**CS6053 Artificial Intelligence and Machine Learning**

**Coursework Group Report**

Hasan Khan ¬ [hak0816@my.londonmet.ac.uk](mailto:hak0816@my.londonmet.ac.uk) ¬ 21002595

Fabian Hasan ¬ [fah0550@my.londonmet.ac.uk](mailto:fah0550@my.londonmet.ac.uk) ¬ 22044754

Contents

[1.0 Introduction 3](#_Toc198845590)

[1.1 Requirements and Success Criteria 3](#_Toc198845591)

[2.0 System Architecture and File Structure 4](#_Toc198845592)

[3.0 State Space Modules 7](#_Toc198845593)

[4.0 Logical Inference Module 12](#_Toc198845594)

[5.0 Findings and Evaluation 26](#_Toc198845595)

[6.0 User Interface 30](#_Toc198845596)

[7.0 Advanced Features 32](#_Toc198845597)

[8.0 Conclusion 38](#_Toc198845598)

[9.0 References 39](#_Toc198845599)

[Figure 1 - Collection of Files 5](#_Toc197105946)

[Figure 2 - Bidirectional Search 8](#_Toc197105947)

[Figure 3 - A\* Search 9](#_Toc197105948)

[Figure 4 - Iterative Deepening DFS 10](#_Toc197105949)

[Figure 5 - Greedy Search 11](#_Toc197105950)

[Figure 6 - bidirectional Search with Forward Chaining 13](#_Toc197105951)

[Figure 7 - Bidirectional Search with Backward Chaining 14](#_Toc197105952)

[Figure 8 - A\* with Forward Chaining 15](#_Toc197105953)

[Figure 9 - A\* with Backward Chaining 16](#_Toc197105954)

[Figure 10 - Iterative Deepening DFS with Forward Chaining 17](#_Toc197105955)

[Figure 11 - Iterative Deepening DFS with Backward Chaining 18](#_Toc197105956)

[Figure 12 - Greedy Search with Forward Chaining 19](#_Toc197105957)

[Figure 13 - Greedy Search with Backwards Chaining 20](#_Toc197105958)

[Figure 14 - Solving Maze using Forward Chaining with Path PART1 21](#_Toc197105959)

[Figure 15 - Solving Maze using Forward Chaining with Path PART2 22](#_Toc197105960)

[Figure 16 - Solving Maze using Forward Chaining with Path PART3 23](#_Toc197105961)

[Figure 17 - Solving Maze using Forward Chaining with Path PART4 24](#_Toc197105962)

[Figure 18 - Solving Maze using Forward Chaining with Path PART5 25](#_Toc197105963)

[Figure 19 - Bidirectional Comparison Forwards Chaining Summary 26](#_Toc197105964)

[Figure 20 - Bidirectional Comparison Backwards Chaining Summary 27](#_Toc197105965)

[Figure 21 – Bidirectional Forward Analysis 28](#_Toc197105966)

[Figure 22 - Bidirectional Backward Analysis 29](#_Toc197105967)

[Figure 23 - Maze Menu Main 30](#_Toc197105968)

[Figure 24 - Menu with the DFS and BFS Search 31](#_Toc197105969)

[Figure 25 - Maze Creation Menu 32](#_Toc197105970)

[Figure 26 - Maze Generation with Obstacles 33](#_Toc197105971)

[Figure 27 - Custom Maze Menu with BFS 34](#_Toc197105972)

[Figure 28 - Maze Search Visualisation 35](#_Toc197105973)

[Figure 29 - Maze Search Graph Comparison 36](#_Toc197105974)

[Figure 30 - Maze Search Visualisation comparison 36](#_Toc197105975)

[Figure 31 - Visualisation for custom maze 37](#_Toc197105976)

[Figure 32 - Custom Maze Graph Visualisation 37](#_Toc197105977)

# 1.0 Introduction

Solving mazes is a classic challenge in artificial intelligence because it tests how well different algorithms can find their way through a problem with obstacles and choices. In this project, we focused on a 5×6 grid maze, where the goal is to guide an agent from the top-left corner to the bottom-right, avoiding obstacles along the way.

The main goal in this assignment was to build a Python program that could solve this maze using several different methods. We ultimately implemented a range of state-space search algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening DFS (IDDFS), Greedy Best-First Search, A, and Bidirectional Search. Alongside these, we also created a logical inference system using First-Order Logic, with both forward and backwards chaining. This allowed us to compare traditional search techniques with rule-based reasoning.

Our objectives were clear in that we wanted to find the best path from start to goal, make the reasoning process transparent, compare how each method performed and provide an easy-to-use command-line interface. The project was a group effort and each member contributed by programming their specific algorithm that we chose at the start and then helping with the main menu, visualisations and other functions.

## 1.1 Requirements and Success Criteria

The main requirements were:

* The maze must be a 5×6 grid, with specific obstacles placed to create a challenge.
* The agent can move in four directions: left, right, up, and down.
* We needed to implement at least two separate Python programs: one for state-space search algorithms and one for logical inference using First-Order Logic.
* Both programs should be able to demonstrate their solutions step by step.
* The project should include a demonstration, showing how the algorithms and inference methods work on the maze.
* The program should always find an optimal path if one exists, especially for algorithms that guarantee optimality.

# 2.0 System Architecture and File Structure

Our project is organised to be modular, easy to understand and straightforward to maintain. The maze model is what represents the grid, obstacles and the agent’s position. We built this using a Maze class that stores the grid as a two-dimensional list, with each cell indicating whether it’s empty, an obstacle, the start or the goal. Each position in the maze is treated as a node and we use a function to quickly find all valid neighbouring nodes for any given position. This setup allows the search algorithms to efficiently explore the maze.

We have many different files, each with a clear purpose. For example, maze\_representation.py manages the maze structure, while assignment\_maze.py sets up the specific 5×6 maze for the assignment. Each search algorithm is implemented in its file, such as greedy\_search.py or astar\_search.py and there are the menu files that provide interactive command-line interfaces for each method. Logical inference is handled in a separate set of files. We also have a visualisation module that helps users see the maze and the search process step by step.

To keep our code flexible, we used some straightforward design principles. For example, we organised the system so it’s easy to switch between different search algorithms by simply calling the right function from the main menu. We also created an adapter to connect the maze model to the logical inference modules which allows both the search algorithms and the logic methods to work with the same maze structure. By making sure each part of the code has a clear purpose, we made it easier to add new features or make improvements in the future.

If you look at the project tree, you’ll see a clear structure with separate folders for the maze, search algorithms, inference methods, command-line interface and tests. This made it easy for us to all edit the right files and delegate each task efficiently

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Collection of Files

* algorithm\_comparison.png – Visual chart comparing algorithm performance.
* algorithm\_comparison.py – Runs and compares all search algorithms on the maze.
* assignment\_maze.py – Sets up the specific 5×6 maze with obstacles for the assignment.
* astar\_menu.py – Menu interface for running and testing the A\* algorithm.
* astar\_search.py – Implements the A\* search algorithm.
* bfs\_search.py – Implements the Breadth-First Search algorithm.
* bidirectional\_menu.py – Menu interface for running and testing Bidirectional Search.
* bidirectional\_search.py – Implements the Bidirectional Search algorithm.
* compare\_algorithms.py – Compares the results and performance of different algorithms.
* dfs\_search.py – Implements the Depth-First Search algorithm.
* greedy\_menu.py – Menu interface for running and testing Greedy Best-First Search.
* greedy\_search.py – Implements the Greedy Best-First Search algorithm.
* iddfs\_menu.py – Menu interface for running and testing Iterative Deepening DFS.
* iddfs\_search.py – Implements the Iterative Deepening Depth-First Search algorithm.
* individual\_menu\_template.py – Template for creating menu interfaces for new algorithms.
* logical\_inference\_backward.py – Implements backwards chaining for logical inference.
* logical\_inference\_forward.py – Implements forward chaining for logical inference.
* logical\_inference\_template.py – Template for building new logical inference modules.
* logical\_main.py – Main driver for running logical inference methods.
* logical\_theory.py – Encodes the maze as logical facts and rules.
* main.py – Main menu for selecting and running all algorithms and inference methods.
* maze\_adapter.py – Connects the maze model to the logical inference modules.
* maze\_representation.py – Defines the Maze class and grid structure.
* maze\_visualization.py – Visualises the maze and the search process.
* menu.py – Provides shared menu logic for the command-line interface.
* requirements.txt – Lists all required Python packages for the project.
* search\_algorithms.py – Contains shared code and helpers for search algorithms.

# 3.0 State Space Modules

The state-space search module is where we put different algorithms to the test, each trying to find a path through the maze from start to goal. We included Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening DFS (IDDFS), Greedy Best-First Search, A and Bidirectional Search. Each algorithm explores the maze in its way and by comparing them, we can see the strengths and weaknesses of each approach.

Breadth-First Search (BFS) explores the maze level by level, always expanding the closest nodes first. This method guarantees the shortest path if one exists, but it can use a lot of memory on bigger mazes. Depth-First Search (DFS), on the other hand, goes as far as possible down one path before backtracking. It uses less memory but doesn’t always find the shortest route. Iterative Deepening DFS (IDDFS) combines the benefits of both, running DFS with increasing depth limits until it finds the goal, so it’s both memory-efficient and complete.

Greedy Best-First Search uses a heuristic (specifically, the Manhattan distance) to always pick the node that looks closest to the goal. This makes it fast, but it doesn’t always find the shortest path. A\\* search improves on this by considering both the cost to reach a node and the estimated cost to the goal, so it’s both fast and optimal when the heuristic is admissible. Bidirectional Search runs two simultaneous searches (one from the start and one from the goal) and meeting in the middle to cut down on the number of nodes explored.

For the heuristics, we mainly used Manhattan distance which is a simple way to estimate how far a point is from the goal in a grid. It works by adding up the number of steps needed to move horizontally and vertically from one cell to another. For example, if you’re at (1, 2) and the goal is at (4, 5), the Manhattan distance is “[1 – 4] + [2 – 5] = 6”. This estimate helps the algorithm decide which paths look most promising, so it can focus on routes that seem to lead directly to the goal.

Heuristics like Manhattan distance are important because they guide the search, making algorithms like Greedy Best-First Search and A\* much faster than methods that explore blindly. A good heuristic should be admissible, meaning it never overestimates the true distance to the goal. This ensures that algorithms like A\* will always find the shortest path if one exists. In our project, using the Manhattan distance made the search more efficient and helped the algorithms avoid wasting time on paths that don’t get closer to the goal. We also looked at how each algorithm performs in terms of time, memory and path quality. BFS and A\* always find the shortest path, but A\* is usually faster because it uses the heuristic. Greedy is the quickest but sometimes misses the best route. DFS and IDDFS are more memory-friendly, but DFS can get lost if the maze is tricky. Bidirectional Search is very efficient for mazes with clear start and end points.

To make everything user-friendly, we built a command-line menu that lets users pick any algorithm, see the maze and watch the search unfold step by step. The visualisation module shows the path and explored nodes, making it easy to understand how each algorithm works. This setup not only helps with testing and comparison but also makes the project accessible for anyone who wants to learn about search algorithms in action.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Bidirectional Search

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - A\* Search

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Iterative Deepening DFS

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Greedy Search

# 4.0 Logical Inference Module

The logical inference module takes a different approach to solving the maze by using rule-based reasoning instead of a traditional search. Here, the maze is described using logical facts and the solution is found by applying inference methods from First-Order Logic (FOL). This lets us see how logical reasoning can be used to solve pathfinding problems systematically.

To represent the maze, we use facts like which cells are open, where the obstacles are and the positions of the start and goal. These are written as logical statements, such as position (0, 0) for an open cell or obstacle (1, 2) for a blocked one. We also define rules that describe valid moves, for example “You can move from (X1, Y1) to (X2, Y2) if both are open, they are next to each other and (X2, Y2) is not an obstacle.” This setup allows the inference engine to reason about possible moves and build a path through the maze.

We implemented two main inference strategies: forward chaining and backwards chaining. Forward chaining starts from the initial facts (like the agent’s starting position) and applies rules step by step to discover new facts, gradually building a path to the goal. It’s thorough and explores all possible moves but can be slower if there are many options. Backwards chaining works the other way around: it starts from the goal and tries to work backwards, asking what needs to be true to reach the goal and then checking if those conditions can be met from the start. This can be more efficient in some cases, especially if the goal is hard to reach.

To connect the maze model to the inference engines, we built a MazeAdapter. This adapter translates the maze’s grid structure into logical facts and makes sure the inference methods can use the same maze as the search algorithms. This keeps the code consistent and avoids duplication.

Throughout the inference process, the program prints each step, showing which rules are applied and what new facts are discovered. This makes the reasoning transparent and helps users follow the logic behind the solution. In our tests, forward chaining was more comprehensive but sometimes slower, while backwards chaining could find more optimal paths but struggled with complex layouts. Both methods highlight the strengths of logical reasoning for problem-solving, especially when you want to see exactly how a solution is built step by step.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - bidirectional Search with Forward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Bidirectional Search with Backward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - A\* with Forward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - A\* with Backward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Iterative Deepening DFS with Forward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Iterative Deepening DFS with Backward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Greedy Search with Forward Chaining

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Greedy Search with Backwards Chaining

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Solving Maze using Forward Chaining with Path PART1

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Solving Maze using Forward Chaining with Path PART2

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Solving Maze using Forward Chaining with Path PART3

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Solving Maze using Forward Chaining with Path PART4

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Solving Maze using Forward Chaining with Path PART5

# 5.0 Findings and Evaluation

To see how well our algorithms and logical inference methods worked, we tested them all on the same 5×6 maze using Python. We ran each method multiple times on the same computer to keep things fair. For each run, we measured how long it took to find a path, how many nodes were expanded and the length of the path found. We also looked at how much memory each method used, though for this small maze, memory was not a big issue.

The results showed clear differences between the methods. Breadth-First Search (BFS) and A\* always found the shortest path, but A\* was usually faster because it used the Manhattan distance heuristic to guide its search. Greedy Best-First Search was the quickest overall but sometimes missed the shortest path because it only looked at which move seemed best right away. Depth-First Search (DFS) and Iterative Deepening DFS (IDDFS) used less memory, but DFS could get stuck or take longer if the maze was tricky. Bidirectional Search was very efficient, especially when the start and goal were far apart.

For logical inference, forward chaining checked all possible moves and was very thorough but could be slower. Backwards chaining worked from the goal and sometimes found a more direct path but could struggle if the maze was more complicated.

Below is the Bidirectional comparisons and analysis.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Bidirectional Comparison Forwards Chaining Summary

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Bidirectional Comparison Backwards Chaining Summary

A screenshot of a computer program

AI-generated content may be incorrect.

Figure – Bidirectional Forward Analysis

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Bidirectional Backward Analysis

# 6.0 User Interface

We designed the user interface to be straightforward to use, so anyone of all ages can try out the different algorithms and see how they work. The main way to interact with the project is through a command-line menu. When you run the program, you’re greeted with a clear menu that lets you pick which search algorithm or logical inference method you want to use. Each option is labelled, and you can select algorithms like BFS, DFS, A, Greedy, IDDFS, Bidirectional Search or try out the logical inference demonstrations. Once you choose an algorithm, the program guides you through the process step by step. You can see the maze; the path being explored and the final solution. For some algorithms, you can also compare results or see a visualisation of the search process. The output is well formatted and easy to follow, and there are prompts to help you move through each stage.

A screenshot of a computer

AI-generated content may be incorrect.

Figure - Maze Menu Main

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Menu with the DFS and BFS Search

# 7.0 Advanced Features

To make the project more flexible and comprehensive, we included several advanced features beyond the basic requirements. One of the most significant enhancements was our decision to implement six different search algorithms, even though only four were required. Our focus was on A\* Search, Iterative Deepening Depth-First Search (IDDFS), Greedy Best-First Search and Bidirectional Search, as these were the core algorithms specified for the assignment. In addition to these, we also implemented Breadth-First Search (BFS) and Depth-First Search (DFS) as extra algorithms. Including BFS and DFS allowed us to compare our main methods with these classic approaches, giving us a broader perspective on the strengths and weaknesses of each strategy.

BFS is well-known for always finding the shortest path in an unweighted maze, while DFS explores as far as possible along each branch before backtracking. By adding these two extra algorithms, we were able to see how their performance and results stacked up against the more advanced methods like A\* and Bidirectional Search. This made our evaluation more thorough and helped us better understand the trade-offs between different search strategies.

Visualisation is another key aspect of our project. The visualisation module provides clear, step-by-step displays of the maze, the search process and the final solution path, making it much easier to follow how each algorithm operates and to communicate results to users. Below is a visualisation of a feature where the user generates a custom maze:

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Maze Creation Menu

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure - Maze Generation with Obstacles

A screenshot of a computer program

AI-generated content may be incorrect.

Figure - Custom Maze Menu with BFS

A maze search visualization grid

AI-generated content may be incorrect.

Figure - Maze Search Visualisation

A graph with a bar

AI-generated content may be incorrect.

Figure - Maze Search Graph Comparison

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure - Maze Search Visualisation comparison

A screenshot of a maze

AI-generated content may be incorrect.

Figure - Visualisation for custom maze

A graph on a screen

AI-generated content may be incorrect.

Figure - Custom Maze Graph Visualisation

# 8.0 Conclusion

In this project, we set out to solve a challenging maze using a variety of search algorithms and logical inference methods. By implementing six different algorithms we were able to compare a wide range of strategies and see how each one performed in practice. Our logical inference module, with both forward and backwards chaining, provided a different perspective on problem-solving and made the reasoning process transparent.

Through testing and comparison, we found that algorithms like A\* and Bidirectional Search offered the best balance of speed and path quality, while Greedy was the fastest but not always optimal. Logical inference methods were slower but gave clear, step-by-step explanations of how solutions were found. The project’s modular design, user-friendly interface, and visualisation tools made it easy to experiment with different methods and understand their results.

# 9.0 References

* GeeksforGeeks (2025) ‘Breadth First Search or BFS for a Graph’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ (Accessed: 30 April 2025).
* GeeksforGeeks (2025) ‘Depth First Search or DFS for a Graph’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ (Accessed: 30 April 2025).
* GeeksforGeeks (2025) ‘Iterative Deepening Depth First Search (IDDFS)’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-search-iddfs/ (Accessed: 30 April 2025).
* GeeksforGeeks (2025) ‘A\* Search Algorithm’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/a-search-algorithm/ (Accessed: 30 April 2025).
* GeeksforGeeks (2025) ‘Greedy Best First Search Algorithm’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/ (Accessed: 30 April 2025).
* GeeksforGeeks (2025) ‘Bidirectional Search’, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/bidirectional-search/ (Accessed: 30 April 2025).
* Wikipedia (2025) ‘Breadth-first search’, Wikipedia. Available at: https://en.wikipedia.org/wiki/Breadth-first\_search (Accessed: 30 April 2025).
* Wikipedia (2025) ‘A\* search algorithm’, Wikipedia. Available at: https://en.wikipedia.org/wiki/A\_search\_algorithm (Accessed: 30 April 2025).
* Computerphile (2014) ‘A\* Pathfinding Algorithm - Computerphile’, YouTube. Available at: https://www.youtube.com/watch?v=ySN5Wnu88nE (Accessed: 30 April 2025).
* Python Software Foundation (2025) ‘Python 3.12 Documentation’, Python.org. Available at: https://docs.python.org/3/ (Accessed: 30 April 2025).
* Python (2025), available at: <https://www.w3schools.com/python/>. (Accessed 10th April 2025)