Designing Intelligent Agents CW1

Harry Bridgen

Introduction

This project explores the design and evaluation of several autonomous agents, within a simulated maze environment. The agents are tasked with navigating the procedurally generated maze, starting at the top left corner, making their way to the bottom right corner. The aim is to compare different pathfinding approaches in terms of their efficiency, effectiveness and steps taken to reach the goal.

The environment is implemented in Python, using PyGame for optional rendering. The program generates random mazes, which can be resized based on constants found in the constants.py file. Four agents are implemented: Random, Depth-First search, A\* and Greedy Heuristic. These implementations can be found in the brain.py file. Each agent is evaluated 1000 times on different maze configurations, allowing for an average performance metric to be collected. Each round of testing generates a new maze, and each agent solves the same maze, to ensure fair testing and controlled variables for each round.

The focus of this coursework is not a complex visualization, rather comparing how different Intelligent Agents and pathing algorithms respond to randomly generated mazes environments.

Background

This project focuses on comparing different algorithmic agents tasked with solving a pathfinding problem: starting at the top left and finding a path to the bottom right through a procedurally generated maze. The maze environment simulates a constrained partially visible space, similar to real problems such as network pathfinding, road/GPS navigation or game agents.

To achieve this, I first wrote a simple algorithm to procedurally generate mazes, then I designed and developed 4 intelligent agents: Random movement, Depth-first search, greedy heuristic and A\*.

The random agent moves without regard for the goal, only avoiding walls and previously visited cells. While not efficient, it serves as a baseline comparison, highlighting the value of more sophisticated search algorithms.

The greedy heuristic agent evaluates neighbouring cells based on their Manhattan distance to the goal (i.e. the total number of horizontal and vertical steps required to reach it) then chooses the cell with the lowest value. While it does reach the goal faster than the random agent, it can take inefficient paths, especially if the maze has a lot of paths leading to dead ends.

The Depth-First Search (DFS) agent explores as far as possible down one path before backtracking. While it doesn't guarantee the shortest path, it performs reasonably well in simpler mazes. Its behavior is somewhat like the classic 'keep your hand on the left wall' rule, though DFS explores any unvisited direction rather than following a physical wall.

Finally, the A\* Search agent combines the actual cost from the start (g(n)) and the estimated cost to the goal (h(n)) to choose paths that are both cost-effective and directed. A\* is considered optimal and complete when using an admissible heuristic, and it often finds the shortest path more efficiently than Greedy or DFS.

This range of agents allows us to examine how these different approaches to pathing behave within randomly generated mazes, and we can paint a clear picture of the benefits of using A\* and similar algorithms.

Design

The project is designed to evaluate the effectiveness of each Agent through a procedurally generated maze. The system consists of three main components. The environment (Maze.py), the agent (Agent.py) and the brain for the Agent (Brain.py). The Maze.py contains a Maze class, which generates random mazes on demand. It stores the maze a 2x2 list, and this ensures that it is easy to display and the Agents can read the environment easily. The Agent.py file contains the Agent base class, which contains a reference to a Brain class. It also encapsulates the position of the Agent, the goal, the start location and its current path. The Brain.py file contains multiple different types of Brains, each containing a different pathing algorithm for the Agent. These brains represent increasing sophistication:

* **RandomBrain** selects random unvisited directions, serving as a baseline.
* **DFSBrain** explores paths deeply before backtracking, suitable for exploring but not optimized for efficiency.
* **HeuristicBrain** uses a greedy approach based on Manhattan distance to the goal.
* **AStarBrain** combines actual cost (g(n)) and heuristic estimate (h(n)) for optimal, informed pathfinding.

To ensure a fair comparison, each simulation round generates a new maze, and then all agents navigate the same maze. This system (by default) executes 1000 rounds, then averages the steps taken for the mazes. The optional interface is built to render agent movements side-by-side, but this will significantly slow down the program. It is recommended when using larger values for simulation rounds, to turn RENDER in Constants.py to false.

Implementation

The implementation is written using Python, with a modular structure that can be easily extended or modified. Each file encapsulates a distinct responsibility, following good object-oriented practices.

The Maze class generates random grid-based mazes using a depth-first search algorithm. The walls are represented as “1” and open spaces are represented by “0”. The Maze is guaranteed to have a solvable route from the start (top left) to the end (bottom right). Maze generation is procedural and happens at the start of each simulation round.

The Agent class is a wrapper around a given “Brain” object. It stores the Agent’s current position, start and end goals, and its full path history. The key method “run\_stepwise()” initializes the Agent with a given Maze and iteratively calls “next\_move()” function which is implemented by the Brain.

The Brain class is implemented as a sub-class of a base Brain class. Each implementation must have two key methods: “reset()”, which prepares for a new simulation round, and “next\_move()”, which calculates the next step for the Agent.

The optional visual output is handled by PyGame and can be toggled via the RENDER constant in constants.py. When enabled, a UI appears that shows 4 identical mazes, each with a different Agent. PyGame is a Python wrapper for the SDL graphics library, allowing lightweight 2D rendering.

Experiments

To evaluate the efficiency of each Agent, the program automatically runs a predefined number of rounds. This is found in constants.py: SIMULATION\_COUNT. Each experiment prints a list of each simulation round, including the steps taken by each Agent. Then, at the end, it returns the average steps for each Agent. This allows us to not only see specific timings, but a look at the average steps for each Agent.

The key metric used is the number of steps taken to solve the maze by each Agent. This gives us a clear image of how effective that Agent is in relation to others.

Results (400-500 words)

Each Agent was tested across 1000 mazes. The primary metric used is the number of steps taken to complete each maze. After averaging the results, the average number of steps taken by each Agent was:

|  |  |
| --- | --- |
| **Agent** | **Average Steps (1000 rounds)** |
| A\* | 622.67 |
| Heuristic | 768.46 |
| Random | 1684.76 |
| DFS | 2572.23 |

Table 1 - Averages

A screenshot of a graph

AI-generated content may be incorrect.

Figure 1 - Averages

The data clearly shows that A\* is the most effective algorithm for solving the mazes. Heuristic followed closely behind, showing comparable performances but is slightly less effective on average. The random Agent was significantly less effective, requiring almost double the number of steps on average as A\*. Interestingly, the DFS algorithm performed significantly worse on average than even the random Agent, requiring almost 1000 more steps than the random Agent on average. However, deeper analysis through distribution and trend graphs revealed insightful and nuanced behaviour.

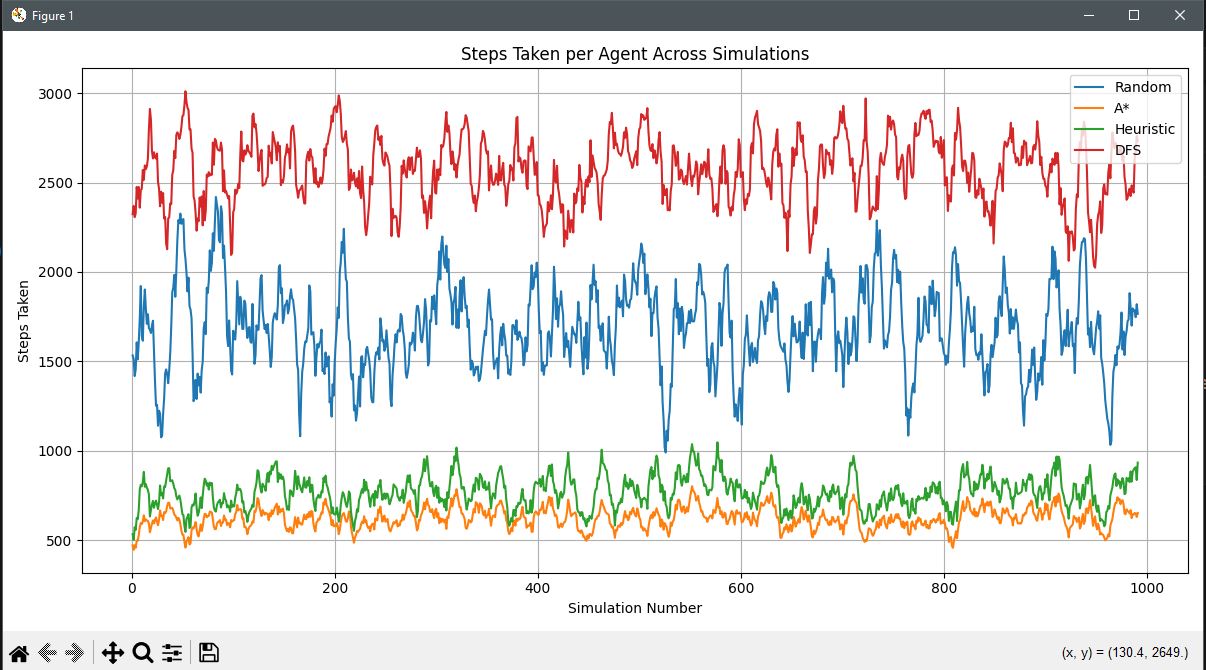


Figure 2 – Rolling Averages

Figure 2 shows rolling averages, using a window size of 10. It uses each Agents performance in each simulation round over 1000 rounds. Examining this graph, we can see that A\* and Heuristic are very consistent in their performance, while random has a very big span, and DFS also has a larger span, but slightly smaller than random. This shows that the heuristic Agent and A\* Agent are very reliable, stable and consistent for solving these kinds of problems.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 3 - Box Plot

Figure 3 shows a box plot of each Agent’s step distribution. This plot offers additional insight. While DFS performances worse than Random on average, DFS has *many* outliers to the bottom, some performing as good as the lower bound for the A\* and Heuristic boxes. This suggests that DFS sometimes finds a very efficient path through the Maze. In contrast, Random’s performance is more uniform, with very even distribution of steps taken. Despite this, random does appear to be the more efficient algorithm overall for these specific mazes.

Conclusion

This project successfully shows and compares the efficiency of four autonomous pathfinding agents within a procedurally generated maze environment. The results show that sophisticated search methods such as A\* and Heuristic dramatically outperformed strategies such as Random and DFS. A\*, as expected, performed the best on average and was the most stable in averages steps taken. While DFS did sometimes produce fast results, its lack of direction showed to be much more of a factor in its poor performance. Random search, though it performed poorly, was surprisingly stable compared to DFS. These finding align with the expectations of agent-based navigation, reinforcing the value of intelligent and sophisticated search algorithms to improve performance and efficiency.

This project, over time, could be extended to include more Agents, different maze environments, partial observability and much more. These additions would allow for the research and development of more complex pathing algorithms.