CS7IS2 - Artificial Intelligence: Assignment 1

Priyansh Nayak
Student ID: 25350660

Contents

1 Introduction

This project investigates two major approaches to solving maze navigation problems: classical search algorithms and Markov Decision Process (MDP) methods. The objective is to compare how these fundamentally different techniques behave in terms of solution quality, computational cost, and scalability.

The search-based methods implemented are Depth-First Search (DFS), Breadth-First Search (BFS), and A* search with both Manhattan and Euclidean heuristics. These algorithms expand nodes selectively, exploring only the portion of the maze necessary to reach the goal.

The MDP-based methods implemented are Value Iteration and Policy Iteration. Unlike search, these methods compute utilities over the entire state space and produce an optimal policy rather than a single path. The maze is formulated as an MDP with deterministic transitions, step penalties, and discounted rewards.

Experiments are conducted across varying maze sizes and openness levels to evaluate runtime, nodes explored, memory usage, and convergence behaviour. The results highlight the structural differences between local search-based planning and global policy computation.

2 Implementation and Design Choices

2.1 Maze Representation and Generation

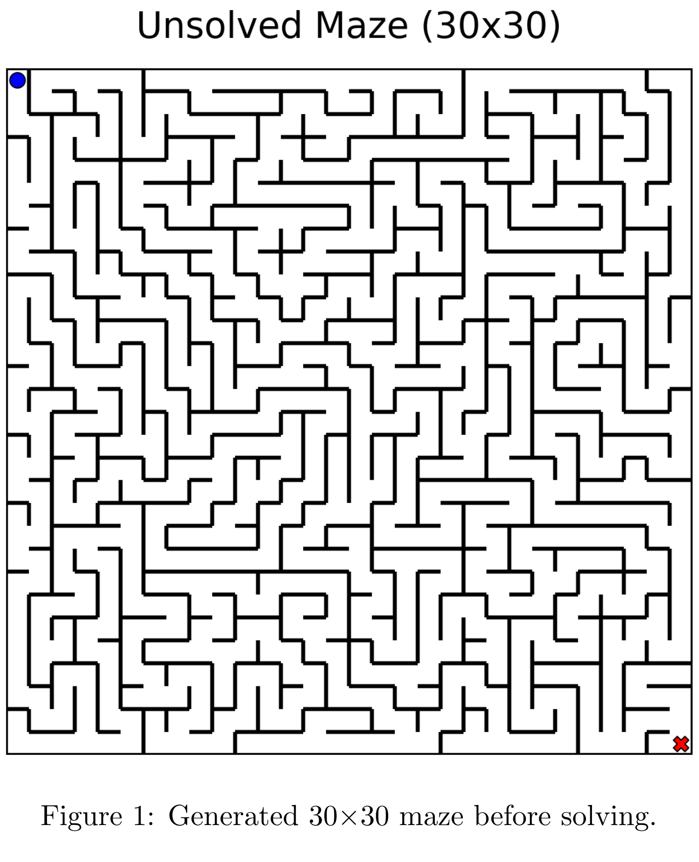

Figure 1: Generated 30×30 maze before solving.

The maze is represented as a two-dimensional grid of cells, where each cell maintains information about the presence or absence of walls in the four cardinal directions (North, South, East, West). This representation allows efficient neighbour lookup and ensures that movement constraints are explicitly encoded in the state structure. The start and goal cells are fixed at opposite corners of the grid to ensure consistent experimental conditions.

The maze generator was implemented from scratch rather than using an external library. This was done to retain full control over structural properties of the maze and to enable systematic experimentation. The generation procedure follows a depth-first recursive backtracking strategy. Starting from an initial cell, the algorithm repeatedly selects a random unvisited neighbour, removes the wall between the current cell and the neighbour, and continues recursively. When a dead-end is reached, the algorithm backtracks until another unvisited neighbour is found.

This process produces a *perfect maze*, meaning that there exists exactly one unique path between any two cells. While such mazes are structurally clean, they provide limited diversity for algorithm comparison because there are no alternative routes.

To introduce structural variability, an additional `openness` parameter is applied after maze construction. This parameter randomly removes a proportion of internal walls, creating loops and multiple possible routes. An openness value of 0 produces a perfect maze, while higher values progressively increase connectivity and reduce structural constraints. This mechanism allows controlled experimentation on how search and MDP algorithms respond to increased branching and path redundancy.

Implementing the generator manually ensured deterministic reproducibility through fixed random seeds and provided direct control over maze size and openness, both of which are central experimental variables in this study.

2.2 Search Algorithms

The maze can be modelled as an unweighted graph where each open cell represents a node and edges connect adjacent cells that are not separated by walls. Movement is restricted to the four cardinal directions, and each step has uniform cost.

2.2.1 Depth-First Search (DFS)

Depth-First Search explores the state space by expanding the most recently discovered node first, using a stack-based frontier. DFS is memory-efficient in practice but does not guarantee optimal solutions in graphs with multiple paths. In the context of maze solving, DFS often finds a valid path quickly, but the resulting path length can be significantly longer than the shortest possible path.

2.2.2 Breadth-First Search (BFS)

Breadth-First Search expands nodes level by level using a queue-based frontier. Since all moves have equal cost, BFS is guaranteed to find the shortest path in terms of number of steps. However, it may explore a large portion of the maze before reaching the goal, especially in highly open environments.

2.2.3 A* Search

A* search combines uniform-cost search with heuristic guidance. Nodes are selected for expansion based on the evaluation function:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the cost from the start node to the current node and $h(n)$ is a heuristic estimate of the remaining cost to the goal. When the heuristic is admissible (never overestimates the true remaining cost), A* is guaranteed to find an optimal path.

Two heuristics were implemented:

Manhattan Distance

$$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

This heuristic reflects the exact shortest-path distance on a grid without diagonal movement (ignoring walls). It is both admissible and consistent in this setting.

Euclidean Distance

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The Euclidean heuristic measures straight-line distance between two cells. It is also admissible but typically provides a weaker estimate than Manhattan distance in a four-directional grid, which may result in more node expansions.

By implementing both heuristics, the effect of heuristic strength on search efficiency can be directly observed while keeping the underlying search algorithm identical.

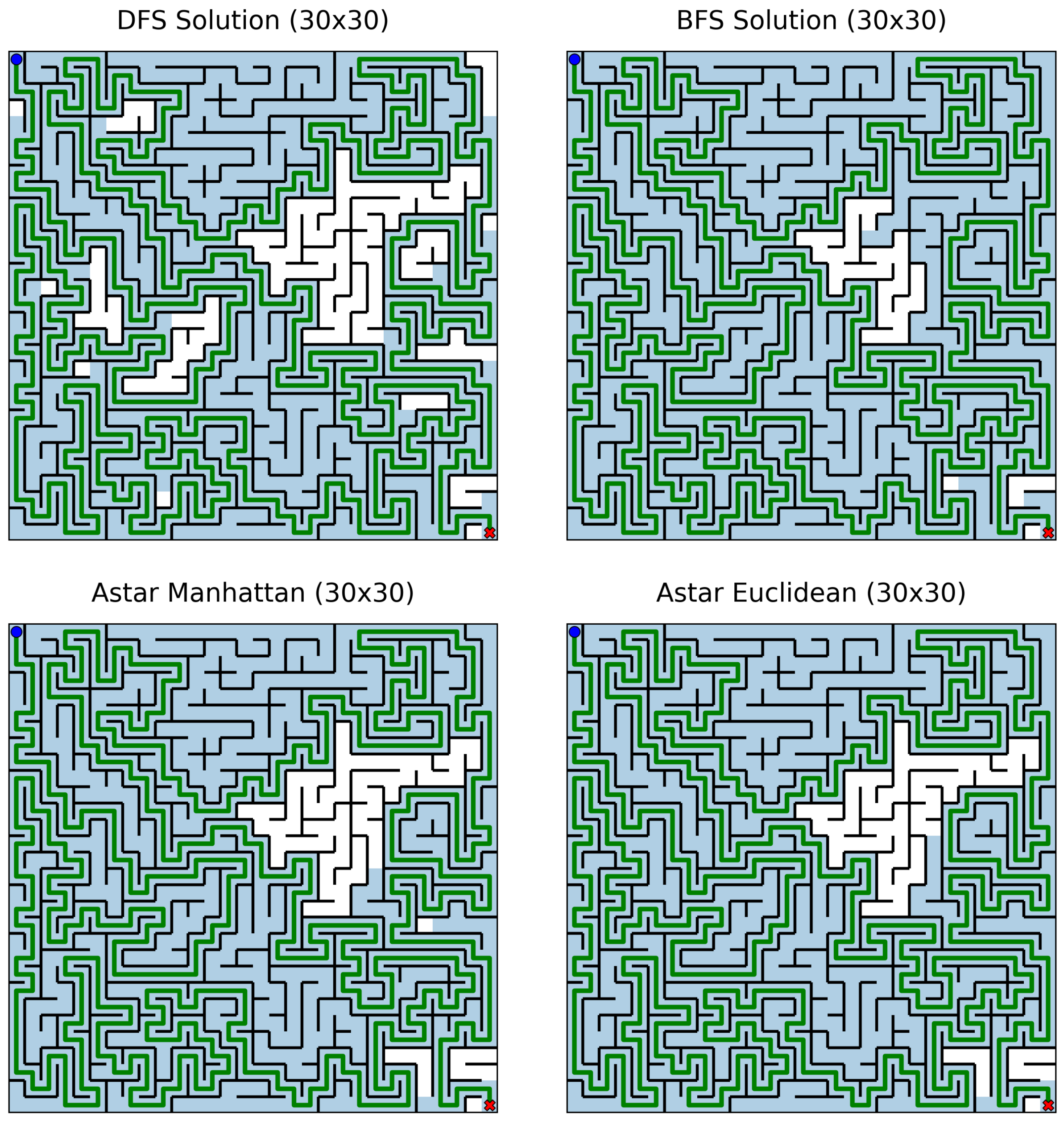

Figure 2: Example solutions on the same 30×30 maze, showing explored states (shaded) and the final path (green).

2.3 MDP Algorithms

To apply MDP methods, the maze is formulated as a finite Markov Decision Process defined by the tuple $\langle S, A, P, R, \gamma \rangle$.

- S: all reachable open cells in the maze.
- A: the four possible movement directions (North, South, East, West).
- $P(s' \mid s, a)$: deterministic transitions; attempting to move into a wall results in remaining in the same state.
- $R(s, a, s')$: a step reward of -1 for each move and a positive reward for reaching the goal state.
- γ: discount factor controlling the importance of future rewards.

The objective is to compute an optimal policy π^* that maximizes the expected discounted return from every state.

2.3.1 Value Iteration

Value Iteration computes the optimal state-value function by repeatedly applying the Bellman optimality update:

$$V_{k+1}(s) = \max_{a \in A} \left[R(s, a, s') + \gamma V_k(s') \right]$$

At each iteration, all states are updated using one-step lookahead until the maximum change between successive value functions falls below a small threshold θ. After convergence, the optimal policy is extracted by choosing, for each state, the action that maximizes the expected value.

Value Iteration updates both the values and the implied policy simultaneously and is guaranteed to converge for $0 < \gamma < 1$.

2.3.2 Policy Iteration

Policy Iteration separates the computation into two alternating steps:

1. **Policy Evaluation:** Compute the value function $V^\pi(s)$ for the current policy by iteratively applying:
$$V(s) = R(s, a, s') + \gamma V(s')$$
where $a = \pi(s)$.
2. **Policy Improvement:** Update the policy by choosing the action that maximizes the one-step lookahead value.

These steps are repeated until the policy stabilizes (no further changes occur). Although Policy Iteration often requires fewer outer iterations than Value Iteration, each iteration is computationally more expensive due to the policy evaluation phase.

2.3.3 Design Choices

A negative step reward encourages shorter paths, while a sufficiently large goal reward ensures that the discounted return still favours reaching the goal even in larger mazes. The discount factor γ was chosen close to 1 to allow long-term rewards to meaningfully propagate through the state space while still guaranteeing convergence.

Unlike search algorithms, MDP methods compute values for the entire state space rather than only the states along a single path. As a result, they produce a complete policy mapping from every state to an action.

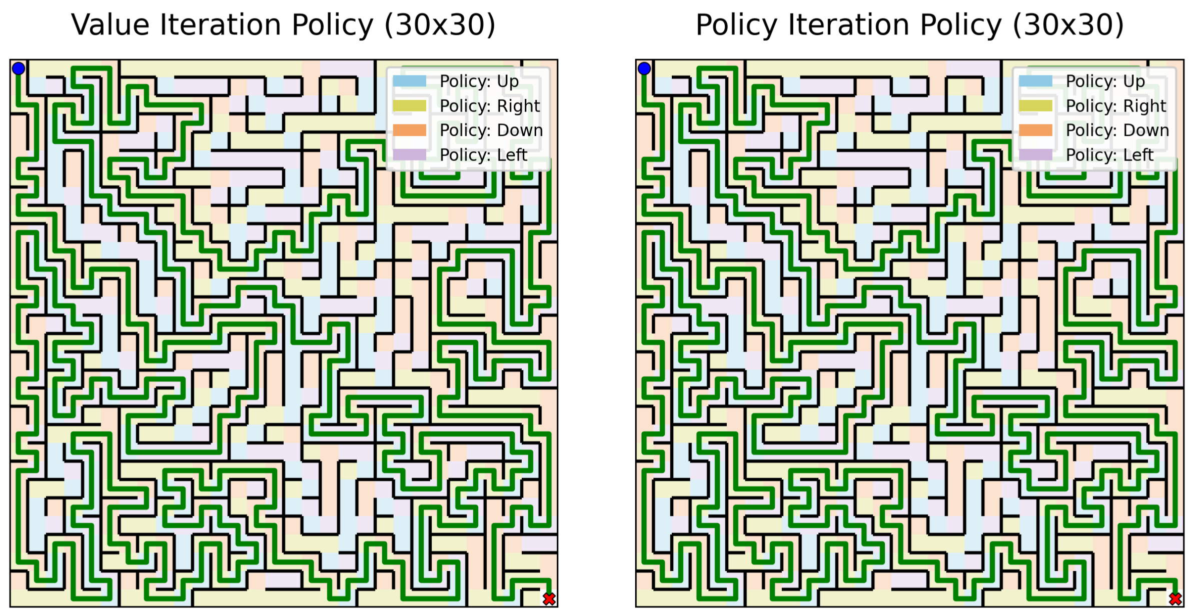
Value Iteration Policy (30x30)
Policy Iteration Policy (30x30)
Policy: Up
Policy: Right
Policy: Down
Policy: Left

Figure 3: MDP solutions on the same 30×30 maze. Both methods compute a full policy over the state space and produce the same optimal path to the goal.

3 Experimental Setup

3.1 Maze Sizes and Openness

Mazes of varying sizes were generated to evaluate scalability. The sizes tested were 10×10, 20×20, 30×30, 40×40, and 50×50 grids. Increasing the grid dimension increases both the number of states and the expected solution depth.

To study structural variation, an `openness` parameter was introduced. Openness values of 0.0, 0.1, 0.2, and 0.3 were tested. An openness of 0 produces a perfect maze with a single path between any two cells, while higher values introduce loops and alternative routes.

For size-scaling experiments, openness was fixed at 0.1. For openness experiments, maze size was fixed at 30×30. Multiple random seeds were used to reduce bias from a particular maze configuration, and results were averaged across runs.

3.2 Evaluation Metrics

The following metrics were recorded:

- **Path Length**: Number of steps in the final solution.
- **Nodes Expanded (Search)**: Total number of states explored.

- **State Updates (MDP)**: Total value updates performed.
- **Maximum Frontier Size (Search)**: Peak memory usage indicator.
- **Iterations (MDP)**: Number of outer iterations until convergence.
- **Runtime**: Wall-clock execution time in seconds.

All experiments were executed on the same machine to ensure consistent timing measurements.

4 Results

4.1 Search Algorithm Comparison

Nodes explored: As maze size increases, the number of expanded nodes grows rapidly (Figure 4). BFS consistently explores the largest number of states due to its level-wise expansion strategy. DFS explores fewer nodes but does not guarantee shortest paths. Among optimal solvers, A* with Manhattan distance expands significantly fewer nodes than both BFS and Euclidean A*, demonstrating the effectiveness of a stronger heuristic in a four-directional grid.

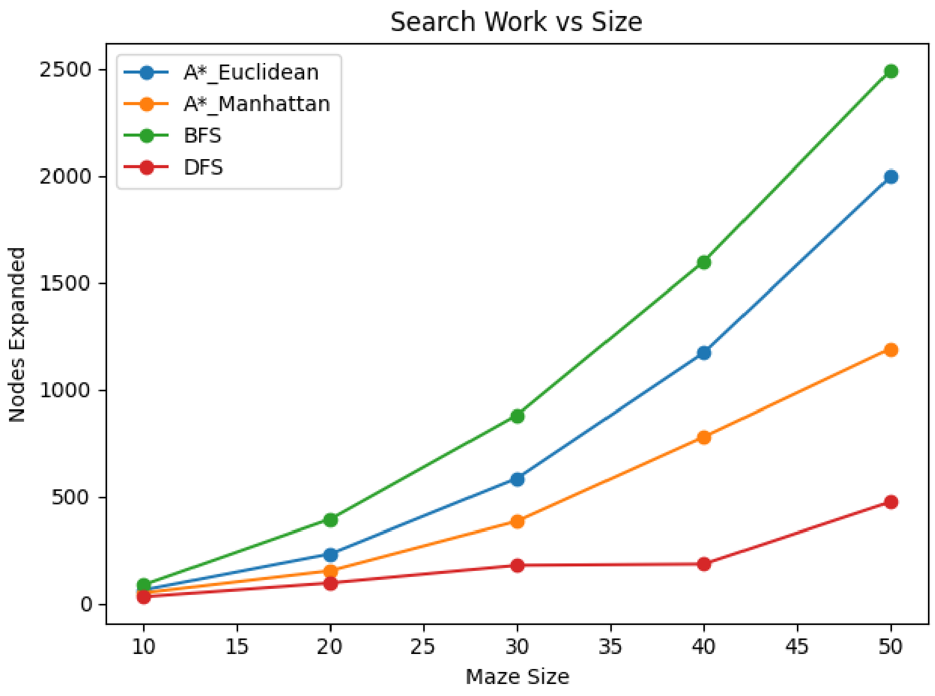

Figure 4: Nodes expanded across maze sizes.

Memory usage: Peak size reflects memory usage (Figure 5). BFS has the largest memory usage because it maintains all nodes at the current depth. DFS generally uses less memory but may accumulate deep branches before backtracking. A* requires intermediate memory due to maintaining a priority queue of states ordered by estimated cost.

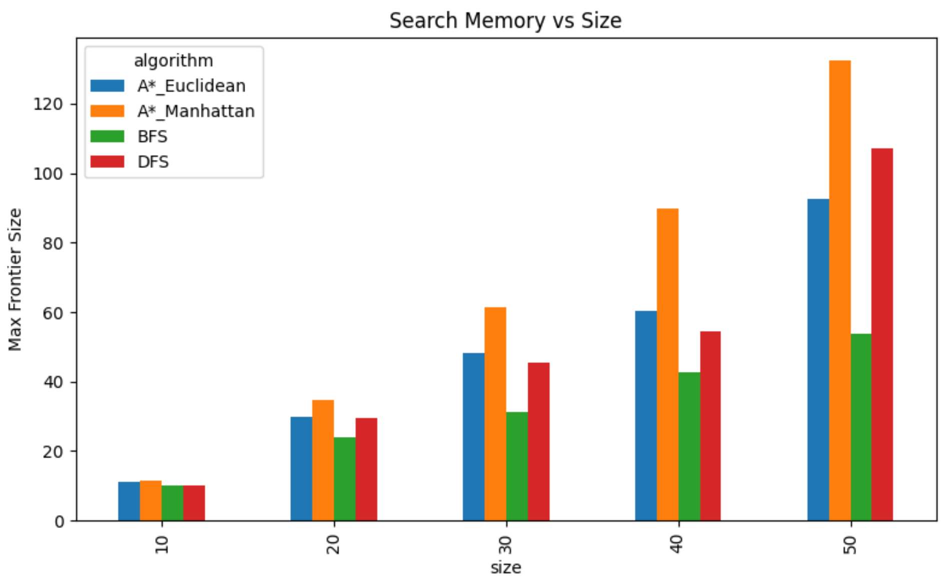

Figure 5: Maximum Memory usage across maze sizes.

Runtime vs openness. Increasing openness introduces additional branching and alternative routes (Figure 6). BFS runtime increases as the memory usage increases. A* remains comparatively stable, particularly with the Manhattan heuristic, as heuristic guidance continues to prioritise promising states. DFS runtime varies depending on early branch selection and the presence of structural shortcuts.

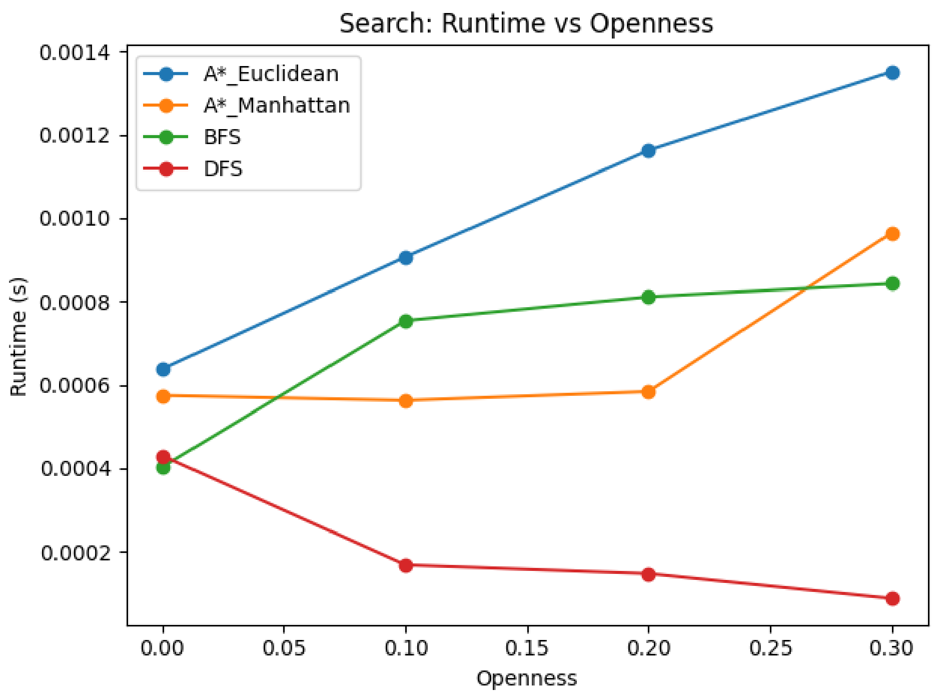

Figure 6: Search runtime as openness increases.

Runtime vs work: Runtime scales approximately linearly with the number of nodes expanded (Figure 7). This confirms that node expansion dominates computational cost in search-based planning. Algorithms that reduce expansions through heuristic guidance achieve proportional runtime reductions.

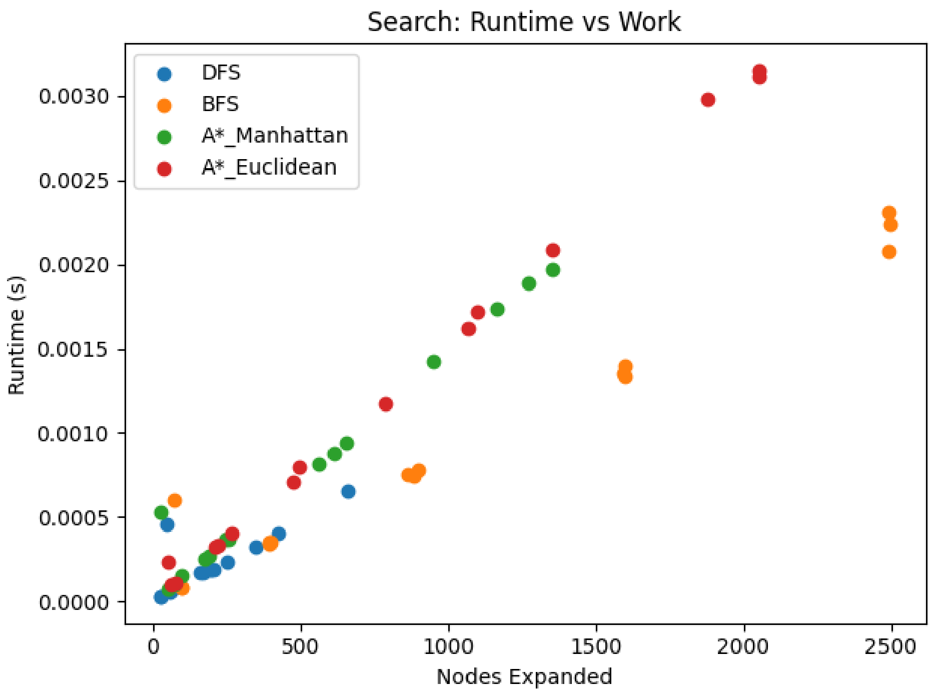

Figure 7: Relationship between nodes expanded and runtime for search algorithms.

4.2 MDP Algorithm Comparison

Work vs Size: Both MDP methods perform updates across the entire state space (Figure 8). The total number of state updates increases sharply with maze size. Policy Iteration performs substantially more updates than Value Iteration due to repeated policy evaluation phases. Unlike search methods, computational effort scales with total state count rather than solution depth.

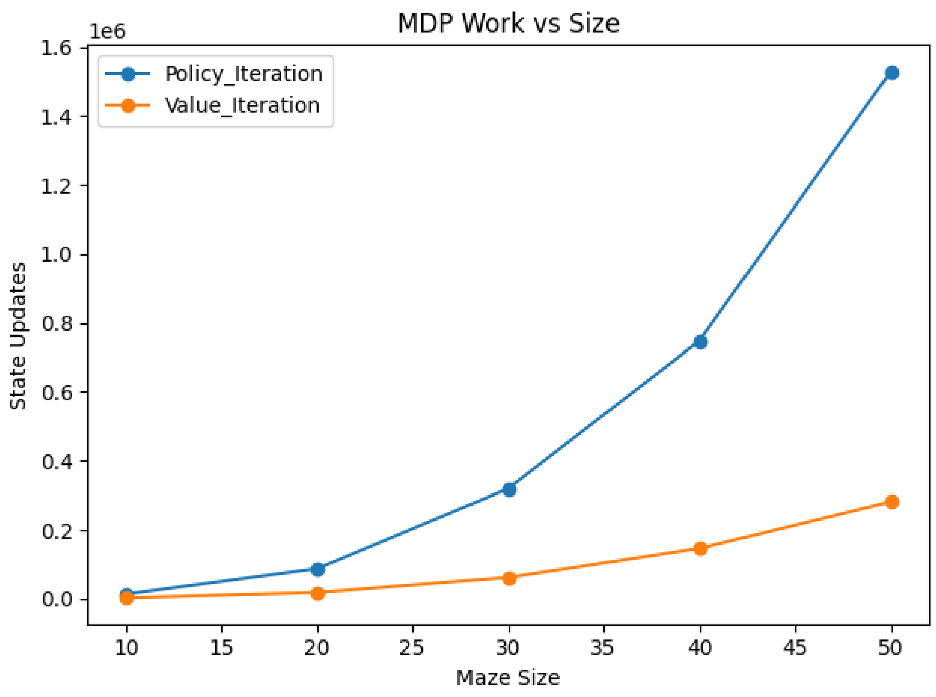

Figure 8: Total state updates performed by MDP algorithms.

Runtime vs Openness: As openness increases, runtime decreases slightly for both methods (Figure 9). Higher connectivity reduces structural bottlenecks and allows value information to propagate more efficiently across states. Policy Iteration remains slower overall due to the computational cost of policy evaluation.

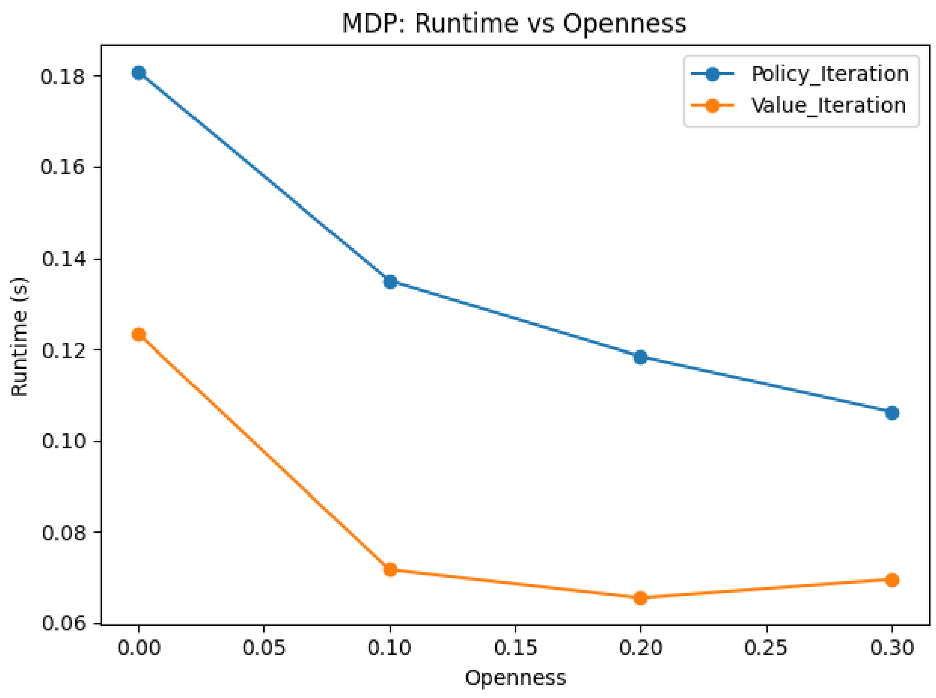

Figure 9: MDP runtime as openness increases.

Gamma sensitivity: As γ approaches 1, convergence slows (Figure 10). Higher discount factors reduce the contraction strength in Bellman updates, increasing the number of iterations required for value stabilisation. Policy Iteration is particularly sensitive to large γ due to repeated evaluation cycles.

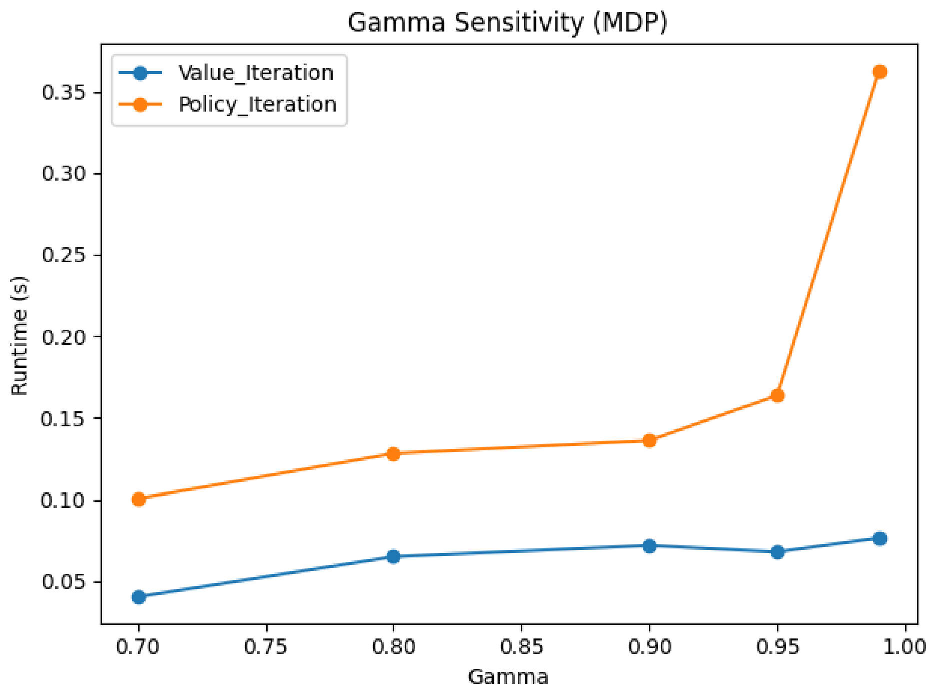

Figure 10: Effect of discount factor γ on runtime.

4.3 All Algorithms Comparison

Execution time: The log-scale comparison (Figure 11) highlights the structural difference between search and MDP approaches. Search algorithms operate on a subset of states and scale with explored depth and branching factor. MDP methods compute values for every state and therefore scale with total grid size. Policy Iteration exhibits the highest runtime, followed by Value Iteration, while all search algorithms remain substantially faster in this deterministic shortest-path formulation.

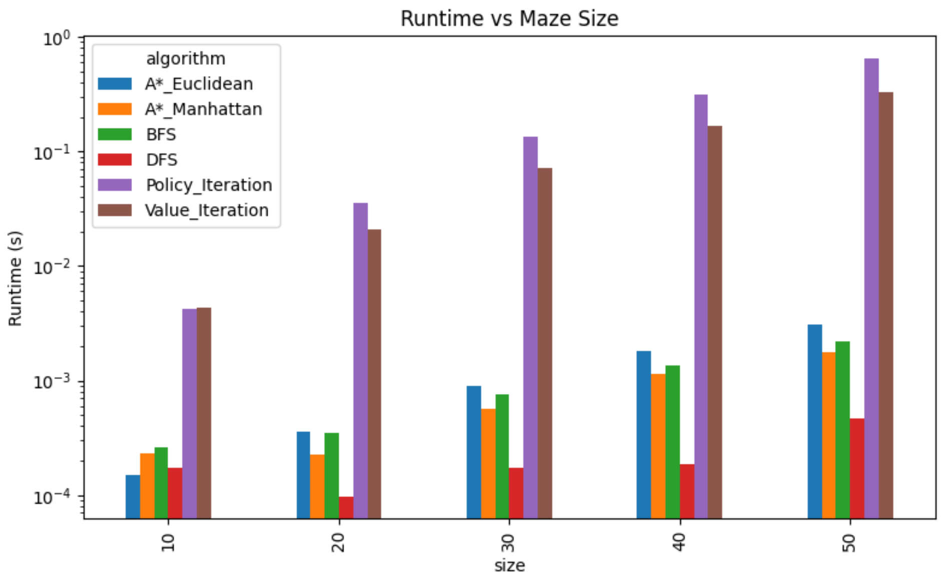

Figure 11: Log-scale runtime comparison across all algorithms.

5 Discussion

The results highlight a fundamental structural distinction between search-based planning and MDP-based planning. Search algorithms compute a single solution path by expanding only necessary states, whereas MDP methods compute value estimates for the entire state space.

This difference explains the large runtime gap observed in Figure 11. Search algorithms scale primarily with solution depth and local branching factor, while MDP methods scale with total state count. As maze size increases, the number of states grows quadratically, which significantly impacts dynamic programming approaches.

Among search algorithms, heuristic strength plays a critical role. A* with Manhattan distance consistently expands fewer nodes than the Euclidean variant, confirming that heuristic accuracy directly affects search efficiency. BFS guarantees optimal solutions but at the cost of greater exploration and memory usage. DFS is computationally light but may produce suboptimal paths.

For MDP methods, Policy Iteration requires fewer outer iterations but performs expensive policy evaluation steps, resulting in higher total runtime. Value Iteration updates values directly and performs fewer total state updates. Sensitivity to the discount factor further illustrates the theoretical contraction properties of Bellman updates: as γ approaches 1, convergence slows.

The openness parameter provides additional insight. Increased openness introduces alternative routes, which affects search branching but can reduce structural bottlenecks for value propagation. This demonstrates how environmental structure influences algorithmic efficiency differently across paradigms.

Overall, for deterministic shortest-path problems, heuristic search offers superior practical performance compared to full-state dynamic programming. However, MDP methods provide complete policies and extend naturally to stochastic environments, where search-based approaches may require modification.