A1 Report

# Implementation

As part of this assignment, various search and MDP algorithms were implemented in Python code. These algorithms were applied to the problem of maze solving, using an AI gym maze environment [cite GitHub maze env]. The maze environment was further adapted to be made compatible with the design and implementation of these algorithms. These implementations are discussed below individually.

## Depth-First Search

The first search algorithm that will be discussed is Depth-First Search, a searching strategy that focuses on trying to find a goal state in a maze by exploring in deep but narrow bursts. Depth-First Search, or DFS, focuses on exploring an environment to its maximum depth before doubling back and trying another route, repeating until either a goal state is located, or all possible avenues of search have been exhausted.

Implementations of this search algorithm are quite similar to Breadth-First Search, however, the key difference in these algorithms is the application of a stack in DFS. The algorithm will always explore the state it has most recently found, leading to a depth-based approach to exploration. This behaviour is perfectly encapsulated by a stack, in which newly added states to the stack are always the ones to be removed first. This is known as last in, first out behaviour, or LIFO.

The Python code formulation of this algorithm can be seen in Appendix 1 [ref appendix 1 code here]. Various experiments were performed with this search algorithm, as well as an analysis of the results. All of this is discussed in the Results and Analysis section below.

## Breadth-First Search

Breadth-First Search, or BFS, is the second search algorithm that will be discussed. BFS is a searching strategy that tries to locate a maze’s goal state through sweepingly wide exploration. It spreads out across an environment in a wave, steadily moving the line between explored and unexplored, also known as a frontier, until it either exhausts the unexplored space or finds the goal state. As a result of its breadth-based exploration, it repeatedly backtracks itself to keep the frontier consistently even, within the allowable limits of the maze environment.

Implementations of this search algorithm, as mentioned previously, are similar to DFS, but are strikingly different in their key data structure used for deciding which states to visit next. BFS makes use of a queue, which operates a policy of first in, first out, or FIFO. This is a perfect representation of the behaviour of BFS, as it will visit all old states before visiting new ones. As a result, it explores the maze broadly, a slow yet thorough strategy.

The Python code formulation of this algorithm can be seen in Appendix 2 [ref appendix 2 code here]. As before, data has been gathered on this searching technique through experimentation and is discussed in sections below.

## A\* Search

The third and final search algorithm to be discussed is A\* Search, a heuristic-based approach that explores more dynamically based on the architecture of the maze environment. A\* explores environments by considering two attributes common to all states: the step cost from the start state, and the heuristic cost of getting to the goal state. The sum of these attributes is called the f score of the state, a value used by A\* to decide which states to explore next.

In this way, A\* is an informed searching algorithm, as it uses attributes of the current maze environment to determine the best course of action, i.e. which state to move to next. This is powered by the heuristic-aspect of the searching, which allows A\* to make educated guesses as to which states will lead to the goal state, and which states will be a waste of time to explore.

A\* makes use of a heuristic to estimate the cost of travelling to the goal state from the current state, enabling the algorithm to favour moving to nodes that are estimated to be close to the goal. Usually, there are two heuristics to choose from: the Euclidean distance and the Manhattan distance. The Euclidean distance is the straight-line distance between two states and is most commonly used in A\* when there are little to no obstacles in the environment. Conversely, the Manhattan distance is the sum of the vertical and horizontal distances between two states, giving a larger cost value but making it more appropriate for environments with walls and other obstacles.

For this implementation, the design made use of the Manhattan distance, as the nature of the maze meant a straight-line path was almost never the solution. For obstacle-based environments, the Manhattan distance gives a better estimation of the cost to the goal, making it the obvious choice in this application. Once the f score is calculated, states are considered using a priority queue. This data structure operates a special case of LIFO, which dynamically rearranges itself smallest to largest based on a key value. In this case, the key value is the f score, meaning A\* explores states with low f scores first, as a low f score creates an optimal path.

The Python code formulation of this algorithm can be seen in Appendix 3 [ref appendix 3 code here]. As before, experimentation, data gathering, and analysis is discussed in sections below.

## MDP Value Iteration

The first Markov Decision Process, or MDP, algorithm to be discussed is Value Iteration. This algorithm considers the environment as consisting of multiple states with individually assigned state values. It uses a special equation that considers transition probability, movement reward, a discounting parameter and a neighbour’s state value to update a state’s value. This process of updating state values is repeated until the state value changes are below a threshold.

MDP algorithms differ from the previously described search algorithms because they operate by pre-calculating the best action, or policy, to take at each state such that every policy always leads to the goal state, rather than exploring the environment with a specific set of rules until the goal state is discovered.

Value Iteration operates by starting all states with a state value of 0, except for the goal state, whose value is a positive number. The algorithm then propagates out from the goal, assigning state values and policies, or actions, to each state until there is a state value convergence. Upon convergence, the state values have not changed sufficiently since the last iteration, meaning a stable path has been found.

State values are determined using the Bellman equation, which is show below:

\*\* Add bellman equation into Latex as formula.

\*\* where, ….blah blah mention deterministic aspect for comparison

Using this equation Value Iteration computes the best state to move to from the current state, assigning a policy to each state in the environment. Once state value propagation is complete, a path to the goal state from every other state has been calculated.

For this design and implementation, a value of 0.9 was chosen for gamma, and a theta of 1xe^-7 was chosen. A high gamma was chosen to promote propagation through the maze of high state values, improving performance on large mazes. Similarly, a theta of such small magnitude was chosen to prevent the algorithm from converging too early, leaving portions of the environment undetermined. A very small theta means that very small state values can be assigned without convergence occurring. As well as that, a large state value of 100 was chosen for the goal, to further promote state value propagation.

The Python code formulation of this algorithm can be seen in Appendix 4 [ref appendix 4 code here]. As before, experimentation, data gathering, and analysis is discussed in sections below.

## MDP Policy Iteration

MDP Policy Iteration is the second and final MDP algorithm that will be discussed. It functions by breaking up the maze environment into multiple states and assigning each state a specific policy to indicate the best move to make to reach the goal. It applies the Bellman Equation in two key steps until all policies of the environment have converged: Policy Evaluation and Policy Improvement.

At start, all states are assigned random policies, or the same policy, as an initialization. Policy Evaluation is then performed using the Bellman Equation. This step calculates the state values across the environment, but only using the single initial policy. Upon value convergence, the algorithm moves on to Policy Improvement. Using a modified Bellman Equation, the best policy for each state is determined. If every existing state policy is found to be the best policy, there has been total convergence, and a path has been found from every state to the goal state.

The modified Bellman Equation used in Policy Improvement is show below:

\*\*equation

\*\*description

As with the implementation of Value Iteration, this implementation of the algorithm is deterministic, meaning the transition probability is a constant of value 1. This is to allow an even and fair comparison between MDP algorithms and search algorithms, such as DFS.

As before, a value of 0.9 was chosen for gamma, and a value of $1.0e-7$ was chosen for the convergence threshold. The same reasoning applies for these design choices; they both promote better value propagation and value convergence in Policy Evaluation. They were also kept exactly the same as in Value Iteration for fairness between the two algorithms.

An interesting aspect of Policy Iteration is that, while it features a modified value iteration technique for a single policy per state, its policy convergence is discrete. What this means is that a policy can only be changed or unchanged; there is no theta to determine if the change is within a threshold. At the same time, Policy Evaluation does feature a theta in its value iteration, which does follow threshold behaviour. This has potential for different convergence behaviours when compared to Value Iteration.

The Python code formulation of this algorithm can be seen in Appendix 5 [ref appendix 5 code here]. As before, experimentation, data gathering, and analysis is discussed in sections below.

# Evaluation

Upon completion of the five algorithm implementations, an evaluation was carried out to determine their performance and operational behaviour. This is broken down below into two sections: Search Algorithms and MDP Algorithms.

## Search Algorithms

The evaluation of the three search algorithms started with a basic test of their execution on a moderately sized maze environment. For this initial evaluation, a maze size of 15 was chosen, meaning the maze featured 225 states, including the start state and the goal state. As each algorithm solved the maze in real time, the explored states were coloured in to visually highlight the scope of exploration. At the end of runtime, the found path was also coloured in to indicate the route found.

Figures 1-3 [ref figures 1-3] show the visual output of each algorithm in a maze environment at the end of runtime. Interesting insights can be drawn from these visual representations. The output for DFS highlights its depth-based approach to solving, as its explored path perfectly matches its final solution path. As the goal state exists at the maximum depth from the start state, DFS is likely to find the goal without much unnecessary exploration. However, this does not mean DFS finds the optimal path, rather, it finds the most readily available path, regardless of efficiency.

This contrasts strikingly with BFS, whose output displays extensive exploration before a solution was found. BFS’s breadth-based approach will explore the majority, if not all, of a maze before finding the goal, especially when the goal state is at max depth. While this is more time consuming than DFS, it pays off because BFS can find the optimal path with this approach. Choosing between these two search algorithms comes down to time and path efficiency: pick DFS if speed is important, but an optimal distance is not, or pick BFS for an optimal solution when time is not important.

Finally, A\* can be considered visually as a mix of DFS and BFS. It exhibits depth-based behaviour, in that it spreads out in narrow channels, but also stems of from these channels to make them broader, a hallmark of BFS. Despite exploring less states than BFS, A\* can still find an optimal path, thanks to its heuristic based approach that makes it more intelligent than the previous two algorithms.

Upon completion of this initial basic evaluation, benchmarking on these three algorithms took places. Each algorithm was executed on the same 10 maze environments, and metrics surrounding their operation were collected. These metrics are the average solve time for each searching strategy, the average number of states that were explored, the average length of each path found, and the average memory footprint of each algorithm. These results and their analysis are discussed in the Results and Analysis section below.

## MDP Algorithms

The evaluation of the two MDP algorithms was much the same as with the three search algorithms, in all respects except for a change in metrics and the resulting visual output. As MDP algorithms operate by pre-calculating values for each state to determine a path, there are no explored states to showcase. Due to this fact, the visual output shows the state values for value Iteration, or the state policy for Policy Iteration.

As with the search algorithms, a maze of size 15 was used, with 225 states that include the goal and start states. For Value iteration, a value for each state is shown in the visual output, rounded to two decimal points. For Policy Iteration, a letter for each state is shown in the visual output, indicating the final policy, e.g. N is North, W is West, etc. As well as these displays, the final path is also coloured to show the solution the algorithms found. Figure [ref visual outputs] shows the visual output of the two algorithms in a maze environment at the end of execution.

As before, some insights can be drawn from the outputs. The many state values shown in value Iteration paint a better picture of how value convergence works. Starting with a state vale of 100 in the green goal, state values are propagated outwards in a slowly decaying fashion. This slow decay is cause by the discount value in the Bellman Equation. As a result of the high discount value, every state value can be observed to be a positive number of no less than 5. An investigation into the effects of the discount value on the algorithms performance is discussed in the Results and Analysis section below.

The visual output for Policy Iteration features each state’s policy displayed in a letter code format to indicate direction. As described previously, this MDP algorithm will calculate a path to the goal from every state, which is highlighted by the policies. At every dead end, the policy is to turn back, and all policies at boundary walls lead the agent back to the central path coloured by the algorithm, perfectly highlighting the global solution this algorithm has found.

Upon completion of this initial basic evaluation, benchmarking on these two algorithms took place. As with the search algorithms, each algorithm was executed on the same 10 maze environments, and metrics surrounding their operation were collected. These metrics are the average solve time for each searching strategy, the average number of iterations before convergence, the average length of each path found, and the average memory footprint of each algorithm. These results and their analysis are discussed in the Results and Analysis section below.

# Results and Analysis

Upon evaluation of each algorithm, results on their performance and behaviour were gathered to determine certain behavioural traits of each algorithm, as well as for comparison individually, as well as between Search and MDP. These results and their analysis are discussed below, broken up into three sections: All Algorithms, where the metrics common to all algorithms are shown and discussed, Search Algorithms, where the metrics exclusive to search algorithms are shown and discussed, and MDP Algorithms, where the metrics exclusive to MDP algorithms are shown and discussed.

## All Algorithms

When evaluating all algorithms together, three core metrics were measured that are common to all algorithms: their solve time, their path length, and their memory footprint. The solve time represents the exact time in seconds it took each algorithm to find a solution to the maze environment. The path length is the shortest path from the start state to the goal state that the algorithm was able to find for the maze environment. Finally, the memory footprint is the total memory consumption in bytes of the algorithm by the end of execution.

These three metrics were measured by running each algorithm 10 times on the same 10 mazes, on maze sizes from 5 to 60 in steps of 5. After measuring each metrics for 10 executions of a given maze size, the average was found for each algorithm and plotted on graphs for analysis. Figure [ref all algo plots] showcases these graphs, which colour code each algorithm onto the same plot for easy comparison.

The first sets of results to analyse are the solve time metrics. From maze size 5 until 45, MDP Policy Iteration can be seen to take the most time, likely due to its two-step process that involves a double convergence. However, after maze size 45, BFS becomes the slowest algorithm to maze solve. This result is inline with the expected performance of BFS, as its behaviour strives to reach as much of the maze as possible while penetrating its depth. Naturally, on large maze sizes, this results in very small gains on its frontier, meaning very slow solving times.

Oppositely, DFS has a near constant solve time that ranks fastest of all algorithms. Again, this is expected behaviour from this algorithm, as it focuses on narrow penetration of the maze. As the goal state in each maze is at max depth, it only takes DFS a handful of channels to find the goal state, leading to fast solve times. A\* ranks in the rough centre of the two measured extremes, highlighting its efficiency in search due to its heuristic approach.

The second set of results to analyse are the path length metrics. Immediately, there is a clear trend among the algorithms. Excluding maze size 5, all maze sizes have DFS finding longer paths than every other algorithm. In fact, every other algorithm is equal in path length for all maze sizes, an easily explainable phenomenon. This equality of results means all four algorithms were able to find similarly optimal path through the maze, which lead to level results after averaging. For every algorithm this occurred for, finding the optimal path is entirely likely.

BFS explores so much of the maze environment that it can easily tell where the optimal path is, while A\*’s informed searching makes it incredibly likely to find the optimal path at every execution. As for the MDP algorithms, they consider the entire maze environment when calculating paths, and it allows them to find the optimal path with little extra effort. The behaviour of DFS being the outlier and longest path is another hallmark of the algorithm.

As discussed previously, DFS’s main advantage is its speed to find the goal state, but this is almost always eclipsed by the fact that it achieves this by finding any available path. In a maze environment with multiple paths, the path it finds is rarely the optimal path. Hence, it performs the worst in path length. Interestingly, at maze size 5, all algorithms are equal in path length, which leads to the conclusion that since the maze is so small, with 25 states, the task of finding the optimal path is not as hard as with the other mazes. Most likely, there are much fewer paths to the goal, making the chances of finding the optimal path randomly much higher.

The third and final set of results common to all algorithms are the memory footprint metrics, which measure how much space each algorithm consumed by the end of execution. As with the solve time, BFS is clearly the largest consumer of memory, a trend which is present for all maze sizes. Once again, this is caused by the operational behaviour of BFS; as it visits and explore more states than any other algorithm, it consumes more memory keeping track of where it’s been. By the end of execution, it has nearly every state stored.

Conversely, DFS has minimal memory use, as it solves each maze environment so fast it barely explores any states, causing it to record very little data. This is inline with observations from the solve time graph. The remaining algorithms occupy the rough centre of the graph, with A\* falling lower than Value Iteration and Policy Iteration, both of whom are equal. As Value Iteration and Policy Iteration do not explore add focus more on calculations in rigid data structures, their graphs are reliably smooth and low, as the amount of data they consume at each run are directly proportional to the maze size. As A\* is smart about its exploration, it consumes less memory than BFS, but still more than DFS, as there are still a number of states to visit when finding the optimal path.

## Search Algorithms

Following the evaluation of all algorithms on common metrics, algorithm-specific metrics were measured. For search algorithms, this was their number of explored states when finding a solution. This metric represents the number of states the agent had to travel to before reaching the goal, and thus finding a path from the start state to the goal state. As before, this metric was measured by running each search algorithm 10 times on the same 10 mazes, on maze sixes from 5 to 60 in steps of 5. Once measurements were complete, the average for each maze size and algorithm were found.

Figure [ref explored graph] displays the graph of these results, which colour code each algorithm onto the same plot for easy comparison. As with previous graphs, BFS is ranked the highest for this metric, and DFS is ranked lowest. Both of these results are expected given the behaviour of each algorithm; BFS explores breadth-first, meaning it visits almost every state in a maze before reaching the goal state. DFS, on the other hand, explores depth-first, meaning it visits minimal states before reaching the goal state. An interesting observation in the BFS graph is that it’s a smooth plot, suggesting there is a specific relation between maze sizes and explored states that is constant for the algorithm.

A\* is ranked in the middle of these two, which is backed up by how A\* functions. Using a heuristic, it makes informed decisions about which states to visit, meaning it visits as little states as it can while also finding an optimal route. This is why it ranks below BFS, which explores many states regardless of their importance to an optimal route, and why it ranks higher than DFS, which only cares about reaching the goal state as quickly as it can, regardless of state importance or the path it finds.

## MDP Algorithms

The MDP-specific metrics were the last to be measured and analysed, and consisted of a single metric, as well as an investigation into the effects of varying a parameter of these algorithms. The metric specific to MDP algorithms is Iterations, also known as the number of cycles each algorithms takes before converging and finding a solution to the maze environment. As before, the Iterations metric was measured over 10 executions of the MDP algorithms on the same 10 mazes, on maze sizes from 5 to 60 in steps of 5. Once final measurements were made, the average for each maze size and algorithm were calculated.

Figure [ref iterations graph] shows the graph of these results, which colour code each algorithm such that they can be viewed in the same plot. For the entirety of the graph, Policy Iteration ranks slightly lower than Value Iteration, but not by a large margin. This suggests that Policy Iteration converges faster than Value \iteration, in terms of iterations. However, when examining results on the solving time graph, it was very clear that Policy Iteration has a longer runtime than Value Iteration, meaning it takes longer to converge with respect to time. These results appear to be conflicting, but they are explainable. As part of Policy Iteration, the Policy Evaluation step existing to converge the existing policy using state values. This is somewhat equivalent to Value Iteration as a whole, but Policy Iteration must do this step and another, which is then considered an iteration.

It is likely that Policy Iteration runs slower because it is performing many more calculations per iteration than value Iteration. At the same time, because of these extra calculations, Policy Iteration makes more progress each iteration towards reaching the goal state and solving the maze, meaning it should converge in fewer iterations. This is the exact behaviour observed in these results, making it expected behaviour given each algorithms operation.

One final evaluation on the MDP algorithms is an investigation into the effects of changing the discount, or \gamma, in the Bellman Equation. For this investigation, Value Iteration will be used. Based on the equation, it can be assumed that decreasing the discount will reduce the propagation of state values, as each neighbour shares less and less of their state value when using the equation. In practice, the results are even more intriguing.

Figure [ref gamma figures] showcases two visual outputs of the Value Iteration algorithm, the first with $\gamma = 0.9$, and the second with $\gamma = 0.5$. As seen in the images, the state values are far lower with a smaller discount. As previously mentioned, this is t be expected due to the level of propagation resulting from a small \gamma. However, it is also observed that many state values are displayed as zero. What this means is that the real state value is smaller than two decimal places and cannot be shown, but it also means that the value of theta, or threshold for convergence, starts to become very important at lower discount values.

If such small state values can be calculated, then there is a distinct possibility that they may be discarded due to early convergence, as they are beyond the threshold accidentally. Further investigation confirmed this, as lowering the value of \gamma below 0.5 resulted in a maze solving failure, explainable through early convergence. With a low discount value, the propagation of state values grows weaker the further from the goal state the calculations are made, meaning that convergence may occur early, leaving states near the start without a properly calculated route to the goal state, hence failure. The only solution to this problem would be to increase the state value of the goal, or to reduce the value of theta, pausing convergence for more iterations.