Intro To Artificial Intelligence
Maze Solver - Project Report

Noy boutboul 206282691

Mark Fesenko 321208605

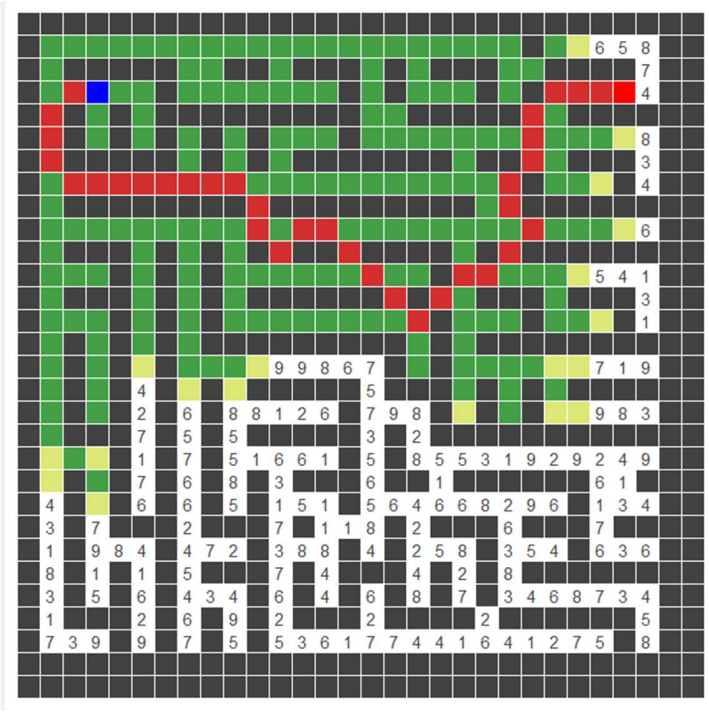


Table of Contents

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:

- 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

Annotations:

- *Folder*
- *Class*
- *function*

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 $O(1)$ 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.
- **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 h is not admissible, thus exists $h(n)$ that oversestimate the goal -> $h(n) \geq h^*(n)$. Because the minimum moves required to reach the goal is* ***exactly*** *$h(n)$, and the optimal price ($h^*(n)$) is lower than the minimum required moves. One of the moves has to be negative, and this is a contradiction. So h 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 h is not admissible, thus exists $h(n)$ that oversestimate the goal -> $h(n) \geq h^*(n)$. Because the minimum moves required to reach the goal is* ***at least*** *$h(n)$, and the optimal price ($h^*(n)$) is lower than the minimum required moves. One of the moves has to be negative, and this is a contradiction. So h 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.

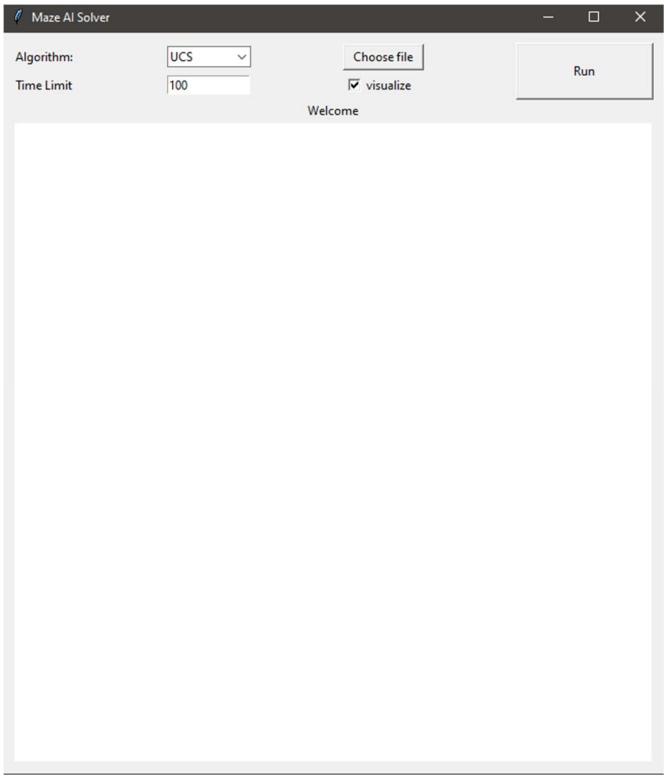
Utilities:

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.

Comparisons

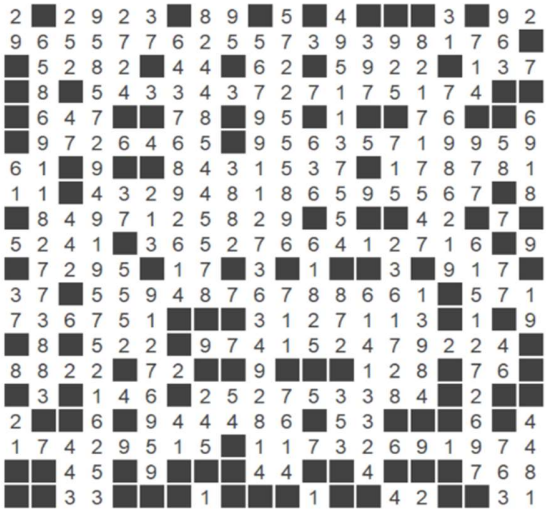
Test Results:

In our comparisons we ran the algorithms on different mazes, with varying wall densities and a time limit of 50 seconds. The result are the average of 5 runs. Test results are shown here in tables (mazes provided with the program, output files are not as there were too many).

We start with a small maze of size 20*20, with a few walls.

Name: maze20wall2.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	274	0.07299	Y	0.03690	1.32399	12.52543	1	11.18705	20
Astar	movesCount	274	0.07299	Y	0.02796	1.32399	11.71281	1	11.15827	20
BiAstar	minimumMoves	461	0.02386	Y	0.07381	1.74644	11.50361	1	10.87772	20
BiAstar	movesCount	505	0.02178	Y	0.05190	1.76097	11.25809	1	11.54435	20
IDAstar	minimumMoves	5643	0.00354	Y	0.84299	1.54019	13.92008	0	6.74050	20
IDAstar	movesCount	5893	0.00339	Y	0.88958	1.54353	12.68482	0	6.74601	20
IDS	-	11169	0.00170	Y	0.69626	1.63325	-	-	-	0
UCS	-	282	0.07092	Y	0.01998	1.32590	-	1	11.42253	20

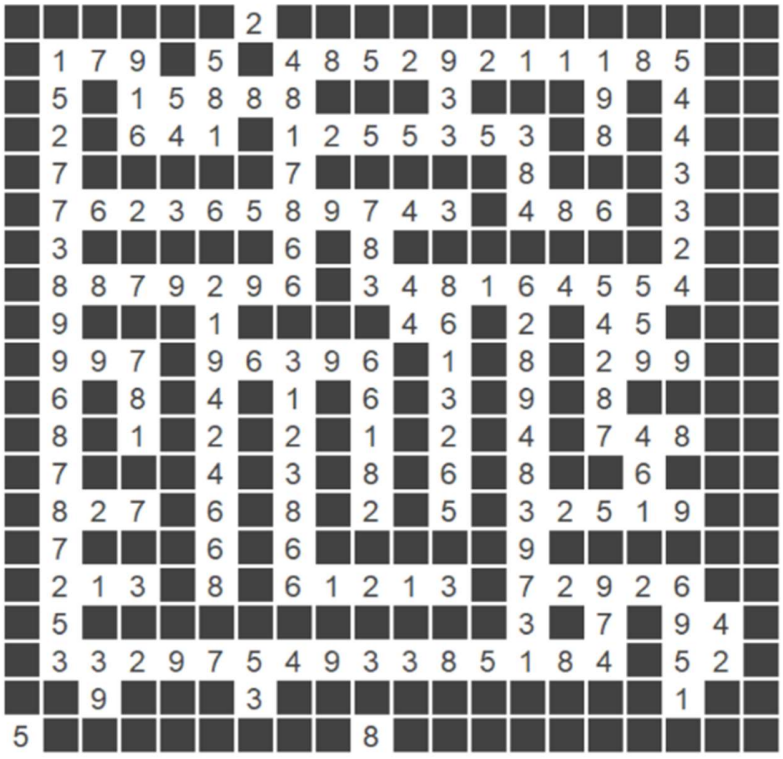


maze20wall2

Next, also a 20*20 maze, with higher density of walls.

Name: maze20wall16.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	165	0.13939	Y	0.01699	1.24856	16.29056	1	13.27868	23
Astar	movesCount	165	0.13939	Y	0.01296	1.24856	11.63962	1	13.35	23
BiAstar	minimumMoves	274	0.04744	Y	0.02992	1.53999	14.77541	1	13.24509	24
BiAstar	movesCount	297	0.04713	Y	0.02895	1.50184	11.52947	1	14.04587	27
IDAstar	minimumMoves	8027	0.00286	Y	1.19401	1.47830	18.06652	0	8.67009	23
IDAstar	movesCount	8835	0.00260	Y	1.36205	1.48447	12.81861	0	8.76768	23
IDS	-	5070	0.00453	Y	0.23942	1.44906	-	-	-	0
UCS	-	172	0.13372	Y	0.00997	1.25082	-	1	13.55737	23

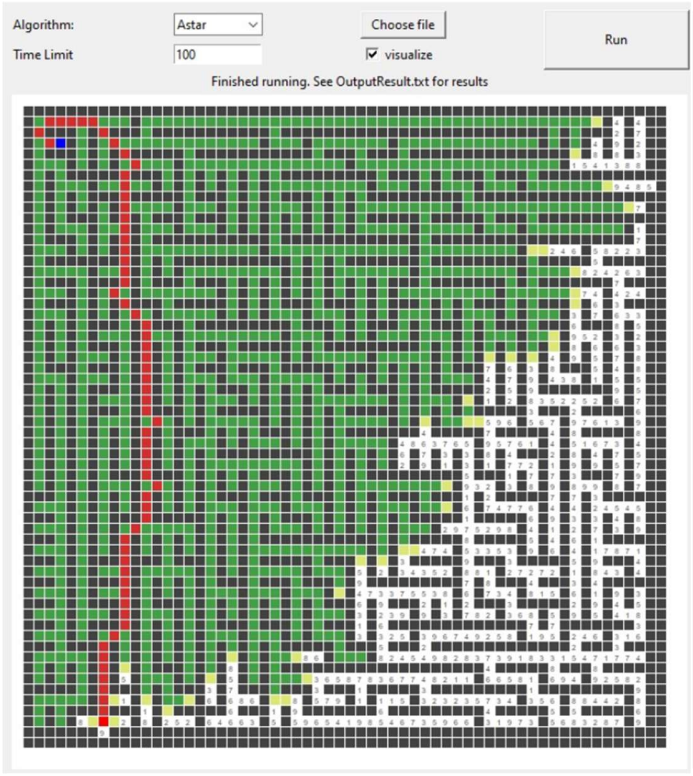


maze20wall16

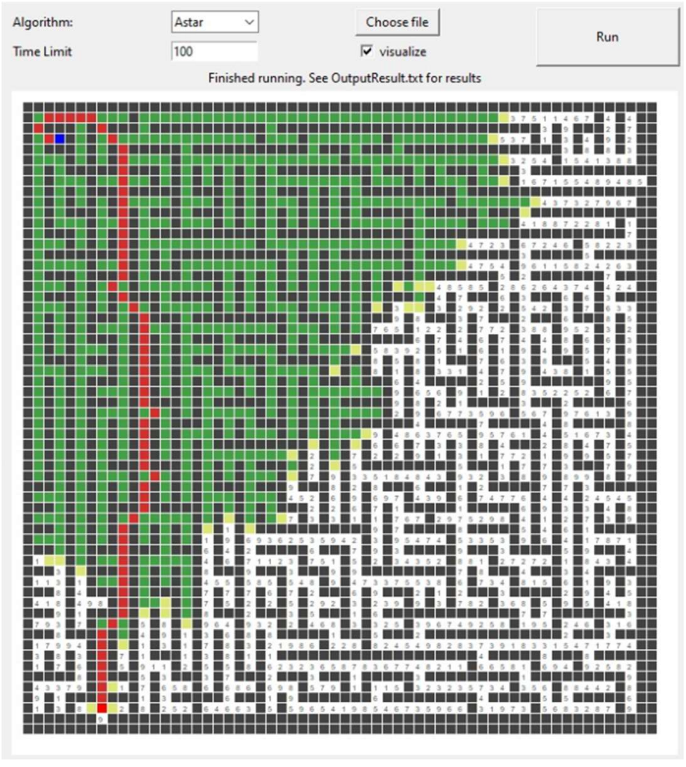
A maze of size 60*60, with high wall density.

Maze: maze60highDensity.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	776	0.08118	Y	0.18415	1.11140	64.27549	2	33.12558	63
Astar	movesCount	1141	0.05521	Y	0.12472	1.11822	33.99872	2	40.05642	63
BiAstar	minimumMoves	978	0.03987	Y	0.22041	1.19309	57.44358	1	33.25631	62
BiAstar	movesCount	1336	0.03368	Y	0.23636	1.17344	33.04901	1	38.49104	64
IDAstar	minimumMoves	66505	0.00094	Y	14.52185	1.19276	61.86782	0	19.58412	63
IDAstar	movesCount	98828	0.00063	Y	15.26526	1.20028	38.95418	0	24.00048	63
IDS	-	30541	0.00206	Y	1.89527	1.17811	-	-	-	0
UCS	-	1333	0.04726	Y	0.15558	1.12098	-	2	43.10695	66



MovesCount heuristic

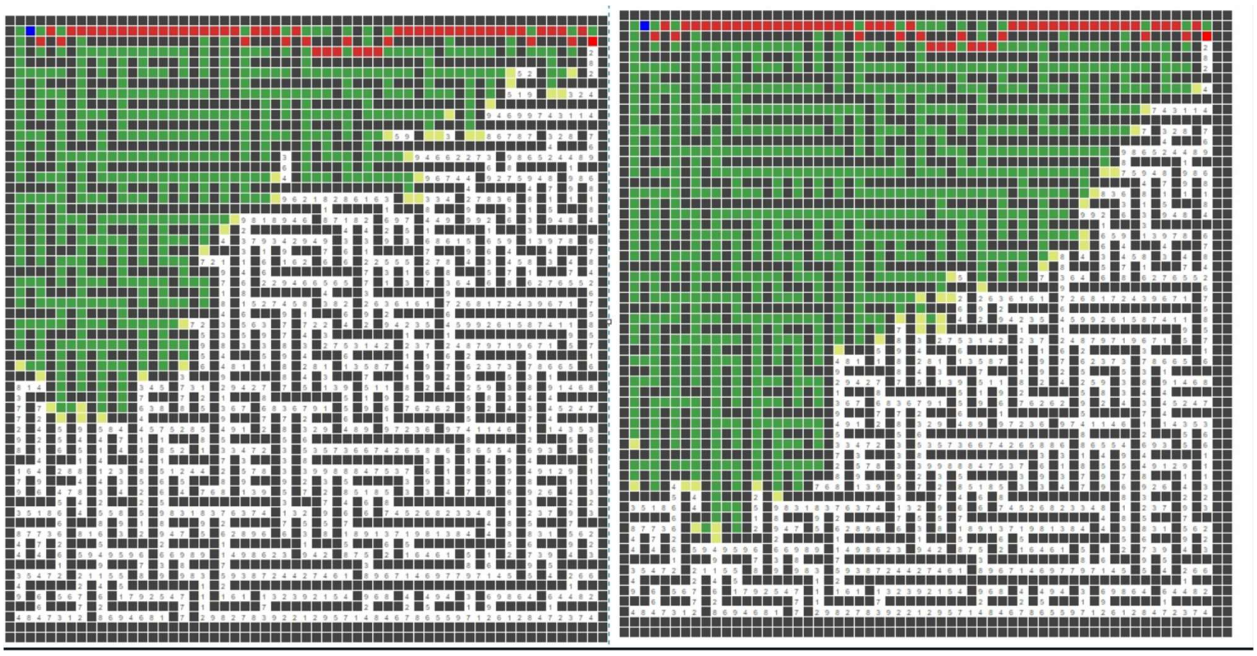


MinimumMoves heuristic

A maze of size 60*60, with high wall density.

Maze: maze60highDensity2.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	566	0.09717	Y	0.07686	1.12215	58.34184	1	28.36	55
Astar	movesCount	860	0.06395	Y	0.06754	1.13071	36.07196	1	33.34262	55
BiAstar	minimumMoves	777	0.04247	Y	0.14561	1.22345	50.46538	1	28.42194	55
BiAstar	movesCount	1016	0.03543	Y	0.07878	1.21206	34.00989	1	31.35922	54
IDAstar	minimumMoves	39228	0.00140	Y	7.10379	1.21204	56.42832	0	15.90438	55
IDAstar	movesCount	56341	0.00097	Y	10.88488	1.22005	41.56072	0	19.13261	55
IDS	-	21265	0.00258	Y	1.46048	1.19862	-	-	-	0
UCS	-	1085	0.05069	Y	0.06083	1.13550	-	1	37.61180	59

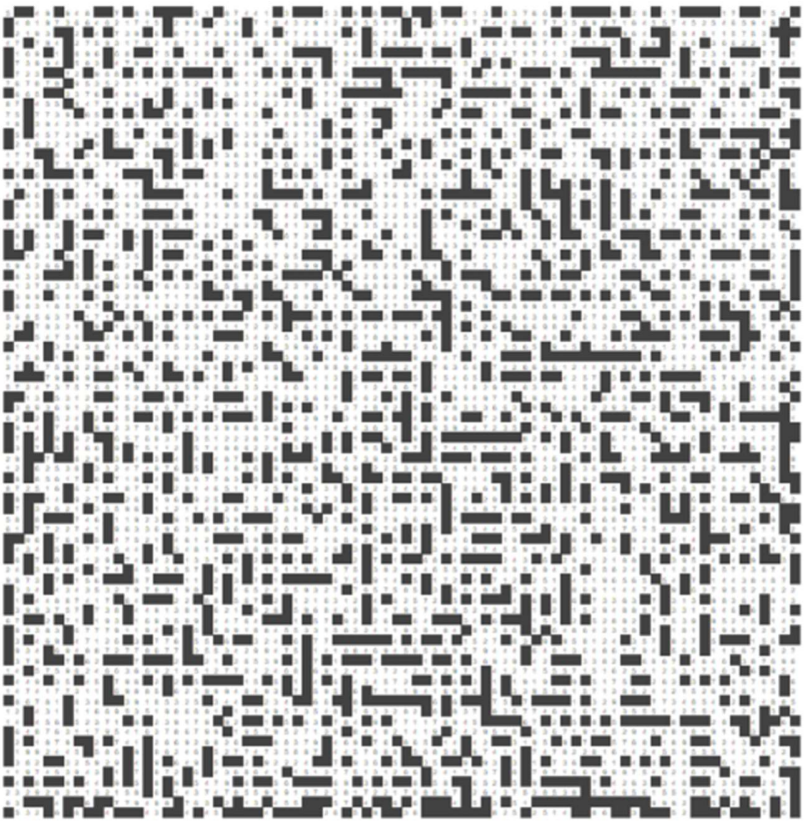


MinimumMoves heuristic *MovesCount heuristic*

A maze of size 80*80, with moderate walls density.

Maze: maze80wall2.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	4216	0.02514	Y	0.84179	1.08192	49.54117	1	57.11894	106
Astar	movesCount	4220	0.02511	Y	0.50068	1.08193	48.43395	1	57.14363	106
BiAstar	minimumMoves	7827	0.00728	Y	1.55492	1.17033	46.63898	1	54.65046	106
BiAstar	movesCount	8617	0.00638	Y	1.04227	1.17910	49.28312	1	56.85831	106
IDAstar	minimumMoves	-	-	N	-	-	-	-	-	-
IDAstar	movesCount	-	-	N	-	-	-	-	-	-
IDS	-	-	-	N	-	-	-	-	-	-
UCS	-	4659	0.02232	Y	0.55554	1.08460	-	1	57.33297	106

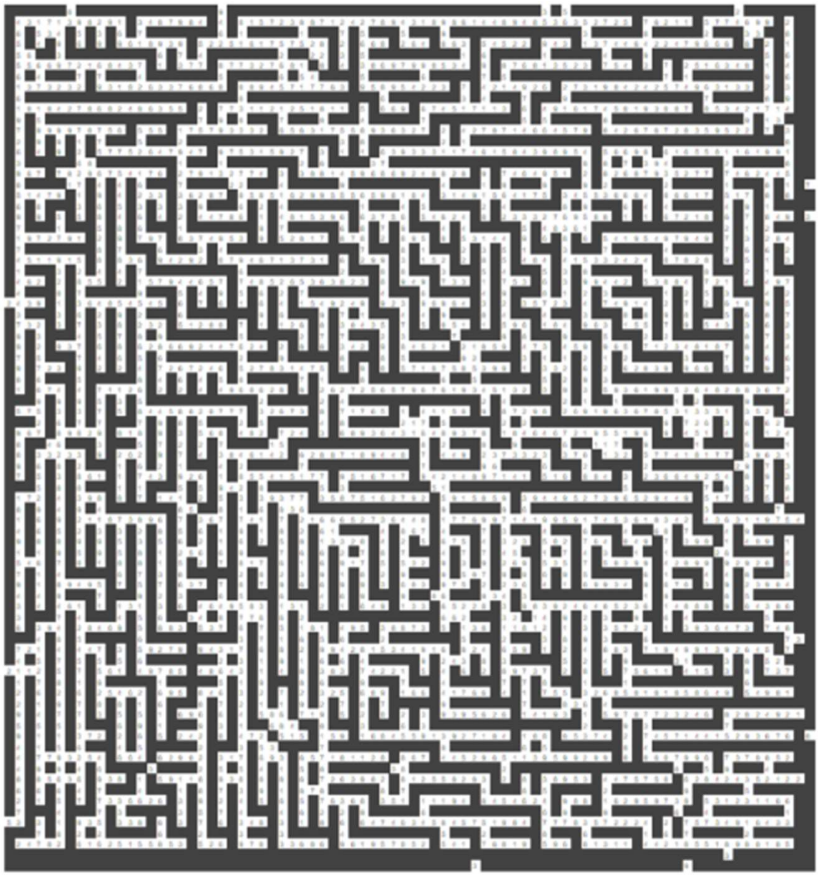


maze80wall2

Maze size 80*80 with high density of walls.

Name: maze80wall20.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	3153	0.03583	Y	0.32513	1.07389	69.12456	1	63.48061	116
Astar	movesCount	3182	0.03551	Y	0.21645	1.07398	48.94624	1	63.34046	116
BiAstar	minimumMoves	5734	0.01011	Y	0.53457	1.16091	65.36722	1	63.41862	116
BiAstar	movesCount	5914	0.00963	Y	0.40791	1.16459	51.14841	1	65.12638	113
IDAstar	minimumMoves	-	-	N	-	-	-	-	-	-
IDAstar	movesCount	-	-	N	-	-	-	-	-	-
IDS	-	-	-	N	-	-	-	-	-	-
UCS	-	3187	0.03545	Y	0.16658	1.07399	-	1	63.61549	118

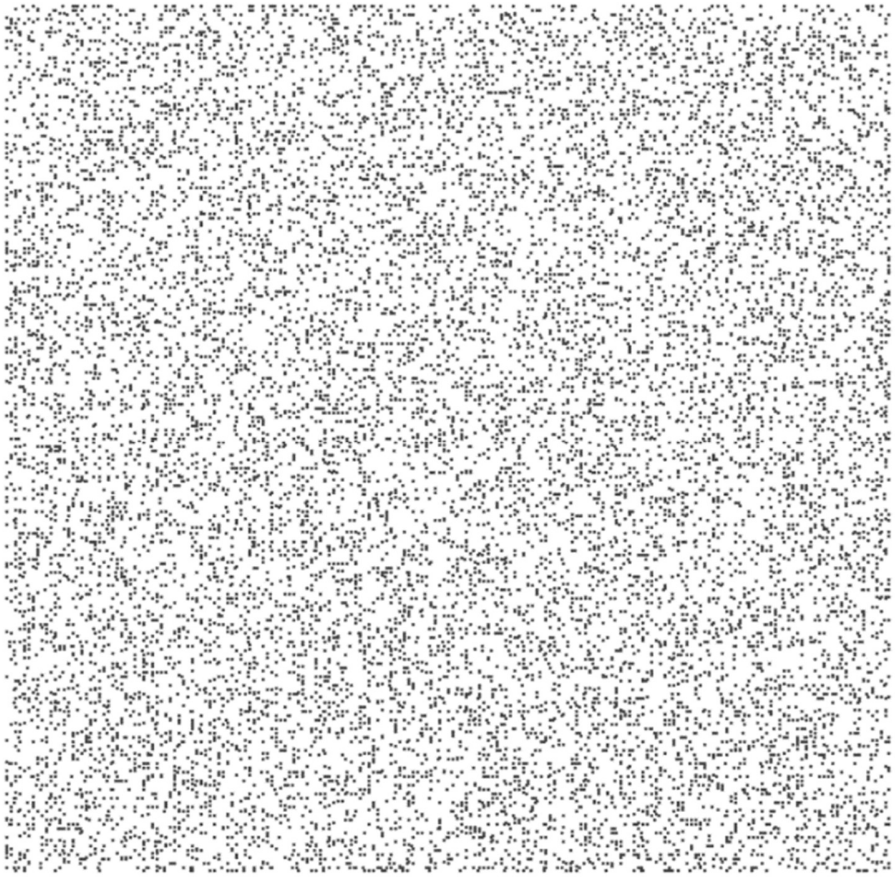


maze80wall20

A big maze of size 400*400, with a low density of walls.

Name: maze400sparse.txt

Algorithm	Heuristic name	N	Penetration rate	Success (Y/N)	Time (sec)	EBF	avg H value	Min	Avg	Max
Astar	minimumMoves	142806	0.00380	Y	37.09958	1.02205	263.11142	1	283.35102	544
Astar	movesCount	142806	0.00380	Y	29.54109	1.02209	262.77232	1	278.98282	544
BiAstar	minimumMoves	-	-	N	-	-	-	-	-	-
BiAstar	movesCount	286451	0.00098	Y	49.82194	1.04573	263.02573	1	298.09815	544
IDAstar	minimumMoves	-	-	N	-	-	-	-	-	-
IDAstar	movesCount	-	-	N	-	-	-	-	-	-
IDS	-	-	-	N	-	-	-	-	-	-
UCS	-	146516	0.00369	Y	18.97634	1.02223	-	1	273.30536	544



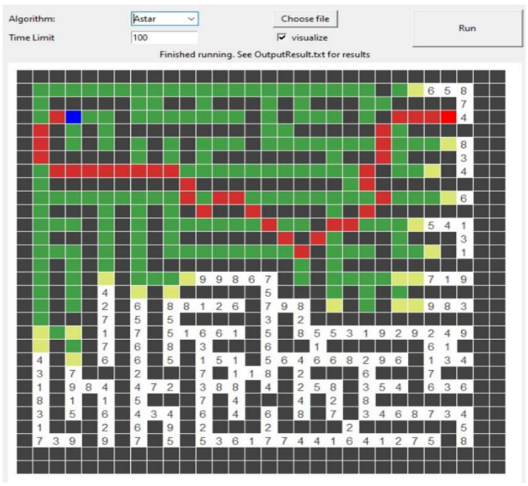
maze400sparse (demonstration of the wall density, without actual values)

Uninformed search vs Informed search:

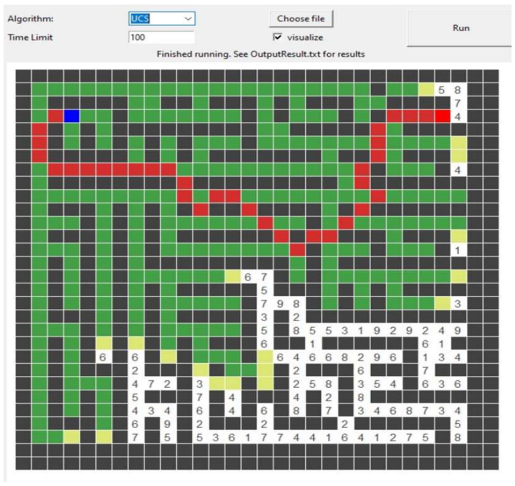
Comparing UCS (uninformed algorithm) to Astar (informed algorithm) we learn a few things:

- UCS algorithm typically explores more nodes than Astar (image below)
- Penetration rate of Astar is higher than UCS (as a result of the last observation)
- Running speed of UCS are better than Astar (most of the time)
- Astar tends to get a better EBF than UCS.

Even though UCS expands more nodes than heuristic algorithms and its penetration rate is lower, it has better running times due to less calculations to perform and no preprocessing whatsoever.



AStar



UCS

AStar need to explore less nodes to get to find the goal because it's informed of its goal node, while UCS explores all nodes without any information of the goal node

Bi-Astar, the algorithm we initially thought to be the best one, isn't giving such good results because of its stopping condition in which it's not enough for the two paths to intersect. Eventually it ends up expanding more nodes and consumes more time.

Heuristics Comparison:

From our tests we learn that Astar algorithm is the best of our heuristic algorithms, and that Minimum Moves heuristic gives better results than moves Count heuristic, even though it has an additional preprocess time of calculating distances from each node to the goal node, Minimum Moves has a better understanding of the maze and thus giving it the ability to predict accurately the distance to goal which leads to expanding less node, better EBF and penetration rate.
The difference can be seen in 'maze60highDensity.txt' page 11, and in 'maze60highDensity2.txt' page 12. (proof of admissibility is inside heuristic documentation page7)

Conclusions:

During the process of testing and comparing we encountered 3 main factors that affect the performance of the algorithms:

- **Maze properties (size, price range, walls density)**
  Bigger mazes takes more time to solve. Mazes that have high walls density, are more difficult to solve due to false paths that reaches near goal node but are blocked (one of the motivations of Minimum Moves pre-processing BFS). Mazes with big range of prices are bounding the heuristic values as the values are set to the lowest price that is in the maze.
- **Code quality and efficiency**
  Code efficiency plays a big rule when running this algorithms, we can see it clearly when comparing UCS with an informed algorithm that does more operations, UCS usually wins when it comes to speed just because it has less actions to take even though it explores much more. Thus, it is important to write the algorithms efficiently.
- **Heuristic quality**
  Our first tries of using Manhattan-Distance heuristic for example, failed because it didn't fit our maze terms. It is important to find the suitable heuristic to improve the performance.

Interesting optimizations we developed during the process:

- Iterative deepening speed up with saving "visited" list and not revisiting nodes we already explored (further information at IDAstar \ IDS documentation)
- Scaling up the heuristic according to the maze lowest cost (further information in heuristic documentation)
- Adding BFS search as pre-processing to MinimumMoves heuristic (further information in MinimumMoves documentation).