# Intro To Artificial Intelligence Maze Solver - Project Report

Noy boutboul 206282691

Mark Fesenko 321208605

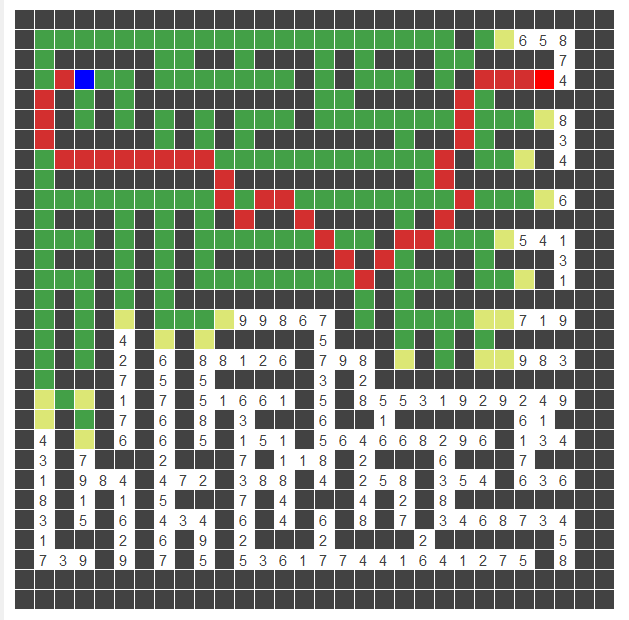


Table of Contents

[Intro To Artificial Intelligence Maze Solver - Project Report 1](#_Toc60689922)

[Tools & environment 3](#_Toc60689923)

[Overview 4](#_Toc60689924)

[Program code architecture 5](#_Toc60689925)

[Documentation 6](#_Toc60689926)

[Entities: 6](#_Toc60689927)

[Data Structures: 6](#_Toc60689928)

[Algorithms: 6](#_Toc60689929)

[Heuristics: 7](#_Toc60689930)

[Utilities: 7](#_Toc60689931)

[GUI: 8](#_Toc60689932)

[Maze Generator: 8](#_Toc60689933)

# Tools & environment

* Python 3.7
* PyCharm version 11.0 (IDE)
* Libraries used:
* HeapDict: we used HeapDict to implement a data structure that combines minimum heap and a hash table
* Tkinter & Turtle: we used this libraries to create the GUI interface and the visualization.
* Auto-py-to-exe: we used this library to bundle our code and generate an executable file.

# Overview

In this project we develop an independent agent that can solve a given maze using various search algorithms - both informed and uninformed while using heuristics we developed.

The goal of the agent is to solve the maze with the cheapest path possible.  
The maze is represented by NxN matrix of costs, starting point coordinates and goal point coordinates. The agent can move to all 8 adjacency direction.

Working and developing this project required research and deep understanding of the algorithms, programming it required a lot of code optimizations and complexity optimizations. One of our major challenges was to come up with the right kind of heuristic and to think of code optimizations.

We offer 5 different search algorithms: Bi-Astar, AStar, ID-Astar, UCS, IDS to that the agent can solve the maze with while using 2 kinds of consistent heuristic.

We provide a GUI interface to run our program, in which you can load mazes, set a time limit, and visualize the solving algorithm and the result path.

The results output is via txt file, which is generated after a run is completed in the same directory as ‘output\_results.txt’ in which you will find statistics of the run.

# Program code architecture

* Main:

Annotations*:*

* *Folder*
* *Class  
  function*
* Entities:
* Maze
* Node
* Algorithms:
* UCS + UCS\_visualized
* IDS + IDS\_visualized
* Astar + Astar\_visualized
* IDASTAR + IDASTAR\_visualized
* BiAstar + BiAstar\_visualized
* Data Structures:
* HeapDict
* Heuristics:
* Heuristics
  + Moves Counter
  + Minimum Moves
* Utilities:
* Utilities
  + Read file \ Write files
  + Calculate run statistics.
* GUI:
* GUI
* GUI interface
* Scripts:
* Maze generator

# Documentation

*General flow* of solving a maze is like so – We open a problem file via utility function, analyze it and generate the entities and variables that are passed into the solving algorithm. After executing, the algorithm passes the statistics to another utility function that prints the results.  
in case of a visualization, we run visualized algorithm (for example UCS\_visualized instead of regular UCS) that is painting its steps along the way, and creates visualizations.

## Entities:

* **Maze** – keeps the data about the maze and functions that relate to the maze.  
  holds the mazes matrix, starting node, goal node and size
* **Node** – keeps the data of a specific node (cell) in the maze. This entity plays a significant role in this program. Each node holds its coordinates, cost, heuristic value, depth, and father node which is used to backtrack when reaching a solution to generate the solution path.

## Data Structures:

In order for our code to run fast, we had to invest in choosing the right data structures.  
In this project we used minimum heaps, hash tables and HeapDict which is a data structure that combines a minimum heap and a hash table. This enabled us to search a node with and reduce its value with a cost of Using this dramatically changed the run time of the algorithms.

* **Hash Table** – we used python’s unsorted dictionary as hash table.
* **HeapDict** (Hash Table + Minimum Heap) – implemented a wrapper for this module

## Algorithms:

* **UCS** – We implemented this algorithm in a classic way, maintaining a frontier priority queue and explored hash table, always expanding the node which has the current cheapest path. This is done until reaching the goal.
* **Astar** – We implemented this algorithm via Best First Search template. Instead of minimum heap holding key values as path costs, the keys are now F values which are path costs + heuristic values thus by a small change to USC we get Astar.
* **BiAstar** – To implement this algorithm we duplicated the frontier of Astar. Each node that is about to expand is first checked up if it’s already been explored at the other frontier. Once we find a node that is explored in both frontiers we keep the search going until we satisfy the extra condition of optimality (until the sum of the evaluations in frontiers is lower than the evaluation of the intersected frontiers). Once we satisfy this condition we concatenate the paths and return it as a solution.
* **IDS** – After encountering enormous run time when implementing this algorithm with recursion in classic way, we decided to improve it. First we implemented a depth limited search iteratively via best first search template with key values as minus depth. Then we implement IDS as a loop with increasing depth limit as for each iteration we run a depth limited search. We optimized it by saving a visited list, which allows us to not visit a cell twice. The run time results were dramatically decreased due to this optimization.
* **IDAstar** – After the success with implementing IDS iteratively, we did the same thing here, and used the same optimizations. Our IDAstar is basically IDS, and for each iteration we run best first search template with key values as evaluated path costs.

## Heuristics:

We implemented and tested 2 heuristics (comparison report later on)

* **Minimum Moves –** The main idea is to calculate the exact minimum moves given a maze with walls. This heuristic has pre-processing of running BFS on the entire maze. Finding and calculating the minimum moves from each position to goal. The heuristic value is then returned for each position in the maze.
* ***Proof of admissibility****. Given maze with positive prices. Let’s assume is not admissible, thus exists that oversestimate the goal -> . Because the minimum moves required to reach the goal is* ***exactly*** *and the optimal price () is lower than the minimum required moves. One of the moves has to be negative, and this is a contradiction. So is admissible.*
* **Moves Counter –** This heuristic calculates the minimum number of moves (any kind, diagonal or regular) that is required to reach the goal node (given there is no walls in the path). And returns this number as the heuristic value.
* ***Proof of admissibility****. Given maze with positive prices. Let’s assume is not admissible, thus exists that oversestimate the goal -> . Because the minimum moves required to reach the goal is* ***at least*** *and the optimal price () is lower than the minimum required moves. One of the moves has to be negative, and this is a contradiction. So is admissible.*

Heuristics Optimizations:We added **scaling optimization**. We noticed that if the prices are very high, e.g. 20x20 maze with prices of 1000+. The range of heuristics values of Moves Counter is 0-28, this has very low effect on the total price. So we keep the minimum price of the maze and multiply it by the heuristic values. Given 20x20 maze with prices of 1000+ a heuristic value that used to be 10 is now 1000\*10, making the heuristic scale up well with the maze prices.

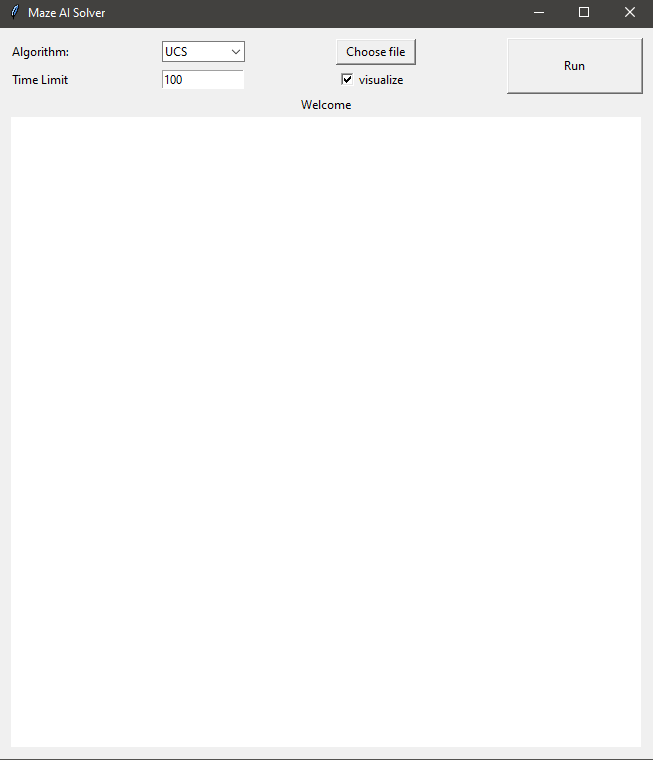
## Utilities:

Utilities class is where we implemented our utility functions. Such as reading the problem file, writing the output file, calculating the statistics after a run, etc.

## GUI:

We added a GUI interface to our project in which you can load maze problem files, set running time limit, select the solving algorithm, run your problem with or without visualizing the algorithm run the solution path.

* **GUI** – a singleton class that is responsible for painting the tiles into the screen while the algorithm runs.
* **GUI\_interface** – a class that responsible for building up the GUI interface. Has both logic and styling of buttons, windows, report statuses, text boxes, etc.



We recommend running our program via this GUI interface. Note that the visualization takes a lot of time, so if you are interested in testing the algorithms speed run it without the visualization.

## Maze Generator:

In order to test our program we needed mazes, so we wrote a script that generates a maze.  
You are welcome to try the script, just run it on its own. You can set the maze size (note that mazes bigger than 200 takes time to generate), you can set the starting point, goal point, walls density (1-9, 1 minimum amount of walls, 9 maximum amount of walls). At the end of the execution the script will print the maze matrix.  
Disclaimer - parts of this script were taken from a GitHub project and was not written by us.