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

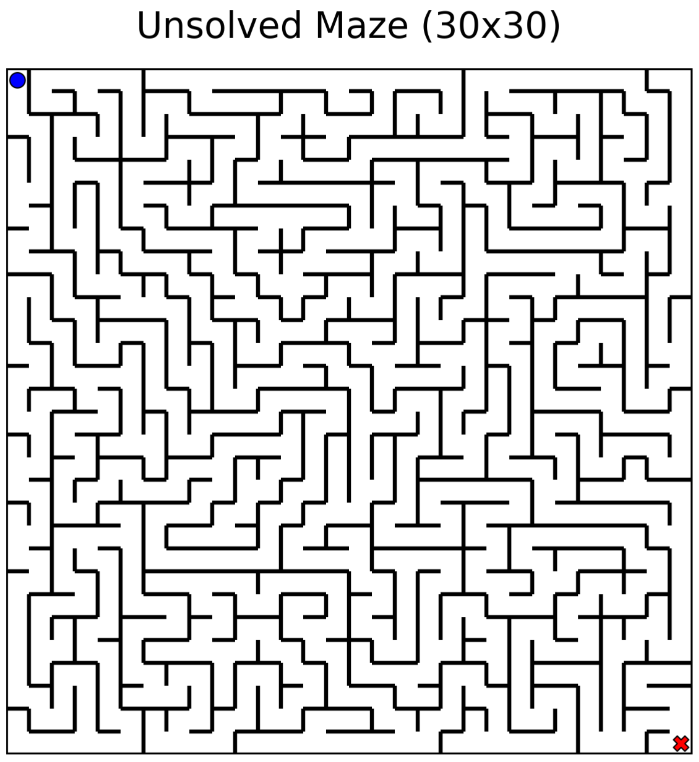

Figure 1: Generated 30×30 maze before solving.

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 approach. 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.

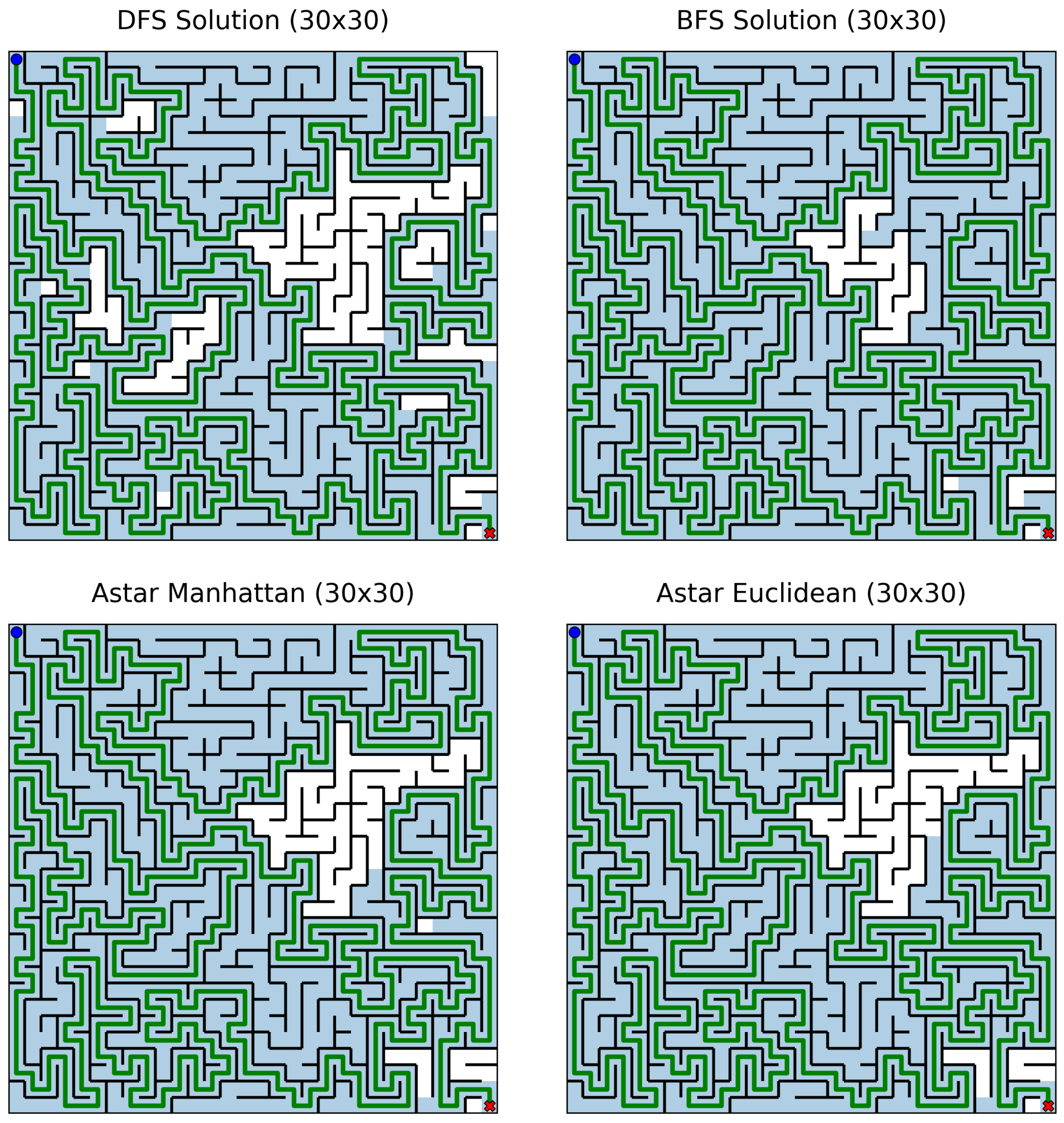

Figure 2: Example solutions on the same 30×30 maze, showing explored states (shaded) and the final path (green).

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 experiments) for each move and a positive reward for reaching the goal state (100 for experiments).
- 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:

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

Since the maze transitions are deterministic, each state–action pair has a single successor, so the summation reduces to a single update in implementation.

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 selecting, for each state, the action that maximizes the expected value.

Value Iteration is guaranteed to converge for $0 < \gamma < 1$.

2.3.2 Policy Iteration

Policy Iteration alternates between policy evaluation and policy improvement:

1. **Policy Evaluation:**

$$V^\pi(s) = \sum_{s'} P(s' \mid s, \pi(s)) \left[R(s, \pi(s), s') + \gamma V^\pi(s') \right]$$

   Under deterministic transitions, this again reduces to a single successor update.

2. **Policy Improvement:**

$$\pi(s) = \arg\max_{a \in A} \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma V^\pi(s') \right]$$

These steps are repeated until the policy stabilizes. Although Policy Iteration typically 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.

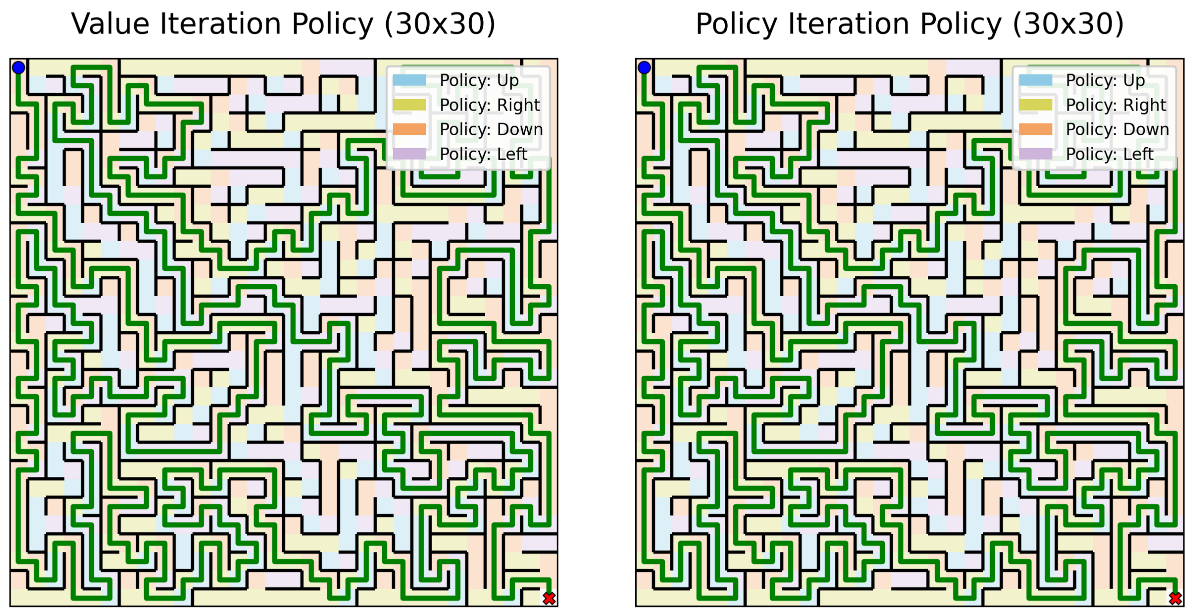
Value Iteration Policy (30x30)
Policy Iteration Policy (30x30)
Policy: Up
Policy: Right
Policy: Down
Policy: Left

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.

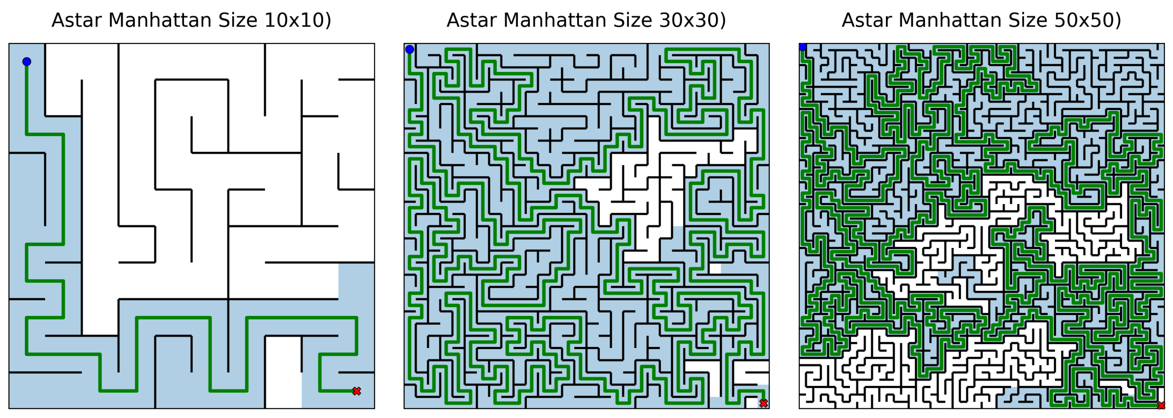
Astar Manhattan Size 10x10)
Astar Manhattan Size 30x30)
Astar Manhattan Size 50x50)

Figure 4: Effect of increasing maze size on solution structure using A* (Manhattan heuristic).

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.

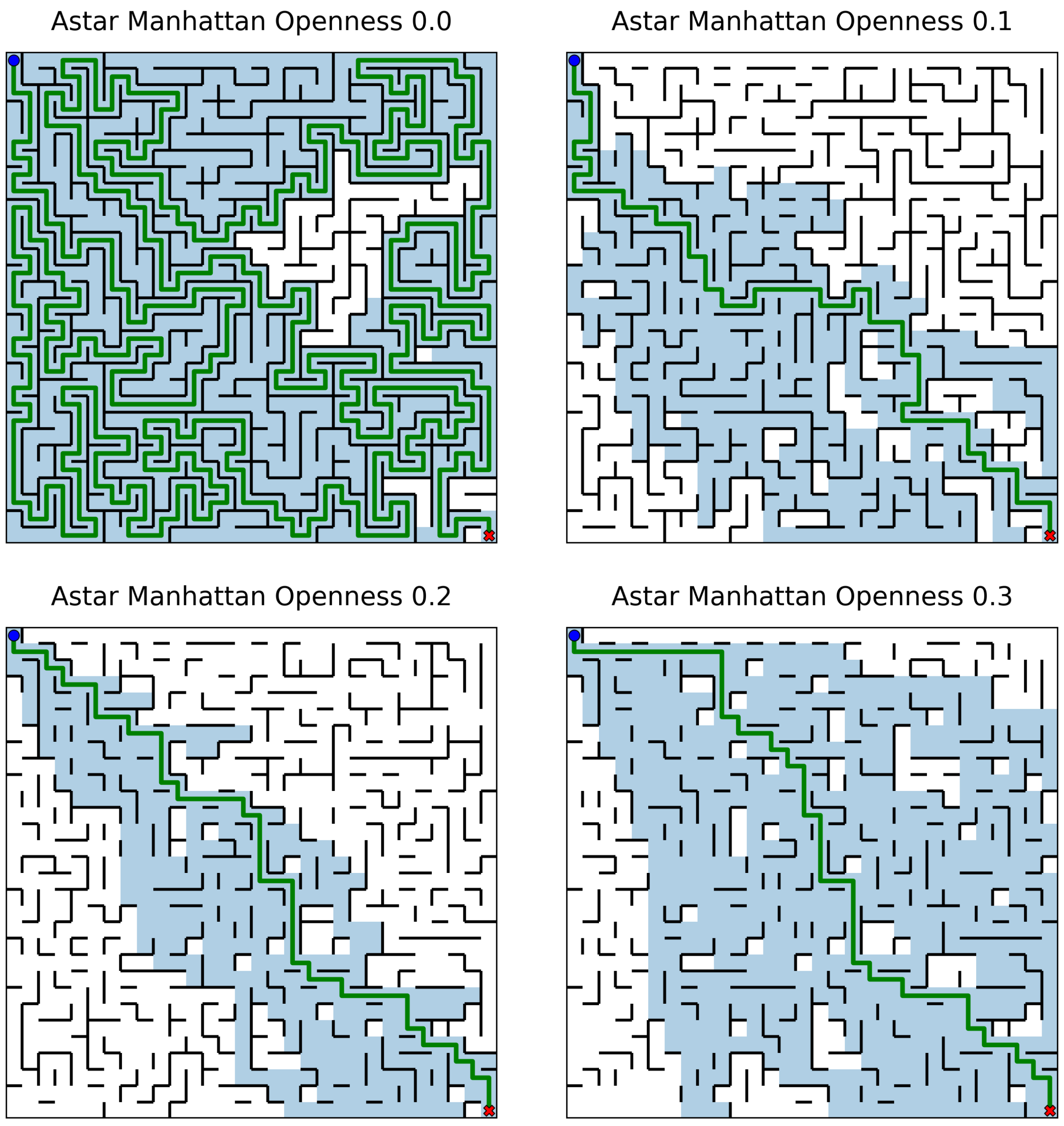
Astar Manhattan Openness 0.0
Astar Manhattan Openness 0.1
Astar Manhattan Openness 0.2
Astar Manhattan Openness 0.3

Figure 5: Effect of increasing openness (0.0–0.3) on maze connectivity using A* (Manhattan heuristic). Higher openness introduces loops and alternative routes, increasing branching and reducing structural constraint.

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 Theoretical Complexity Analysis

4.1 Search Algorithms

Let b denote the branching factor and d the depth of the optimal solution.

Depth-First Search (DFS) DFS explores one branch of the search tree as deeply as possible before backtracking.

- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$

DFS is not optimal in general, as it may return a non-shortest path.

Breadth-First Search (BFS) BFS expands nodes level by level.

- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$

When all step costs are equal, BFS is complete and optimal.

A* Search A* selects nodes according to

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

where $g(n)$ is the path cost from the start node and $h(n)$ is a heuristic estimate of the remaining cost.

If $h(n)$ is admissible, A* tree search is optimal; if $h(n)$ is consistent, A* graph search is optimal. Worst-case time and space complexity remain exponential in d, but stronger heuristics reduce the effective branching factor and significantly decrease node expansions in practice.

4.2 MDP Algorithms

Let $|S|$ denote the number of states and $|A|$ the number of actions.

Value Iteration Value Iteration applies the Bellman optimality update:

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

Each iteration evaluates all actions for all states and sums over successor states, giving a per-iteration complexity of

$$O(|S|^2|A|).$$

In the deterministic grid used here, each state–action pair has a single successor, so practical behaviour is closer to $O(|S||A|)$ per iteration. However, values must still be computed for the entire state space.

Policy Iteration Policy Iteration alternates between policy evaluation and policy improvement. Policy evaluation updates values for all states under a fixed policy, while policy improvement maximizes over actions for each state.

Although Policy Iteration often converges in fewer outer iterations than Value Iteration, each iteration is computationally heavier due to the evaluation phase. Overall complexity remains polynomial in $|S|$ and $|A|$, but scales with the full state space.

Since $|S|$ grows quadratically with maze size ($|S| = n^2$ for an $n \times n$ grid), MDP methods scale less favourably than single-path search methods in deterministic environments.

5 Results

5.1 Search Algorithm Comparison

Nodes explored: As maze size increases, the number of expanded nodes grows rapidly (Figure 6). 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.

Table 1: Search performance summary across maze sizes (mean values).

Size	Algorithm	Runtime (s)	Nodes	Path	Memory
10	DFS	0.00017	32	22.7	10
10	BFS	0.00026	88	19.3	10
10	A* Manhattan	0.00023	51	19.3	11
10	A* Euclidean	0.00015	64	19.3	11
30	DFS	0.00018	180	145.3	45
30	BFS	0.00076	882	64.7	31
30	A* Manhattan	0.00056	386	64.7	61
30	A* Euclidean	0.00089	586	64.7	48
50	DFS	0.00046	476	317.3	107
50	BFS	0.00221	2493	110.7	54
50	A* Manhattan	0.00176	1192	110.7	132
50	A* Euclidean	0.00308	1996	110.7	93

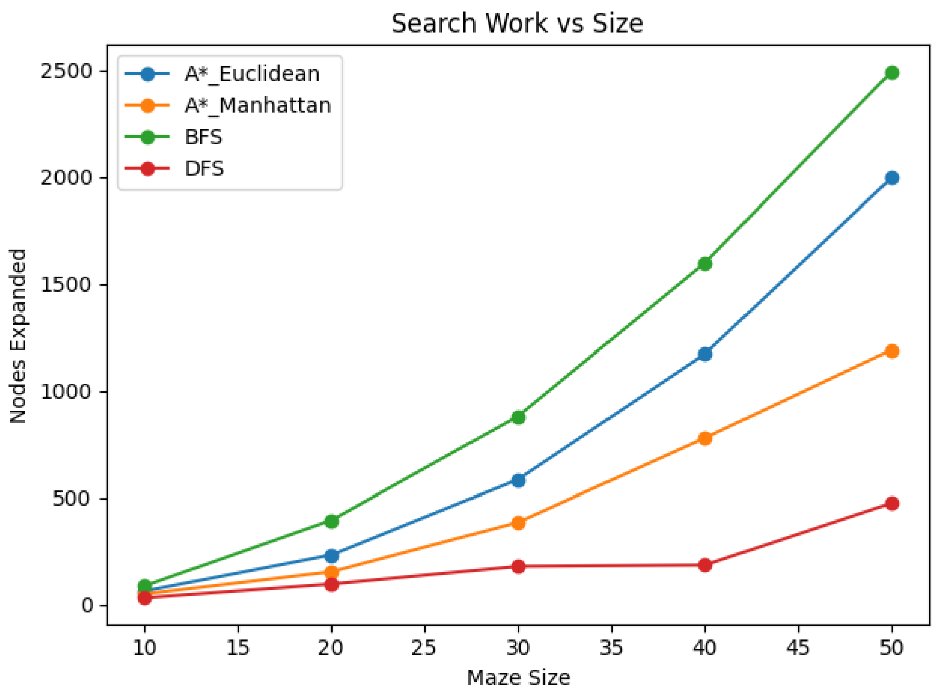

Figure 6: Nodes expanded across maze sizes.

Memory usage: Peak size reflects memory usage (Figure 7). 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.

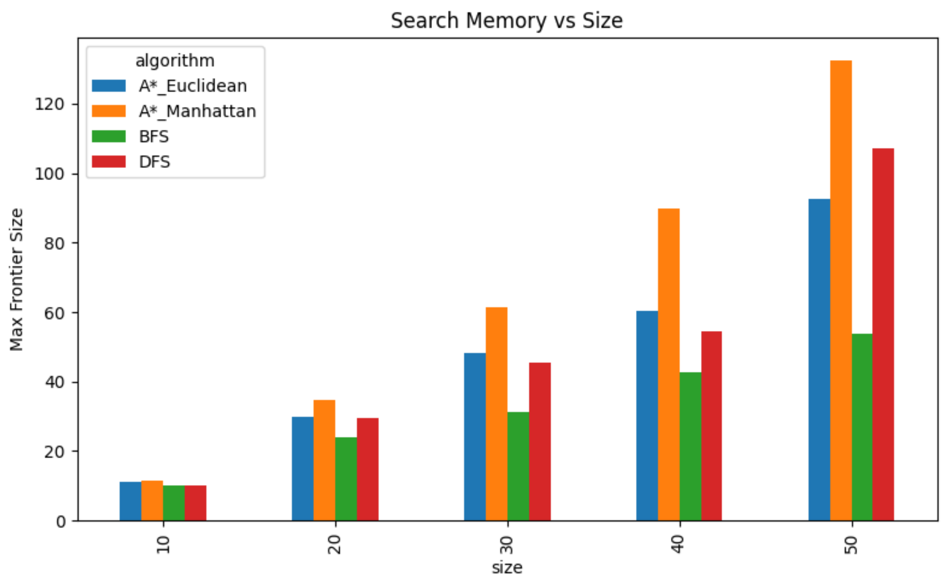

Figure 7: Maximum Memory usage across maze sizes.

Runtime vs openness. Increasing openness introduces additional branching and alternative routes (Figure 8). 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.

Table 2: Effect of openness on search performance (30×30 maze).

Open	Algorithm	Runtime (s)	Nodes	Memory
0.0	A* Manhattan	0.00057	451	6.7
0.1	A* Manhattan	0.00056	386	61
0.2	A* Manhattan	0.00058	389	75
0.3	A* Manhattan	0.00096	627	88

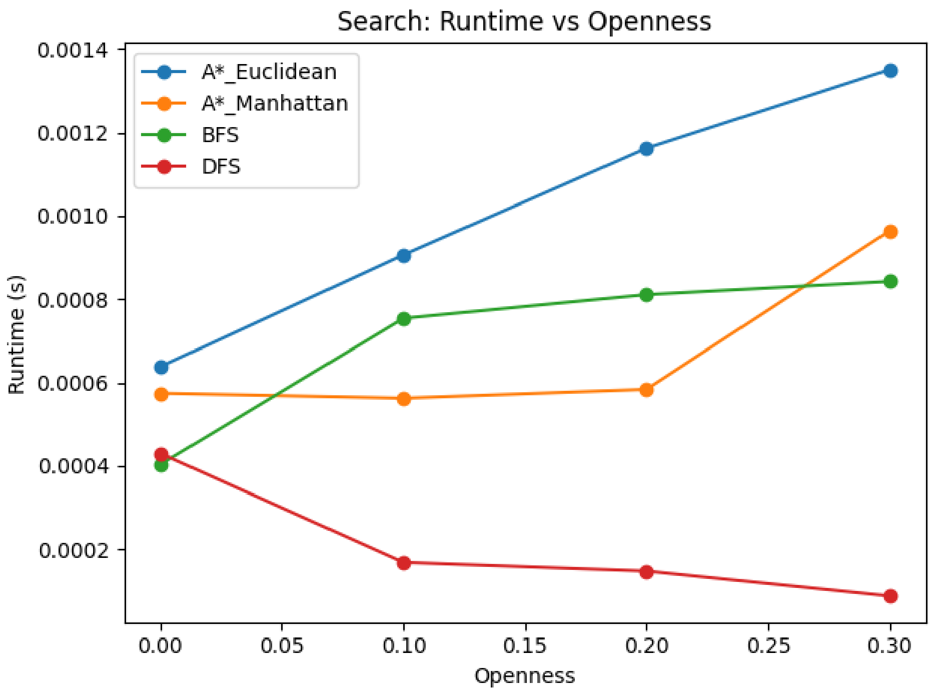

Figure 8: Search runtime as openness increases.

Runtime vs work: Runtime scales approximately linearly with the number of nodes expanded (Figure 9). This confirms that node expansion dominates computational cost in search-based planning. Algorithms that reduce expansions through heuristic guidance achieve proportional runtime reductions.

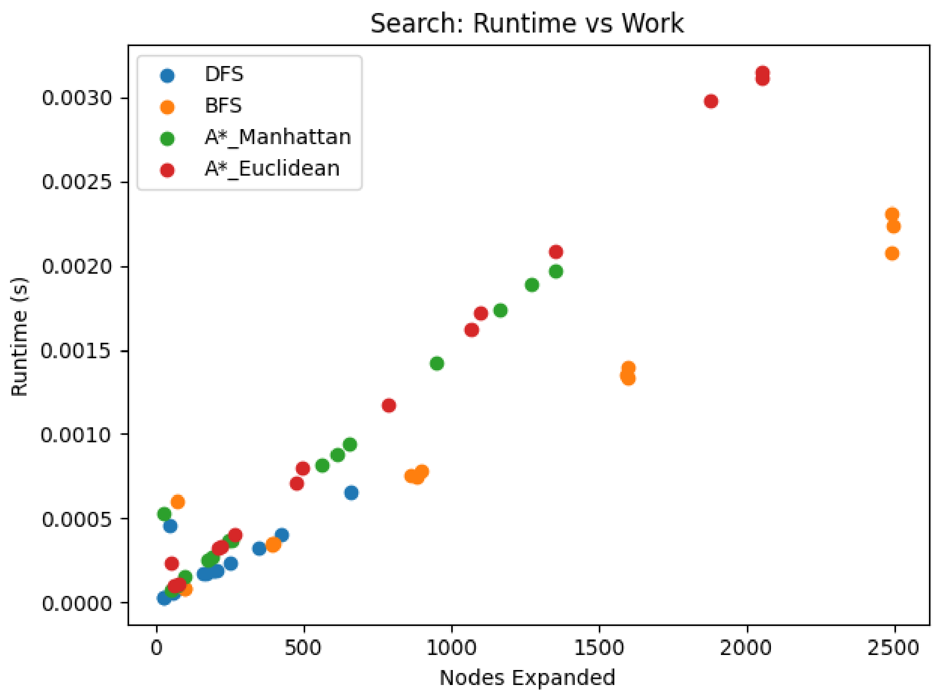

Figure 9: Relationship between nodes expanded and runtime for search algorithms.

5.2 MDP Algorithm Comparison

Work vs Size: Both MDP methods perform updates across the entire state space (Figure 10). 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.

Table 3: MDP convergence statistics across maze sizes.

Size	Algorithm	Runtime (s)	Updates	Iterations
30	Value Iteration	0.0717	61,731	68.7
30	Policy Iteration	0.1328	319,444	43
50	Value Iteration	0.3319	281,554	112.7
50	Policy Iteration	0.6548	1,529,388	73

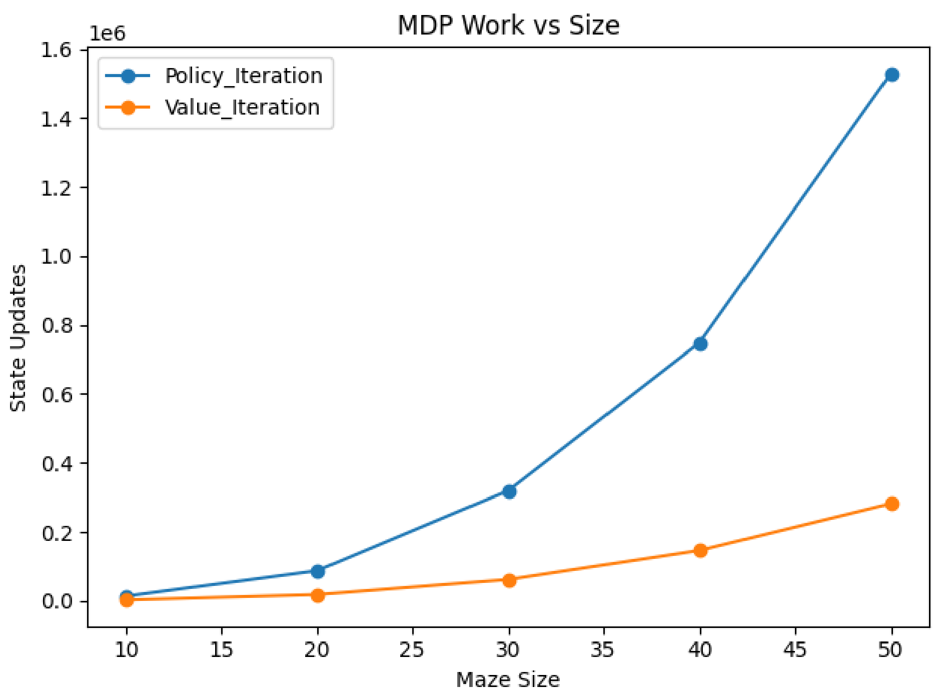

Figure 10: Total state updates performed by MDP algorithms.

Runtime vs Openness: As openness increases, runtime decreases slightly for both methods (Figure 11). 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.

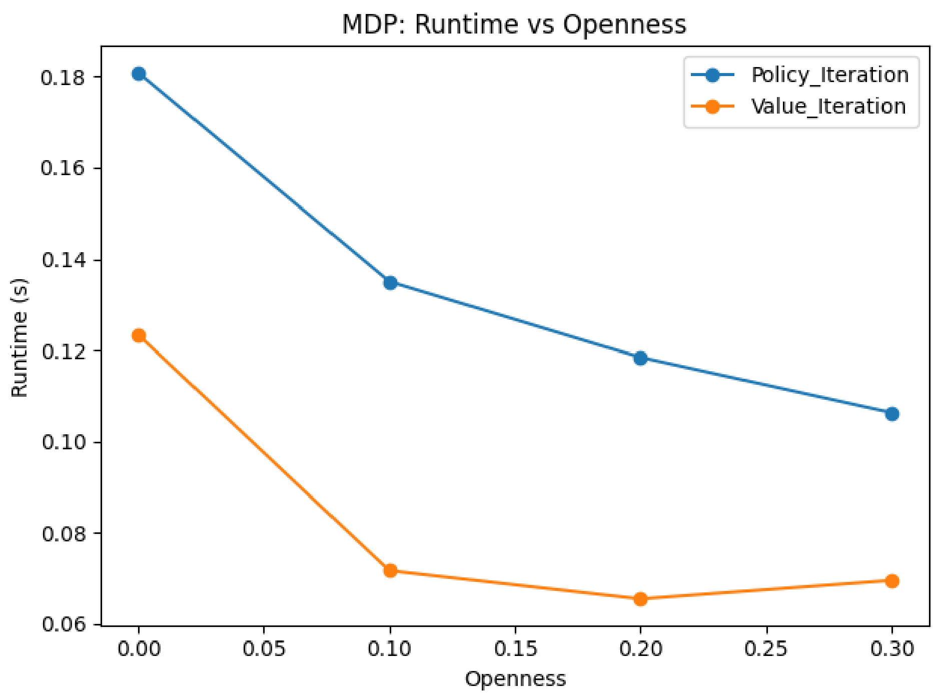

Figure 11: MDP runtime as openness increases.

Gamma sensitivity: As γ approaches 1, convergence slows (Figure 12). 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.

Table 4: Effect of discount factor γ on convergence.

γ	Algorithm	Runtime (s)	Iterations
0.70	Value Iteration	0.0403	40
0.80	Value Iteration	0.0650	63
0.90	Value Iteration	0.0719	69
0.99	Policy Iteration	0.3627	45

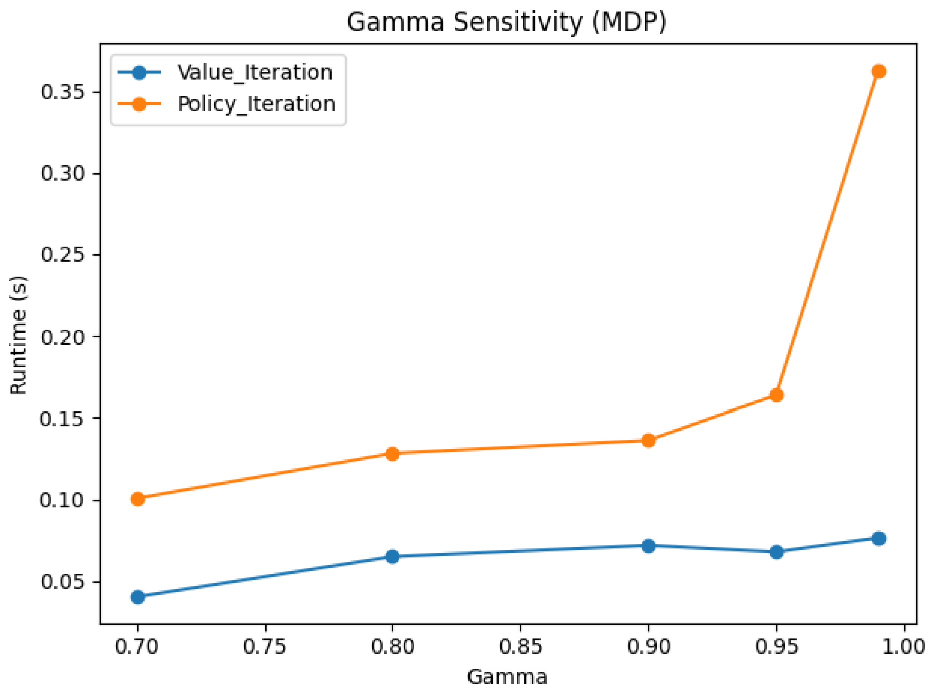

Figure 12: Effect of discount factor γ on runtime.

5.3 All Algorithms Comparison

Execution time: The log-scale comparison (Figure 13) 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.

Table 5: Runtime summary for maze size 50×50 (mean, standard deviation, coefficient of variation).

Algorithm	Mean (s)	Std (s)	CV
DFS	0.000463	0.000171	0.369
A* Manhattan	0.001761	0.000292	0.166
BFS	0.002209	0.000116	0.052
A* Euclidean	0.003081	0.000092	0.030
Value Iteration	0.331898	0.010033	0.030
Policy Iteration	0.654766	0.045890	0.070

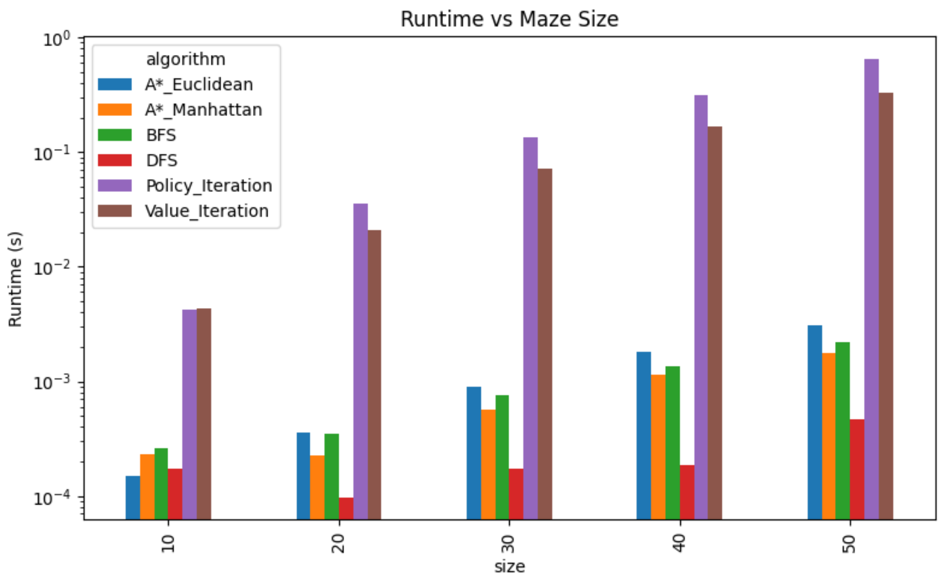

Figure 13: Log-scale runtime comparison across all algorithms.

6 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 13. 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.

7 Conclusion

This project implemented and experimentally evaluated five algorithms for maze solving: DFS, BFS, A* with two heuristics, Value Iteration, and Policy Iteration. The goal was not only to measure performance differences, but to examine how fundamentally different planning paradigms behave under controlled structural variations.

The experiments demonstrate that heuristic search is highly effective in deterministic shortest-path environments. A* with the Manhattan heuristic consistently achieved optimal solutions with significantly reduced computational effort compared to uninformed search. BFS maintained optimality but required substantially more memory, while DFS prioritised speed at the expense of path quality.

MDP-based approaches exhibited markedly different scaling behaviour. Because they compute values for all states, their computational cost increases with total grid size rather than explored depth. While Value Iteration and Policy Iteration both converge to the same optimal policy, their convergence characteristics and computational overhead differ in practice.

The primary contribution of this work lies in the systematic comparison under controlled maze size and openness parameters. By isolating structural factors, the study clearly illustrates when local search methods are sufficient and when global policy computation becomes unnecessarily expensive.

In deterministic grid navigation tasks, heuristic search provides the most efficient solution. However, the broader significance of MDP methods lies in their generality. When uncertainty, stochastic transitions, or long-term reward trade-offs are introduced, value-based methods provide a principled framework that extends beyond simple path finding.

Future work could extend this comparison to stochastic mazes, incorporate probabilistic transitions, or evaluate reinforcement learning approaches that operate without explicit transition models.

A Algorithm Implementations

A.1 Depth-First Search (DFS)

```
def dfs_solver(maze):

    start = maze.start
    goal = maze.goal

    stack = [start]
    visited = {start}
    parent = {}

    while stack:
        current = stack.pop()

        if current == goal:
            break

        for nbr in maze.neighbors(current):
            if nbr not in visited:
                visited.add(nbr)
                parent[nbr] = current
                stack.append(nbr)

    # Reconstruct path
    path = []
    if goal in visited:
        cur = goal
        while cur != start:
            path.append(cur)
            cur = parent[cur]
        path.append(start)
        path.reverse()

    return path
```

A.2 Breadth-First Search (BFS)

```
def bfs_solver(maze):

    start = maze.start
    goal = maze.goal

    queue = deque([start])
    visited = {start}
    parent = {}

    while queue:
        current = queue.popleft()

        if current == goal:
            break

        for nbr in maze.neighbors(current):
            if nbr not in visited:
                visited.add(nbr)
                parent[nbr] = current
                queue.append(nbr)

    # Reconstruct path
    path = []
    if goal in visited:
        cur = goal
        while cur != start:
            path.append(cur)
            cur = parent[cur]
        path.append(start)
        path.reverse()

    return path
```

A.3 A* Search

```
def manhattan(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def euclidean(a, b):
    return ((a[0] - b[0])**2 + (a[1] - b[1])**2) ** 0.5

def astar_solver(maze, heuristic):

    start = maze.start
    goal = maze.goal

    open_set = []
    g = {start: 0}
    parent = {}
    closed_set = set()

    heapq.heappush(open_set, (heuristic(start, goal), start))

    while open_set:
        _, current = heapq.heappop(open_set)

        if current in closed_set:
            continue
        closed_set.add(current)

        if current == goal:
            break

        for nbr in maze.neighbors(current):
            tentative_g = g[current] + 1

            if nbr not in g or tentative_g < g[nbr]:
                g[nbr] = tentative_g
                parent[nbr] = current
                f = tentative_g + heuristic(nbr, goal)
                heapq.heappush(open_set, (f, nbr))

    # Reconstruct path
    path = []
    if goal in parent or goal == start:
        cur = goal
        while cur != start:
            path.append(cur)
            cur = parent[cur]
        path.append(start)
        path.reverse()

    return path
```

A.4 Value Iteration

```python
def value_iteration(maze, gamma=0.9, goal_reward=100, step_cost=-1):

    # Initialise value function
    V = {state: 0 for state in maze.all_cells()}

    # Iterate until convergence
    while True:
        delta = 0
        new_V = V.copy()

        for state in maze.all_cells():

            if state == maze.goal:
                continue

            neighbors = maze.neighbors(state)
            if not neighbors:
                continue

            best_value = float("-inf")

            for next_state in neighbors:
                reward = goal_reward if next_state == maze.goal else step_cost
                value = reward + gamma * V[next_state]
                best_value = max(best_value, value)

            new_V[state] = best_value
            delta = max(delta, abs(new_V[state] - V[state]))

        V = new_V

        if delta < 1e-4:
            break

    # Extract policy
    policy = {}

    for state in maze.all_cells():

        if state == maze.goal:
            policy[state] = None
            continue

        neighbors = maze.neighbors(state)
        best_action = None
        best_value = float("-inf")

        for next_state in neighbors:
            reward = goal_reward if next_state == maze.goal else step_cost
            value = reward + gamma * V[next_state]

            if value > best_value:
                best_value = value
                best_action = next_state

        policy[state] = best_action

    return policy
```

A.5 Policy Iteration

```python
def policy_iteration(maze, gamma=0.9, goal_reward=100, step_cost=-1):

    # Initialise policy
    policy = {}
    gr, gc = maze.goal

    for state in maze.all_cells():

        if state == maze.goal:
            policy[state] = None
            continue

        neighbors = maze.neighbors(state)
        if not neighbors:
            policy[state] = None
            continue

        # Initialise toward goal (heuristic initial policy)
        policy[state] = min(
            neighbors,
            key=lambda n: abs(n[0] - gr) + abs(n[1] - gc)
        )

    # Initialise value function
    V = {state: 0 for state in maze.all_cells()}

    policy_stable = False

    while not policy_stable:

        # Policy Evaluation
        while True:
            delta = 0
            new_V = V.copy()

            for state in maze.all_cells():

                if state == maze.goal:
                    continue

                action = policy[state]
                if action is None:
                    continue

                reward = goal_reward if action == maze.goal else step_cost
                value = reward + gamma * V[action]

                new_V[state] = value
                delta = max(delta, abs(new_V[state] - V[state]))

            V = new_V

            if delta < 1e-4:
                break

        # Policy Improvement
        policy_stable = True

        for state in maze.all_cells():

            if state == maze.goal:
                continue

            old_action = policy[state]
            neighbors = maze.neighbors(state)

            best_action = None
            best_value = float("-inf")

            for next_state in neighbors:
                reward = goal_reward if next_state == maze.goal else step_cost
                value = reward + gamma * V[next_state]

                if value > best_value:
                    best_value = value
                    best_action = next_state

            policy[state] = best_action

            if best_action != old_action:
                policy_stable = False

    return policy
```