Bulgarian Diploma Thesis

Implementation of the A* and the Genetic Algorithm in Path Searching through a Maze

Имплементация на A* и генетичен алгоритъм за търсене на път в лабиринт

Radostina Kisleva, 200035409

Student:  **Date: 20 November 2020**

Supervisor: John Galletly , Date: 20 November 2020

Department of Computer Science, AUBG

Blagoevgrad, 2020

Title: Implementation of the A* and the Genetic Algorithm in Path Searching through a Maze

Author: Radostina Kisleva

Abstract: The project consists of the implementation of two searching algorithms – the A* and the genetic algorithm applied to the pathfinding problem. Pathfinding is the process of obtaining a path from a defined start point to a target point. The searching environment which the algorithms will traverse is a static 2d maze with a pre-defined start and target points. The performance of the algorithms is analyzed and compared. The program is written the Java programming language.

Declaration of authorship:

"The Senior Project/Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of John Galletly. All sources, used in development, are cited in the text and in the Reference section."

Author: Radostina Kisleva

Table of Contents

I. Introduction

1.1 Project topic

The problem of path planning or pathfinding is one of the most researched in the field of artificial intelligence and robotics. Generally speaking, it is described as finding an optimal (or near optimal) collision-free path between a starting point and a target point usually in an environment with obstacles. The affecting factors of path planning are the different kinds of environments and obstacles which can be static or dynamic. In a static environment, the area is known while in a dynamic one the obstacles are moving. Finding the shortest path with a good degree of smoothness which means a path with a low degree of direction change which also avoids collision with other robots and obstacles remains a challenging and complex problem. Path planning is used in a broad range of robotic applications in the industry, medicine and agriculture [1].

Further, pathfinding finds broad application in areas such as transit planning, telephone traffic routing, maze navigation and robot path planning. One industry which is particularly involved with path finding problems is the gaming industry. Path planning finds a broad application in solving the problem of avoiding obstacles in an environment and finding an efficient path on different terrains in games. It is a frustrating problem because traditional searching algorithms soon become overwhelmed by the exponential complexity of the game and more efficient solutions are needed [2].

There are different ways in which researchers have approached the pathfinding problem. The most popular ones include using depth-first search, iterative deepening, breadth first search, Dijkstra's algorithm and different versions of the A* algorithm [2].

Path planning problems eventually come down to graph searching. Generally, there are two main categories of graph searching algorithms which are blind search and heuristic search. Blind search algorithms, whose most typical representative is Dijkstra's algorithm, are considered very inefficient, with a high space-time complexity. On the other hand, heuristic methods use heuristic information to choose the most optimal node to expand from a set which improves significantly the search efficiency. They use a heuristic function to conduct the search and the quality of the heuristic is the determinant of the time required to find the optimal path [3].

1.2 Project overview

In my project, I have attempted to solve the pathfinding problem in a static, maze-like environment using two heuristic searching algorithms, namely the A* algorithm and the genetic algorithm. The selection of what algorithms to use was a difficult task but I decided to go for complex ones because I wanted to obtain good results from the search and display, if not the most optimal, at least one of the near optimal paths. Both algorithms are informed search algorithms. They make the search area smaller and focus on searching this area instead of the whole space.

The A* algorithm has been extensively researched and applied to the pathfinding problem because of its numerous advantages such as a simple principle, easy

realisation, efficiency and ability to find the optimal solution [4]. It is one of the most well-known and used search algorithms which finds application in games and robotics. It was the first algorithm which made use of a heuristic function to search a graph in a best-first manner. A* inspired many altered and improved algorithms [11].

The genetic algorithm is a metaheuristic searching algorithm which has found various applications due to its ability to solve complex problems. Genetic algorithms have been applied successfully to numerous path planning problems because of their ability to search the space of all possible paths and provide the most optimal one [5].

The project can be divided into two parts which are closely intertwined. One is concerned with the implementation of the searching algorithms while the other consists of the logic behind their visualization and the graphical user interface. The algorithms will traverse a 2D static maze which is represented as a square grid with pre-determined starting and target points. Besides the starting point and the destination, the maze will consist of untraversable squares, called “walls” which should be avoided during the search and traversable blank spaces or “gaps”. The user will have the ability to choose the size of the maze in which the algorithms will run and maze variant according to the chosen size. Upon clicking the mouse and triggering the search, the path obtained by the algorithms will be visualised and the user will be able to see what decision has been taken and what nodes have been traversed by the algorithms to find the path to the target in the maze. In the A* algorithm section of the application, the visited nodes as well as the nodes on the optimal path will be displayed in the maze in a logical order. The genetic algorithm will be visualized by consecutively displaying each path in the generation with its fitness value and controlling parameters until the target node is found. In the end, the time for which the search was performed and the number of nodes of the final path will be displayed for each of the algorithms which allows for comparison of their performance.

1.3 Technologies used

The technologies which I have used are the following:

- Java
- IntelliJ
- Java Swing
- Java Abstract Window Toolkit
- JUnit

For the development of the project, I have decided to use the Java programming language. I have not used any third-party libraries or frameworks. Some of the reasons for choosing Java for the development of the project include its portability on different operating systems and the fact that you can run it anywhere anytime. Moreover, considering the fact that visualization is a crucial part of the problem that I am solving, I needed a programming language appropriate for GUI programming. Java offers built-in graphics tools for 2D applications like mine which are simple and user-friendly.

When it comes to the integrated development environment, there are several popular choices when it comes to programming in Java. I decided to use IntelliJ because of

the convenience it offers when it comes to debugging, autocompletion and refactoring. I like its design, the ease and pace of working with it.

Regarding the visualization part of the project, I have kept the user interface simple and minimalistic because the main focus is on the implementation of the algorithms. I have used Java Swing to create the graphical user interface. I have also used Swift's predecessor – Java AWT (Abstract Window Toolkit) which is a platform dependent API for creating Graphical User Interface for Java programs.

In order to test the application and make sure that it works correctly, I had to write unit tests. I used one of the most popular testing frameworks available for Java which is JUnit and more specifically JUnit 5 which is the next generation of JUnit. I used the JUnit Jupiter module which includes new programming and extension modules for writing tests in JUnit 5. More information about the technologies that I have mentioned can be found in section 4.1.

1.4 Interest in the topic

I chose this project because I wanted to challenge myself and learn more about complex searching algorithms and their implementation. It was interesting to tackle one of the most challenging problems in designing realistic Artificial Intelligence in computer games and apply AI search techniques to a real-world problem. The part of this project that was particularly interesting to me was the implementation the genetic algorithm. The genetic algorithm is a part of a group of algorithms called evolutionary algorithms. I find it fascinating that we can take the concept of something that exists in nature such as genetics, apply it to technology and solve complex problems. Another fascinating idea is how the operators of the genetic algorithm which work with a high degree of randomness can solve very complex problems. I also wanted to compare how one and the same problem is solved by two different algorithms, what path each of them chooses and how their performances compare to one another.

II. Specification of the software requirements and their analysis

2.1 Functional requirements and their analysis

2.1.1 List of functional requirements

The functional requirements describe what actions the developed software must support and what the application is supposed to accomplish for the user. They capture the intended behavior of the system. The following are the functional requirements which I have identified for my application:

1. A user should be able to choose the size of the maze.
2. A user should be able to choose a maze variant.
3. A user should be able to reset the maze.
4. A user should be able to run the A* algorithm and see the path obtained in chosen maze.
5. A user should be able to run the genetic algorithm and see the path obtained in chosen maze

2.1.2 Activity diagram

The activity diagram below depicts the flow of activity or the sequencing and coordination in the program. This is the top-level activity diagram for the whole application. First, the program accepts a valid maze based on a choice from the user. Then, it runs either the A* or the genetic algorithm, again based on the preferences of the user. After that, it should produce and display the obtained path in the maze.

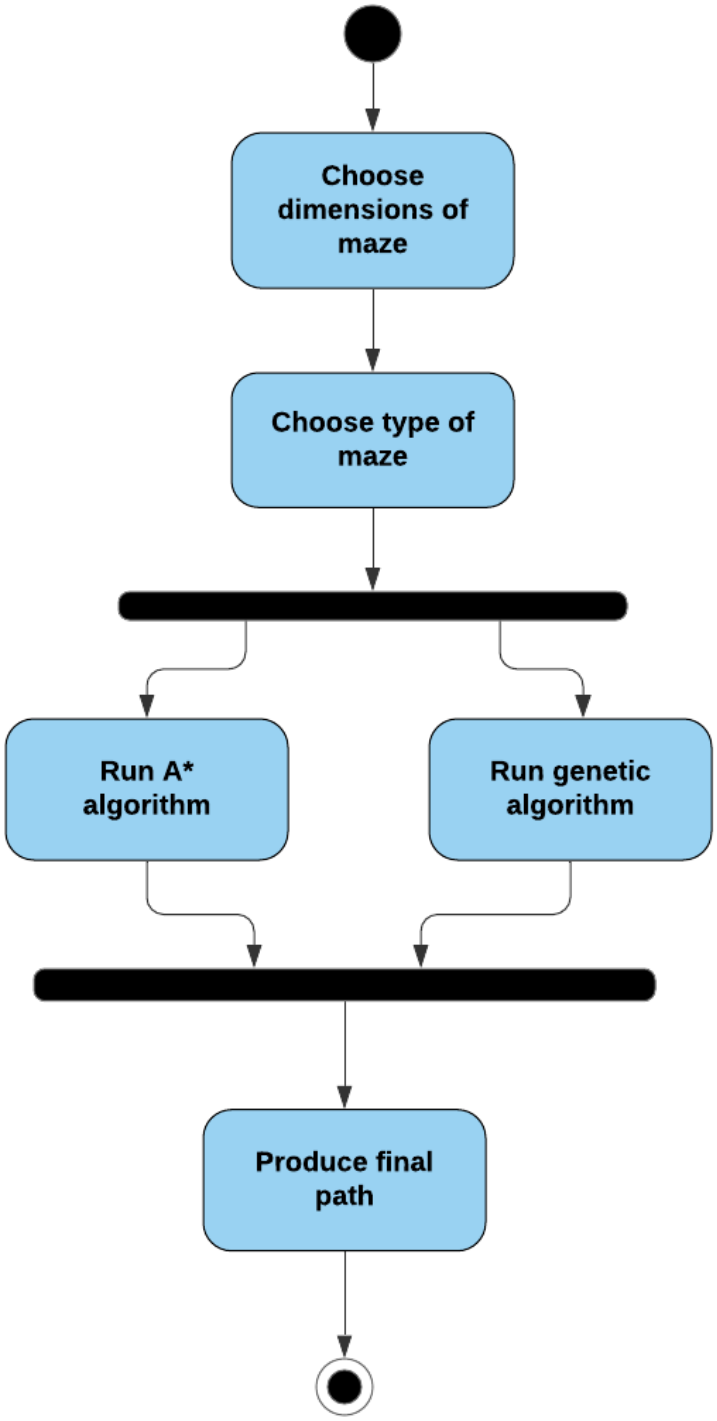


Figure 1 Top-level activity diagram

2.1.3 Analysis of the functional requirements

In the following pages each functional requirement is broken down into a more detailed description. In order to accomplish this task, I have used use case narratives and diagrams which define a set of interactions between an actor and the system in order to achieve a goal. Each use case is initiated by a user which has a particular goal in mind and is considered successful if the said goal is accomplished. Each use case narrative includes the goal in mind, the actor, any preconditions, triggering actions, the flow of events, exceptions that may occur and the post-condition.

- *A user should be able to choose the size of the maze.*

In order to visualize how the algorithm works, an appropriate environment should be chosen first. In my program, the algorithms search for a path in a maze. The user should have the freedom to choose what type of maze will be used by the algorithm. The first step is to choose the size of this maze. The user will be given a drop-down menu through which they can choose the size of the maze in which they want to visualize the algorithm. There will be several available variations for maze sizes such as 15x15, 20x20, 25x25, etc. These numbers represent the number of squares in each row and each column of the maze. Choosing the size is the first part of a two-step sequence which results in returning an appropriate two-dimensional array which represents the chosen maze. Once the user chooses the size, the grid in the frame is updated and changes its size and appearance according to the chosen numbers. The user can see the size of the maze visually and proceed with the next step.

Use case narrative:

Use case name	Choose maze size
Goal in mind	User chooses a set of numbers and the system returns a maze with this size which the algorithm will traverse.
Actor	User
Conditions	An option is chosen from the particular drop-down menu.
Trigger	The system performs the action automatically when the user clicks on an option.
Scenario (typical flow of events)	1. User chooses a set of numbers from the drop-down menu. 2. System updates the size of the grid in the frame. 3. System displays the maze in the frame.
Exceptions	No exceptions can occur at this point. If the user doesn't choose size, there is a pre-selected default option.

Post-condition	A maze with chosen size is visualized in the frame.

Use case diagram:

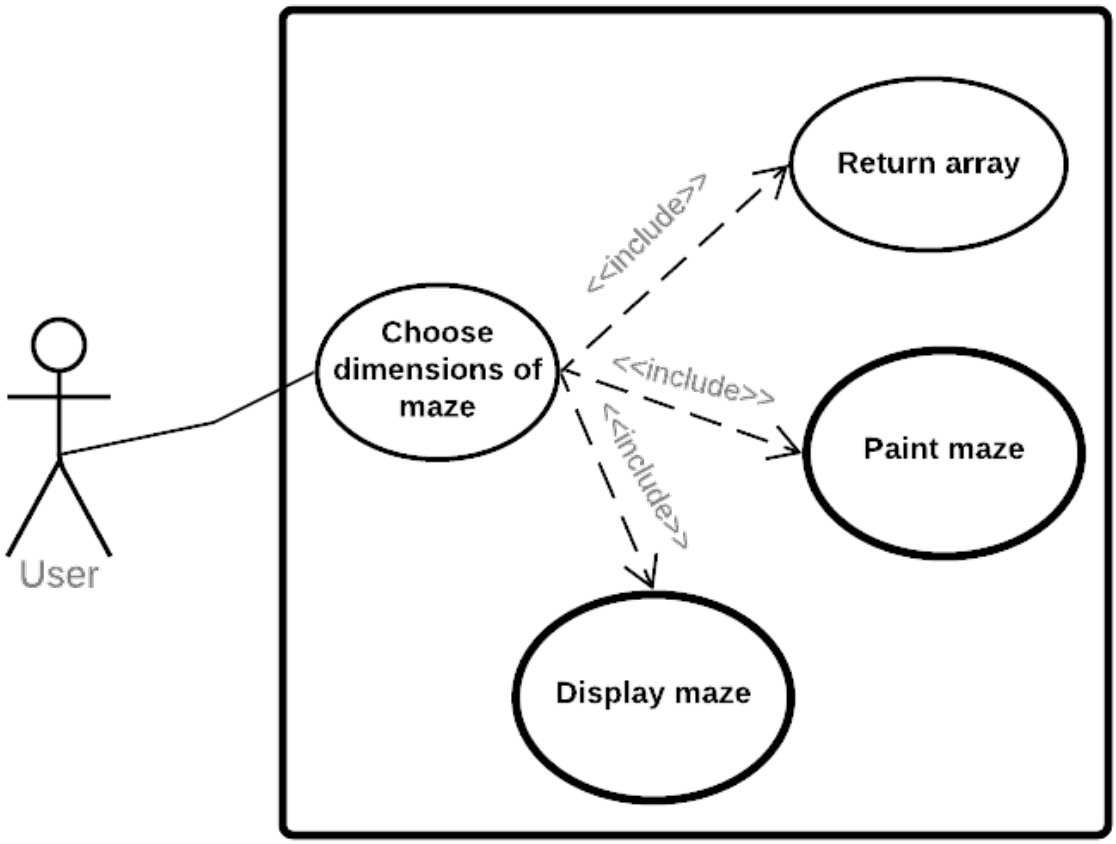


Figure 2 Use-case diagram for first requirement

- *A user should be able to choose a maze variant.*

After choosing the size, the user should be able to choose the appearance of the maze in which they want to see the searching take place. This process will be handled by another drop-down menu through which the user can choose a particular version of the maze. The different versions differ in terms of density of the obstacles and may also differ in the terms of the location of the starting and the target points. When the user chooses an option and clicks a button, the appropriate two-dimensional array which represent the maze is found and returned by the program. The different values in this two-dimensional array are what is used in order to construct the maze. Lastly, the maze is displayed in the frame.

Use case narrative:

Use case name	Choose maze variant
Goal in mind	User chooses a variant of the maze based on their preferences with different obstacles and

	complexity which they want the algorithms to traverse
Actor	User
Conditions	1. A size is selected, or default size is used. 2. A maze variant is selected from a drop-down menu.
Trigger	The “Generate maze” button should be clicked.
Scenario (typical flow of events)	1. User chooses one of the variants of the maze from a drop-down menu. 2. User clicks “Generate maze button”. 3. System finds the appropriate two-dimensional array which represents this choice. 4. System paints the maze based on this two-dimensional array. 5. System displays the maze in the frame.
Exceptions	No exceptions can occur at this point. If the user doesn’t choose a maze variant, there is a pre-selected default option.
Post-condition	The chosen variant is visualized in the frame.

Use case diagram:

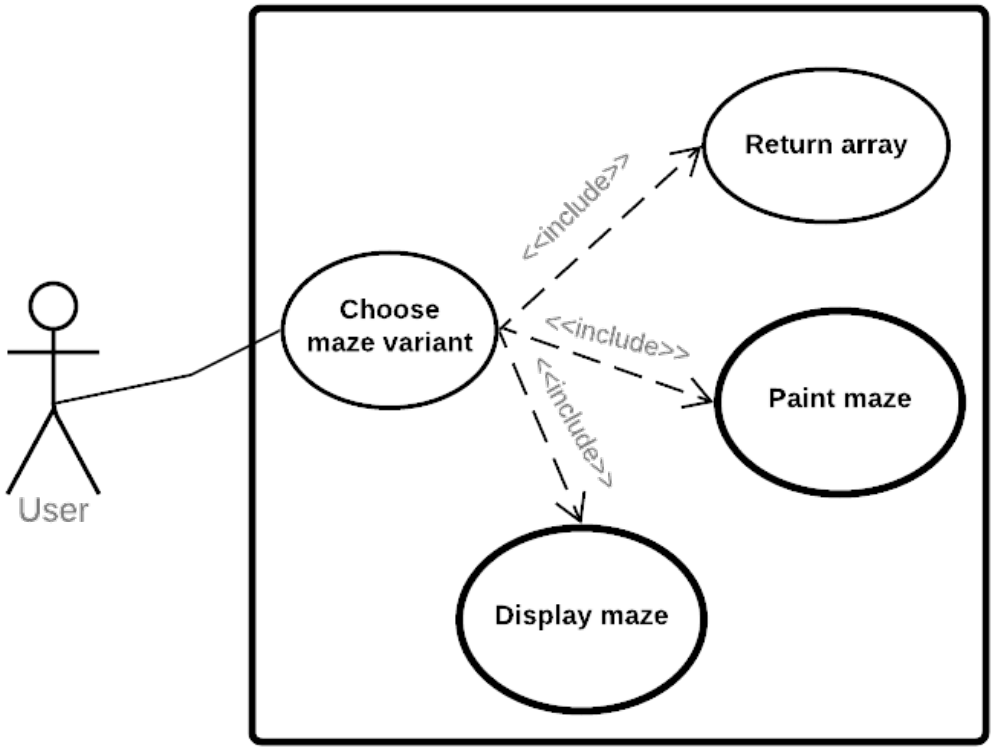


Figure 3 Use-case diagram for second requirement

- *A user should be able to reset the maze.*

After the user has chosen particular maze size and variant, if they wish to, they should be able to reset the maze which will result in reverting the choice back to the default state and giving the user the ability to start choosing a new maze.

Use case narrative:

Use case name	Reset maze
Goal in mind	User wants to choose a new maze.
Actor	User
Conditions	The user has previously chosen a maze which now they wish to reset.
Trigger	The “reset maze” button is clicked.
Scenario (typical flow of events)	1. User presses the “reset maze” button. 2. System returns an empty maze. 3. System updates maze size if needed.
Exceptions	No exceptions can occur at this point.
Post-condition	An empty maze is displayed in the frame.

Use case diagram:

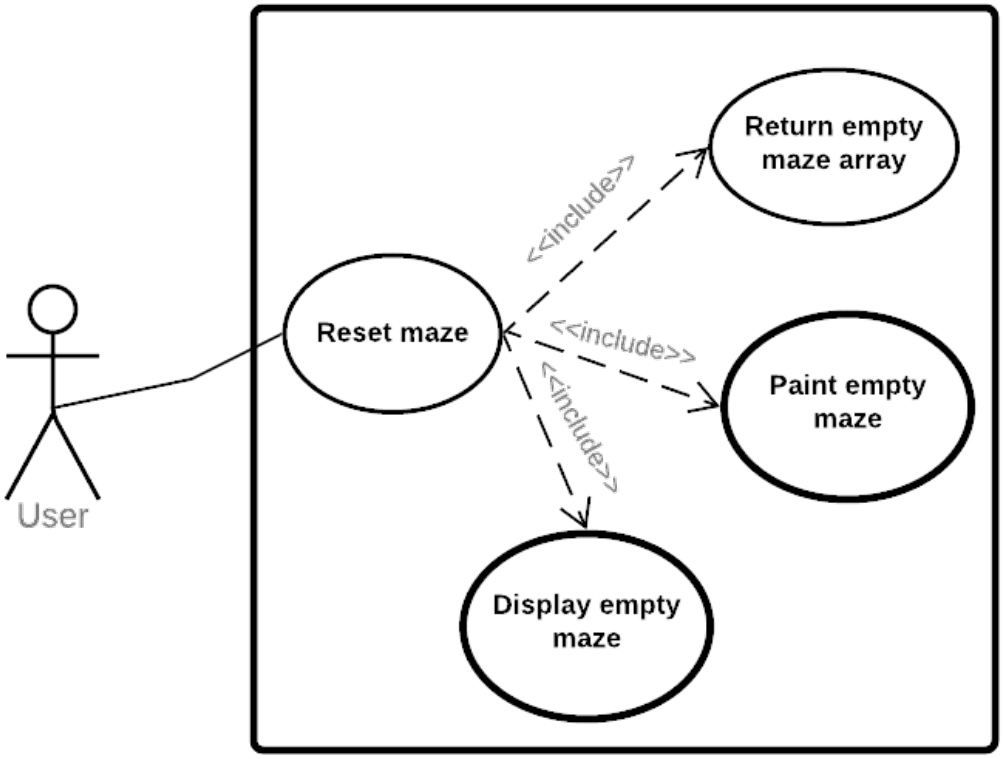


Figure 4 Use-case diagram for third requirement

- *A user should be able to run the A* algorithm and see the path obtained in chosen maze.*

The main objective of the whole application is having a working implementation of the two algorithms, one of which is the A*. After the program has processed the preferences of the user about the maze, it should be able to run the algorithm in the chosen maze. The algorithms themselves are not concerned with the drawings that the user sees but rather go back to the two-dimensional array that represents the chosen maze and turn that array into a graph with nodes to be traversed. These nodes have several attributes like an index, a list of neighbors and a value which indicates if they have been visited or not. Each node also has a heuristic value which is used by the algorithm when it makes the decision of which node to expand to get to the target. This heuristic value is the Euclidean distance which is the straight-line distance from the current node to the target. The walls are made visited to indicate that the algorithm cannot go there. After the graph is built and the heuristic values are set, here is a rather simplified version of how my implementation of the A* algorithm works in pseudocode:

```
let openList = empty list of nodes
let closedList = empty list of nodes
add startNode to the openList

while openList is not empty
  let currentNode = node with least heuristic value
  remove currentNode from openList
  add currentNode to closedList

 if current node == goal
   backtrack and find the parents of each node to obtain the path

 get neighbors(children) of currentNode
 for each neighbor in neighbors
   if neighbor is in closedList
     continue to beginning of for loop

   update heuristics of neighbors
   neighbor.g = currentNode.g + distance between child & current
   neighbor.h = distance from child to end
   neighbour.f = neighbor.g + neighbor.h

   if neighbor is in openList
      if neighbour.g > distance from start to neighbor through current
         continue to beginning of for loop
   add neighbour to openList
```

After the algorithm has run, we are left with two list – one which holds all of the visited nodes and another which keeps tracks of the nodes which comprise the final path.

The second part of the requirement is to show the path obtained by the algorithm in the chosen maze. First, the maze board is drawn again. The program goes through

the visited nodes obtained from the algorithm and paints the corresponding squares in the maze in purple. The shade of purple depends on the heuristic value of that node. The nodes closest to the target have the darkest shade of purple. After the visited nodes are displayed, the program uses the second list with the nodes which construct the optimal path and paints the corresponding squares in the maze in yellow. The final result includes visualization of the visited nodes, visualization of the final path from the start to the goal, the number of the visited nodes, the number of the nodes of the final path and the time for which the path was found.

Use case narrative:

Use case name	Run the A* algorithm
Goal in mind	User wants to see the path that the A* algorithm obtains in the chosen maze.
Actor	User
Conditions	A maze must be chosen before the algorithm runs. The user must have chosen the “A* algorithm” option from the drop-down menu.
Trigger	The “Start Search” button is pressed.
Scenario (typical flow of events)	1. User chooses the A* option from the drop-down menu. 2. User presses “Start Search” button. 3. System gets the chosen maze from previous step. 4. System runs the A* algorithm using the chosen maze. 5. System displays the visited nodes by painting the squares of the maze corresponding to them in purple. 6. System displays the nodes of the final path by painting the squares of the maze corresponding to them in yellow. 7. System returns the time for which the target was found, the number of visited nodes and the number of nodes of the final path.
Exceptions	A maze must be provided before the algorithm runs. The two-dimensional array which represents the maze must be in the correct format. For example, each row and column should have the same number of values. Also, the target node should be reachable and there must be a start and a target indicated.

Post-condition	The final result includes: - the chosen maze with obstacles - the visited nodes painted in purple - the path from the start to the target painted in yellow - the time for which the target was found - the number of visited nodes - the number of final path nodes

Use case diagram:

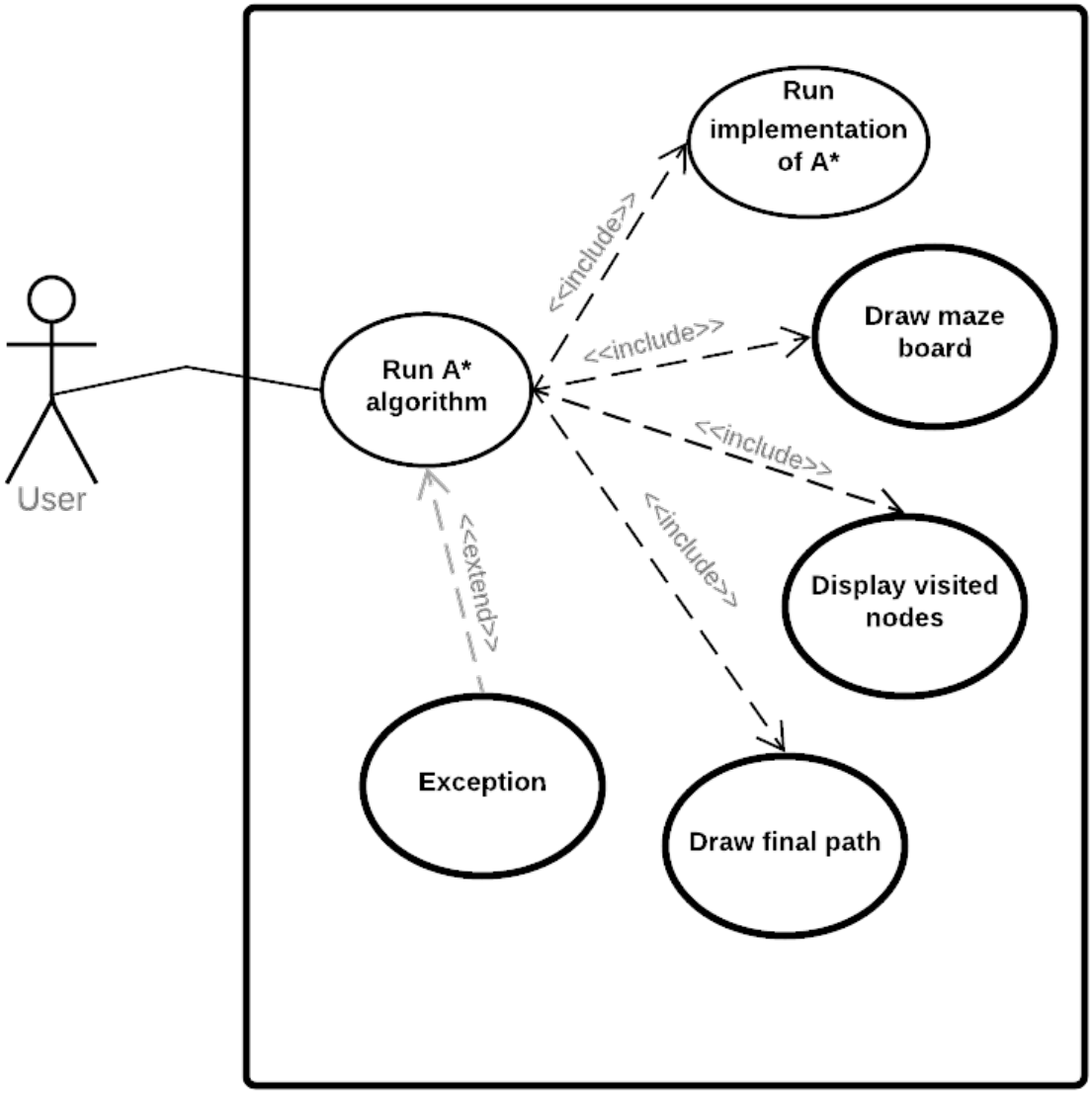


Figure 5 Use-case diagram for fourth requirement

- *A user should be able to run the genetic algorithm and see the path obtained in chosen maze.*

This is the second part of the main objective of the application – to have a working implementation of the genetic algorithm. Similar to the A*, the GA also needs a maze to traverse. After the user has chosen what kind of maze they want to use, the

procedure here is the same – a two-dimensional array which represents a maze is turned into a graph with nodes to be traversed. My implementation of the genetic algorithm uses four main operators which are selection, crossover, mutation and elitism. Since the algorithm is inspired by the evolutionary theory in genetics, such terminology is used. It works with controlling parameters which need to be supplied before the algorithm starts running. The performance of the algorithm depends on these parameters. It starts by creating an initial population of solutions to the problem. The different solutions are also called individuals or chromosomes. These chromosomes are paths in the maze. Each solution has a fitness value which evaluates how "fit" or how "good" it is with respect to the problem. The calculation of this fitness value is done repeatedly while the GA is running. The rest of the implementation of the algorithm in my program is described in the pseudocode below:

```
let initialPopulation = empty population;
initialPopulation.evaluateIndividuals();
while (the maximum limit of generations is not reached)
      newPopulation = empty population
      //keep the "fittest" individuals from the initial population
      //by moving them to the new one
      newPopulation = elitism(initialPopulation)
      //while the new population doesn't have the same number of
      //individuals as initial
      while (size(newPopulation)+eliteNumber != size(initialPopulation)
            //choose the best individuals and perform crossover
            parents = selectParents();
            offspring = crossover (parents.first, parents.second);
            if (randomNumber < mutationRate)
               offspring.mutation;

            //whether to add the offspring or a parent to the new pop
            if (randomNumber < probabilityCrossover)
               newPopulation.add(offspring);
            else
               newPopulation.add(oneOfParents);
            newPopulation = evaluateIndividuals();
            //override old population with new one
            initialPopulation = newPopulation;
```

After each iteration of the for loop, the program will output the number of the generation for which it is currently running and the fitness value of the fittest individual. It will display the current solution in the frame even though it might not be the path that reaches the target. The point is to see how the values of the algorithm change, how it progresses and changes with each generation in order to reach the target. If the path cannot be found in the current population, the algorithm runs again but with different values of the controlling parameters. This update is automatic. Using higher values of these parameters allows the algorithm to have better chances of finding the path to the target in the next iteration. The algorithm runs until, at the end, it reaches the target. When this happens the animation of the changing paths is stopped, and the final successful path remains in the frame so that it can be observed.

Use case narrative:

Use case name	Run the genetic algorithm
Goal in mind	User want to see the path that the genetic algorithm obtains in the chosen maze.
Actor	User
Conditions	A maze must be chosen before the algorithm runs. The user must have chosen the "Genetic algorithm" option from the drop-down menu.
Trigger	The "Start Search" button is pressed.
Scenario (typical flow of events)	1. User chooses the genetic algorithm option from the drop-down menu. 2. User presses "Start Search" button. 3. System gets the chosen maze from previous step. 4. System runs the genetic algorithm using the chosen maze. 5. For each iteration, system displays current number of the generation, the fitness value of the fittest individual and the path in the maze. 6. When the target is reached, system stops the animation and displays the final path.
Exceptions	A maze must be provided before the algorithm runs. The two-dimensional array which represents the maze must be in the correct format. For example, each row and column should have the same number of values. Also, the target node should be reachable and there must be a start and a target indicated. Appropriate values for the controlling parameter must be provided.
Post-condition	The final frame includes: - the chosen maze with the obstacles - the current generation number - the fitness value of the fittest individual - the current solution as a path drawn in the maze - a message whether the path is successfully found or not - the values of the controlling parameters - the time for which the target was found - the number of final path nodes

Use case diagram:

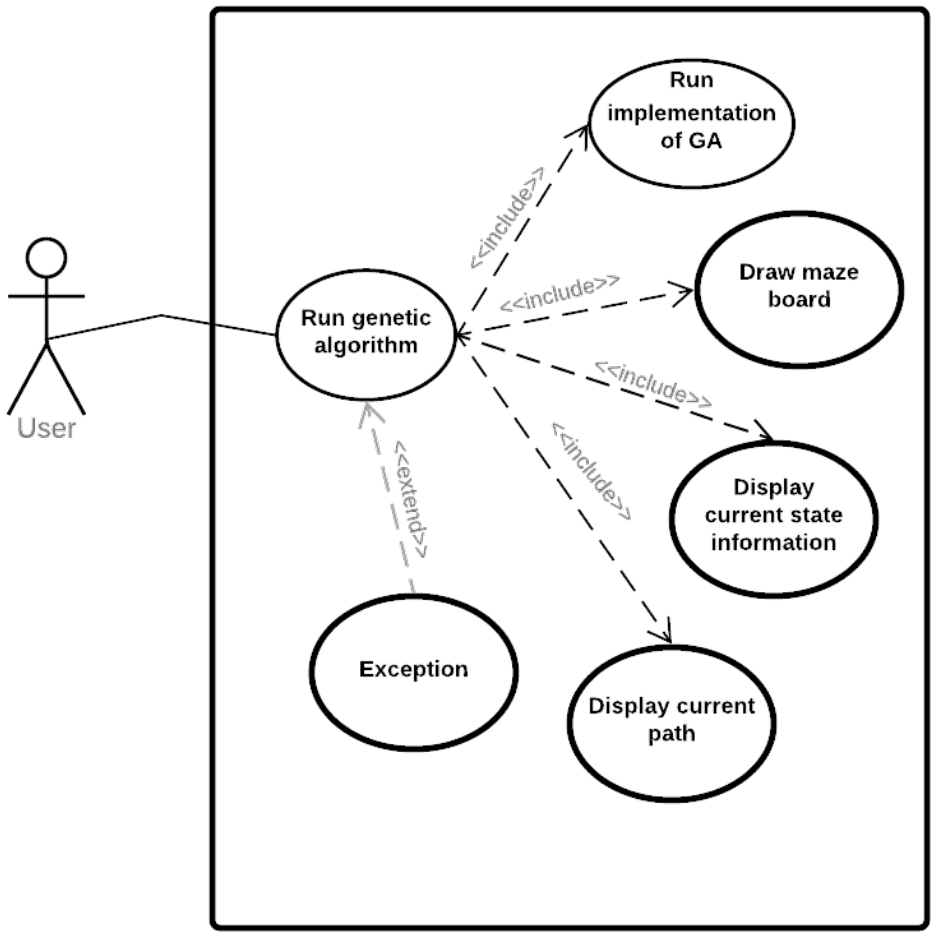


Figure 6 Use-case diagram for fifth requirement

2.2 Non-functional requirements

The non-functional requirements describe constraints related to the implementation of the application such as performance, security or reliability requirements. Below is a list of these requirements in my application and a short description of their meaning.

1. The application should be able to run on any system that has the Java Virtual Machine installed.

What is meant by this requirement is that the program should be platform-independent and be able to be run on any operating system. One very important feature of the Java programming language is its portability. This means that Java compiles to Java Bytecode which runs on the Java Virtual Machine. The JVM converts the byte code into something the operating system can read and run. Being written in Java, the application should be able to run on any operating system that has the JVM installed.

2. The system must be able to respond to a search request within 2 seconds.

The program must be able to start the search within 2 seconds after the user clicks the “Start Search” button. This requirement is concerned with how fast the system can accept a request and start processing it.

3. The structure of the code should follow good coding standards.

In the software development industry, there are certain coding standards that should be followed. Writing high-quality code is not easy but it is necessary in case the application will be extended in the future and further improved. The code should be easy to understand, consistent and properly commented. It should implement the separation of concerns principle and have a well-structured and organized software architecture. The different functionalities that exist within the code should be separated. There are many coding practices which can increase the maintainability of the code so that it can be easily modified at any point in time.

2.3 Constraints

Constraints are restrictions in the degree of freedom in providing a solution. The scope of the application is to successfully implement and apply the two searching algorithms to the problem of pathfinding and analyze their performance in terms of advantages and disadvantages. However, it should be noted that the final goal is not to find the shortest path to the target. While the path obtained by the algorithms will be a near optimal one and definitely better than using blind search techniques, it is not the shortest one. Also, in my application, I have allowed only four movement directions – up, down, left and right. Diagonal movements are not allowed. If they were, the paths obtained would have been shorter.

The fitness function which the genetic algorithm uses can be further improved so that it converges less often. It should also be noted that the performance of the GA depends on the size of the maze and on the values of the controlling parameters. The performance heavily depends on those numbers and while I have tried to the best of my ability to tune these parameters properly for the different sizes of the maze, in some specific cases it might take longer for the algorithm to reach the target.

Certainly, there is room for improvement in the implementation of these algorithms and there is an abundance of literature available on this topic where researchers have modified the algorithms or even merged them with others in order to enhance their performance and find more optimal paths. Unfortunately, the time scope of the senior project did not allow me to dive deeper into these techniques and apply some of them in practice.

III. Design of the software solution

3.1 Pathfinding plan of action

The way pathfinding algorithms find a solution is by searching through a search space. The essence of the search is picking an option and putting the others aside in case the first one does not lead to the goal. The algorithm continues picking, testing and expanding until a solution is found [13]. The algorithms used for pathfinding fall under the category of local search methods. They perceive the elements of the search space as nodes of a graph with edges and move from item to item along the edges. The nodes have particular components which include the state in the search space to which the node corresponds, the node that generated

this node (parent node), the count of nodes on the path from the root to current node (depth of the node) as well as the path cost from the root to the node [13].

Similar to other optimization problems, the pathfinding problem can be approached by using informed or uninformed search methods. Uninformed search, also called blind search, encompasses algorithms that do not use information about the path cost from the current node to the target. The expandability of these approaches is limited as their execution time expands exponentially with the size of the problem which in the case of pathfinding is depicted as the size of the environment model. The informed search methods provide more intelligent ways to explore the search space which accordingly reduces the execution time [14].

Informed search methods are called heuristic methods. A heuristic function is defined as a function that calculates the cost estimate of reaching the target from a particular current state. In other words, the node that is judged to be the closest to the target node is expanded first. An example of a good heuristic function applied to the pathfinding problem is the straight-line distance to the goal state [13]. The A* algorithm is an example a heuristic search method. There also exist metaheuristic methods which provide a higher-level operation to find, create or choose a heuristic. The genetic algorithm is an example of a population-based metaheuristic.

3.2 Description of the algorithms

As I have already mentioned, the most important functionality of the application is the implementation of the two algorithms, namely A* and the genetic algorithm (GA). Below I have given an overview of these algorithms before moving on to my implementation of them and applying them to the problem of pathfinding.

3.2.1 A* Algorithm

The A* algorithm is a path planning algorithm that assures to find a path with a low traversal cost under the right conditions. It works by building all of the paths that lead to the goal, one at a time and ranking each of them for consideration using the following formula:

$$f(x) = \mathrm{g}(\mathrm{x}) + h(x)$$

where *x* is one of the partial paths, *f* is an estimated final cost of the path through the partial path *x* to the target, *g* is the cost to traverse *x* and *h* is an estimate of the remaining distance from the end of *x* to the target [6].

When the heuristic is equal to zero, A* becomes Dijkstra's algorithm and all of the neighbouring nodes are expanded. Instead, The A* improves Dijkstra's algorithm by using a heuristic approach which means that only the areas that look favourable are examined. A good heuristic function which makes a precise estimation of the cost can significantly improve the algorithm and make it a lot faster. If the heuristic overestimates the real cost by a little bit, the outcome will be a faster search because less nodes will be traversed [2].

There are several types of heuristics – Euclidean, Manhattan and Diagonal Shortcut. For my project, I have used Euclidean heuristic which is defined as the straight-line distance from the starting to the target point. Each computation involves two powers and a single square root which makes it computationally expensive. The following formula is used to measure the distance, where (x_{path}, y_{path}) are the coordinates of the point at the end of the currently examined partial path and (x_{goal}, y_{goal}) of the destination point: [6]

$$Distance = \sqrt{(x_{path} - x_{goal})^2 + (y_{path} - y_{goal})^2}$$

During the process, the A* algorithm will create two lists, one defined as an Open set which keeps track of all the nodes ready to be searched next and Closed set which stores the nodes which have been searched. The current node is considered the parent node and is stored in the open set. Its surrounding nodes are searched and the ones with minimum *f(x)* value are put into the closed set. This goes on until the target node is in the open set. Finally, getting the coordinates of all nodes in the closed set returns the optimal path. The algorithm chooses the next parent node using the *f(x)* value [7].

The procedure which the algorithm uses in pseudocode is shown below [8]:

```
1. Store start_node into Open Set with f (starting node) = h
   (starting node)
2. while Open Set is not empty {
     a. take from Open Set current_node with lowest f
        (current_node) = g(current_node) + h (current_node)
          i. if (current_node == target_node) {
             break; → solution found}
     b. Generate child_node from each current_node
     c. Foreach (child_node of current_node) {
        Set child_node_cost = g(current_node) + w(current_node,
        child_node)
          i. if (child_node is in Open Set {
                    if (g(child_node) <= child_node_cost
                    continue; (jump to point 3)
         ii. else if child_node is in Closed Set {
                    if g(child_node) < child_node_cost
                    continue; (jump to point 3)
                    move child_node from Closed Set to Open Set
        iii. else {
             add child_node to Open Set
             set h(child_node) = heuristic distance to the
             goal}
     d. Set g(child_node) = child_node_cost
     e. Set the parent of child_node to current_node
        }
3. Add current_node to Closed Set
   }
```

```
4. If (current_node != target_node)
   Exit with error (Open Set is empty)
```

A* is probably the most preferred algorithm in the field of pathfinding because of its useful properties. First, it is guaranteed to find the path to the target node if such exists. This path will be optimal if $h(x)$ is always less than or equal to the cheapest path cost from end of x to the goal. A* also uses the heuristic more efficiently compared to other search methods and examines fewer nodes when finding the optimal path. It performs best in smaller search spaces where there are less squares to search [2].

3.2.2 Genetic Algorithm

Genetic algorithms are inspired by the concept of Darwinian evolution. The idea is based on the evolutionary concept of natural selection and genetics. The algorithm uses a mathematical requirement, and this is an important characteristic when it comes to issues of optimization [5]. By mirroring genetics and evolution, the algorithm can be used to solve real-world problems.

It starts by initializing a collection of solutions (chromosomes). After that genetic operations such as selection, crossover and mutation are applied on some of the solutions in the population. New solutions are generated from the current population by a fixed in advance selection method. The most fundamental part of the algorithm, during which new solutions are generated, are the crossover and mutation operators. The overall performance of the algorithm is determined by these operators. During mutation, a local change is randomly created to discover a new search space which replaces the worst solutions in the population. The solutions are evaluated based on a fitness function during each iteration of the main for loop. Once the algorithm stops, the best solution is returned [5].

Below is the pseudo code for the genetic algorithm based on the pathfinding problem [14]:

```
Generate randomly the initial population (set of feasible
paths) using the Euclidean distance heuristic
   while (generation number < max generation number) do
        Fitness function
        Elitist selection
        Rank selection
        repeat
             choose randomly two paths (parents) from current
             generation
        if (random number generated < crossover rate) then
             if (parents have common cells) then
                  perform the crossover
                  move the resulting paths to the next
                  generation
             else
                  choose other two parents
             end if
```

```
            else
                move the parents to the next generation
            end if
        until (next generation size < max population size)
        for each path in the next generation do
            if (random number generated < mutation rate) then
                choose randomly a gene and replace it
                if (resulting path is feasible) then
                    replace old path with new one
                else
                    choose randomly two cells C1 and C2 from the
                    path
                    remove all the cells between C1 and C2
                    connect C1 and C2 using the greedy approach
                    based on Euclidean distance heuristic
                end if
            end if
        end for
end while
```

Generally, a genetic algorithm is implemented with the following components [9]:

- **Genetic representation of individuals (encoding of chromosomes**) – the algorithm requires a genetic coding of the individuals. Different types of encoding can be used depending on the problem. Binary encoding is the most widely used technique. Every chromosome is represented as a string of 0s and 1s. Examples are shown below. This kind of encoding allows for many possibilities of the chromosomes.

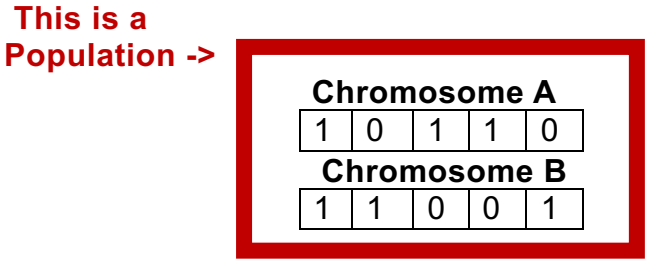
Chromosome A	101110000111000111
Chromosome B	111100000110001111

  Applied to the path planning problem, with this type of binary encoding, each gene will consist of integers which could be interpreted as follows:
  {00} – move up
  {01} – move right
  {10} – move left
  {11} – move down
  The potential solutions to the problem (the chromosomes) are paths (position sets) which link two pre-determined positions – a starting node and a target node.

  The fitness function and the genetic operators use this encoding, so it is very important for the performance of the algorithm.

- **Initial population** - the algorithm starts the search with an initial population of potential candidates. A population is a collection of chromosomes. What

this initial population will be is of crucial importance for the efficiency of the algorithm. The initial population is usually created randomly which increases the convergence time of the algorithm. The size of the population is one of the parameters of the GA which is generally fixed during the execution.



- **Fitness function** - once the initial population is created, the algorithm must assign to each individual a value which reflects its quality. For that, it uses a fitness function. In each of the generations, the fitness function assigns a value to each individual in said population based on its quality. The size of the path is the main criterion by which the fitness is evaluated. Several principals are considered such as distance, safety and smoothness. The implementation of a good fitness function is important because this information is used to select individuals from the population for mutation, reproduction and in the end for choosing the best solution.

- **Selection** – this is the operator which chooses the parents that will construct the next generation. The better the fitness value, the higher the chance to be selected for mating. This process is a simulation of the survival of the fittest principle found in nature. There are several selection methods such as rank selection, elitist selection, roulette wheel selection with the most popular being tournament selection [14] which I have used in my application. During tournament selection individuals are selected and a tournament is run among them. Only the fittest one passes to the next generation.

- **Crossover** – this operator is applied after the selection has been made. During crossover, the genetic information of the selected parents is combined in order to increase the fitness value of the new individual. The new chromosome will possess the best characteristics of both of its parents. There are different types of crossover such as one-point crossover and two-point crossover. In my application, I have used one-point crossover. It is performed by choosing one crossing position and exchanging genes between the two parents to the right of the crossover line as illustrated by the diagram below.

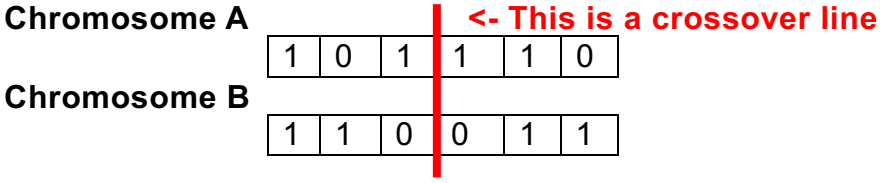
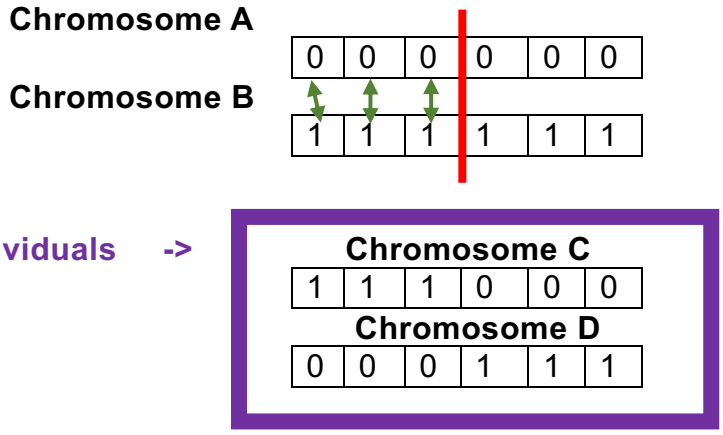
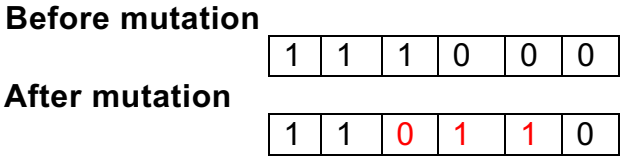


Illustration of how crossover is performed:



- **Mutation** – the goal of this operator is to improve the diversity. During mutation, a gene is randomly selected and replaced with a new one. The purpose of this operator is to add new information in a random manner and help to prevent getting stuck in a local optimum. Individuals are randomly selected for mutation and during it their genes are randomly replaced. The resulted path is evaluated and if it feasible, the individual is passed to the next generation [14].



3.1.3 Restrictions and limitations of the algorithms

When it comes to the A* algorithm, some uncertainties may arise. For example, discovering that the path being followed does not lead to the goal. Moreover, the algorithm is generally considered to be computationally expensive. In particular when it comes to the maze traversal problem, the algorithm would expand many nodes only to find out that the path does not lead to the goal. Then it would start examining other regions which were initially considered bad. This wastes a lot of time and resources [10].

In addition to that, the algorithm depends heavily on the heuristics. The final result would not be optimal at all in the absence of good heuristics. The search would also take a lot of time. The heuristic is expected to get the process closer and closer to the goal but if a sudden change occurs in it, the search will be destroyed. For example, since the heuristic function at any point in the maze depends on its distance from the target, when the algorithm reaches second to last position and is at a point surrounded by walls, the heuristic will increase suddenly. Likewise, if the solution is a bundle of bad moves followed by many good moves, the function will decrease from high to low. Such fluctuations in the value of the heuristic function damage the algorithm's performance [10].

When it comes to the genetic algorithm, since the algorithm works in a grid map and does not have control over the population diversity, the problems that most often arise are early convergence, slow convergence speed and too much time consumed by the algorithm.

Because of the above-mentioned constraints, numerous improvements, additions and suggestions have been proposed by researchers regarding the A* and the genetic algorithm which tackle various of their problems. Most often the algorithms are combined with other algorithms in order to achieve better results.

3.3 Description of the user interface

3.3.1 Environmental model

Coming back to my application, in order to solve the pathfinding problem and illustrate the process, the searching should be conducted in the appropriate environment. As I already mentioned, the algorithms traverse a set of vertices which are connected by edges and thus depict a graph. This graph or more specifically the visited nodes which represent the paths should be visualized in an intuitive way which is easy to understand by anyone, not only by people who have knowledge about graph theory. This illustration should show where the obstacles can be found, what are the traversable regions, as well as where the start and the target are located. In other words, there must be a drawing of a maze where all of this information, as well as the produced paths, will be visualized.

There are different ways to represent and draw the maze. The method I found the most appropriate in order to depict the maze is through the use of a grid. A grid is a network of intersecting parallel lines. The grid consists of adjacent square shaped blocks all of equal size which is considered to be 1. The square grids are the most popular ones used in games and robotics.

The maze is constructed using a two-dimensional array which holds values that represent the walls, gaps, starting point and destination. The number 0 in the two-dimensional array is considered to be a traversable blank space which can be passed through. The corresponding squares are painted white in the representation of the maze. The number 1 in the array represents a wall or an obstacle which the algorithm should avoid and cannot pass through in order to get to the target. The squares that correspond to the 1s are painted in black. The numbers -1 and 9 represent the start and the target respectively. There is only one starting point and only one target point. The start is painted in green and the target in red. Figure 2 holds a visual representation of the above-mentioned concepts with the value that each of the squares corresponds to in the two-dimensional array. The outlines of the maze are always depicted as walls which gives a better look of the maze as a whole. In all mazes which will be used in my application, the start and the target are always placed inside the maze, but this doesn’t have to be the case. The start and target points can also be placed along the outer walls of the maze.

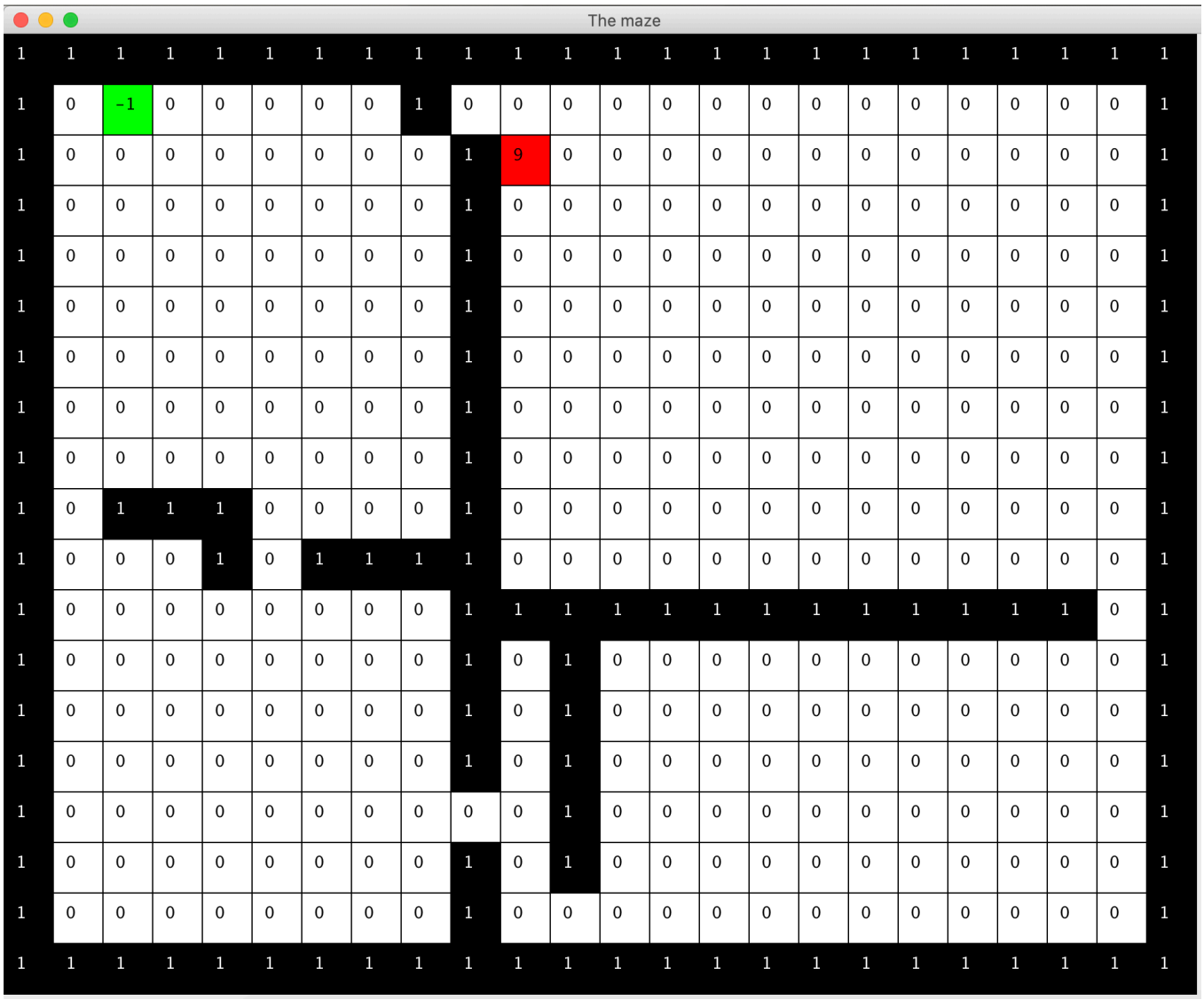


Figure 7 The maze the final path will be visualized in with walls, gaps, start and target points

The square grids limit the number of neighbours to four. The neighbours can be extended up to eight, but I have only allowed four movement directions in my application. The directions are up, down, left and right and diagonal movements are not allowed. If the square is located next to a wall, the number of choices is decreased to three, two or one.

The maze given to the application should be of appropriate size. If it goes over a particular size, it would be computationally impossible for the algorithm to find a path. Hence, I have restricted the size of the maze grids to the following: 15x15, 20x20, 25x25, 30x30, 35x35. This does not mean that the algorithms cannot find paths in larger grids. In fact, I haven't found the largest grid size for which the algorithms fail. The main point of this restriction was to provide mazes which are guaranteed to always work and to illustrate how the algorithms perform in different size in regard to time and path length. Since the application works with mazes of different sizes, when the overall size is change by the user, the size of each individual square changes in accordance with it so that mazes of different sizes can fit inside the same frame.

3.3.2 Main frame

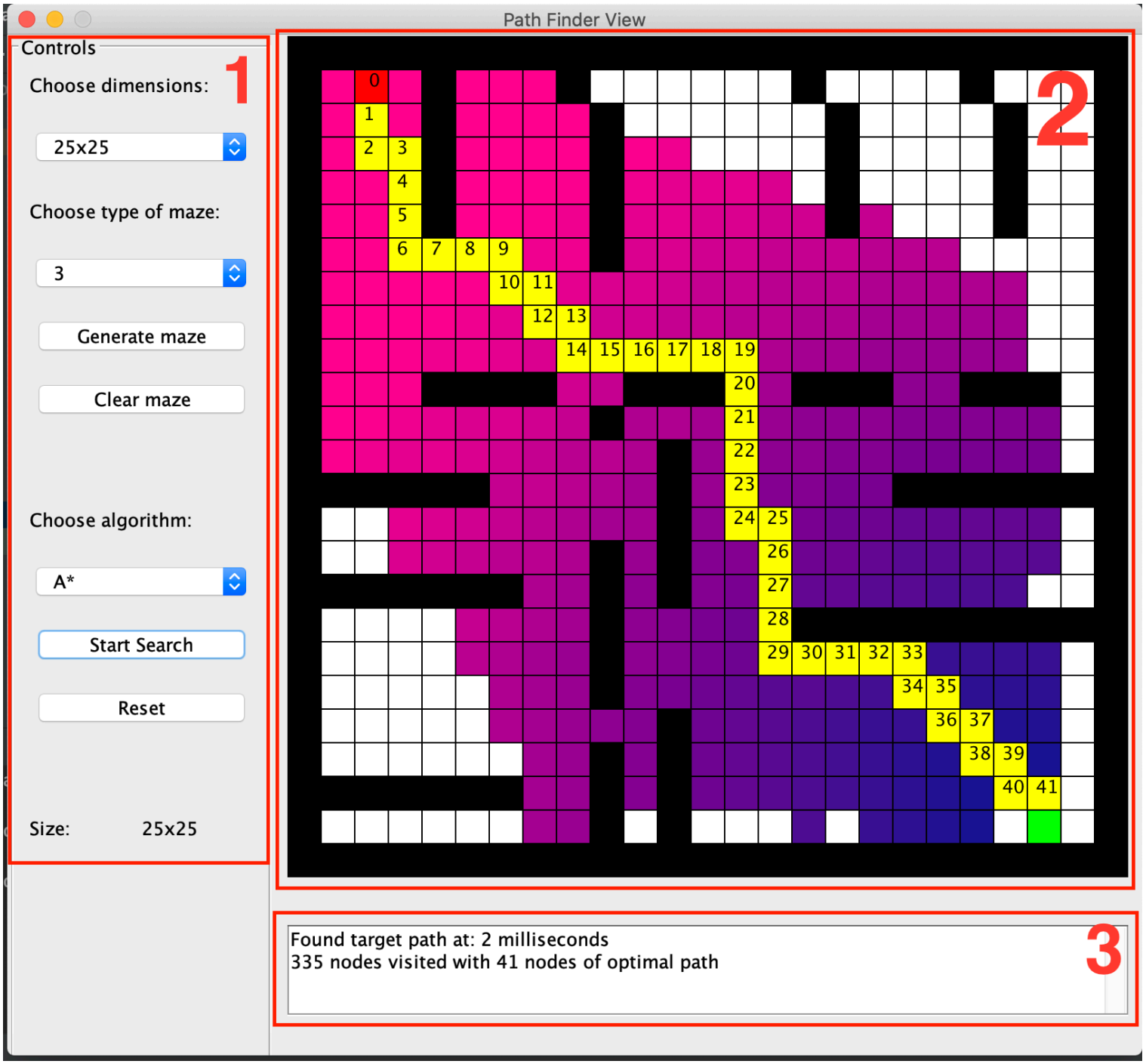


Figure 8 The main window for the program

In order to explain the user interface better, I have divided the main frame into three parts each labeled with a number from 1 to 3. I will explain the elements and purpose of each one of them below.

Panel 1 - as you can see, it is titled “Controls” and through this panel the user is controlling the application. In the first drop-down menu the user chooses the size of the maze. There are several options which are 15x15, 20x20, 25x25, 30x30 and 35x35. When an option is chosen, an empty maze with the chosen size appears in the

maze board so that the user can see visually how big such a maze is. With the second drop-down menu the user can choose the maze that they want to use. There are four different options for each of the size which differ in terms of complexity and density of obstacles. There are four numbers which represent four different mazes. In order to see the maze which each option represents in the maze board, the user needs to click the “Generate maze” button. They can play with the different sizes and types of mazes until they choose the right one. The “Clear maze” button reverts the previously chosen maze back to an empty one. Even if the user doesn’t click the “Clear maze” button but simply chooses new size and maze type, the maze will change in the maze board. After the user is happy with the maze, they can choose an algorithm from the third drop-down menu. There are two options – genetic and A*. The “Start Search” and “Reset” buttons are self-explanatory. One initiates the search in the chosen maze with the chosen algorithm. The other is used after the search is finished. The maze is reset so that the user can choose a new one and initiate a new search. The last label shows the size of the currently chosen maze and it changes dynamically when different mazes are switched.

Panel 2 – this is the main maze board. This panel changes according to the different combinations chosen with the “Controls” in Panel 1. It can show an empty maze or a maze with obstacles, as mentioned above. The empty maze can be with any of the five available sizes. This is also where the algorithms are visualized so we can say this panel is the most important part of the frame. When the algorithms are run, it shows the maze once again with the start and target nodes and then the algorithms start working.

For the A* algorithm, the board displays the squares corresponding to the visited nodes by the algorithm in the graph in purple while squares matching the nodes on the final path are yellow. As you can see in the figure above, the purple squares are painted in slightly different shades of the color. The shade depends on how close the square is to the target one. The nodes further away from the target are displayed with a lighter shade of purple while the nodes closest to the target have a darker purple colour. The shade is determined based on the value of the heuristic function in that particular node. This can serve as a visualization of how the A* algorithm makes a decision based on the heuristic information and how it determines which node to visit next. There is also animation when you run the A*. First the visited nodes (the purple ones) appear gradually, starting from the nodes closest to the start and getting to the ones closer to the target in the same in which the algorithm itself works. Then the final path (the yellow one) is visualized with an animation as well. Starting from the target and then gradually getting to the start in the same way the final path is obtained by the algorithm as well.

The user interface connected to the genetic algorithm has its own specifics and looks differently. Refer to the picture below for a visual representation. Here only the nodes on the final path are displayed. If the maze is more complex, the genetic algorithm might need to run several times with different values of the parameters. The parameters change automatically and the path in the frame changes in real time as well. The frame switches all the time with the different paths. I decided that it would be a good idea to show each of the different chromosomes (or paths) obtained by the algorithm instead of just the final one because in this way we can see how the algorithm itself is working and how with each iteration it gets closer and closer to the

target. Once the algorithm has found the target, the animation stops, and the user can see the final path that has been found. Again, the path is displayed in yellow in order to stand out and the word “Searching finished” appears below the panel.

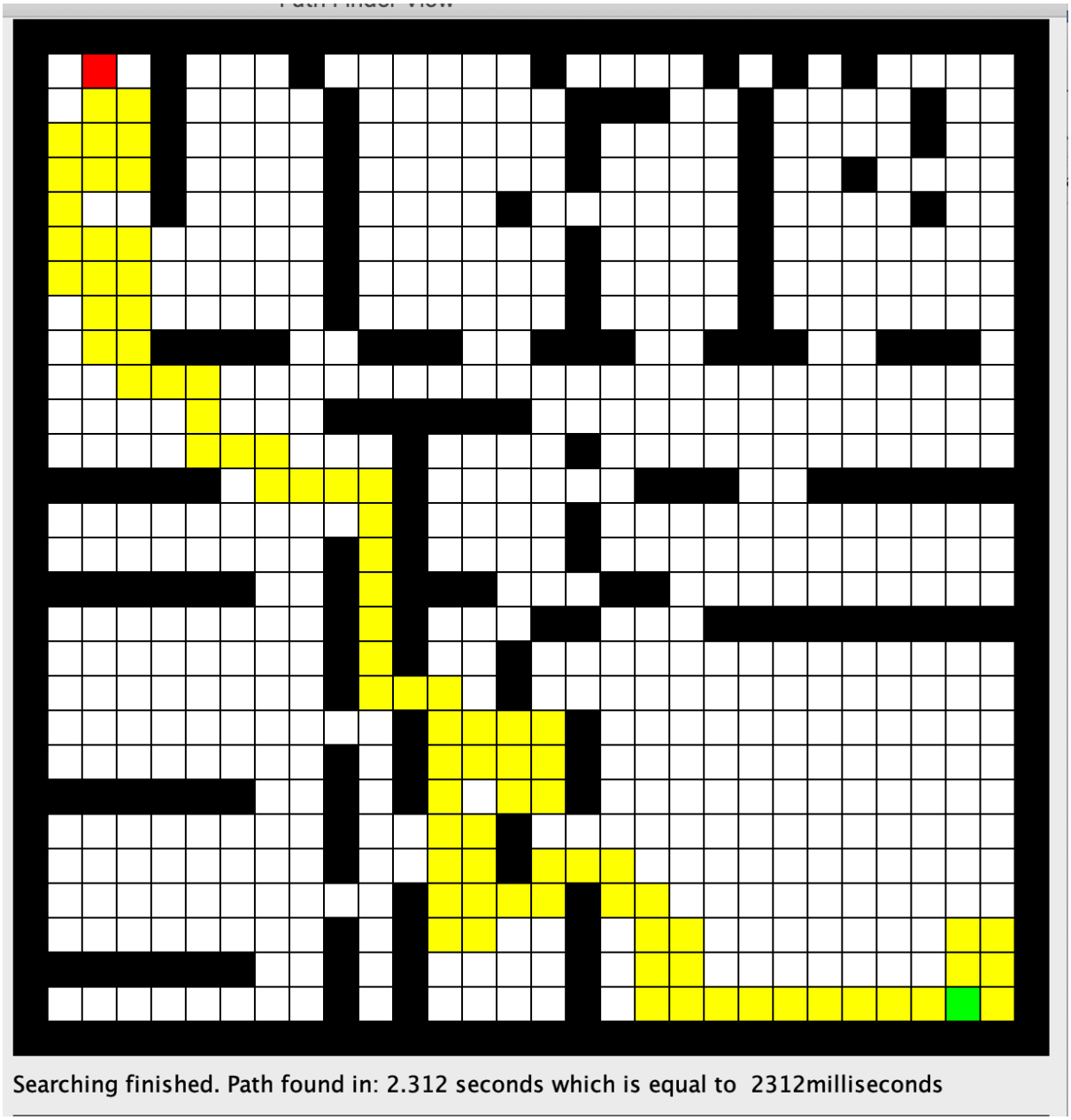


Figure 9 A successful run for the genetic algorithm and the obtained path

Panel 3 – this panel shows analytics information about the algorithms during or after they have finished searching. For the A*, it shows how many milliseconds it took for the algorithm to find the path as well as how many nodes have been visited and how many nodes the final path consists of. This is important to know in order to evaluate the performance of the algorithms. When it comes to the genetic algorithm, again, things look a little different. The panel first shows the current values of the controlling parameters which the algorithm is currently using. It also shows the number of the current generation and the fitness value of the fittest individual in this generation. These values change dynamically and are scrollable. When the genetic algorithm finishes it is indicated that the search has finished and the generation in which the

solution was found, as well as the fitness value of the successful chromosome are displayed.

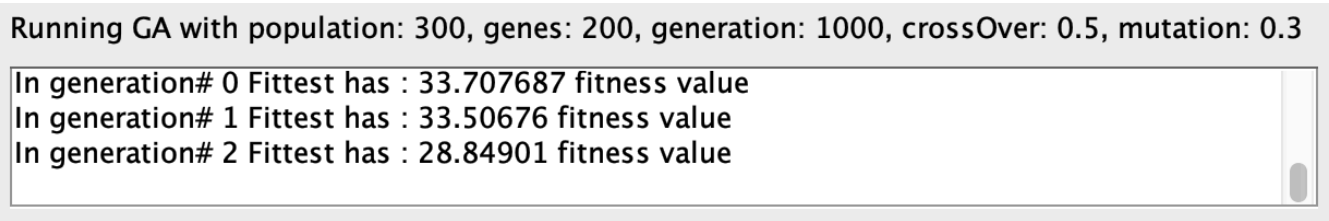


Figure 10 Information displayed while the genetic algorithm is running

3.4 Software Architecture

3.4.1 Overview

Software architecture is the organization of the software system including its components, the relationships between them and to the environment. It is concerned with how the code is structured and organized in order to achieve the required functionality. The different functionalities that exist within the code must be separated and components that share common or similar functionalities should be grouped together. This is one of the main practices of writing high-quality code.

The most common way to separate functionalities is to group them into three different categories – User Interface code functions, Application Logic code functions and Data Access code functions. In order to do that, the best practice is to use a defined software architecture in order to separate the related sections of code. The two most common architectures are layers and MVC. When designing this project, I considered using both of those architectures but in the end, I decided to use the layered architecture.

The layered architecture makes it easy to separate the user interface from the logic of the application, the logic from the data access and thus make separation of concerns easier. The application logic layer does not need to worry about the formatting and presentation of data as that is a concern of the presentation layer. Generally speaking, the layers of the architecture are the user interface layer, the application logic layer and the data access layer although large and complex application may contain four or more layers.

This separation of concern is achieved by source code organization. The layered architecture can be practically implemented by using namespaces or packages according to the syntactical constructs available in the particular programming language. The different components of the layered architecture can be independently deployed, maintained and updated. The concept of the layered architecture is simple, easy to learn and can be seen at first grasp. If the layers become large, they can be split into smaller layers.

The figure below illustrates the layers in my application. In the pages below, I will explain in detail what is included in each of the layers and how the layers interact with each other.

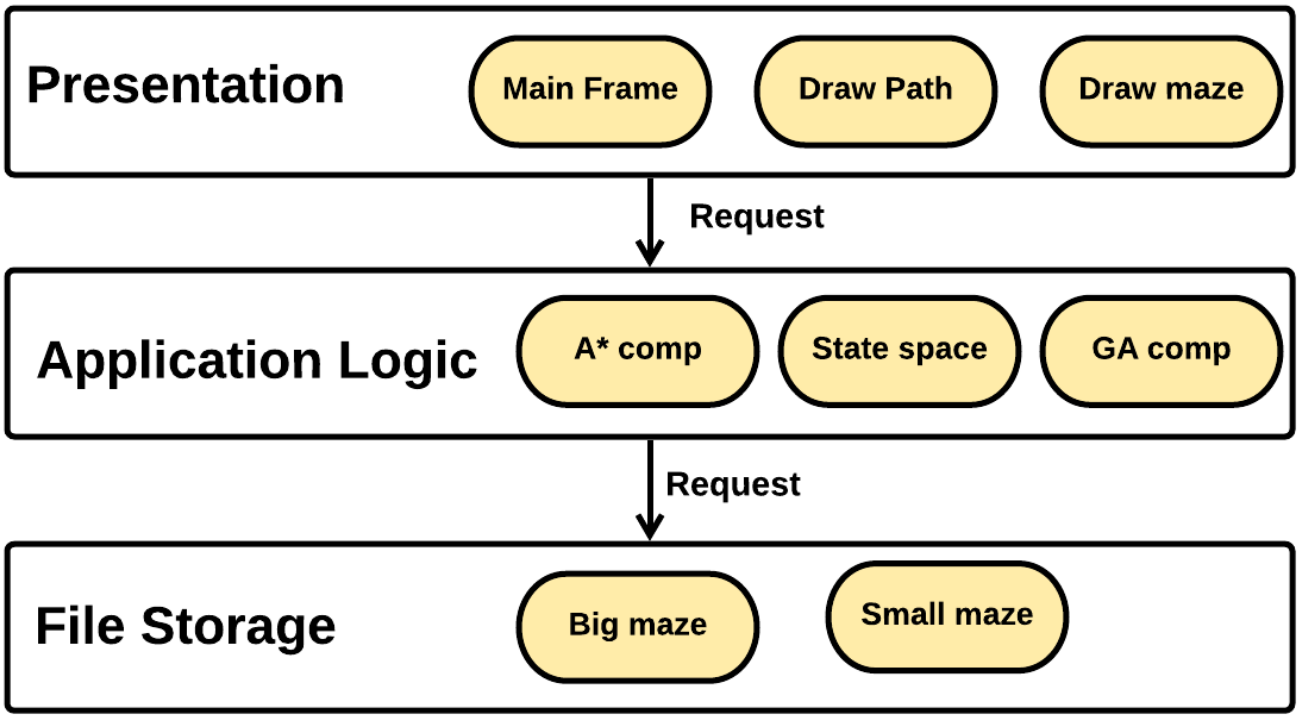


Figure 11 The different layers in my software architecture

The user interface layer usually includes all classes which are responsible for displaying the UI to the user or sending a response back after data has been processed. The main purpose of the UI is to translate tasks and their results into something the user can understand. In my application, the presentation layer consists of the classes which are responsible for the user interface such as the classes with the code responsible for displaying and working with the main frame of the application which includes the user controls and the maze board. In this layer are also the classes and methods responsible for drawing the maze, the paths by the two algorithms and the animation. According to the data input by the user, the presentation layer will make a request to the application logic layer.

The application logic layer includes all of the logic that is required by the application used in the implementation of its functional requirements. This layer makes logical decisions and evaluation, processes commands and performs calculations. In my application, this layer is responsible for the main computations and consists of the implementation of the algorithms which processes the user input which in this case is a maze and returns the desired output. It encompasses the logic behind the program. This layer is the most crucial part of the entire program. As I mentioned above, if a layer becomes too large, it can be split into several smaller layers. In my case, the application logic layer is very large so I have split in into three parts – A* logic sub-layer, genetic algorithm logic sub-layer and state space sub-layer which itself can be split into an A* part, GA part and common part which is used by both of the algorithms. The A* logic sub-layer consists of the core implementation of the algorithm and the data which is produced after the algorithm has finished. Same applies to the GA logic sub-layer. The state space sub-layer includes the logic for creating and working with the graph and its nodes which is data needed by both of the algorithms in order to perform the search. The graph is constructed using a two-dimensional array which

represents the maze chosen by the user. This two-dimensional array is requested from the third layer which is the File storage layer.

In the data access layer, information is stored and retrieved either from a database or a file system. This information is given to the application layer for processing and then to the user. Even though my application does not utilize access to a database system, as a third layer can be considered the files where the different two-dimensional arrays which represent the mazes are stored and retrieved so that the algorithms can use them.

3.4.2 Components

A component is a logical unit block of a system which offers a somewhat higher abstraction than classes. In fact, it can be defined as a set of classes with a well-defined interface that provides a given functionality. The components are conceptual stand-alone design elements. In order to visualize the components in a system, UML component diagrams are used. Their purpose is to assist in modeling the implementation details and ensure that each functional requirement is covered and planned in development. A component diagram decomposes the actual system into several high levels of functionality. Each component is responsible for one distinct purpose and only interacts with elements that are essential for it.

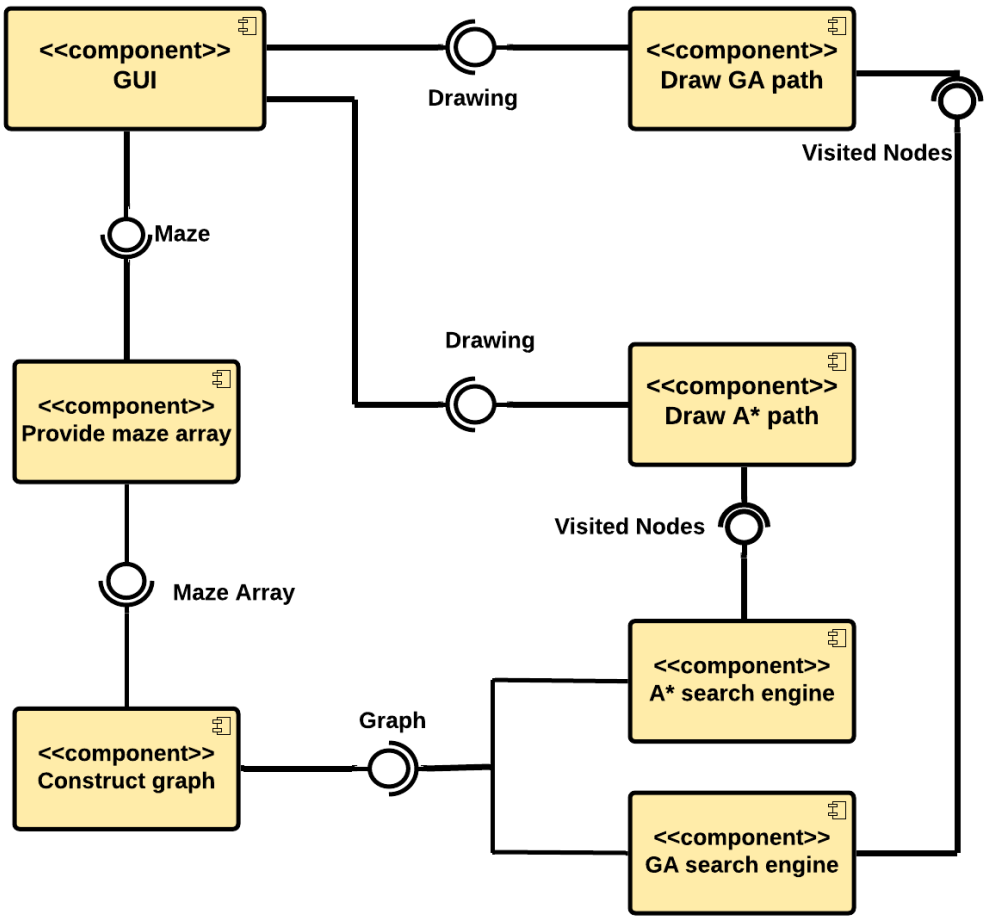


Figure 12 Component diagram

This is the component diagram for my project. It is a visual representation of the main components which I have identified for the application and how they interact with each other in order to conduct the main processed in the application.

The **GUI component** encompasses the module responsible for the inputs by the user as well as the produced output after it has been processed and the search has finished. In my application, the user provides the system with a maze which they prefer to use for the search.

This maze is then passed to the **Provide maze array component** which, based on the input by the user received via the drop-down menus, returns the appropriate array which corresponds to the chosen maze. The component which needs this array is the **Construct graph component** which constructs the graph out of the provided array, sets the indices of the nodes and their heuristic values.

The constructed graph is then used by the **A* search engine** or **Genetic algorithm search engine**. As you can see from the diagram, there is no interaction between these two modules. They are completely separated because the user can run only one of them at a time but not simultaneously. They work with the same maze data and it can very easily be compared how each of them works if we run them in the same maze, but these two components are not intertwined with each other. The logic behind the algorithms is very complex itself and this is a rather simplified diagram of how they actually work. The complex implementation details are skipped for simplicity and in order to only visualize the grand scheme of things. The algorithms produce data such as how many nodes have been visited during the search and the final path found by the algorithms.

This is used by the **Draw Path component**. The array of visited nodes is turned into a drawing and in this way the path is visualized. As you can see, there are two different drawing components for each of the algorithms because the paths are drawn in different ways and different logic is used. The drawing of the maze is then returned back to the **GUI component** and the user can see the result of the data that has been processed.

3.4.3 Package structure

A package is a collection of logically related UML elements. It is a group of similar model elements and can include elements of different kinds, including other packages. In order to simplify complex class diagrams, such as the one in this project, classes are grouped into packages. In order to explain the package structure, I have included diagrams for each of the three main packages of my application. A package is represented by a rectangle with a tab on the top. Packages can be nested inside one another. The rectangles inside each package represent the classes that live inside the package. Ideally, I would be able to include the methods that each class consists of but due to the complexity of the project, the classes and methods are too many and such a diagram would be too large and difficult to understand. That's why I have decided to keep it simple and write only the class name in the diagram and write brief descriptions about the services and functions each class provides.

3.4.3.1 Presentation package

The classes in this package are concerned with the input of data by the user and the output of results after the search has been done by the algorithms. First, you can see the diagram with the sub-packages which reside in this package and the names of the classes that live inside them. After that, I have written an overview of the classes which comprise this package.

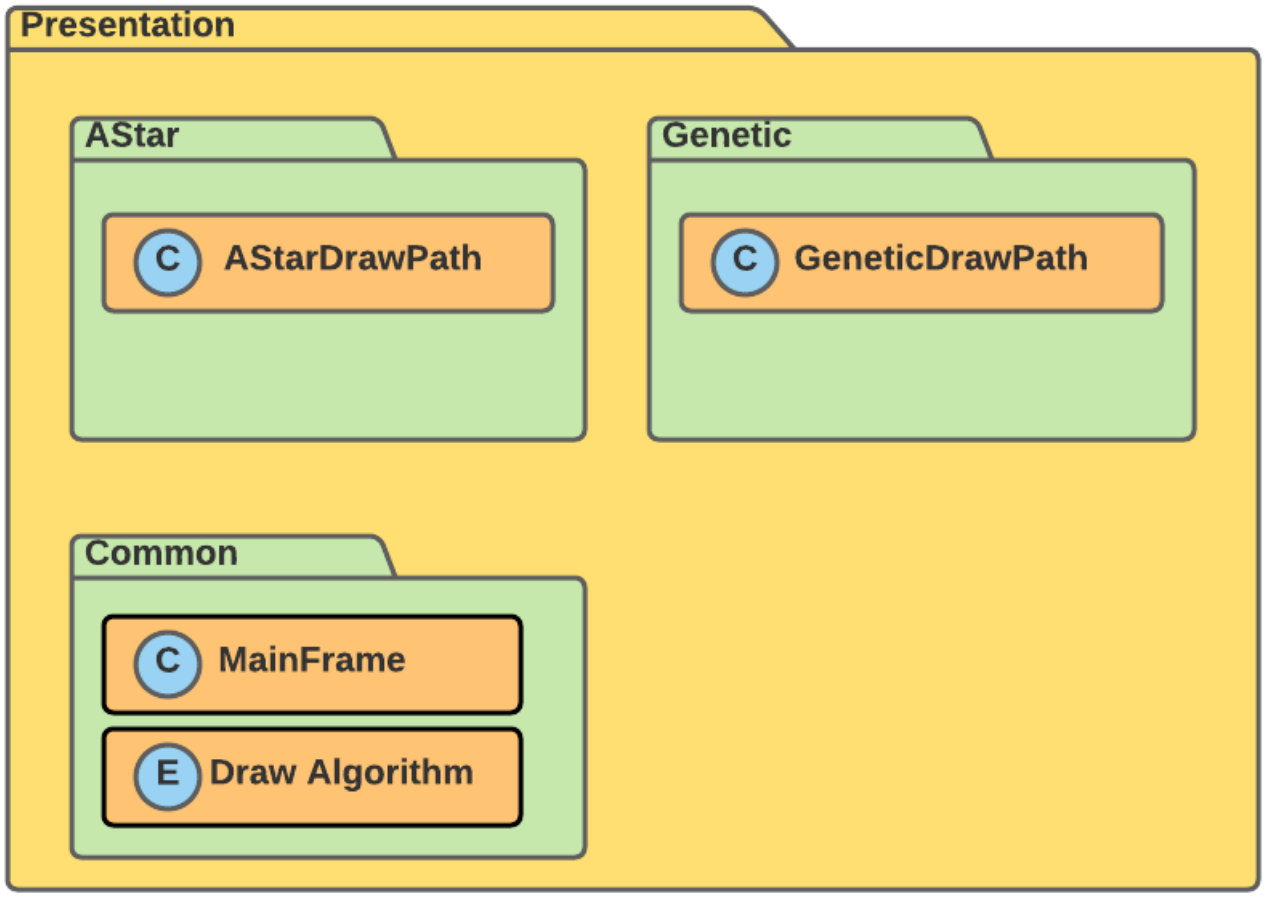


Figure 13 Presentation package structure and the classes that live inside it

- AStarDrawPath.java - This class's purpose is to visualize the result of the search done by the A* algorithm. Its most important attributes include an array of the visited nodes, another array which holds the nodes of the optimal path and a two-dimensional array with heuristic values. It also holds two synchronized lists that return thread-safe collections that will be used in the animation part of drawing the path. The class includes a method that draws the visited nodes by going through each one of them, getting their heuristic value and based on this value assigning specific numerical values which interpret the colour of each square. The lower the heuristic value of the particular node, the more intense will be the shade of purple in that box. There is another method which goes through the nodes of the optimal path and gives their blocks the colour of yellow. Another important method here is the one responsible for drawing the visited nodes and final path using an animation.

- GeneticDrawPath.java – similar to the A*, this class is responsible for drawing the final path found by the genetic algorithm. The method used is to look at the visited nodes of the graph, get their index and paint them in a yellow colour so that they are distinguished from the black and white ones.

- MainFrame.java – this is a very important class as most of the code for the visualization of the main frame which the user utilizes for the input and for seeing the output is here. Swing applications present their GUIs in a JFrame which is the top-level container needed to add the other components. Here live the parameters for how the frame optically looks, how big each of the components is in terms of its width and height and how it is positioned relative to the other components. The labels, drop-down menus, buttons, panels, maze board which are all a part of the main frame are defined here. This class holds the code for drawing the maze in the frame. In order to draw the maze, the elements needed are a maze array, preferred cell size and the number of all cells. The drawing is done by overriding the paintComponent() method from the JComponent class. Here squares with a particular size are defined which the maze will consist of. Then, it builds the maze based on the 0s and 1s hard coded in the two-dimensional array. If the value is 0, the block is painted black, if 1 – white, -1 stands for green and 9 for red. I have also included black border around each block in order to outline it.

  The class has instances of the two classes for drawing the paths for the two algorithms, mentioned above. Here is the logic for deciding which method to be called based on the choice provided by the user and what text to be shown in the white panel below the drawing of the path. Error messages need to appear if something is not right, so some error-handling is also implemented here. There is a separate method with the behaviour of each of the buttons in the frame such as reset, generate maze, update maze board, clear maze, update maze size, etc.

3.4.3.2 Application Logic package

This package is responsible for the main computations and the implementation of the algorithms used for processing the user input and returning the desired output. It holds the main logic needed for the application to work. I decided to split it in two parts. The first is the implementation of the algorithms themselves and the classes each of them needs. The second part holds the search space logic. The search space is what is explored by the searching algorithm in order to find the solution. Before the problem can be solved, it needs a representation of the search space that the algorithms can understand. The search space, generally speaking, consists of a graph or as set of nodes representing each state of the problem and edges between the nodes representing the moves from one state to another, an initial state and a goal state. The search space can be a tree as well but, in my case, it is a graph. I needed a representation of this graph and this is what the classes in the Search Space package are used for. Since the algorithms work with the same kind of graph, some of the classes here can be shared between them. In contrast with the graph, the nodes have some differences which are significant and I decided it would be better if they are separated. Hence, each of them has their own node class.

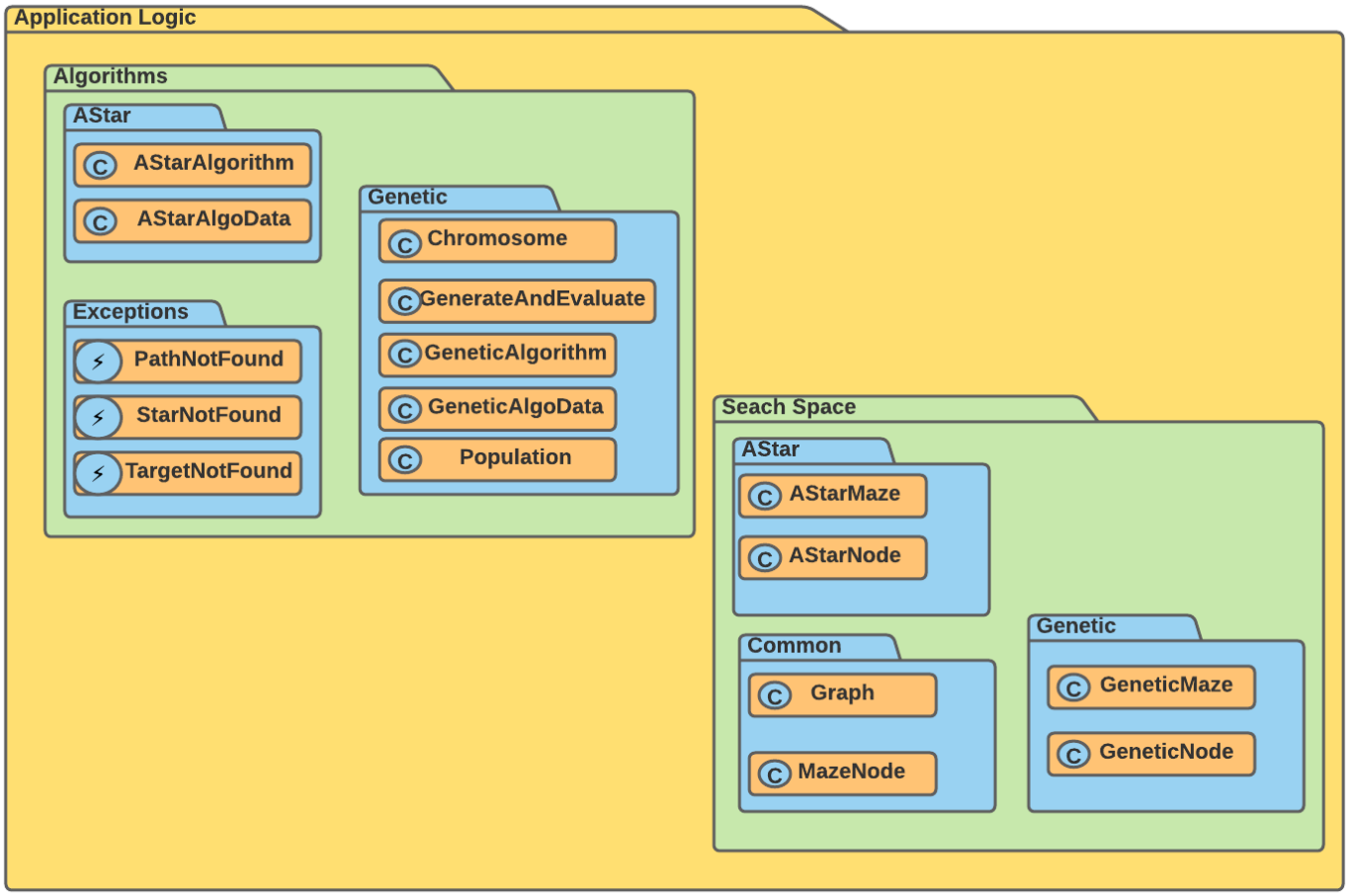


Figure 14 Presentation logic package and the classes that live inside it

3.4.3.2.1 Algorithms sub-package

- **AStar sub-sub-package**

  - AStarAlgorithm.java – here is the main part of the implementation of the A* algorithm. One of the fields of the class is for the priority queue, needed by the algorithm which is called an open list. This priority queue is used in order to choose the node with the minimum(estimated) cost for expansion. There is also a Boolean array which represents the closed list or the nodes which have been visited so far. The nodes which have been visited have the value true assigned to their corresponding index. The class also holds two array lists which keep track of the nodes visited and the nodes which comprise the optimal path. There is also a "timer" which counts for how many milliseconds the search was performed. At first, the priority queue is created, solutions are compared and put in the right order based on their heuristic value. The nodes with the lowest heuristic values are moved to the front of the priority queue. Then the node with the lowest final cost is removed from the queue and added to the array with the visited nodes. Its value in the closed list becomes true. The neighbours of this node are checked and if they are not visited, they are added to the priority queue. Their f(x) and g(x) values are updated, and their parent is set. The algorithm stops when a removed node from the priority queue is the target node. When this happens, starting from the target node, the parents of all of the nodes are retrieved and added to a new array which

by the time it has reached the starting node, holds all of the nodes which comprise the optimal path from the start to the goal. This is how the algorithm is implemented in my program in a nutshell.

- AStarAlgorithmData.java – this class holds the arrays with visited nodes, the optimal path, the time for which the algorithm finished the search and the maze that was used. I created it as an easy way to pass lots of data as a single object instead of passing multiple parameters which were needed to draw the path. It is used in the classes that draws the path which needs this information in order to illustrate the result of the search in a more human-friendly way.

- **Genetic sub-sub-package**

- Chromosome.java - a chromosome is a solution that is made up of multiple genes. Hence, in this class, there is an array which holds all of the genes and represents the chromosome. Each chromosome also has a fitness value and a set length of the genes which is one of the controlling input parameters of the genetic algorithm. The class consists of a method which initializes a chromosome. As I have already mentioned, the program works with binary encoding of the chromosomes. A chromosome is created by going through all of the genes and assigning 0 or 1 to each of them. The choice is determined by a comparison of a randomly generated number with a set probability value. Then a fitness value is assigned to each of the chromosomes using a method in the GenerateAndEvaluatePath class.

- Population.java - a population is a collection of multiple chromosomes. In this class the population is initialized. There is function which finds and return the fittest chromosome in this population based on the fitness values. The class offers a method which allows for the insertion of a chromosome in a particular place in the population.

- GenerateAndEvaluatePath.java – in this class a path is generated from the genes or in other words the data returned by the genetic algorithm is applied to the pathfinding problem. Each chromosome is a solution and each solution in this case is a path. The way to turn the binary genes into a path is the following. The genes which are an array of 0s and 1s are used to show different movement directions. Every pair of genes indicates a movement direction – {0,0} indicates moving up, {0,1} – going right, {1,0} – left and {1, 1} – going down. Each of these pairs of genes is converted to a decimal number. The decimal numbers are used to construct a path and this path is then evaluated. The path is constructed by using a for loop. Starting from a current node and based on the number in the decimal array, one of current node's neighbours is chosen, the current node is added to the visited nodes, the neighbour becomes the current node, and the next iteration starts.

  The next step is to evaluate this path and assign a fitness value to it. The fitness value is an evaluation of how good a path is. The less the value of the fitness function, the better is the path. The fitness function for the nodes is calculated in the following way. There is a method which finds the distance of each of the nodes of the path to the target node. Another method is used to indicate the
  closest node to the target that was found in the current path or the minimum index which was closest to it. It is possible that the target node can be in this final path

and if that is a case, the loop stops, and the fitness value is returned. If that is not the case, the index of the node closest to the target in the path is returned. The path and the fitness value are returned whether the target has been found or not. This is way we are able to see each path in the frame and see how the genetic algorithm works visually.

- GeneticAlgorithm.java - this class holds the implementation of the genetic algorithm in my application. The genetic algorithm runs by using controlling parameters such as the size of the population, the length of the gene, the maximum limit of generations, the tournament size, cross over and mutation rate. Changing these parameter affects the performance of the algorithms. The implementation goes through the main operators of the genetic algorithm which are selection, crossover, mutation and elitism. The main method is called *performSearch().* First an initial population is created. In the main loop, two parents are selected from the initial population. They are the fittest chromosomes in this population. Crossover is performed on them and a new individual is created. Since single crossover point is used, the genes from 0 to the crossover point are taken from the first parent and the genes from the crossover point to the end – from the second parent. After that comes the mutation process during which a gene in the new individual is randomly changed. It is then randomly decided whether this individual will be added to the new population. If not, one of the two parents is chosen to be added. At the end of the loop, the fittest individual from the population is returned and displayed along with the value of its fitness function. If this is the path to the target, the program stops. If the last 10 fitness values are the same, that means that the algorithm is probably stuck next to a wall. The program is stopped, and it runs again from the beginning until it finds the path but this time the values of the controlling parameters change.

  In the next method, the controlling parameters for the genetic algorithm are changed. This is where the genetic algorithm engine is created. An array is declared for each of the parameters which holds appropriate values for a particular size of the maze which enables the genetic algorithms to work with appropriate parameters and thus find the target faster. It shows to the user the values of these parameters after each iteration and the path which they produce. If the path is found, the GA engine stops. If not, a message is output that the path couldn't be found in the current iteration and the program keeps running by replacing the parameter with better ones.

- GeneticAlgorithmData.java - similar to the case with A*, I created this class as an easy way to pass lots of data as a single object instead of passing multiple parameters which are needed to draw the path.

- **Exceptions sub-sub-package**

  These three exceptions are to be thrown in the main frame class of the application when there is no path or the start or the target node cannot be found. Instead of exceptions, error messages appear for the user. They are used in the unit tests part of the application which I will explain later.

3.4.3.2.2 Search space sub-package

- **Common sub-sub-package**

  - MazeNode.java – this class is an abstract class which is inherited by AStarNode.java and GeneticNode.java. The class is abstract because the classes that inherit from it have very similar fields and methods. The private fields are the node index and an boolean value which indicates whether the node has been visited or not.

  - Graph.java – This class holds a list of all of nodes and by using this list instantiates the graph. This graph will be traversed by the algorithm. It is defined as a generic class which can use nodes either of type AStarNode.java or GeneticNode.java depending on the type of algorithm.

- **AStar sub-sub-package**

  - AStarNode.java - in graph theory, nodes (also referred to as vertices) are the fundamental unit of which a graph is formed. Since pathfinding eventually comes down to graph searching, a graph should be created and traversed. In my implementation, the nodes have several attributes - an index, an indicator whether the node has been visited or not, a list of neighbours, a parent node or in other words the node before the current one in the path, as well as the three values used by the A* algorithm also called heuristic values. This class inherits from MazeNode.java.

  - AStarMaze.java - this class is concerned with the representation of the maze which will be used by the algorithm. Based on a two-dimensional array which represents the maze that the user has chosen, the search space is created i.e. the graph is initialized with a set of nodes. The logic behind this is to start with one node, find its neighbours and add the nodes to a list one by one thus building the graph. The $f(x)$, $g(x)$ and $h(x)$ values are also calculated here based on the distance from the current node to the target one. As I mentioned in previous sections of this thesis, I have used the formula for Euclidean distance which is a popular formula used with the A* algorithm in order to calculate the heuristics.

- **Genetic sub-sub-package**

  - GeneticMaze.java - this class is responsible for the construction of the maze when the genetic algorithm is used. Here the graph is also initialized by a list of nodes. Starting with one node, the neighbours are found and used to build the graph. The process is similar to the one used in the A* algorithm but here heuristic values are not set for each node.

  - GeneticNode.java -since the GA is also a graph-traversal algorithm, it needs a graph. The graph that will be traversed is made up of nodes. This class holds the fields and methods needed for working with these nodes which include the index, a list of neighbours and a boolean value to indicate whether it has been visited or not. This class inherits from MazeNode.java.

3.4.3.3 Access mazes package

This is the last package in the application. The application might not have a database, but it still needs to store somewhere all of the maze arrays that the search can be done in. These mazes are the two-dimensional arrays given to the algorithms in the form of graphs with nodes and turned into a drawing of a maze with walls and blank spaces.

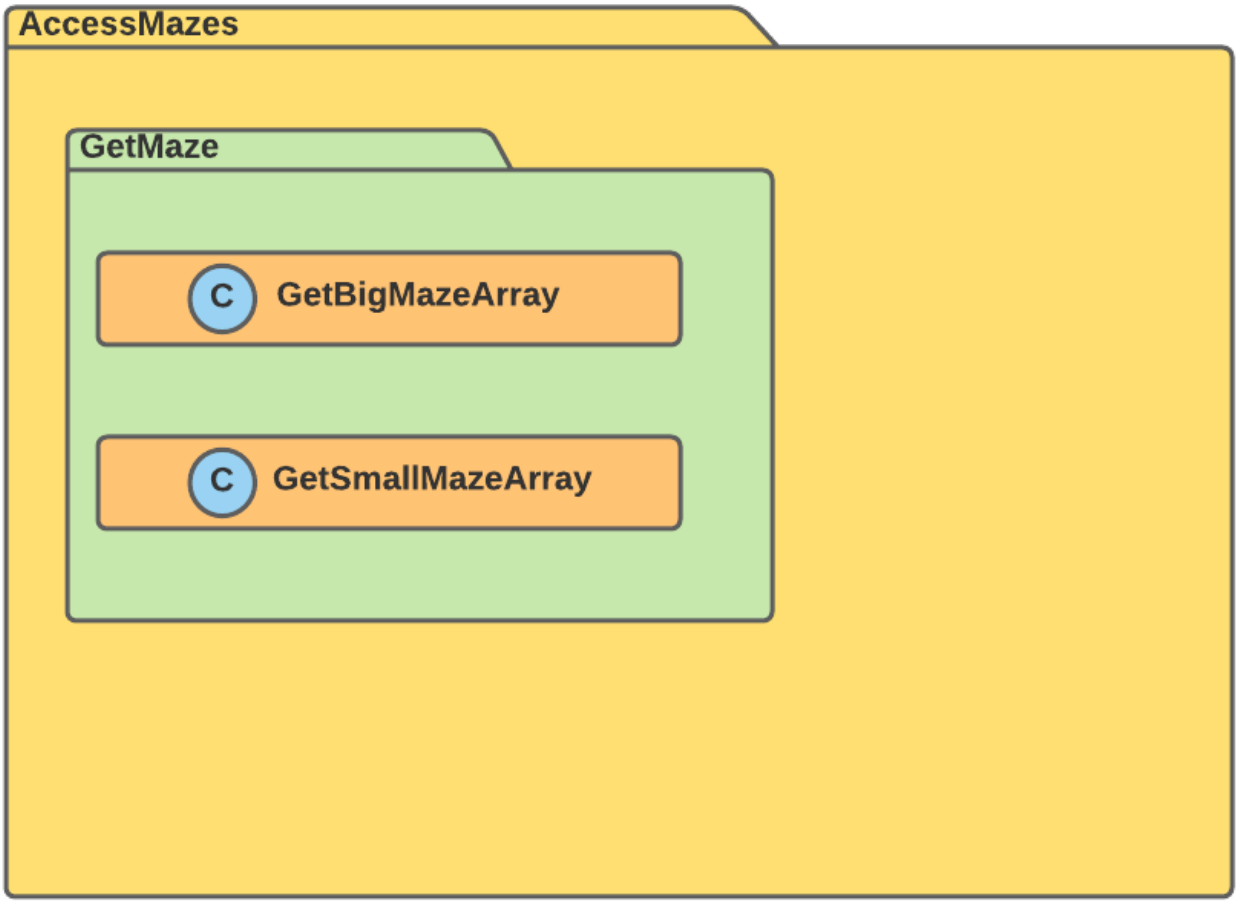


Figure 15 Access mazes package and the classes that live inside it

- GetBigMazeArray.java – in this class live the two-dimensional arrays which represent the mazes with size 30x30 and 35x35. There are four versions of the maze with size 30x30 and four different versions with size 35x35. The versions are very different and the purpose of that is to allow the algorithms to be visualized in different environments and see how they perform. The two-dimensional arrays have the same number of rows and columns and consist of numerous 1s and 0s, one -1 for the start point and one 9 for the target point. There is a method in this class that returns the appropriate maze based on the values chosen by the user.

- GetSmallMazeArray.java – similar to the previous class, in this class live the two-dimensional arrays which represent the mazes with smaller sizes such as 15x15, 20x20 and 25x25. There are different maze versions for each size so in total there are 12 arrays in this class. Again, the versions are very different from each other. The two-dimensional arrays also have the same number of rows and columns and have the values of 0, 1, 9 and -1.

The last class in the whole application which doesn't really fit into any of the packages is the PathFinerMain.java class. This is the entry point to the application where the main() method is located. This is the class that the user needs to run in order to make the frame appear and to start working with the application. There is an instance of MainFrame() declared here.

3.5 Security and reliability

Since this is not a web application, or an application that uses a database, there is no need to utilize any complicated database preservation or encryption schemes. It doesn't require user authentication, and anyone can use it and have the same privileges. This is not an application that requires internet access or the creation of user accounts but is intended to be a standalone desktop application. That's why, in terms of security, there is not much to protect against here.

In terms of reliability, the provided user interface aims to protect against errors and wrong inputs as much as possible. The input that the algorithms needs in order to run is a maze or more specifically a two-dimensional array that consists of 0s and 1s with the same number of values in each row. There should be only one starting and target point and the target should be reachable in at least one way. Mazes in the correct format are included in the application and as long as the user sticks to the user interface and chooses one of them, the algorithms will run, and no errors will be produced. However, if someone decides to interfere directly with the program files and code and gives the application an array that is not in the correct format, appropriate error messages will be displayed in the panel and the console. What I have tried to do is to handle as many of the exceptions and errors as possible and provide meaningful error messages instead of letting the system crush. The error-handling I have provided is explained in more detail in the Testing part of this thesis.

Another reliability aspect to consider are the computational resources of the system. If the algorithms, especially the genetic algorithm, run in a bigger or complex maze, the program might consume a lot of system resources in terms of high memory and CPU usage. This can slow down other currently running processes on the system which should be considered by the users.

IV. Implementation

4.1 Technologies used

4.1.1 Computing platform, programming language & IDE

As I have already mentioned in other parts of this thesis, the program is written in Java. In order to run the application, the system needs a Java platform which is a set of programs that facilitate the compilation of programs written in Java. The Java platform includes an execution engine called the Java Virtual Machine, a compiler and libraries. Java is not specific to any processor or operating system and the Java platform can be used on different types of operating system and is set to run identically on all of them. The compiler turns source code into Java bytecode which is the same no matter what hardware or operating system the program is run on. With that being said, the

bytecode allows the program to run on any system that has the JVM installed. While Java is platform-independent, the JVM is not and every OS has its own JVM but that is of no concern for the users. To conclude, the program should be able to run on any platform.

When it comes to the user-interface, I have used Java Swing and the components might look slightly different on the different operating systems. Swing uses the system's look and feel to make an application look like a native application and uses the design principles of the system it is run on. However, that is not a big concern in my case because even though the components, such as the buttons and drop-down menus might look different, this is not of crucial importance for the working of the application. The part which is important is the implementation of the algorithms and their visualization which looks the same anywhere. I have developed the program on a system which runs Mac OS, but I have also tested it on a Windows machine and the design of the application is preserved although the components look different.

The program was written using the latest version of IntelliJ professional edition. There are multiple IDEs available for working with Java and I was doubtful whether to use Eclipse or IntelliJ as they are the most popular Java IDEs. Both have their advantages and disadvantages but, in the end, I chose IntelliJ. The main functionalities that drew me to it were the smart code completion, chain completion, detection of duplicates, reliable refactoring tools, inspections and quick fixes. I like its design, the ease and pace of working with it.

The project was developed on a system that has the Java 8 installed. It was developed using JRE version 11.0.4 and is not compatible with the earliest versions like 1.8, for example. Also, in order to run the unit tests, JUnit 5 requires Java 8 or higher.

4.1.2 Programming language

Besides Java's portability which was described in the previous point, there are other reasons for which I decided to use Java for the development of this project. Java is one most popular programming languages among developers. It is an object-oriented language which provides a neat modular structure and makes it easier to solve complex problems. As such, it supports the four major building blocks of object-oriented programming such as abstraction, encapsulation, polymorphism and inheritance. These practices provide simplicity, code reusability, extendibility and security which improve the development process. In addition to that, Java has a large strong and active community support with lots of people providing help on Stack Overflow and other forums. This combined with high-quality documentation is a huge advantage when working with any programming language or technology.

Another reason is the richness and extensiveness of the Java API which includes numerous methods and classes that can help with every purpose. Java gives API to I/O, organizing, utilities, XML parsing, etc. Java offers a great collection of packages for data structures which are very useful when coding algorithms and which are not available in C++, for example. For example, I used the **java.util.PriorityQueue** class from the java utility package to implement the priority queue needed by the A* algorithm. The data structures supported by the Java utility package can perform a

broad scope of functions and are very powerful. There are other functions available for a faster development of algorithms as well.

The second reason has to do with visualization which is an essential part of the application. I needed a programming language that has a good toolkit of working with graphics. Java provides easy-to-use features like creating and managing windows and adding components to them. There is an abundance of pre-defined widgets available to use. Moreover, Java's built-in graphics tools for 2D applications like the one I am developing offer simplicity and ease of use. Java also supports a fairly easy option to implement an animation which is what I will do in my program.

4.1.3 Libraries, frameworks and technologies

The project was developed using only the Java built-in functions or the Java API which is a library of pre-written classed that is a part of the Java Development Environment. No external frameworks or libraries were used in the development process. In order to write unit tests, I used JUnit 5 which is framework for writing tests in Java. Below is a detailed description of some of the technologies that I have used:

- Java Swing - Java Swing is a Java graphical user interface (GUI) which includes a large set of widgets. It is a part of the Java Foundation classes (JFC) which consists of a group of features for developing GUIs, adding graphics functionalities and interactivity to Java applications. Swing contains packages for developing desktop applications in Java. It is included in the Java standard edition. Swing contains built-in controls such as buttons, panes, sliders, toolbars [12]. Even though I realise that Java FX is becoming the new standard when creating Java GUI, one of my objectives with this application is to keep things related to the user interface fairly simple and focus more on the implementation of the algorithms. For this reason, it didn't really matter whether I would use Swing or Java FX, so I decided to go for Swing because of my familiarity with it, its simplicity and the ease of use which it provides.

- Java Abstract Window Toolkit – Java AWT is Swift's predecessor and is a platform dependent API for creating Graphical User Interface for Java programs. It is also a part of the JFC – the standard API for providing GUI for a Java program. The most important class for me, from the AWT that I made use of is Graphics2D class which is a fundamental class for rendering 2-dimensional shapes, text and images in Java. Its purpose is to provide more advanced control over geometry, coordinate transformations, colour management, and text layout. Since, the program visualizes the path in a grid, another essential class is the Rectangle class which specifies an area in a coordinate space that is enclosed in an object's upper-left point (x, y) with its width and height. Another important class was the Color class since I am using different colours for the visualization of the path, the visited nodes and the maze. The class is used to encapsulate colours in the default RGBA colour space where RGB stands for red, green, blue, alpha. The values of the RGB part range from 0.0 to 0.1. The value of alpha determines the opacity of the color. Based on the tuning on these values, the different colors are created. This was used by me for the creating of the different shades of purple.

- JUnit 5 – is the new generation of JUnit which is one of the most popular testing frameworks available for Java. It is a test framework that uses annotations to identify methods that specify a test. It is an open source project hosted on GitHub. There is direct support to run JUnit tests on the JUnit platform on IntelliJ and tests can be run there. JUnit is composed of several different modules which are the JUnit Platform, JUnit Jupiter and JUnit Vintage. JUnit platform is a foundation for launching testing frameworks on the JVM. JUnit Jupiter combines a new programming model and extension model for writing tests. JUnit Jupiter is the API against which tests are written and the engine that understands it. JUnit Vintage is used for running tests on older versions such as JUnit 3 and JUnit 4 [15].

4.2 Installation requirements

The easiest way to run this application is on IntelliJ IDEA community or professional edition. The instructions provided below will assume IntelliJ is used but they can easily be translated to other IDEs as well as the sequence of steps required is the same. The first requirement is the system on which the program will run to have the JDK installed for the appropriate operating system. After that, the project folder should be opened with an IDE of choice. Before the program can run, the SDK should be set up. The way to do that is to go to File → Project Structure → Project tab. In the project SDK section, choose version 11 or higher. Click "Apply" and then "OK". The project is not able to run on the earliest versions of the SDK such as 1.8, for example.

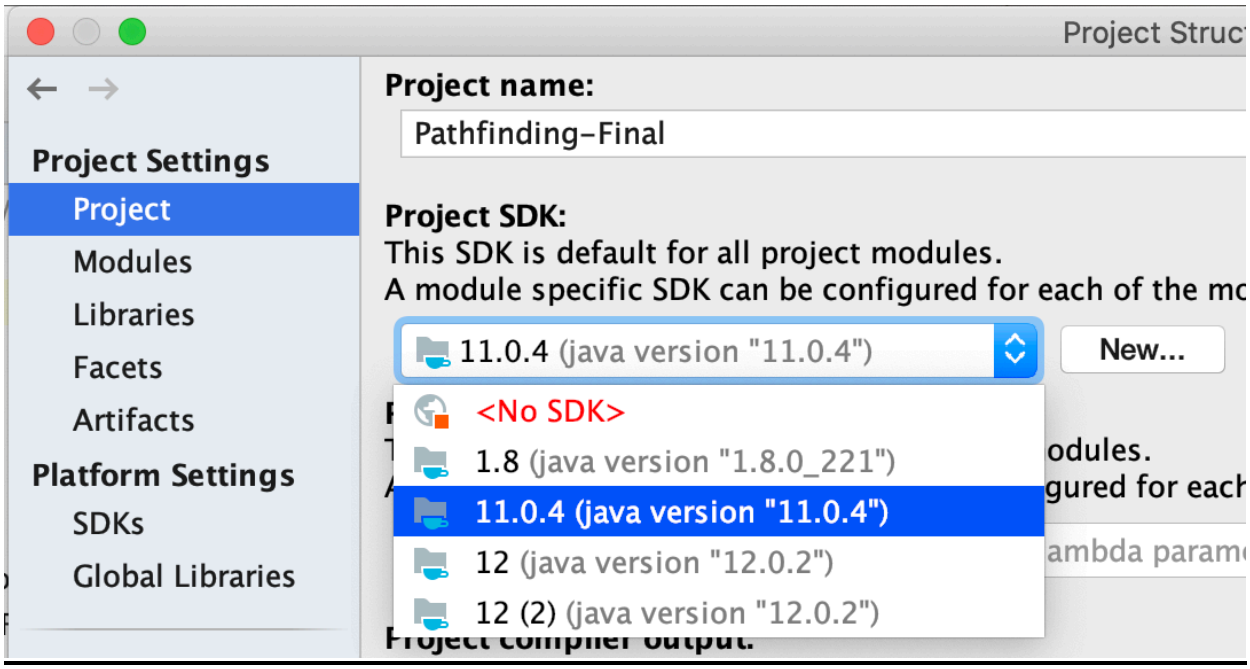


Figure 16 The SDK version that needs to be chosen when running the project

After the SDK is set up, go to the "Modules" tab from the same menu on the left.
The **src** package should be marked as a Source Folder and the **tests** package should be marked as Test Source Folder. To do that, click on Sources from the top menu and then on the blue **src** folder and then on Tests and on the green **tests** folder. Click "Apply" and then "OK".

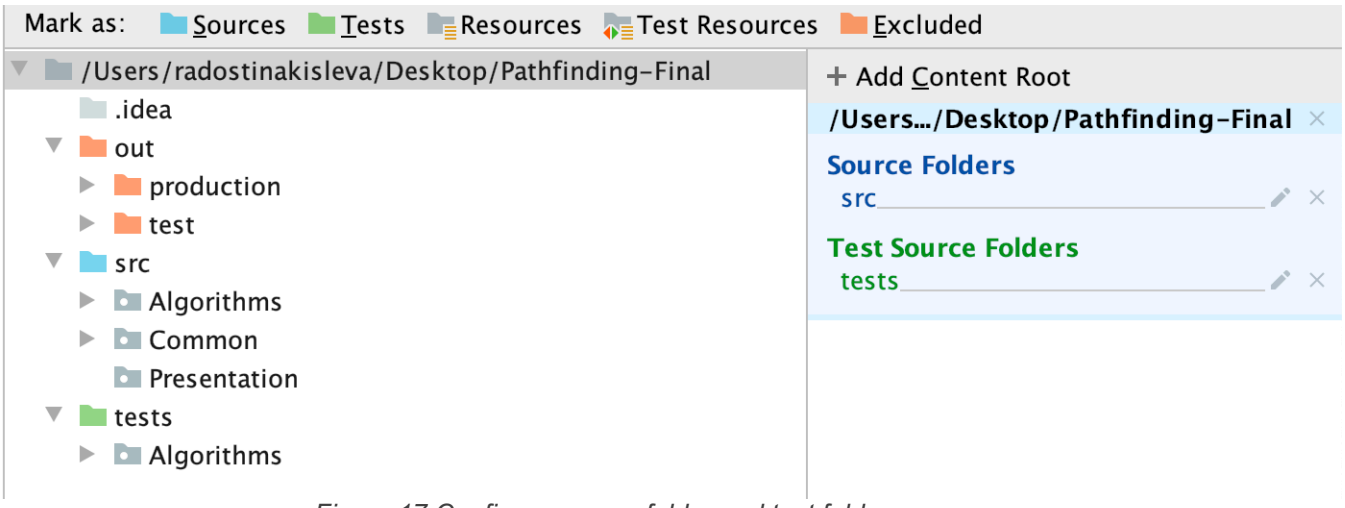


Figure 17 Configure source folder and test folder

The project should be able to run by right clicking on the main class which is PathFinderMain.java and choosing the "Run" option. The main window should appear and the user should be able to start working with the application.

In order to run the unit tests, Junit 5 must be set up. Go to File → Project Structure → Libraries and click on the "+" button. In the menu that appears select "From Maven". The following window will appear:

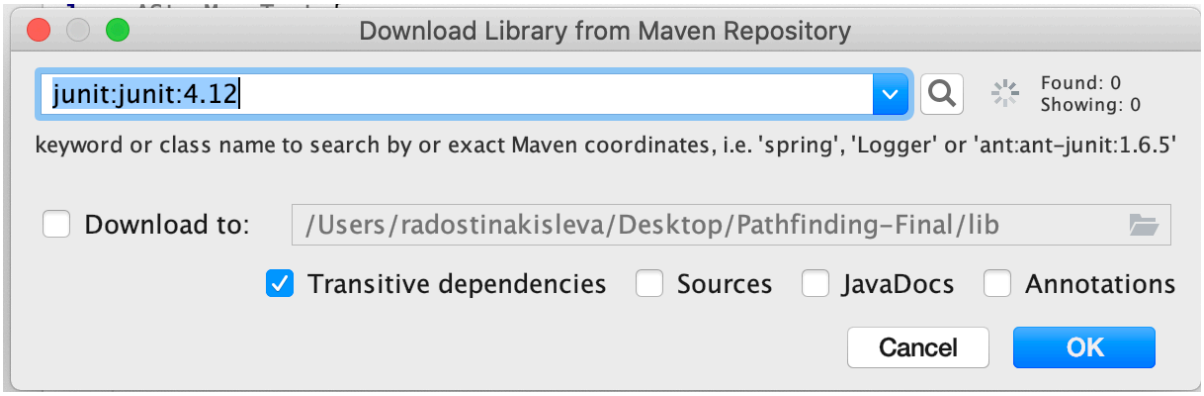


Figure 18 Download JUnit library

In the search box type **junit:junit:4.12**. After that click OK. Press the "+" again and this time type **org.junit.jupiter:junit-jupiter:5.4.2** in the search box. Click Ok then "Apply", "OK" again and exit the window. The test should now be able to be run by right clicking on the test class and choosing "Run".

4.3 Code fragments

Below I will include some code fragments of implementation features which I find interesting.

The first code fragment shows the implementation of the main loop of the A* algorithm. On each iteration, the current node is removed from the open set and added to the list of visited nodes. If the target node is found, the optimal path is

constructed. The system outputs the milliseconds for which the search was performed. The current node's neighbours are added to the queue if they have not been there already which is checked through the closed set. Their heuristic values are updated if they are not inside the open set. If the path is not found an exception is thrown which says that no path has been found.

```
while (true) {
    current = openSet.poll();

    if (current == null) {
        break;
    }

    nodesVisited.add(current);
    int currentIndex = current.getCellIndex();
    if (currentIndex == targetNodeIndex) {
        this.optimalPath = reconstructPath(current, startNodeIndex);
        finishedAtMSec = (System.currentTimeMillis() - milliStart);
        System.out.println("Searching finished. Optimal found in: " +
(finishedAtMSec) + " milliseconds");
        return;
    }
    this.closedSet[current.getCellIndex() - 1] = true;
    neighbours = current.getNeighbours();

    for (AStarNode n : neighbours) {
        updateOpenSet(current, n);
    }
}
finishedAtMSec = (System.currentTimeMillis() - milliStart);
throw new PathNotFound("Searching finished. No path found in: " + (finishedAtMSec)
+ " milliseconds");
```

The code fragment below shows the method which returns the nodes which are a part of the final path. After the current node is removed from the priority queue, the algorithm examines its neighbours and chooses the one with a better heuristic value to visit next. Each node is programmed to keep track of its predecessor or the node which led to it. Starting from the target node, the method adds the predecessor of each of the nodes to the variable path. The loop terminates when the path has reached the starting node.

```
private List<AStarNode> reconstructPath(AStarNode current, int startNodeIndex) {
    List<AStarNode> path = new ArrayList<>();

    while (current.getCellIndex() != startNodeIndex) {
        path.add(current);
        current = current.getParent();
    }
    path.add(0, current);
    return path;  }
```

The next code fragment shows a part of a method which initializes a graph. Its purpose is to calculate and assign a heuristic value to each of the nodes in the graph. First, the row number and the column number of the target node are obtained. After that, the heuristics are calculated using a nested for loop which goes through all of the nodes. The x coordinate is calculated by subtracting the row of the target node from the total row number and the y coordinate is calculated by doing the same thing but this time subtracting the target's column number from

the total number of columns. After that, the Euclidean distance is calculated, and the heuristic value of the node is set.

```
//the index of the target
int idx = getTargetIndex();

//error handling code here skipped for simplicity
int iTarget = idx / (this.columnNumber);    //row number of the target
int jTarget = (idx % this.columnNumber) - 1;   //column number

for (int i = 0; i < this.rowNumber; i++) {
    for (int j = 0; j < this.columnNumber; j++) {
        double euclideanDistance = getEuclideanDistance(iTarget, jTarget, i, j);
        this.nodesArray[i][j].setHeuristicValue(euclideanDistance); //assign the h
value

private double getEuclideanDistance(int iTarget, int jTarget, int row, int col) {
    double dx = row - iTarget;
    double dy = col - jTarget;
    return Math.sqrt(dx * dx + dy * dy); // Euclidean Distance
}
```

One interesting feature which my program has is to colour the squares of each of the visited nodes based on their heuristic value or based on how far they are from the target node. It was implemented by adjusting the values of the Color class constructor. The heuristic value of the node is taken and multiplied by a number less than 10 and then divided by 255. The remainder of this division is the value for the red part of the colour, keeping in mind that each colour consists of values for red, blue and green. The parameters for the green and blue are always 20 and 100 respectively. Since the value for the red will be different for each node, this will create a unique shade of purple. This feature is illustrated in the pictures in the next two section of the thesis. The colour of the squares closest to the target is almost blue, while the nodes further away have pinkish hues.

```
int heuristicDivider = getHeuristicDivider(mazeColNo);
Color color = new Color((int) this.heuristicMatrix[i][j].getHeuristicValue() *
heuristicDivider % 255,
        20,
        100).brighter();
drawSingleCell(g2, idx, color, "");
```

The following code fragment is a part of GenerateAndEvaluatePath.java. This is the part where the fitness function is calculated. It is done by calculating the distance remaining from the final node of each of the paths to the target.

```
float sumDistance = 0;
for (GeneticNode geneticNode : nodesVisited) {
    int nodeIndex = geneticNode.getCellIndex();
    float dist = findDistance(nodeIndex, finishNodeIndex, mazeColumns);
    sumDistance += dist;
}
float fitnessValue = sumDistance / (nodesVisited.size() * 2);
fitnessValue += findDistance(nodesVisited.get(nodesVisited.size() -
1).getCellIndex(), finishNodeIndex, mazeColumns);
```

The code fragment below shows how the final path of the genetic algorithm is drawn. For each of the nodes that are a part of the final path or in other words in the nodes visited array, the index is divided on the count of the cells. That's how

it is determined which cells to paint. The rectangle is painted in yellow and the outline is black.

```
int index, i, j;
for (GeneticNode node : geneticAlgorithmData.getNodesVisited()) {
    index = node.getCellIndex();
    i = index / cellCount;
    j = (index % cellCount) - 1;

    graphics2D.setColor(Color.YELLOW);
    graphics2D.fillRect(j * cellSize, i * cellSize, cellSize, cellSize);
    graphics2D.setColor(Color.BLACK);
    graphics2D.drawRect(j * cellSize, i * cellSize, cellSize, cellSize);
}
```

The following code fragment resides in MainFrame.java and is responsible for updating the maze size in the frame when new size is chosen. When the user chooses particular set of numbers which represent a size from the drop-down menu, the size will be collected by parsing the string to int. Then the algorithm draw type will be updated. Since here there is no algorithm, the algorithm type is No Algorithm. The maze array will be created based on the chosen size and lastly the repaint will be called by updating the maze view. If there is a drawing that is already in the frame, it will be deleted and updated according to the new information.

```
 private void updateMazeDimensions() {
    String selectedDimension = (String) cmbMazeDimensions.getSelectedItem();
    assert selectedDimension != null;
    cellCount = Integer.parseInt(selectedDimension.split("x")[0]);

    drawAlgorithm = DrawAlgorithm.NoAlgorithm;
    createMazeNodes();
    updateMazeView();
}
```

V. Testing

Testing is a very important part of the software development cycle. It is a method used in order to check whether the software product meets the expected requirements and to ensure that it doesn't have any defects or in other words "bugs" that disrupt the smooth user experience. Reliable, secure and high performing software is very highly valued. Testing can be done using manual or automated tools. The main goal of the testing process is to discover errors that exist but are not easy to spot for the developer and also to prevent from missing or not fully implementing a particular requirement.

There are different types of tests and not all testing types are applicable to all projects but depend on its nature & scope. For my project, I decided to perform acceptance and unit tests the results of which, I will present below. The way the tests were done is by executing the program with both wrong and accurate data and seeing how the software performs. The expected result when wrong data is passed is the software to produce the desired error messages in order to notify the user that something is wrong. When correct data is passed the software is expected to stick to the project's objectives and fulfil the functional and non-functional requirements. In the pages below I will analyze the program's behavior under different circumstances in order to evaluate whether the functional and non-functional requirements have been met.

Before I start, I want to say that no software is perfect, and that no developer can possibly think of all the things that can go wrong and protect against them. This is why projects undergo software quality assurance which is a practice that monitors the software engineering process and methods in order to assure the quality of the software and the conformance to certain standards. The process is extensive but luckily in my situation the project is not that big and doesn't have that many functionalities. I have attempted to address all of the errors I could think of and prevent the program from crushing as much as possible but there are most probably some bugs that I haven't thought of that are still present in my project.

There are two main parts to my project which are the graphical user interface and the implementation of the algorithms. The fact that there are two such components means that there could be bugs related to both and these bugs need to be addressed. Errors in the GUI are fairly easy to spot while errors in the algorithms and the logic are a nightmare. Luckily, in my application the algorithms are visualized in a more human-friendly way which made errors in the way they work easier to spot.

5.1 Acceptance tests

I will start with the acceptance tests that I conducted for my application. Acceptance test is a description of the behavior of a software product performed as usage scenario. It is conducted in order to access whether the software meets the expected requirements and specifications. The result of an acceptance test is either pass or fail. They are called acceptance tests because usually through such tests, the customer, client or another entity determines whether to accept the system or not. There are usually acceptance criteria that need to be met. Acceptance tests usually make use of the black box testing method and are carried out manually. When it comes to the software development life cycle, acceptance testing is the last level of software testing which is done right before the system becomes available for use and production.

The best way to perform these tests is to go through the requirements one by one and see whether the software has met these requirements. Usually in acceptance testing, the requirement might have to be split into a list of smaller features and each of these features needs to be tested separately but, in my case, most of the requirements have a small scope and usually only one feature is put under the microscope. Note that the tests below only test whether the software satisfies a particular requirement. I have left the part of how the software performs when invalid data is passed to it for the unit test part of this thesis as most of the error-handling work I have done is tested through the unit tests.

5.1.1 First and second requirement – choose size and type of maze

This requirement is implemented through drop-down menus through which the user can choose the maze size and the type of maze. There are five different choices for maze size and four different mazes for each of those sizes, so it is safe to say that the user has plenty of choices of mazes to choose from - a total of 20. The choice is made though drop-down menus, so it is impossible for the user to input wrong data at this stage. Drop-down menus offer a lot more security than input fields, for instance, where anyone can write anything. Even if the user does not click on any of the choices in the drop-down menu, there is a default option, already chosen. Errors at this stage can

occur if the user decides to go into the program files and tamper the source. They can try to add their own version of a maze, but I will talk about how the software performs when this happens later. No error will be output at this point because as I said if no size is provided, there is an option chosen by default. However, if no maze variant is provided from the second drop-down menu, the search will be performed in a map without any obstacles. I have left it like this because I think that it is interesting to see how the algorithm finds the target without any obstacles and the nature of kind of the decisions it makes which can then be compared with the path with some obstacles present.

The way to test, that this requirement is fulfilled, is manually and also visually. I had to test the application with different versions of the maze in order to check whether the drawing in the maze board changes in accord with the changes I am making. It is visually easy to spot if the drawn maze is not with the same size as chosen one. The frame changes dynamically when I make different changes which proves at least visually that the program works. However, I needed to also make sure that the drawings of the mazes are correct and that they represent accurately the type of maze the search will be performed in or in other words if they represent the two-dimensional array accurately.

In order to do this, I extracted the logic for building the maze in a different program that only deals with that and passed the two-dimensional arrays which represent mazes to it. It is easy to see whether the visualization is correct because I can easily see the positions of the 0s and 1s in the array and whether the same positioning has been maintained in the drawing of the mazes. This proves that the arrays in the application and therefore the graphs created will accurately represent what the user sees in the maze board drawing.

5.1.2 Third requirement – the user should be able to reset the maze

This requirement is also implemented through the GUI and consists of the user clicking a button in order to clear the current maze from the maze board and choose a new one. Again, this feature is easy to be tested manually as it is a visualization feature. After a particular maze has been chosen, the click of the button should make the frame turn into an empty maze grid. It is easy to see that this feature is implemented, and that the software works correctly.

5.1.3 Fourth requirement – run the A* algorithm and see the path obtained in chosen maze

Here we are getting to the more interesting parts of the application which is related to the implementation of the searching algorithms. The requirement can be rephrased in another way which is:” the program should have a working implementation of the A* algorithm.” The implementation of a complex algorithm consists of many parts and many things can go wrong. One thing to keep an eye for here are logic errors which can be really frustrating and heavily disrupt the development process. It is one thing to read about how an algorithm works and another to implement it with code.
In the grand scheme of things, it can be said that the user is indeed able to run the A* algorithm and see the path obtained by it in the chosen maze because it is easily noticeable in the frame. We can see that the target is reached, and a path is drawn to

it starting from the start. In the earlier stages of development, the program worked without a visualization tool and at first, I only saw the output produced by the algorithms as simple strings in the console which informed me what nodes have been visited by showing their index and indeed the index of the last visited node corresponded to the index of the target. Later, it became even easier to check that the algorithm indeed works by adding the visualization tool which shows the path obtained from the start to the target as well as the visited nodes.

The visualization makes it easy for anyone to see how the algorithm works without having any knowledge about graphs, nodes and heuristics. It also makes testing a lot easier as errors can be spotted immediately instead of writing numerous printing statements in order to see where the software fails.

The requirement does not explicitly state that the path to the target needs to be obtained as long as the search results in some kind of a path. I have tested the algorithm in many different environments and mazes, and it was always able to find a path to the target as long as such a path exists, and the target is not blocked in any way. My testing was hindered by the way I was creating the mazes which was manually, and which took a long time. Because of this reason, I was not able to test how the algorithm performs in mazes with a bigger size than 40x40. However, based on what I read in scientific research done on the A*, the algorithm has a pretty high success rate when it comes to finding paths in mazes with a size up to 100x100. I did test the algorithms with the 20 provided mazes as well with others that didn't make it to final program and the paths obtained by the algorithm were pretty good and close to the ones which a human would choose, keeping in mind that diagonal moves are not allowed. Based on all of this, I can conclude that the project has a working implementation of the A* algorithm.

The second part of this requirement is to make sure that the algorithm performs the search in the chosen maze. Again, this is easy to test with the visualization tool that the application offers. In the windows where the search is performed, I can see that the maze is exactly the same as the chosen one with every single maze chosen which proves that the tools for choosing the maze works correctly and the correct maze is given to the algorithm.

5.1.4 Fifth requirement – run the genetic algorithm and see the path obtained in chosen maze

Besides the A* algorithm, the program should also have a working implementation of the genetic algorithm. Similar to what I said in the previous point, the visualization part of the program makes it easier to visualize that the algorithm is indeed running and working. After each iteration of the main for loop of the genetic algorithm, we can see in the console and on the screen, the fittest individual and its fitness value. When the target is found, the program outputs the fitness value of the fittest individual. When the path cannot be found after ten iterations with the same fitness value, the controlling parameters of the genetic algorithm are changed automatically. Each of the chromosomes, also called solutions, which in my case are paths is displayed on the screen in the maze even if it is not the path to the target. The display of each of the paths shows visually how the how the genetic algorithm works and what decisions it

makes which can be used for debugging purposes even though it takes some more time now until the path is found.

Similar to A*, the genetic algorithm must be supplied with a maze in the correct format. After that, the algorithm sequentially performs the needed operations without any “help” from the user. The performance, similar to the case with A*, depends on how far the start and the target are positioned relative to each other. Logically the further they are, the more of the maze area needs to be traversed in order to get to the target. In addition to that, when it comes to the genetic algorithm, the performance is heavily influenced by some controlling parameters and their values are important for how the algorithm will run. These parameters also need to be tested with different combinations of their values in order to be sure that the genetic algorithm will work properly and have a satisfactory performance. I decided that the best place to describe the process of testing the controlling parameters is in this section. I will present my findings below. They are particularly tailored to my application and it is not guaranteed that these parameters will work in other applications which use the genetic algorithm.

In my implementation of the genetic algorithm I used single point crossover, binary encoding of the chromosomes and tournament selection. It was difficult to find the right parameters in order to optimize the performance and even though I found some useful resources online, most of the work had to be done by trial and error in order to see what parameters work best in the different kind of mazes and which combinations give the fastest and best results. The controlling parameters in my implementation of the GA are the following: population size, length of the genes, maximum limit of generations, tournament size, crossover rate, mutation rate and a Boolean value which indicates whether elitism is allowed or not.

The resource under [16] notes that there is still no general theory which would outline the parameters for the genetic algorithm for any problem but there are some useful recommendations which were discovered as a result of some empiric studies of GAs which were conducted with binary encoding which applies to my case. It is recommended that the crossover rate is relatively high – around 80-95%. The **crossover rate** shows how often crossover will be performed. If it is not performed, the offspring is an exact copy of one of the parents. If the crossover rate is 100%, it means that all offspring will be made from crossover. If the crossover rate is 0% then the new population will be a copy of the solutions from the old population. The **mutation rate** should be very low – around 10%. This rate shows how often parts of the chromosomes will be mutated. If there is no mutation, offspring will be copied after crossover without any change. If the mutation rate is 100%, the whole chromosome will be changed whereas if it is 0% nothing will be changed. The mutation rate is used in order to prevent the GA to fall into a local extreme but if it is performed too often the search may turn into a random search. That’s why it should be used in moderation.

The **population size** indicates how many chromosomes are in one population. If there are not enough chromosomes, the generic algorithm has less possibilities to apply crossover and thus it will not be able to explore much of the search space which will damage the search behavior. However, if there are way too many chromosomes in a population, the genetic algorithm slows down. The source notes that when it comes to the population size, it is rather puzzling to observe that big population size does not improve the performance of the genetic algorithm when it comes to the speed of finding

solutions. **Elitism** should be used when there is no other method for saving the best chromosomes. The **length of the gene** is 2 times the node path length as each pair of genes indicates a movement direction. The **maximum limit of generations** indicates for how many iterations the main loop will run. For the rest of the parameters, I either couldn't find appropriate recommendations or couldn't find suggestions that fit my problem, so I had to find the right values completely by myself, using the trial-and-error method. After many runs of the genetic algorithm which are performed, here are the most appropriate values for each of the parameters based on the size of the maze.

	Population Size	**Length of Gene**	**Maximum limit of generations**	**Tournament Size**	**Crossover Rate**	**Mutation Rate**	**Allow elitism**
15x15	100	200	400	30	0.75	0.15	yes
20x20	200	200	400	30	0.85	0.15	yes
25x25	300	300	600	30	0.85	0.25	yes
30x30	300	400	800	30	0.95	0.25	yes
35x35	300	500	800	30	0.95	0.25	yes

I discovered that in order to improve the performance of the algorithm the length of the gene and the maximum limit of generations should be increased. The speed and time for which the solution was found really did increase as it was noted in the source. For bigger mazes, the length of the genes should logically be more considering that if it not enough, the target will not be able to be reached psychically because there will not be enough genes to reach it. Increasing the population size, didn't really make a different for the speed or the performance of the algorithms, probably because the mazes I used are not too large. The number of generations should comply, to an extent, with the computational resources of the specific machine because it can put pressure on it if the number is too large. When it comes to the mutation rate, if the maze is too complicated or big, increasing the mutation rate means that if the algorithms gets stuck at some point for example at a corner, adding more random genes might lead to more random directions which is what we are looking for. In my case, it helped to find the solution more quickly, but we can see more random behaviors and movement directions in the final path which results in longer paths in the end. Increasing the number of genes means that the algorithm is given the chance to explore more areas of the graph. This was crucial when tuning the parameters for the different size of the maze.

5.2 Unit tests

As I already mentioned, the second type of tests that I used in order to validate that my program is working correctly is unit tests. A software test is a piece of software that executes another piece of software in order to confirm that the code produces the expected result (state testing) or the expected sequence of events (behavior testing). Unit tests are used to validate that a piece of the whole program works correctly. When tests are set up to run automatically, this helps to recognize regressions introduced by

changes in the source code. Having high test coverage helps to ensure that adding new features will not require an extensive number of manual tests. Unit tests specifically are a piece of code that executes specific functionality to be tested and asserts a certain expected behavior or state. A unit test tests a small portion of the code which could be a method or a class, for example [17].

In order to write the unit tests, I have used the Junit 5 testing framework for Java. The tests are created in a separate source folder in order to keep the test code separate from the real code. The folder with tests in my project is named “tests” and is separated from the main “src” folder. I have written software for critical parts of the application which might create problems and make the program fail or crush and also for testing the logic of the algorithms. Before I start explaining the tests, I want to mention that before starting to write these tests, I had to perform code refactoring and separate some of the functionalities that I wanted to test in separate methods. This is required so that they can be used for unit testing. Ideally, this would be done during the development process when the developer knows exactly what parts of the code need to be tested.

The first set of tests is related to the mazes, their creation and problems that might arise with them. The maze in this program represent the data that the algorithms need in order to run. One of the things to consider is whether the program logic works correctly, and the graph is properly initialized from the provided maze. This is a concern of the developer and problems might occur due to logical errors.

The other aspect of working with mazes has to do with making sure that the mazes which are input are in the correct format which is a concern of the users of the application and a concern of the application to handle errors appropriately. As I have already briefly mentioned in other parts of this thesis, I have insured that as long as the user sticks to the designed user interface and chooses one of the mazes which I have provided, no errors can occur as the available mazes are correct and are suitable to be traversed by the algorithms. However, that doesn’t mean that error handling should not be implemented in the project because if someone obtains the source code and decides to tamper with it by adding their own custom maze, a big mess can occur very quickly. Also, if any other developers work on this project in the future, it is important for them to know what rules they should follow and what not to do. A user of this application needs to be able to know what mistake they have made and be able to correct it in the future. Ensuring proper error handling makes it easier for the user to use the code and also makes the code easier to maintain.

In the next few pages, I will first explain a particular error and then check whether the software reacts as expected in case of occurrence of the error through unit tests.

The first set of unit tests are concerned with the logic of the application and whether the nodes, graph, start and target index are properly set and initialized from the provided maze. The first unit test checks whether the application works correctly and whether the array with nodes is property initializes from the provided maze array. The way to check this is to make sure that the number of rows and columns of the graph and the maze array match. Here is the test for the GA, but there is the same one written for A* as well.

```
@Test
void test_maze_nodes_array() throws Exception {
    GeneticMaze maze = new GeneticMaze(GetMazeArray.Maze(20, 1));
    assertTrue(maze.getNodeArray().length > 0);

    assertEquals(maze.getNodeArray().length, maze.getRowNumber());
    assertEquals(maze.getNodeArray()[0].length, maze.getColumnNumber());

    System.out.println("Row: " + maze.getRowNumber());
    System.out.println("Column: " + maze.getColumnNumber());
    System.out.println("Nodes row: " + maze.getNodeArray().length);
    System.out.println("Nodes Column: " + maze.getNodeArray()[0].length);
}
```

For this test, one of the maze arrays already in the program is used. asserEquals() checks whether the array with nodes has the same number of rows as the maze array. Then the numbers are output. The screenshot below shows that the test has passed, and the nodes array has been correctly initiated with the right number of rows and columns.

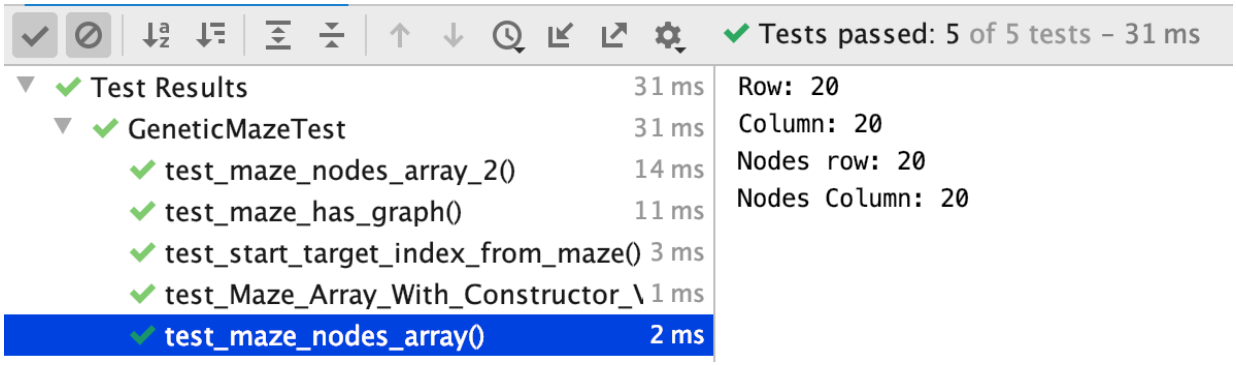


Figure 19 Result of JUnit test of the nodes array

The next test is connected to this one and is very short. It checks whether the nodes array is null when an empty maze array is provided. This test passes as well as you can see from any of the screenshots and the nodes array is indeed null.

```
@Test
void test_maze_nodes_array_2() throws Exception {
    AStarMaze maze = new AStarMaze(new int[][]{}, 400);
    assertNull(maze.getNodeArray());
}
```

The next test asserts whether the graph that is created from the list of nodes is properly initialized. The graph nodes count should be equal to the size of the maze (row * column). Here is the code for test for the A*, but there is the same one written for the GA, as well. In this test, first, the maze is initialized from one of the mazes provided in the program. The number of values in that maze is output which is calculated by multiplying the number of rows by the number of columns. It has size of 20x20, so the expected number here is 400. Then, the test asserts whether the size of the graph is bigger than 0 and asserts whether the size of the maze and the size of the graph are the same. There is a second part of this test which checks whether when an empty maze is provided, the graph is also empty. If the logic is correct, the expected result is that the graph will be null.

```
 @Test
void test_maze_has_graph() throws Exception {
    AStarMaze maze = new AStarMaze(GetMazeArray.Maze(20, 1), 400);
    System.out.println("Maze array count: " + maze.getRowNumber() *
maze.getColumnNumber());

    assertTrue(maze.getGraph().getNodes().size() > 0);
    System.out.println("Graph nodes count: " +
maze.getGraph().getNodes().size());

    assertEquals(maze.getRowNumber() * maze.getColumnNumber(),
maze.getGraph().getNodes().size());

    maze = new AStarMaze(new int[][]{}, 400);
    assertNull(maze.getGraph());
    System.out.println("Graph nodes count_2: " + 0);
}
```

Here is the result of the unit test and as you can see the test passes and the program performs correctly. The error message visible in the screenshot is produced when the maze array is empty.

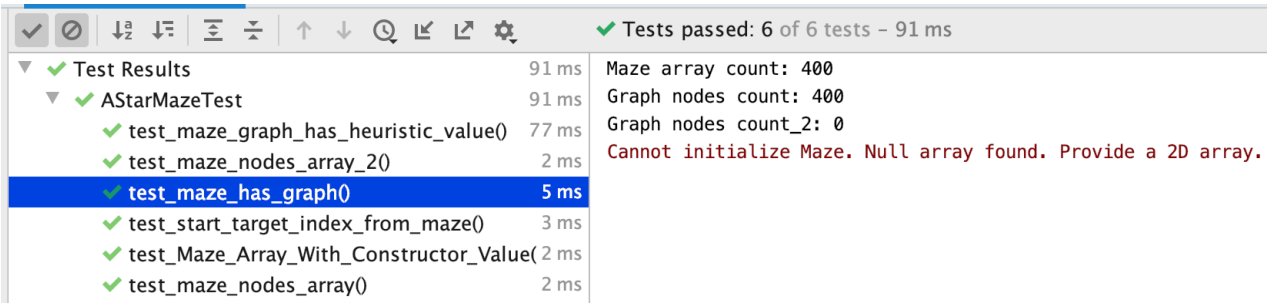


Figure 20 Result of JUnit test of whether graph exists

The next piece of logic to check is whether the start index and the target index are correctly set when the maze array is not null and is in the correct format and also when the maze array is null or in other words when a maze is not provided at all. Here is the unit test for the GA, and there is a similar one for the A* as well.

```
@Test
void test_start_target_index_from_maze() throws Exception {
    GeneticMaze maze = new GeneticMaze(GetMazeArray.Maze(20, 1));

    assertTrue(maze.getStartIndex() >= 0);
    System.out.println("Start index_1: " + maze.getStartIndex());

    assertTrue(maze.getTargetIndex() >= 0);
    System.out.println("Target index_1: " + maze.getTargetIndex());

    maze = new GeneticMaze(new int[][]{});

    assertEquals(-1, maze.getStartIndex());
    System.out.println("Start index_2: " + maze.getStartIndex());
    assertEquals(-1, maze.getTargetIndex());
    System.out.println("Target index_2: " + maze.getTargetIndex());
}
```

First, the maze is initialized, and it is checked whether the start index exists. The index is printed out. The same procedure is done for the target index as well. Then, an empty maze is provided, and the start and target indices are printed out as well. The expected result is that they will be -1 since that is the default number assigned to the start and target which changes when new values, taken from the array, are assigned to them. Here is the result from the unit test. As you can see, the program works correctly, and the start and target indices are properly assigned when they are provided. If they are not, which is the case with an empty maze, their values stay -1 which is the default.

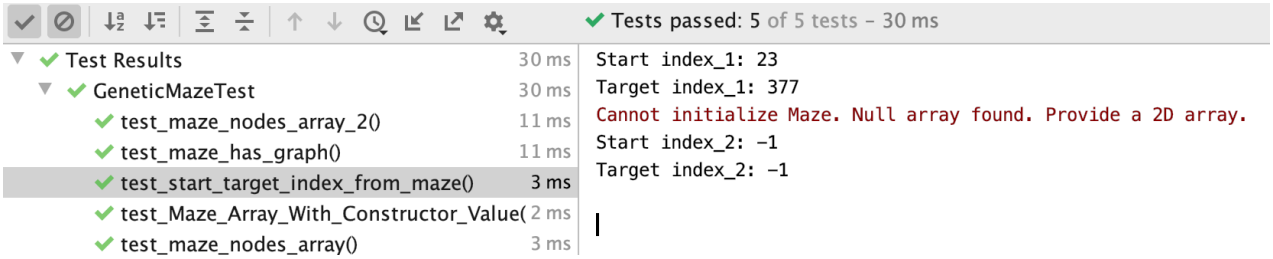


Figure 21 Result of JUnit test of whether start and target are set

The next test will examine whether the heuristic value is set for each of the nodes of the graph when running the A*. The heuristic value is very important for the algorithm and it is crucial to see whether it is properly set. For each node of the graph, it is asserted whether the heuristic is greater than 0 and then it is printed.

```
@Test
void test_maze_graph_has_heuristic_value() throws Exception {
    AStarMaze maze = new AStarMaze(GetMazeArray.Maze(20, 1), 400);
    for (AStarNode node : maze.getGraph().getNodes()) {
        assertTrue(node.getHeuristicValue() >= 0);
        System.out.println("Node: " + node.toString());
    }
}
```

Here is the output of the test. As you can see, it is true that the heuristic value is set and is greater than 0. Looking at the values, I can also see that the nodes closer to the target have the smallest heuristic values which leads me to believe that this piece of logic is working correctly. The number of neighbors is also correct. The rest of the values are still not set because the A* algorithm hasn't run yet.

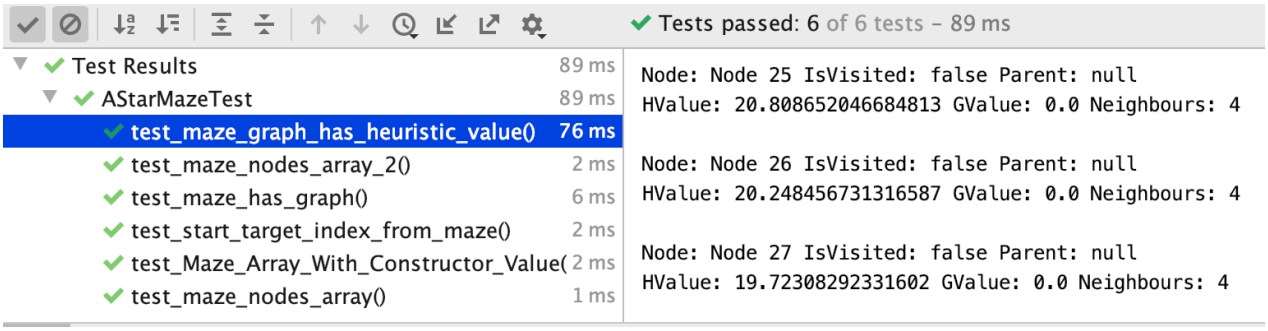


Figure 22 Result of JUnit test of whether heuristics are set

The second set of unit tests, connected to the maze, have to do with errors in the maze input which means that there is something wrong with the two-dimensional array given to the program which is a mistake made by the user that the program should catch

and inform them about. The first such error to “handle” is the input of an empty maze or trying to run the algorithms with no maze. The program should notify the user if this happens. Below is the unit test and there is one such test written for the A* and one for the GA. They are identical with the exception that a different constructor is used and because of that I have only included one of them.

```
    @Test
    void test_Maze_Array_With_Constructor_Value() throws Exception {
        AStarMaze maze = new AStarMaze(new int[][]{}, 400);

        System.out.println("Row: " + maze.getRowNumber());
        System.out.println("Column: " + maze.getColumnNumber());

        assertNull(maze.getGraph());
        assertNull(maze.getNodeArray());

        assertTrue(maze.isMazeArrayEmpty(maze.getMaze()));
    }
```

In this test, the maze is initialized with an empty array. Then the number of rows and columns is output which is expected to be 0. assertNull() checks whether the object is null. The method isMazeArrayEmpty() returns true if the maze array is empty. Here is what happens when the test is run. As we can see, the test passed. The number of rows and columns is 0 when the maze is empty, and an error message is output which informs the user what error has occurred.

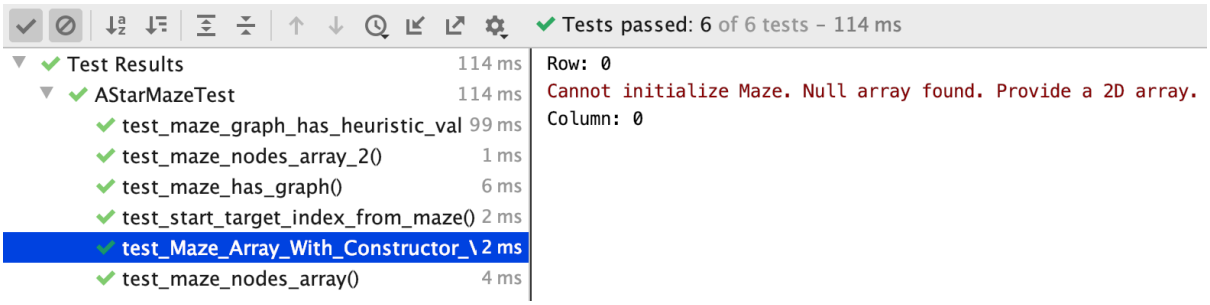


Figure 23 Result of JUnit test to test maze constructor

Here is the error message printed in the frame for the user to see.

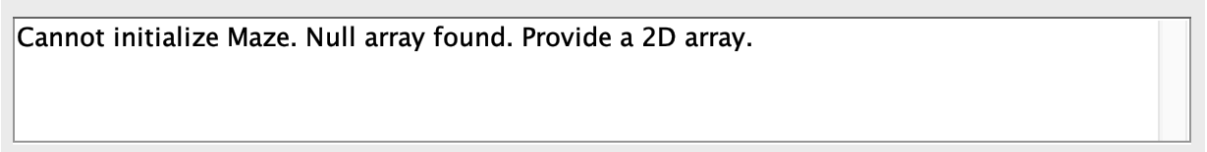


Figure 24 The output in the panel when no maze is provided

The next error has to do with inputting a maze with no target. The algorithms cannot work in such a maze. Here is the unit test which I wrote in order to check if the proper exception is thrown.

```
@Test
void test_optimal_path_with_no_target()  {
    try {
        AStarAlgorithm algo = new AStarAlgorithm(testMaze_1, 400);
```

```
        AStarMaze maze = algo.getData().getMaze();
        assertEquals(1, maze.getRowNumber());
        assertEquals(testMaze_1[0].length, maze.getColumnNumber());
        assertNotNull(maze.getGraph());
        assertEquals(0, maze.getGraph().getNodes().size());
        assertEquals(-1, maze.getTargetIndex());
        algo.performSearch();
        assertNotNull(algo.getData().getOptimalPath());
        assertEquals(0, algo.getData().getOptimalPath().size());
    } catch (TargetNodeNotFound | StartNodeNotFound | PathNotFound |
EmptyMazeArray ex) {
        ex.printStackTrace();
    }
}
```

The algorithm is given a maze without a target which I created specifically for the test. First, it is checked whether the graph has the same size as the given maze. Then searching is performed in this graph. Since there is no target, the size of the final path is expected to be 0 and an exception is to be thrown. Here is the output of the test. It works correctly.

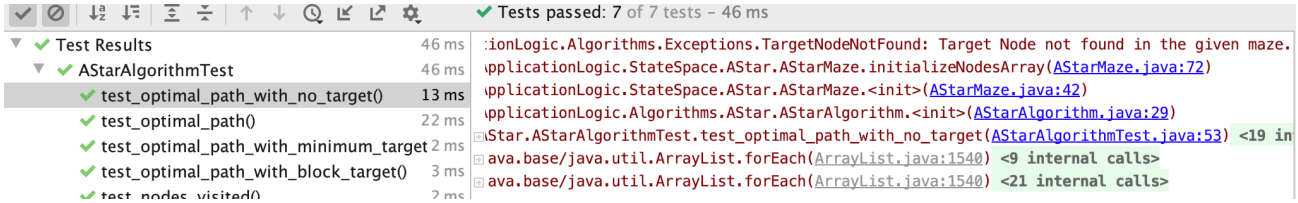


Figure 25 Result of unit test when no target is found

And here is the error, given to the user in the display of the frame:

Figure 26 The output in the panel when no target is found

The next unit test shows what happens when the target is blocked by walls and there is no way to get to it. In this case, a final path cannot be obtained because such a path doesn't exist. Below is not the whole test but a fragment of it as the test is quite long. The algorithm is given a maze that has walls around the target. After the search is performed, it is asserted whether the length of the optimal path is 0 or less. The result printed is correct, the final path has a size of 0 and an exception is thrown.

```
void test_optimal_path_with_block_target() throws Exception {
    AStarAlgorithm algo = new AStarAlgorithm(testMaze_3, 400);
    AStarMaze maze = algo.getData().getMaze();
    …
    assertEquals(13, maze.getTargetIndex());

    try {
        algo.performSearch();
    } catch (PathNotFound pathNotFound) {
        pathNotFound.printStackTrace(); }
    assertTrue(algo.getData().getOptimalPath().size() <= 0);
```

```
    System.out.println("Optimal path size: " +
algo.getData().getOptimalPath().size());
}
```

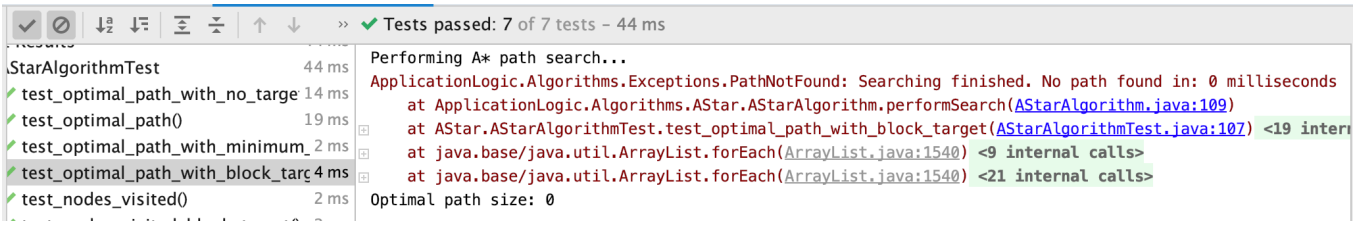


Figure 27 The result of the unit test when the target is blocked and cannot be reached

The next few unit tests test the logic of the algorithms once again but this when the maze input is correct. The first test verifies whether some nodes are visited after the search is performed. The test asserts that the nodes visited array is not null and prints the size.

```
@Test
void test_nodes_visited() throws Exception {
    AStarAlgorithm algo = new AStarAlgorithm(GetMazeArray.Maze(20, 1),
400);
    try {
        algo.performSearch();
    } catch (PathNotFound pathNotFound) {
        pathNotFound.printStackTrace();
    }
    assertNotNull(algo.getData().getNodesVisited());
    System.out.println("Nodes visited: " +
algo.getData().getNodesVisited().size());
}
```

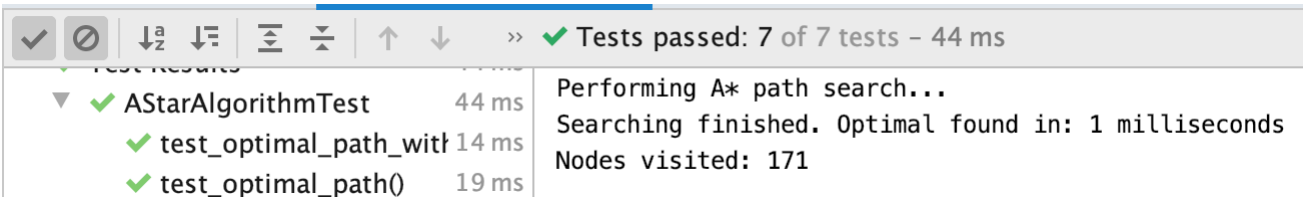


Figure 28 Result of test when the input is correct. Nodes visited count shown.

The following test examines whether a path is found by the algorithm when the input is correct, and the algorithm runs. First the search is performed and then it is asserted whether the final path is not null, and its length is bigger than 0. Here is the result. The algorithm indeed finds a path.

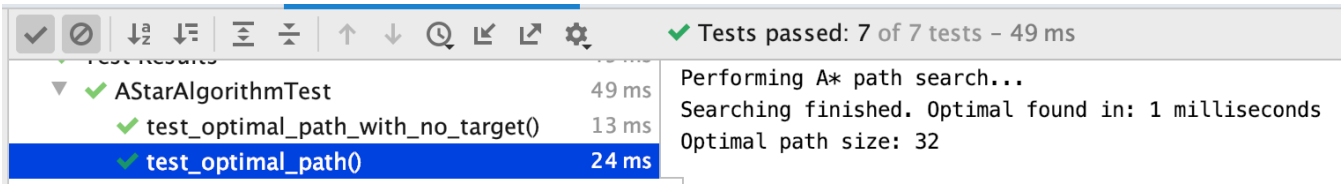


Figure 29 Result of test when the input is correct. Final path nodes count shown

I have written more unit tests which can be found in the source code of the project which unfortunately I will not be able to discuss here due to the page restriction of this thesis.

VI. Results and conclusion

My biggest achievement with this project, in my opinion, is the implementation of the two searching algorithms and applying the implementation to the pathfinding problem in a static maze-like environment. I was able to successfully visualize the search in the maze by displaying the final path obtained, utilizing animation and other features which aim to show the inner workings of the algorithms. I was also able to implement a user interface which allows the user to control the environment in which the search is performed and be able to visualize it more easily. I will discuss the final results obtained by each of the algorithms. The performance of the algorithms will be reviewed in the third part of this section where I considered it appropriate to also compare the performance of the algorithms with one another.

6.1 A* algorithm results

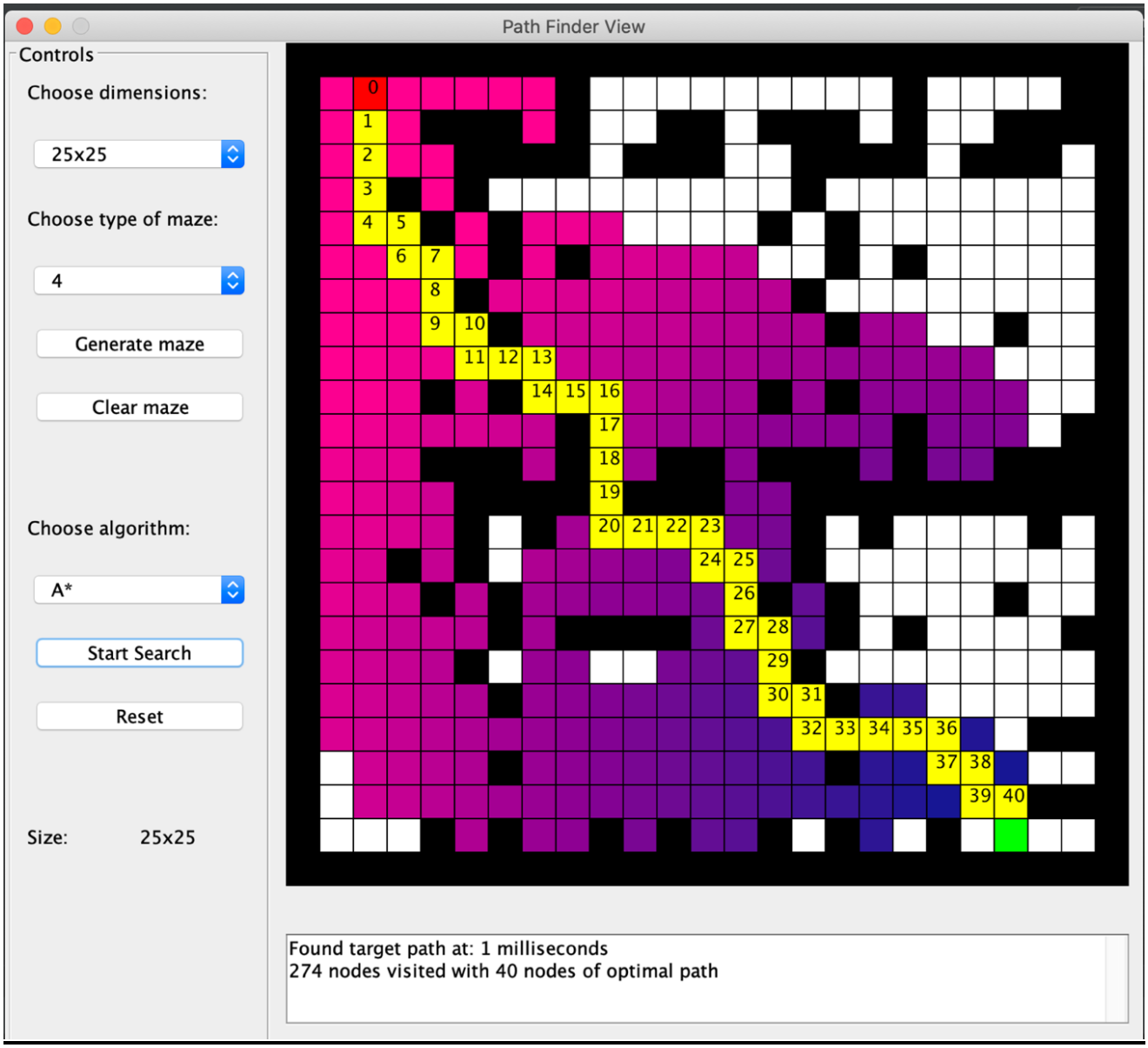


Figure 30 That the maze looks like after the A has run. The purple squares are the nodes visited and the yellow squares form the final path*

The frame above visualizes the final result obtained from the search done by the A* algorithm. The A* start is able to produce good results, closer to the shortest path in a reasonable computational time. The nodes visited are visualized in the maze as purple squares while the optimal path is visualized in yellow. The time the search took, and the length of the final path are displayed as well.

The tests show that the algorithm managed to adapt itself in different environments with different obstacles, starting point and ending point. The algorithm was able to reach the target node in 100% of the test cases and was able to solve the problem quickly in all given situations.

6.2 Genetic algorithm results

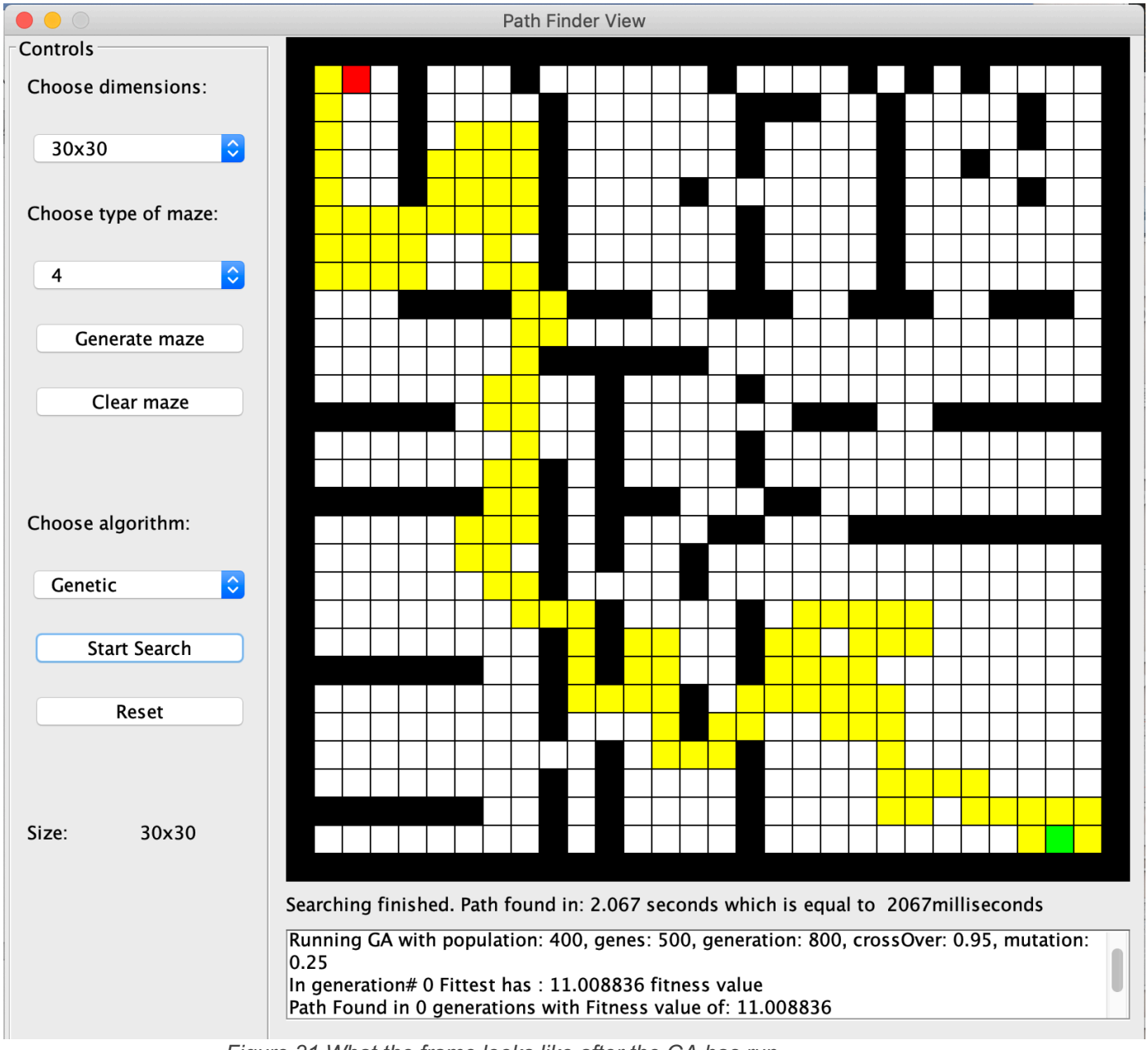


Figure 31 What the frame looks like after the GA has run

The window above shows the final result after the search obtained by the GA has finished. It was able to produce a result in mazes with different sizes, with different density of the obstacles and with differently positioned start and target. Each time the algorithm runs, the controlling parameters, the number of the current generation, the fitness value of the individual and the path in the frame are displayed.

The algorithm runs with different combinations of the controlling parameters until the final path is obtained. If a particular combination of parameters has not led to the target, a message is printed, and the search continues automatically with different values. When the search is finished, the length of the path and the time for which the algorithm run are displayed as well as the fitness value of the solution. The paths obtained from the genetic algorithm are not the shortest paths but are better than a random search.

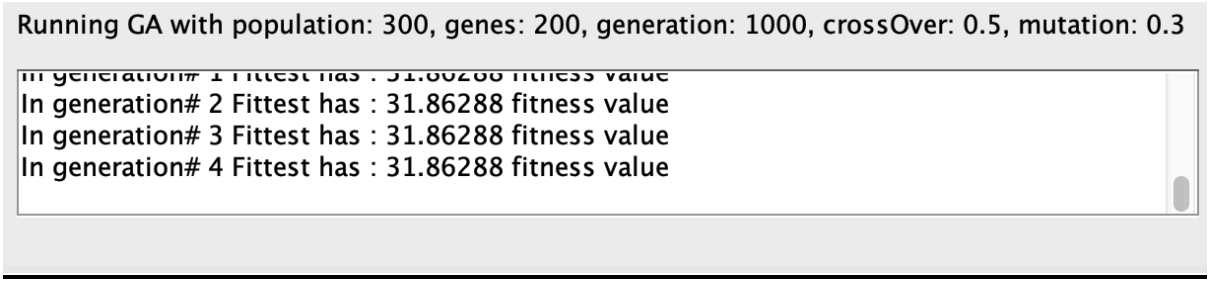


Figure 32 The output in the panel while the GA is running. Shows the GA parameters and fitness values.

6.3 Discussion about the algorithms' performances and comparison

The parameters which I will use in order to compare the algorithms are the time it takes to find the solution in milliseconds as well as the length of the acquired final path. The table below shows the mean of the path length and the time the search took for the two algorithms. The results were obtained after running each of the algorithms in four mazes of a particular dimension. For example, the A* was run in four mazes with size 15x15 and the mean was recorded. After that, the genetic algorithm was run in the same four mazes and the mean was recorded. The procedure was repeated for the five different maze sizes. Here are the obtained results:

	A*		Genetic	
	Average Time	Average Path Length	Average Time	Average Path Length
15x15	0.75 milliseconds	24 nodes	242 milliseconds	40 nodes
20x20	1 millisecond	31 nodes	586 milliseconds	49 nodes
25x25	1 millisecond	41 nodes	1339 milliseconds	73 nodes

3030	1 millisecond	53 nodes	3164 milliseconds	84 nodes
35x35	2 milliseconds	59 noes	8000 milliseconds	111 nodes

Results show that the genetic algorithm has significantly worse computational time compared to the A* which is not surprising considering that the A* progressively and gradually builds the final path while the genetic algorithm performs local search to improve an existing solution and the time heavily depends on the number of generations. As the maze gets larger, the computational time further drops down. The algorithms perform well when the size of the maze is not very large. The density of the maze mattered as well since with less obstacles the algorithms had more pathways to progress through which is important particularly for the genetic algorithm. Big mazes with a particularly high level of density encompass much more dead ends which result in early convergence of the GA. It is eventually able to find a path, but it takes more iterations and requires bigger length of the genes and maximum limit of generations. It is also important not to forget that the genetic algorithm uses a lot of random numbers while searching. Sometimes very different values in terms of time are produced in the same maze after two consecutive iterations. In order to obtain more precise results and be able to draw stronger conclusions, the algorithm need to be run a lot more than 4 times in each maze size. Unfortunately, I do not have the knowledge of how to use any testing software and perform automatic tests, so I had to perform the search manually which is very time-consuming. An idea for future development is to build a testing software which runs the GA hundreds of times and at the end produces a result about its performance but that is far beyond the current scope of this project.

The A* seems to be optimal and efficient. The results it obtains are satisfactory and seem to be close to the shortest path, keeping in mind that diagonal movements are not allowed. One noticeable disadvantage of the A* is that the number of nodes it visits in order to get to the target is still high and grows exponentially with the size of the maze.

The results obtained from the above table show that the A* is able to provide a lot better results than the genetic algorithm in terms of time. When it comes to path lengths, the results obtained from the GA are not as poor relative to those concerned with the time.

The results obtained from the GA are not particularly satisfactory and indicate that when it comes to solving mazes, the A* is a much better choice. The main reason for this is the fact that the steps the genetic algorithm needs to go through are very time-consuming and extensive in order to be applied for the pathfinding problem where optimal time and path length are important. However, the results obtained from the genetic algorithm would certainly be better in comparison with blind searching algorithms.

6.4 Conclusions

I can say that the final result exceeded my initial hopes. At the beginning, my main goal to achieve was to implement the searching algorithms and obtain good results from them. I was hoping for some kind of visualization as well, but I wasn't sure what was going to come out of it. Even though the user interface and the drawings are relatively simple now, they are extremely useful and turned out to be a significant part of the project and increased significantly its complexity. They certainly exceeded any expectations I had initially. As the time passed during the semester, I kept thinking of new features and things to improve and add to this project. I didn't expect that the user interface would turn out the way it looks now with the user being able to control what they use and see to this extent.

What I created initially were two applications was consisted of the implementations of the two algorithms and a console output of the visited nodes obtained from the search. Then I was able to create a single maze for each algorithm and illustrate the path in it. Then I gradually made the maze and path drawings look more and more sophisticated. I was able to add animation for the A* and show visually the heuristic value of each of the nodes and the manner in which the algorithm itself performs the search with the nodes visited first appearing first. For the GA, I was able to show all paths obtained by it until the final one is found which also looks like an animation with the paths switching in real time as the parameters are changing at the same time at the panel below the path. These features illustrate graphically how these complex algorithms work and how they are able to make decisions. I greatly improved the searching time for the genetic algorithm by tuning the parameters and finding appropriate values for each maze size. Finding the right values and combinations for six different parameters in five cases was definitely a very time-consuming task and the work done was not trivial at all. It was important to find appropriate values because these parameters are very important for the favorable time of the algorithms. They should be large enough to ensure the genetic diversity but not too large because then the computational costs will be enormous.

The last addition was the frame with the buttons, drop-downs, panels and labels which added a lot of additional functionalities to the application and made it a lot more complex. It was not easy to add such a complex user interface without having it in mind at the beginning and planning for it. The code to be refactored numerous times and it ended up looking a lot different than it did at the begging.

The problems that I faced were numerous, but I managed to solve a lot of them which is the best learning experience. They mainly appeared due to my unfamiliarity with the algorithms and their complexity. Even though I was familiar with other searching algorithms, the genetic algorithm was completely new for me and it is a rather big bite for a beginner. I had to read a lot of theory in order to get my head around those concepts. Another issue was the fact that I first developed the two algorithms separately and then had to merge the two programs into what you see now. It took a lot of tedious hours of refactoring and figuring out how to display them both in one frame which changes size and is able to display different things according to the chosen algorithm. If I had started with the current user interface or somethings similar, I could have saved myself a lot of hours of refactoring work.

Even though, I am very satisfied with the result, there is always room for improvement. I had many ideas for improvements during the development process but unfortunately due to the time restrictions, I was not able to implement them. The first thing was to improve the fitness function of the genetic algorithm in order to improve its performance and the searching behavior. An interesting improvement to the genetic algorithm that I read about in one of my sources and wanted to try to implement was to use a method known as many islands algorithm which divides the population into groups, prohibits reproduction between groups and thus allows different genetic strains to survive that might otherwise strive out. This is done in order to maintain the diversity of the population and in this way improve the possibility of reaching an optimal solution to the problem [18].

The resource also discussed other improvements that could be applied to the primary structure of the genetic algorithm but applying these techniques required more extensive research on my side and considerable amount of time in order to think of ways to implement them in my solution which is understandable considering the fact this was my first time coming face to face with the genetic algorithm. I admit I have a lot more to learn and experiment in order to be comfortable with more complex concepts related to it.

I had the idea of implementing a random maze generator which would be able to generate mazes with different size and densities, but the idea failed because I did not have enough time to work on it. My even more ambitious plan, when it comes working with the mazes, was to be able to allow the user to draw their own maze by clicking on a square in the grid and thus to be able to make it a “wall” or a “gap”. I even started implementing it, but I soon realized that it is much harder than I had initially thought, and I would simply not have enough time to do it.

In terms of the knowledge that I acquired, I have learned a plethora of new things. Starting from graph theory, going through implementation of searching algorithms and creation of graphical user interface in Java and ending at unit testing. Even though there is room for improvement when it comes to the implementation of the algorithms, the process was very educational and extremely useful for me. I gained more confidence as a developer because I was able to solve many different problems that I encountered during these three months. I was able to dive deeper into topics I have not had the opportunity to explore before, read a lot and do a lot of research in order be able to apply theoretical concepts to a real-world problem.

VII. References

[1] M. Nazarahari, E. Khanmirza, S. Doostie. “Multi-objective multi-robot path planning in continuous environment using an enhanced genetic algorithm”, *Expert Systems with Applications,* Volume 115, 2019, Pages 106-120, https://doi.org/10.1016/j.eswa.2018.08.008.

[2] X. Cui, H. Shi. “A*-based Pathfinding in Modern Computer Games.”, 2010.

[3] Lin, Y. C., Su, K. L., & Chang, C. H, “Development of the searching algorithm with complexity environment for mobile robots.” *Applied Mechanics and Materials, 284-287*, 1826, 2013, doi:http://dx.doi.org/10.4028/www.scientific.net/AMM.284-287.1826

[4] Hong-Mei, Z., Ming-Long, L., & Yang, “Safe path planning of mobile robot based on improved A* algorithm in complex terrains”. *Algorithms, 11*(4), 44, 2018. doi:http://dx.doi.org/10.3390/a11040044

[5] D. Maddi, A. Sheta, A. Mahdy, and H. Turabieh, “Multiple Waypoint Mobile Robot Path Planning Using Neighbourhood Search Genetic Algorithms”. *Association for Computing Machinery*, New York, NY, USA, 14–22. doi:https://doi.org/10.1145/3388218.3388225

[6] R. Leigh, S. J. Louis and C. Miles. “Using a Genetic Algorithm to Explore A*-like Pathfinding Algorithms”, 2007 *IEEE Symposium on Computational Intelligence and Games,* Honolulu, HI, 2007, pp. 72-79, doi: 10.1109/CIG.2007.368081.

[7] B. Li, Chaoyi Dong, Q. Chen, Y. Mu, Z. Fan, Q.Wang, and X. Chen. 2020. “Path planning of mobile robots based on an improved A*algorithm.” *Association for Computing Machinery*, New York, NY, USA, 49–53. doi: https://doi.org/10.1145/3409501.3409524

[8] A* Algorithm pseudocode. https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf

[9] C. Lamini, S. Benhlima, A. Elbekri, “Genetic Algorithm Based Approach for Autonomous Mobile Robot Path Planning”, *Procedia Computer Science*, Volume 127, 2018, Pages 180-189, https://doi.org/10.1016/j.procs.2018.01.113.

[10] R. Kala, A. Shukla, and R. Tiwari. 2009. “Robotic path planning using multi neuron heuristic search”. *Association for Computing Machinery*, New York, NY, USA, 1318–1323. doi: https://doi.org/10.1145/1655925.1656167

[11] Z. Abd Algfoor, M. Shahrizal Sunar, H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games", *International Journal of Computer Games Technology*, vol. 2015, Article ID 736138, 11 pages, 2015. https://doi.org/10.1155/2015/736138

[12] What is Java Swing? – Definition from Technopedia. *Technopedia, Inc*. Retrieved 2020-11-08. https://www.techopedia.com/definition/26102/java-swing

[13] Russell, S. J., & Norvig, P, “Artificial intelligence: A modern approach.” Englewood Cliffs, N.J: Prentice Hall, 1995.

[14] Alajlan, Maram & Koubaa, Anis & Chaari, Imen & Bennaceur, Hachemi & Ammar, Adel. “Global path planning for mobile robots in large-scale grid environments using genetic algorithms”. *2013 International Conference on Individual and Collective Behaviors in Robotics - Proceedings of ICBR 2013*. 1-8. 10.1109/ICBR.2013.6729271, 2013.

[15] S. Bechtold, S. Brannen, J. Link, JUnit 5 User Guide, https://junit.org/junit5/docs/current/user-guide/

[16] XIII. Recommendations. *Introduction to genetic algorithms.* https://www.obitko.com/tutorials/genetic-algorithms/recommendations.php

[17] L. Vogel, Unit Testing with Junit – Tutorial, 2016, https://www.vogella.com/tutorials/JUnit/article.html

[18] Jonasson, Anton, Simon Westerlind. “Genetic algorithm in mazes. A comparative study of the performance for solving mazes between genetic algorithms, BFS and DFS”.