметь доступ к промежуточным данным. В такой ситуации возможно два подхода: либо на каждом этапе алгоритма выводить изменения на экран, либо сохранять промежуточные данные и выводить их уже после выполнения алгоритма. Был выбран второй способ по двум основным причинам: во-первых, сохранение промежуточных результатов и их использование после завершения алгоритма позволяет просматривать шаги не только по прямому порядку их выполнения, но и в обратном порядке (возможность прокручивать шаги назад); во-вторых, данная реализация позволяет уже при начале визуализации понимать, успешно ли выполнится алгоритм или же нет. Выполнение и хранения результатов происходит в классе *AStar*. Выполнение самого алгоритма происходит в конструкторе при передаче начальной и конечной вершины, выбранной эвристики и самого графа. Всего имеется 4 варианта эвристики: расстояние

Чебышева, Манхэттенская метрика, Евклидово расстояние и нулевая эвристика (в этом случае алгоритм представляет из себя алгоритм Дейкстры).

Суть самого алгоритма – поиск минимального пути в графе при помощи оценок пути от конкретной точки до требуемой точки. Иными словами, при построении пути рассматриваются не все точки подряд, а те для которых эвристическая оценка пути минимальна на данном шаге. В результате выполнения алгоритма мы получаем кратчайший путь, который возможно и не полностью достоверный (так как используется эвристика), но который получен за более быстрое время. Хранение эвристических оценок происходит в мин-куче, что уменьшает время доступа к вершине с наименьшей эвристической оценкой.

На каждом шаге алгоритм запоминает эвристику и эвристическую оценку пути каждой рассмотренной вершины, а также ребро, по которому прошёл алгоритм на данном шаге. Также алгоритм содержит следующие поля: кол-во шагов работы алгоритма, стартовая и финальная вершины, список рёбер в полученном пути и его длина (под длинной пути подразумевается сумма весов рёбер, из которых он состоит). Для каждого поля реализованы соответствующие геттеры.

3.3. Графический интерфейс

Графический интерфейс реализован с помощью библиотеки *JavaFX*. В начале программа загружает макет формата *.fxml*, затем функционал дополняется специальным контроллером, реализованным в классе *MainViewController*, содержащем обработчики событий окна. В результате пользователь получает рабочий графический интерфейс (см. рис 2). Всё рабочее окно можно логически разделить на три части: панель инструментов, полотно для графа и панель для вывода сообщений.

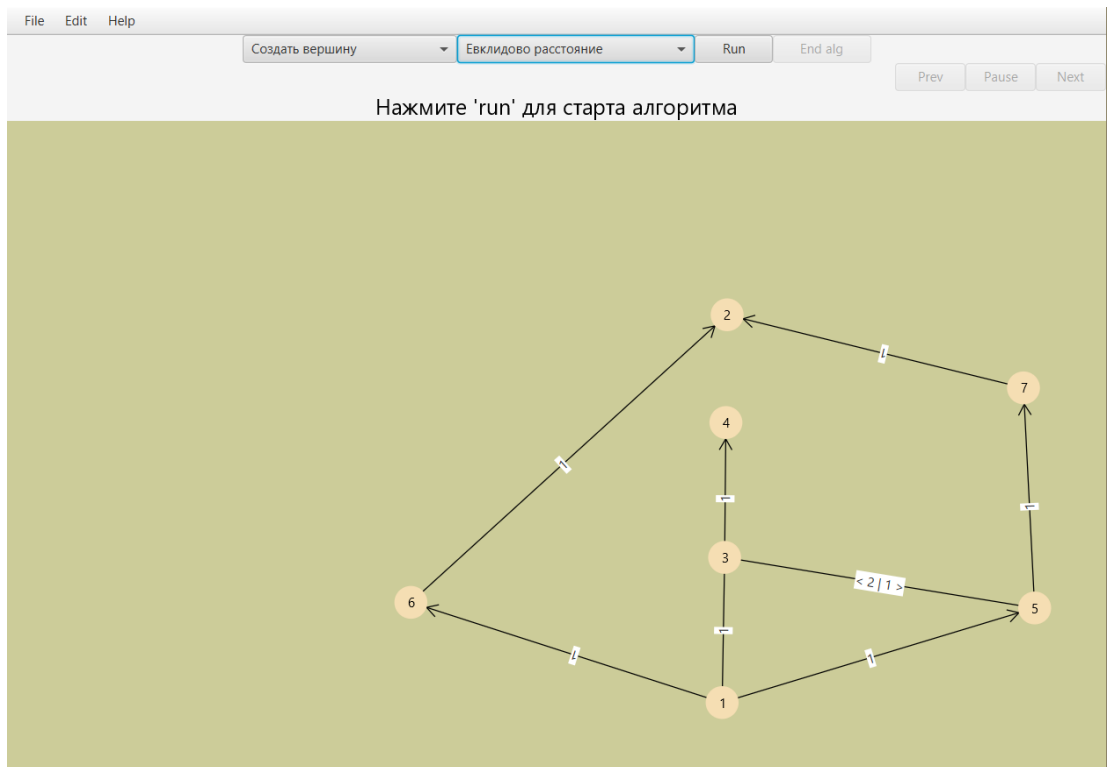


Рисунок 2 – Графический интерфейс пользователя.

Панель инструментов включает себя возможность выбрать действие над полотном (создать вершину, соединить вершины, удалить вершину/ребро), выбрать эвристику, загрузить граф из файла или сохранить его, начать визуализацию алгоритма. При этом некоторые клавиши будут недоступны, пока не будут выполнены соответствующие для них требования. Например, кнопка запуска визуализации не будет доступна, пока не будет выбрана эвристика. Также существует возможность открыть окно с подробным описанием алгоритма.

Панель с сообщениями выводит подсказки для пользователя касательно интерфейса и хода работы, а также информацию о выполнении алгоритма, после начала его визуализации. Методы для вывода сообщений на экран находятся в статическом классе *Message*. Статичность методов обусловлена тем, что возможность послать сообщение необходима из разных уровней визуализации интерфейса.

Работа с полотном реализована в классе *Canvas*, который хранит граф и позволяет интерактивно его изменять. Если в классе графа существуют методы

для добавления в модель рёбер и вершин, то в классе полотна реализованы методы для добавления рёбер и вершин на экран и удаления их оттуда. После каждого изменения происходит отрисовка полотна с начала (для этого созданы отдельные методы) согласно хранящемуся в памяти графу, чтобы избежать возможных ошибок в согласовании графа в памяти и графа на экране. Рёбра и вершины на полотне имеют более сложную структуру, нежели в модели, (ввиду особенностей графического представления), поэтому для каждого из них реализован свой класс – *Edge* и *Node*. Так как класс непосредственно связан с отрисовкой графа, в нём также реализованы методы для чтения и вывода на экран графа из файла, а также сохранения графа на экране в файл. При чтении графа из файла координаты вершин масштабируются, чтобы граф поместился на полотно. Также класс содержит информацию для визуализации результатов алгоритма: рёбра, которые необходимо покрасить, вершины, которые нужно пометить и т.д. – подробнее это будет описано далее.

3.4 Визуализация алгоритма

При запуске алгоритма пользователем, сначала полностью выполняется алгоритм A^* , результаты которого затем передаются в отдельный класс – *GraphVisuals*, который содержит сведения о том, какую информацию необходимо выводить на экран (на полотно) на конкретном шаге. Более конкретно, выводятся следующие промежуточные данные (см. рис. 3): непосредственно сам граф; рёбра, пройденные к моменту данного шага (выделяются зелёным); информация об эвристиках рассмотренных вершин (в виде текста рядом с вершинами) и на последнем шаге найденный путь (выделяется жёлтым).

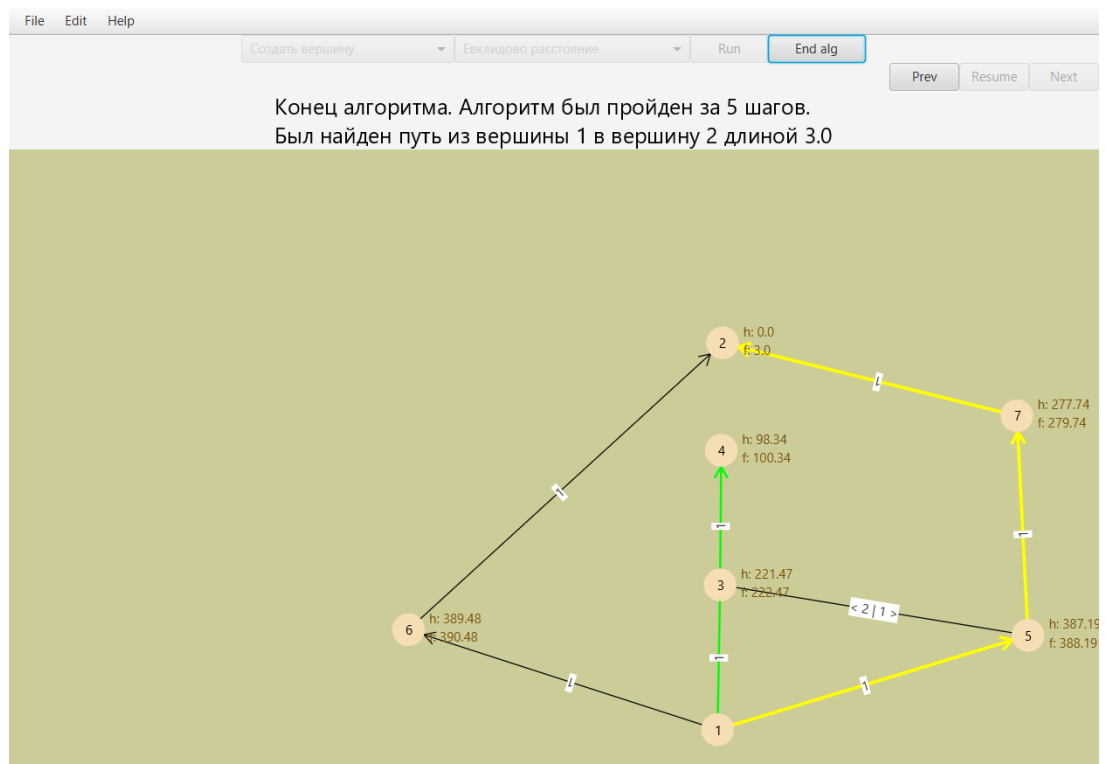


Рисунок 3 – Визуализация алгоритма

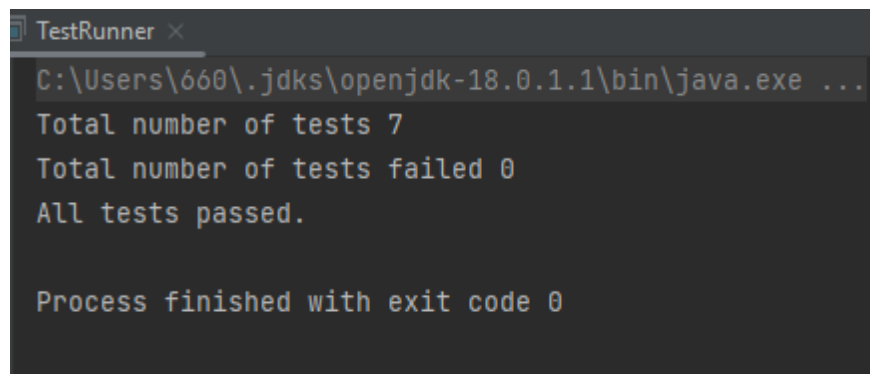
Возможны два режима просмотра алгоритма: переход к следующему шагу происходит автоматически со временем или переход вперед/назад происходит по интеракции пользователя с окном. Для первого режима был добавлен асинхронный процесс с таймером, который вызывает смену шага каждую секунды после начала визуализации алгоритма. Для второго режима в интерфейсе были реализованы кнопки для остановки алгоритма и пошагово перехода вперед и назад.

Полную *UML*-диаграмму классов и взаимосвязей см. в приложении А.

4. ТЕСТИРОВАНИЕ

В проекте было реализовано модульное тестирование с использованием библиотеки JUnit.

Тестирование проводилось для работы алгоритма A^* и для структуры представления графа. Для каждого случая был реализован соответствующий тест-класс, наследованный от *TestCase*, со своими тестовыми методами. Для запуска всех тестов был реализован класс *TestRunner*, который с помощью метода *runClasses()* запускает тестовый набор, составленный из предоставленных классов-наследников *TestCase*.



```
TestRunner x
C:\Users\660\.jdk\openjdk-18.0.1.1\bin\java.exe ...
Total number of tests 7
Total number of tests failed 0
All tests passed.

Process finished with exit code 0
```

Рисунок 4 – Вывод тестов класса *TestRunner*.

4.1. Тестирование структуры представления графа

Для тестирования структуры представления графа был реализован тест-класс *GraphTest*, наследованный от *TestCase*, который с помощью аннотированных(*@Test*) методов *testEdgesInfo()* и *testVerticeInfo()* проверяет, правильно ли обрабатываются и создаются графы, заданные в файлах-тестах.

Перед запуском всех тестовых методов вызывается аннотированный *@BeforeAll* метод *readGraphs()*, который считывает графы из файлов-тестов, создает из этих графов *ArrayList* для дальнейшей обработки и сравнения результатов и запоминает этот список в static переменную *graphList*.

Метод *testEdgesInfo()* проверяет для каждого тестового графа значение поля *edgesInfo* после считывания и обработки, т.е. правильно ли создались ребра графа. Для сравнения ожидаемого результата с фактическим результатом теста используется *Assertions.assertEquals()*. В случае, если результаты не совпадают, генерируется исключение *AssertionFailedError*.

Аналогично, метод *testVerticesInfo()* проверяет для каждого тестового графа значение поля *verticesInfo* после считывания и обработки, т.е. правильно ли создались вершины графа.

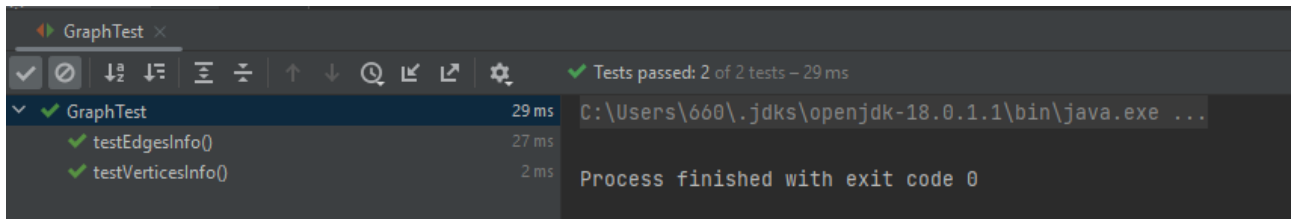


Рисунок 5 – Вывод тестов класса *GraphTest*. Без ошибки.

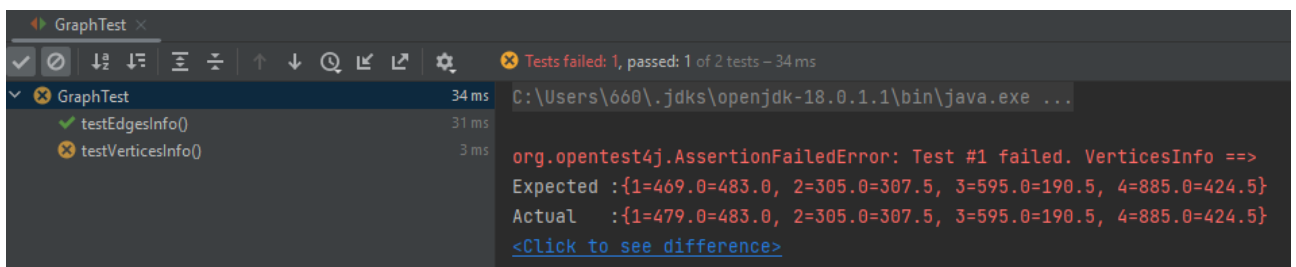


Рисунок 6 – Вывод тестов класса *GraphTest*. В случае ошибки.

4.2. Тестирование кода алгоритма A^* .

Для тестирования работы алгоритма A^* был реализован тест-класс *AStarTest*, наследованный от *TestCase*, который с помощью аннотированных(*@Test*) методов *testEdgesSteps()*, *testFinalPath()*, *testCountSteps()*, *testPathLen()*, *testHeuristics()* соответственно проверяет, корректно ли:

- запоминаются промежуточные ребра, по которым «ходил» алгоритм;
- находится итоговый кратчайший путь;
- считается количество шагов, за которое кратчайший путь был найден;
- вычисляется длина кратчайшего пути;
- вычисляются эвристики смежных вершин на каждом шаге

в процессе выполнения алгоритма A^* к графам, заданным в файлах-тестах.

Перед запуском всех тестовых методов вызывается аннотированный *@BeforeAll* метод *readGraphs()*, который считывает графы из файлов-тестов, создает из этих графов *ArrayList* для дальнейшей обработки и сравнения результатов и запоминает этот список в static переменную *graphList*. Также создается *ArrayList <AStar> resList*, который хранит сгенерированные

конструктором объекты *AStar*, результаты выполнения алгоритма по каждому графу из *graphlist*.

Метод *testEdgesSteps()* проверяет для каждого тестового графа значение поля *edgesSteps* объекта *AStar* после выполнения алгоритма, т.е. правильно ли запомнились промежуточные ребра, по которым «ходил» алгоритм. Для сравнения ожидаемого результата с фактическим результатом теста используется *Assertions.assertEquals()*. В случае, если результаты не совпадают, генерируется исключение *AssertionFailedError*.

Метод *testFinalPath()* проверяет для каждого тестового графа значение поля *FinalPath* объекта *AStar* после выполнения алгоритма, т.е. правильно ли найден итоговый кратчайший путь. Ожидаемые результаты сравниваются с полученными с помощью *Assertions.assertEquals()*.

Метод *testCountSteps()* проверяет для каждого тестового графа значение поля *CountSteps* объекта *AStar* после выполнения алгоритма, т.е. правильно ли подсчитано количество шагов, за которое кратчайший путь был найден. Ожидаемые результаты сравниваются с полученными с помощью *Assertions.assertEquals()*.

Метод *testPathLen()* проверяет для каждого тестового графа значение поля *PathLen* объекта *AStar* после выполнения алгоритма, т.е. правильно ли вычислена длина кратчайшего пути. Ожидаемые результаты сравниваются с полученными с помощью *Assertions.assertEquals()*.

Метод *testHeuristics()* проверяет для каждого тестового графа значение поля *PathLen* объекта *AStar* после выполнения алгоритма, т.е. правильно ли вычислены эвристики смежных вершин на каждом шаге. Ожидаемые результаты сравниваются с полученными с помощью *Assertions.assertEquals()*.

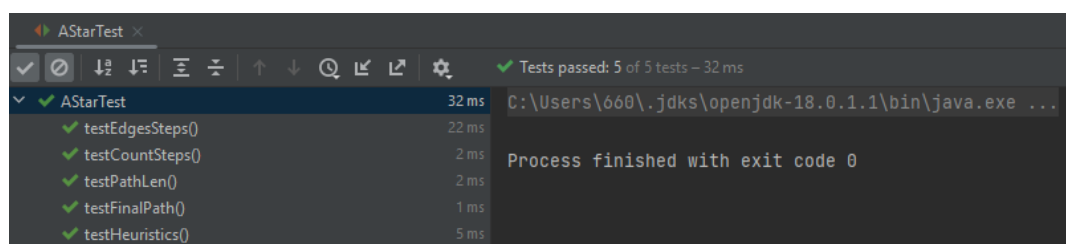


Рисунок 7 – Вывод тестов класса *AStarTest*. Без ошибки.

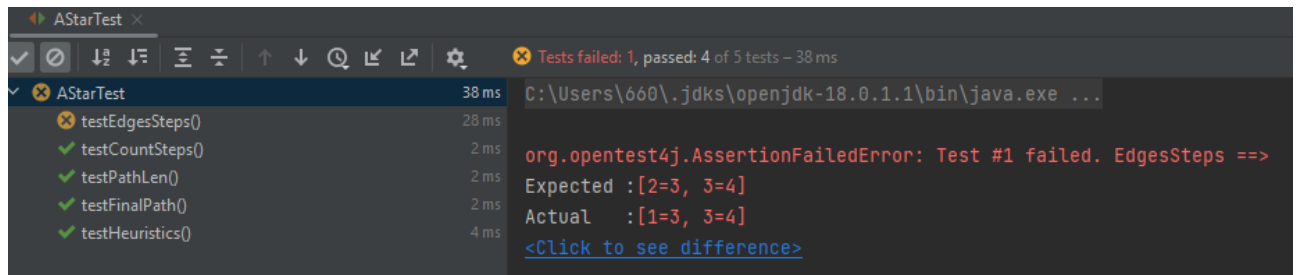


Рисунок 8 – Вывод тестов класса *AStarTest*. В случае ошибки.

ЗАКЛЮЧЕНИЕ

В ходе практики были изучены основные разделы языка программирования — *Java*, а именно: базовый синтаксис, объекты, классы и пакеты, обработка ошибок, ввод-вывод в файл.

Главной задачей практики была разработка графического интерфейса для наглядного демонстрирования работы алгоритма A^* . Для этого был итеративно разработан визуализатор алгоритма A^* на основе языка *Java*, с использованием библиотеки *JavaFX*. Для ускорения времени разработки графического интерфейса был использован *FXML* файл.

Для совместной работы была использована система контроля версий *git*, что позволило каждому участнику наиболее быстро работать над своей частью программы. Реализованный визуализатор позволяет в полной мере решать поставленную задачу, обладая при этом удобным и понятным пользователю графическим интерфейсом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация к JavaFX - FXML. URL:
https://openjfx.io/javadoc/18/javafx.fxml/javafx/fxml/doc-files/introduction_to_fxml.html (дата обращения: 01.06.2022).
2. Документация к JavaFX - запуск приложения при помощи maven. URL:
<https://openjfx.io/openjfx-docs/> (дата обращения: 05.06.2022).
3. Документация к Junit. URL:<https://junit.org/junit5/docs/current/user-guide/>(дата обращения: 05.06.2022).
4. Описание алгоритма A*. URL:
https://en.wikipedia.org/wiki/A*_search_algorithm

ПРИЛОЖЕНИЕ А

UML-ДИАГРАММА

