# Intro To Artificial Intelligence Maze Solver - Project Report

Noy boutboul 206282691

Mark Fesenko 321208605

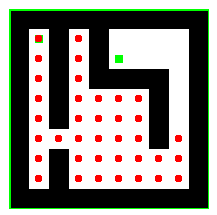


Table of Contents

[Intro To Artificial Intelligence Maze Solver - Project Report 1](#_Toc58848619)

[Tools & environment 3](#_Toc58848620)

[Overview 4](#_Toc58848621)

[Program code architecture 5](#_Toc58848622)

[Documentation 6](#_Toc58848623)

[Entities: 6](#_Toc58848624)

[Data Structures: 6](#_Toc58848625)

[Algorithms: 6](#_Toc58848626)

[Heuristics: 7](#_Toc58848627)

[Utilities: 7](#_Toc58848628)

# 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
* Heapq: we used heapq to implement priority queue

# 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 (all costs are 1+), 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.

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.

The results are presented as \_\_

# Program code architecture

* Main:

Annotations*:*

* *Folder*
* *Class  
  function*
* Entities:
* Maze
* Node
* Algorithms:
* UCS
* IDS
* Astar
* IDASTAR
* BiAstar
* Data Structures:
* HeapDict
* Priority Queue
* Heuristics:
* Heuristics
  + Moves Counter
  + Diagonal
* Utilities:
* Utilities
  + Read file
  + Write files (TODO)
  + Calculate run statistics.
* 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.

## 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 O(1) and reduce its value with a cost of O(log(n)). Using this dramatically changed the run time of the algorithms.

* **Hash Table** – we used python’s unsorted dictionary as hash table.
* **Minimum Heap** (Priority Queue) – implemented a priority Queue Wrapper
* **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 duplicated the frontier of Astar. While each node that is expanded is first checked if it’s already been explored at the other frontier. Once we find a node that is explored in both frontiers we concatenate their paths and return it as a solution.
* **IDS** – We first implemented a depth limited search 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.
* **IDAstar** – We implemented IDS as a loop with increasing F bound values as for each iteration we run Astar with a limited F value.

## Heuristics:

We implemented and tested 2 heuristics and compared between them (see \_\_\_\_\_ later also admissible proof.).

* **Diagonal -** The main idea behind this heuristic is that diagonal moves are more valuable, thus giving them a lower heuristic value then regular moves.  
  the heuristic calculate the minimum diagonal and regular moves required from the current node to the goal node (given there is no walls in the path), and gives the diagonal moves a cost of 0.58 and regular moves cost of 1.   
  The heuristic value is *(#diagonal moves ) \* 0.58 + (#regular moves) \* 1*
* **MovesCounter –** 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.

## 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.