CS7IS2 - Artificial Intelligence Assignment 1 Report

Kaaviya Paranji Ramkumar

March 2025

1 Introduction

This assignment presents the implementation and comparative analysis of various search and Markov Decision Process (MDP) algorithms for maze-solving problems. A maze generator capable of creating environments of different dimensions was developed to serve as the testing environment for the algorithm implementations. The search algorithms implemented include Depth-First Search (DFS), Breadth-First Search (BFS), and A* to find paths through the mazes. Additionally, two MDP-based approaches - Value Iteration and Policy Iteration - were implemented to solve the same maze problems.

The core of this work involved the careful design of each algorithm to efficiently navigate through maze environments, followed by extensive testing across mazes of varying complexity. Performance data was collected using metrics such as execution time, memory usage, and solution quality to provide a comprehensive comparison between these different approaches. The analysis explores the strengths and limitations of each algorithm, examining not only how search algorithms compare to each other, but also how MDP-based approaches differ from traditional search methods in maze-solving tasks.

2 Methodology

2.1 Maze Generation

Description of the maze generator

The maze generation implementation utilizes a Recursive Backtracker algorithm with enforced boundaries. This approach generates perfect mazes that contain exactly one path between any two points, with no loops or inaccessible areas. The implementation creates a grid where cells are either walls ('#') or paths ('.'), with special markers for start ('S') and goal ('G') positions. Below are listed the key steps involved in implementing the maze generator:

- Initialize maze with custom size with all cells as walls.
- Create a NumPy grid with odd dimensions to ensure proper wall placement.
- Implement depth-first search with backtracking using a stack for finding path.
- Perform random selection of unvisited neighboring cells during path creation.
- Optionally, add multiple paths by randomly removing walls based on a configurable path density parameter.
- Enforce boundary walls that cannot be removed, so that the maze remains contained.

Maze properties

The implemented maze generator produces mazes with the following properties:

- Fully connected structure with enforced boundary walls.
- Default start position at (1,1) and goal position at the bottom-right corner (height-2, width-2).

- Configurable dimensions that are always adjusted to odd numbers to maintain proper wall-path structure.
- Optional path density parameter that can create shortcuts by removing interior walls. This in turn created multiple paths from start to goal position.

For testing purposes, a range of odd-sized mazes was generated:

- 7×7 (very small)
- 11×11, 15×15, 19×19, 21×21, 25×25 (small to medium)
- 31×31, 45×45, 59×59, 67×67 (medium to large)
- 83×83, 101×101 (large)

The maze complexity increases with size, as larger dimensions create longer potential solution paths and a larger state space for algorithms to explore.

Justification for chosen maze sizes

The selection of maze sizes for performance comparison was based on several key considerations:

- **Consistent structure**: All sizes are odd numbers, which aligns with the recursive backtracker algorithm's requirement for proper wall placement between path cells.
- **Graduated complexity range**: The sizes range from very small (7×7) to quite large (101×101), providing a comprehensive spectrum to observe how algorithm performance scales with increasing problem complexity.
- **Finer granularity at critical transition points**: More sizes were selected in the small-to-medium range (11×11 through 25×25) to capture the performance characteristics to see where algorithms begin to show meaningful differences in behavior.
- **Performance constraints consideration**: The largest sizes (83×83 and 101×101) were included to test the limits of the algorithms and the maze generator itself, revealing important distinctions in time and memory efficiency under high computational demands.
- **Statistical significance**: Multiple maze instances were generated at each size to ensure results were not biased by any particular maze configuration, providing more reliable performance metrics.

This carefully selected range of maze sizes enables a thorough analysis of how different search and MDP algorithms respond to increasing problem complexity in terms of execution time, memory usage, and solution quality.

2.2 Algorithm Implementations

2.2.1 Search Algorithms

Depth-First Search (DFS) and Breadth-First Search (BFS)

Both the DFS and BFS implementations share similar design patterns, with key differences in their traversal strategies. These classic graph search algorithms were implemented with careful consideration of several important factors.

- **State representation**: For the representation of states, coordinates (x, y) within the maze grid were chosen as the primary way to track positions. This approach gives an intuitive mapping between the algorithm's internal state and the physical maze structure. Using simple coordinate pairs makes it easy to calculate adjacent positions and requires minimal memory overhead, which becomes especially important when dealing with larger maze sizes.

- **Action space**: The action space was deliberately limited to the four cardinal movements: up, down, left, and right. This design prevents diagonal movements that might allow "cutting corners" through walls while ensuring complete coverage of all possible paths through the maze. Each movement has a uniform cost of 1 unit, simplifying path cost calculations and maintaining consistency across different maze configurations.

- **Goal recognition**: Goal recognition is handled through explicit marking with the 'G' character in the maze array. This approach makes it straightforward to detect when the search has reached the target position, supporting flexible goal placement anywhere within the maze structure. Such clear termination condition helps prevent unnecessary exploration once the goal has been located.

- **Fringe management**: The key difference between DFS and BFS lies in their fringe management strategies. DFS employs a stack data structure that encourages exploring as deeply as possible along each branch before backtracking. This can be advantageous in mazes with long corridors and few branching paths. In contrast, BFS utilizes a queue that ensures all states at a given depth are fully explored before moving deeper, guaranteeing the shortest path will be found when a solution exists.

- **Track visited states:** To prevent cycles and repeated exploration, both algorithms maintain a boolean matrix matching the maze dimensions to track visited states. This approach offers constant-time lookup performance and consumes less memory than alternative approaches like hash sets, especially for larger mazes where memory efficiency becomes critical.

- **Parent pointer map**: For path reconstruction, a parent pointer map is maintained throughout the search process. This dictionary maps each visited state to the state that led to its discovery. When the goal is reached, the solution path can be efficiently reconstructed by working backward through these parent pointers until reaching the starting position. This eliminates the need to store complete paths during the search phase, significantly reducing memory requirements.

A* Search

The A* search algorithm builds upon the foundation of BFS but incorporates a heuristic function to guide the search toward the goal more efficiently. This approach combines the completeness of BFS with additional intelligence about the problem structure.

- **Heuristic function**: The Manhattan distance was selected as the heuristic function for our A* implementation after careful consideration of alternatives. This heuristic is particularly well-suited for grid-based mazes with cardinal movements because it exactly represents the minimum number of steps required to reach the goal in the absence of walls. This property ensures the heuristic never overestimates the true cost, making it admissible and guaranteeing optimal solutions. Additionally, Manhattan distance satisfies the consistency requirement for A*, meaning the estimated cost to the goal never increases by more than the actual cost of moving between states. From a computational standpoint, the calculation involves simple arithmetic operations, adding minimal overhead to each state evaluation.

- **Priority queue implementation**: To manage the exploration frontier efficiently, a priority queue was implemented using a three-part priority tuple: (f_score, counter, position). The f_score combines the known cost from the start (g_score) with the estimated cost to the goal (heuristic), directing the search toward promising paths. The counter serves as a tiebreaker for states with equal f_scores, ensuring consistent behavior and preventing potential oscillations between equally valued states. This approach avoids the need for costly priority queue updates when better paths to already-discovered states are found.

- **Path cost tracking**: Path costs are tracked using a dictionary that maps each explored state to its current best-known distance from the start. When the algorithm discovers a potentially better path to a state, it compares the new path cost with the previously recorded value and updates accordingly. This mechanism ensures that the search always maintains optimal path information throughout the exploration process, even when encountering the same state through different routes.

The implementation also includes comprehensive performance metrics collection to facilitate meaningful algorithm comparisons. These metrics include the number of nodes explored (providing insight into search efficiency), maximum fringe size (indicating peak memory usage), path length (measuring solution quality), and execution time (quantifying computational performance). These measurements allow for data-driven evaluation of how A* performs relative to the uninformed search algorithms across different maze configurations and sizes.

2.2.2 Markov Decision Process (MDP) Algorithms

Value Iteration and Policy Iteration

The Markov Decision Process (MDP) offers a different approach to maze solving compared to traditional search algorithms. Instead of explicitly searching for a path, MDP algorithms compute an optimal policy that specifies the best action to take at each state. Two classic MDP algorithms were implemented: Value Iteration and Policy Iteration.

For the MDP formulation of the maze-solving problem, these are the common key design decisions were made:

- **State representation**: States are represented as coordinates within the maze, similar to the search algorithms. This representation naturally maps to the grid structure while keeping the state space manageable and allowing for direct performance comparisons.
- **Action space**: The action space consists of the four cardinal directions (up, down, left, right), maintaining consistency with the search implementations and realistically modeling movement constraints in a grid-based maze.
- **Rewards**: The reward structure was carefully designed to guide the agent toward the goal while discouraging excessive movements. A large positive reward (+100) is assigned to reaching the goal state, creating a strong incentive to find it. Small negative rewards are assigned to all intermediate states, effectively creating a path cost that encourages finding shorter paths. Additionally, attempting to move into a wall results in remaining in the same state and still incurring the negative reward (-100), discouraging such attempts.
- **Transition model**: The transition model is deterministic in our implementation, meaning that taking an action from a state always leads to the same next state with probability 1.0. This design choice simplifies the MDP while still capturing the essential dynamics of maze navigation and creates a fair comparison with the deterministic search algorithms.
- **Discount factor**: A discount factor γ (gamma) between 0.9 and 0.99 was selected after experimentation with different values. This parameter balances the importance of immediate versus future rewards. Setting gamma close to 1 encourages the agent to consider long-term rewards more heavily, which is appropriate for maze-solving where the goal might be many steps away.

For the Value Iteration algorithm, individual key implementation details include:

- An adaptive convergence threshold that ensures the algorithm converges to an approximately optimal policy while avoiding unnecessary computation.
- Iterative computation of state values until the maximum change between iterations falls below the threshold.
- A threshold of 0.001 that provides a good balance between solution quality and computational efficiency

As for the Policy Iteration, algorithm alternates between policy evaluation and policy improvement steps. For policy evaluation, a system of linear equations is solved to determine the values of states under the current policy. Policy improvement then updates the policy based on these values. This two-step process often converges in fewer iterations than Value Iteration, especially for larger mazes, though each iteration typically requires more computation.

Both MDP algorithms were implemented with a custom maximum iteration limit to ensure termination even for very large or complex mazes. Based on experimentation with different maze sizes, a limit of 1000 iterations was found to be more than sufficient for convergence in tested mazes.

To extract a solution path once the optimal policy has been computed, a path extraction method follows the optimal policy from the initial state until reaching the goal state. This allows direct comparison of path lengths with the search algorithms and provides a concrete solution rather than just a policy.

Performance metrics calculated for both MDP algorithms include:

- Convergence iterations
- Execution time
- Solution path length
- Change in state values between iterations

2.3 Experimental Setup

Code organization and object-oriented design

The experimental framework for this assignment was carefully planned and implemented using object-oriented programming principles to ensure modularity, re-usability, and clean separation of concerns. The code structure follows software engineering best practices, making it easy to extend and maintain.

The code base for maze generation and solving is currently maintained at *https://github.com/prkaaviya/TCD-Artificial-Intelligence-A1*

The project is organized in a hierarchical structure with clearly defined components like:

```
.
|-- Makefile              # For automation
|-- gen_maze.py           # Maze generation module
|-- run.py                # Main execution script to solve maze
|-- mazes/                # Generated maze data
|   |-- text/             # Text representation of mazes
|   |-- visuals/          # Visual representation of mazes
|-- notebooks/            # Analysis and debugging notebooks
|-- results/              # Algorithm output
|   |-- metrics/          # Performance measurements
|   |-- visuals/          # Solution visualizations
|-- solvers/              # Algorithm implementations
    |-- base.py           # Base solver class
    |-- informed.py       # A* implementation
    |-- mdp.py            # MDP algorithm implementations
    |-- uninformed.py     # DFS and BFS implementations
    |-- utils.py          # Utility functions
```

- **Solver classes**: All maze-solving algorithms are implemented as classes that inherit from a common base class (`MazeSolverBase` in `solvers/base.py`). This base class provides shared functionality such as maze loading, visualization, and performance metrics tracking. Using inheritance ensures consistent interfaces across all algorithm implementations while allowing each algorithm to have its specialized behavior.
- **Algorithm implementations**: The algorithms are logically grouped into separate modules:
  - uninformed.py contains DFS and BFS implementations.
  - informed.py contains the A* implementation.
  - mdp.py ontains Value Iteration and Policy Iteration implementations.

Automation with Makefile

A comprehensive Makefile was developed to automate all aspects of the experimentation process, providing a user-friendly interface for running multiple tests and saving overall time taken to conduct testing and analysis. The Makefile includes targets for:

- Maze generation commands to generate mazes of various sizes (7×7 to 101×101) with consistent properties.
- Commands to run specific algorithms on specific maze sizes.
- Execute benchmarking with commands to automatically run all algorithms on all maze sizes.

Such automation proved crucial for ensuring consistent testing conditions across all algorithms and maze sizes. The Makefile also manages the organization of output files, storing results in a structured directory hierarchy that separates metrics data and visualizations.

For example, running `make benchmark` automatically executes all five algorithms (DFS, BFS, A*, Value Iteration, and Policy Iteration) on all available maze sizes, generating comprehensive performance data in a single command. This approach minimizes human error in the experimental process and ensures reproducibility of results.

Maze generation outputs

The maze generation process produces two distinct outputs for each maze size:

- A text file representation of the maze stored as a NumPy array in CSV format (.txt), where characters represent different maze elements: '#' for walls, '.' for open paths, 'S' for start position, and 'G' for goal position. This format allows for easy loading and processing by the solver algorithms.
- A visual representation of the maze saved as a PNG image, where black cells represent walls, white cells represent paths, green marks the start position, and red indicates the goal. These visualizations provide an intuitive way to understand the maze structure and complexity.

All generated mazes are stored in a structured directory hierarchy, with text representations in `mazes/text/` and visual representations in `mazes/visuals/`. This organization ensures that mazes can be consistently referenced by both the solver algorithms and the analysis tools.

Data Collection and Metrics

The framework systematically collects a comprehensive set of performance metrics for each algorithm:

- *execution_time* is measured in seconds, capturing the computational efficiency of each algorithm.
- *nodes_explored* tracks how many states each algorithm visits during execution.
- *path_length* measures the quality of the solution found.
- *iterations* metric is available for MDP algorithms, which tracks how many iterations were needed for convergence.

These metrics are automatically saved to CSV files in the `results/metrics directory`, with naming conventions that clearly identify the maze size and algorithm used. Additionally, visual representations of the solution paths are rendered and saved to the `results/visuals` directory, providing intuitive visualization of how each algorithm navigates the maze.

Such meticulous experimental setup provides a robust foundation for analyzing the performance characteristics of different search and MDP algorithms, ensuring that conclusions drawn from the results are based on systematic and reproducible testing.

3 Results and Analysis

3.1 Search Algorithm Comparison (DFS, BFS, A*)

When comparing DFS, BFS, and A* algorithms against each other, several key performance differences emerge based on the collected metrics that are broadly discussed below.

- **Path quality**: As we can see from "Fig. 1", BFS guarantees the shortest path in unweighted graphs, while DFS often produces inefficient paths as it explores deeply before backtracking. At the largest maze size (101), DFS paths are approximately 8 times longer than optimal. A* consistently finds optimal paths identical to BFS but does so more efficiently by using heuristic guidance.
- **Computational efficiency**: Execution time analysis from "Fig. 2" reveals that all three algorithms have similar asymptotic growth as maze sizes increase, with execution times ranging from 0.1 to 10ms. A* shows interesting performance fluctuations at sizes 21 and 67, we can infer its heuristic might perform differently depending on maze structure. Despite these fluctuations, all algorithms seem to maintain comparable execution times across maze sizes, with slight advantages varying by maze complexity.
- **Memory usage**: "Fig. 3" demonstrates the peak memory usage during maze solving for each algorithm and shows that DFS consumes significantly more memory than both BFS and A*, particularly in larger mazes. At size 101, DFS requires approximately 4.4MB, while BFS uses only about 0.15MB and A* about 0.6MB. This actually contradicts the theoretical expectation that BFS would use more memory, suggesting implementation details might be affecting the measurements or that the maze structure with multiple paths from start to goal can influence memory patterns in the case of DFS.
- **Nodes explored**: BFS explores the most nodes across almost all maze sizes, followed by DFS, with A* exploring the fewest as seen in "Fig. 4". At size 101, BFS examines approximately 5400 nodes, DFS around 4400, and A* only about 4000. This demonstrates A*'s efficiency in avoiding unnecessary exploration. The data also shows an interesting anomaly at size 67 where A* explores significantly fewer nodes, suggesting maze structure may particularly favor A*'s heuristic in certain configurations.
- **Scalability**: As maze dimensions increase, A* demonstrates superior scalability by maintaining the optimal path length while exploring fewer nodes than both DFS and BFS. The logarithmic growth in execution time across all algorithms suggests good scalability, though with different trade-offs: DFS sacrifices path quality for potentially better space efficiency in some implementations, BFS guarantees optimality but explores more nodes, and A* achieves optimality while minimizing exploration.

Why does BFS seem to be more memory efficient than DFS? The theoretical space complexity of DFS is O(d) and BFS is $O(b\hat{d})$, where d is the depth of the solution and b is the branching factor. However, the current implementation shows that practical memory usage can diverge significantly from theoretical predictions due to specific implementation decisions.

The primary factor driving memory consumption in my implementation is the decision to store complete path histories with each frontier node. While this approach simplifies solution path reconstruction, it fundamentally alters the memory usage patterns:

- In DFS, the paths stored with each node grow rapidly as the algorithm explores deeply, creating a multiplicative effect on memory usage. Since DFS paths are significantly longer especially in large maze size (approximately 8 times longer in 101×101 mazes), the memory overhead grows accordingly.
- In BFS, while more total nodes may be explored, the paths associated with each node remain relatively short until the goal is reached. This results in substantially lower memory requirements despite the theoretically higher space complexity.

Additionally, maze complexity can also significantly impact memory consumption in the case of DFS:

- As maze size and complexity increase, DFS tends to explore deeper into "dead-end" paths before backtracking, leading to excessive path lengths as clearly visible in "Fig. 5" where BFS and DFS solution paths are visualized on the maze.

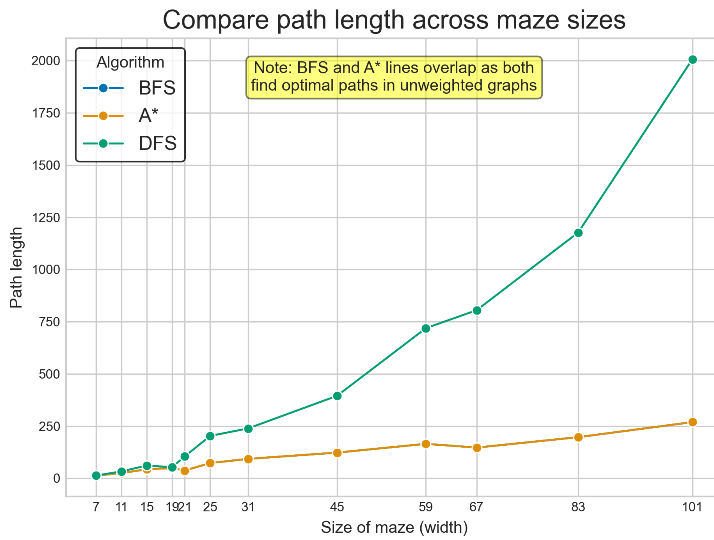


Figure 1: Comparison of path lengths across different maze sizes for search algorithms.

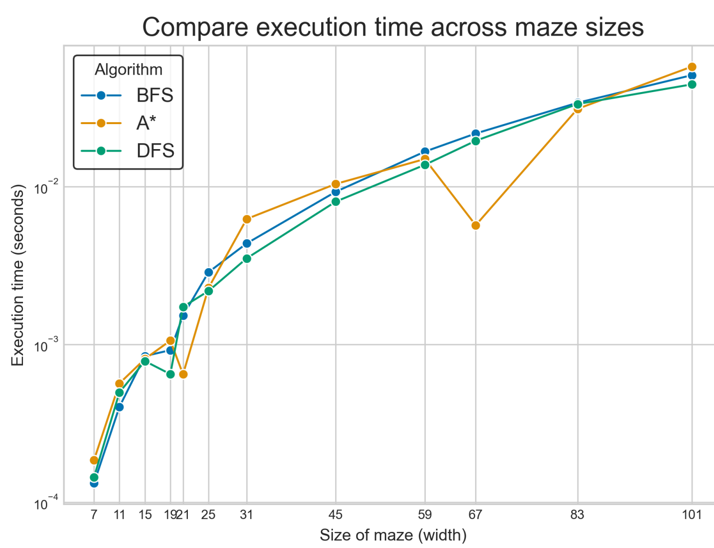


Figure 2: Comparison of execution time across different maze sizes for search algorithms.

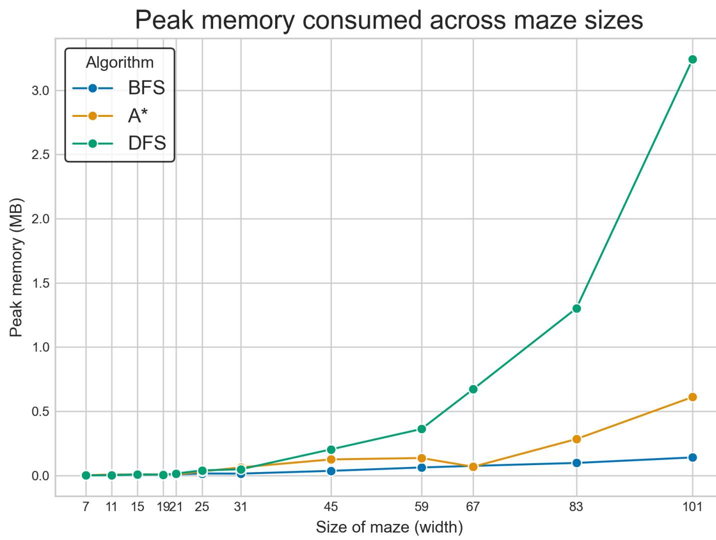


Figure 3: Comparison of peak memory usage across different maze sizes for search algorithms.

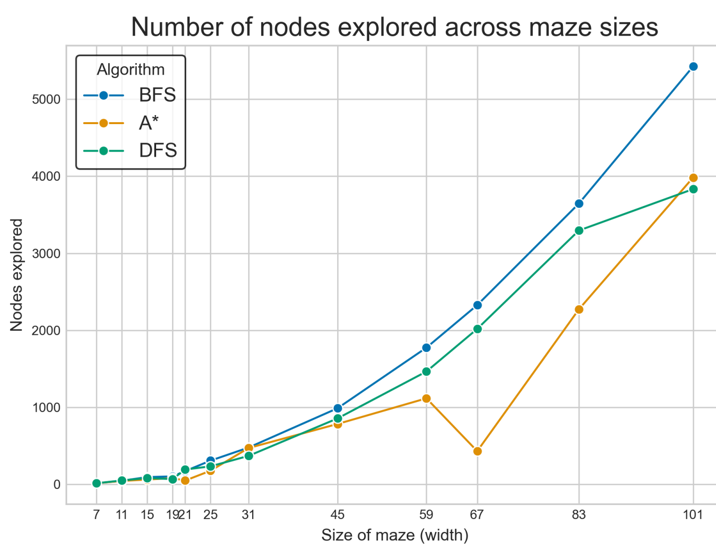
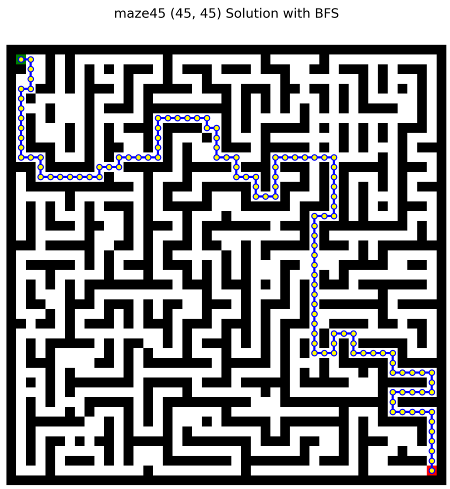
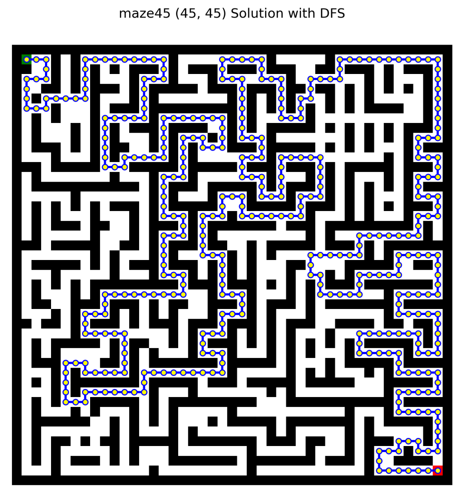


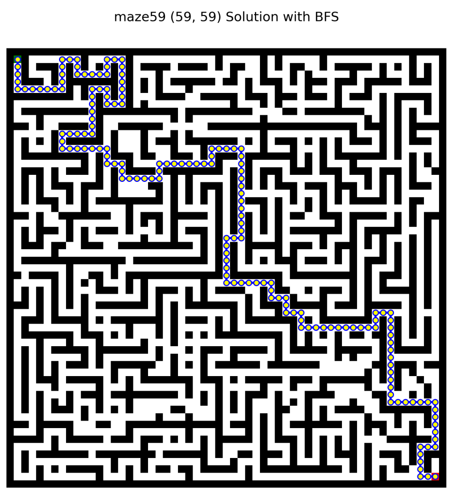
Figure 4: Plot of number of nodes explored for different maze sizes with search algorithms.



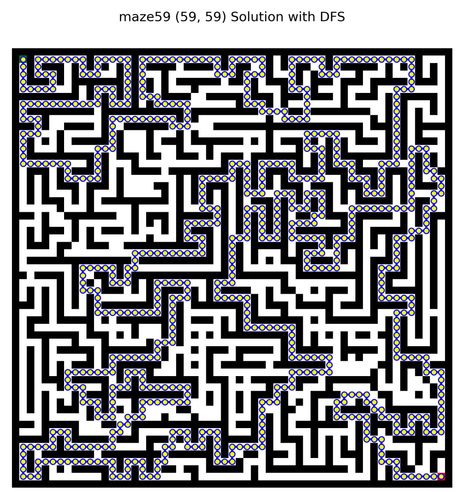
(a) Maze 45×45, BFS solution



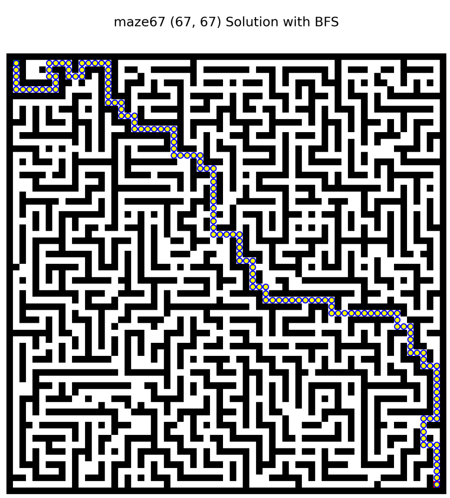
(b) Maze 45×45, DFS solution



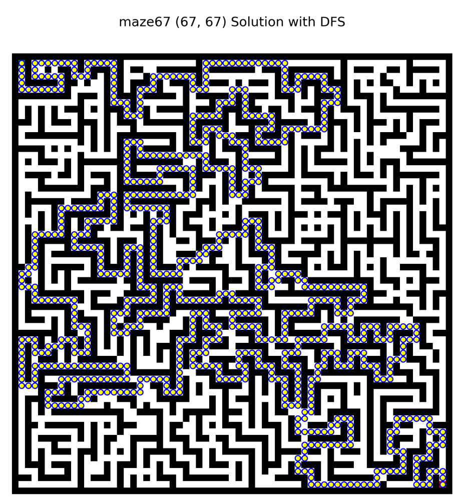
(c) Maze 59×59, BFS solution



(d) Maze 59×59, DFS solution

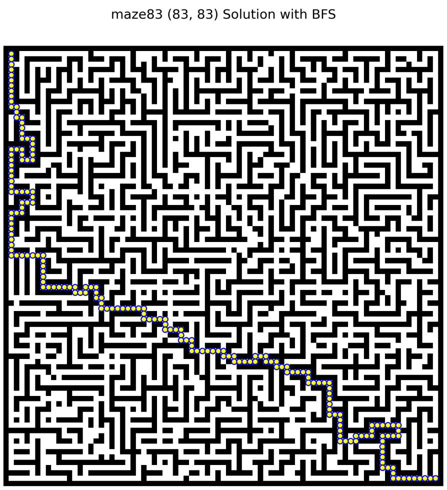


(e) Maze 67×67, BFS solution

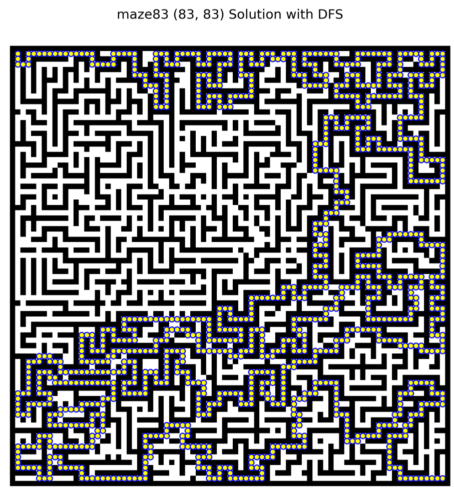


(f) Maze 67×67, DFS solution

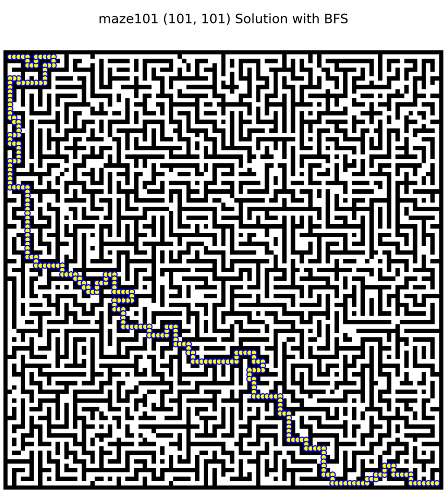
Figure 5: Comparison of BFS and DFS solutions for mazes of sizes 45×45, 59×59, and 67×67. The left column shows BFS solutions, which find optimal (shortest) paths. The right column shows DFS solutions, which tend to explore more of the maze and produce longer paths.



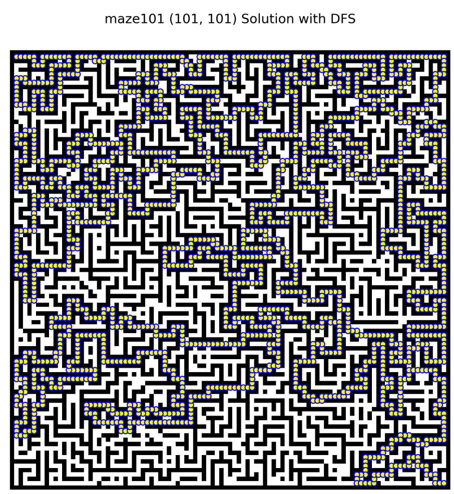
(g) Maze 83×83, BFS solution



(h) Maze 83×83, DFS solution



(i) Maze 101×101, BFS solution



(j) Maze 101×101, DFS solution

Figure 5: Comparison of BFS and DFS solutions for mazes of sizes 83×83 and 101×101. The difference in path length becomes increasingly pronounced as maze size increases.

- The side-by-side comparison of solution paths (particularly in the 83×83 and 101×101 mazes) visually demonstrates how DFS paths meander through large portions of the maze, while BFS paths take direct routes.
- Each additional turn or segment in these winding DFS paths compounds the memory overhead, as every node added to the stack carries a copy of an increasingly long path history.
- In complex mazes with many branching paths, this path-copying behavior creates a multiplicative effect on memory usage that grows more severe with maze size, explaining the steep upward trend in DFS memory consumption shown in "Fig. 3".

This finding highlights an important principle in algorithm implementation: theoretical space complexity assumes minimal state representation, while practical implementations often include additional state information that can significantly alter performance characteristics. When path reconstruction is a requirement, the traditional space advantage of DFS might get negated by the longer paths it generates.

3.2 MDP Algorithm Comparison (Value Iteration vs Policy Iteration)

The performance comparison between Value Iteration and Policy Iteration algorithms for solving Markov Decision Process models of the maze reveals several key differences and huge challenges specific to MDP-based approaches.

- **Solution convergence**: Both algorithms successfully found solutions for smaller mazes, converging to optimal policies. Value Iteration typically required more iterations to converge than Policy Iteration, as evidenced in their respective iteration counts plotted in "Fig. 6". For instance, in this maze solver, Value Iteration often needed 1.5-2x more iterations than Policy Iteration to reach convergence thresholds.
- **Computational efficiency**: A critical finding was the significant computational burden of MDP algorithms on larger mazes as indicated by the time taken to solve the maze in "Fig. 7". While search algorithms (DFS, BFS, A*) completed even for 101×101 mazes in milliseconds, MDP algorithms became prohibitively expensive beyond certain maze dimensions. Policy Iteration particularly suffered from the cubic complexity of its policy evaluation step, making it computationally infeasible for mazes larger than approximately 45×45 cells in my testing environment even after reducing the discount factor.
- **Memory usage**: Both MDP algorithms showed similar memory consumption patterns, proportional to the number of states in the maze as depicted in "Fig. 8". However, their memory requirements were substantially higher than search algorithms on an average for equivalent maze sizes due to the need to maintain complete value functions and policies across all states simultaneously.
- **State evaluations**: "Fig. 9" reveals exponential growth in state evaluations as maze sizes increase, with Policy Iteration consistently requiring 5-10 times more evaluations than Value Iteration. Both algorithms fail to find solutions at larger maze sizes (45×59), despite evaluating up to $10\hat{6}$ states. This show a fundamental scalability issue with MDP approaches in maze-solving, especially as performance degrades significantly beyond 31×31 mazes.
- **Parameter sensitivity**: Both algorithms showed significant sensitivity to parameter choices. The discount factor (γ) particularly impacted convergence speed and solution quality. With higher values (e.g., 0.999) creating precise but computationally expensive solutions and sometimes no solution at all, and lower values (e.g., 0.8) offering faster convergence at the cost of potential solution quality.

MDP Limitation with scalability

A fundamental observation from our experiments is the severe scalability limitation of MDP approaches compared to search algorithms:

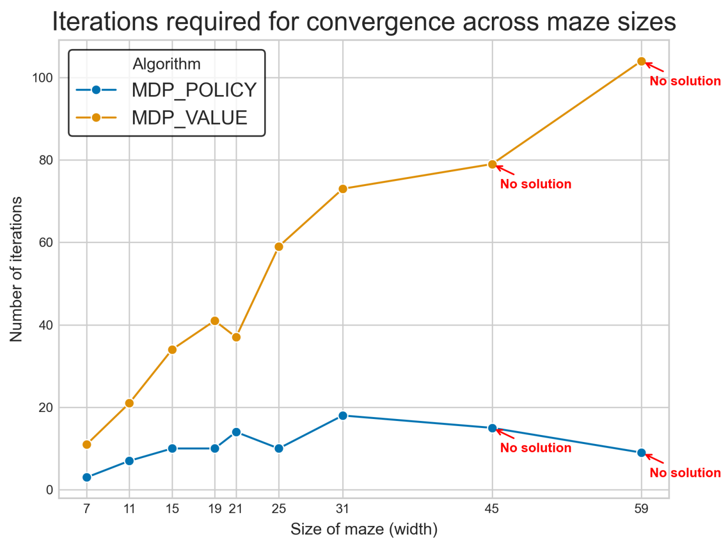


Figure 6: Comparison of iterations required for convergence across different maze sizes for MDP algorithms.

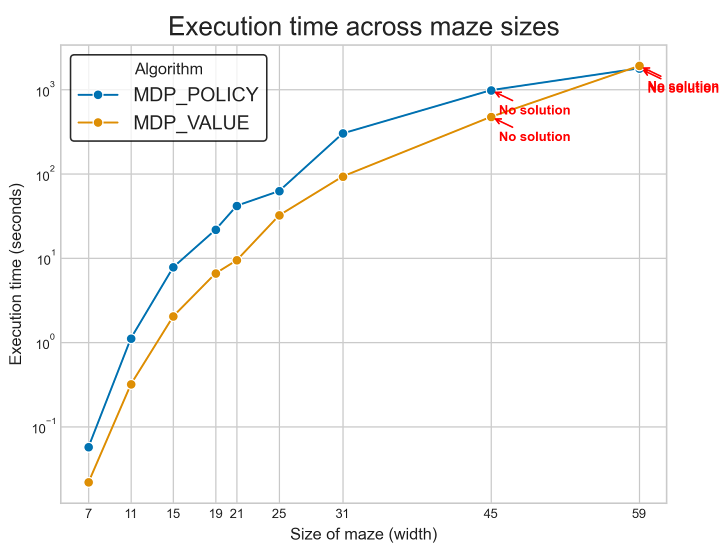


Figure 7: Comparison of execution time across different maze sizes for MDP algorithms, showing the dramatic increase in computational cost as maze sizes increase.

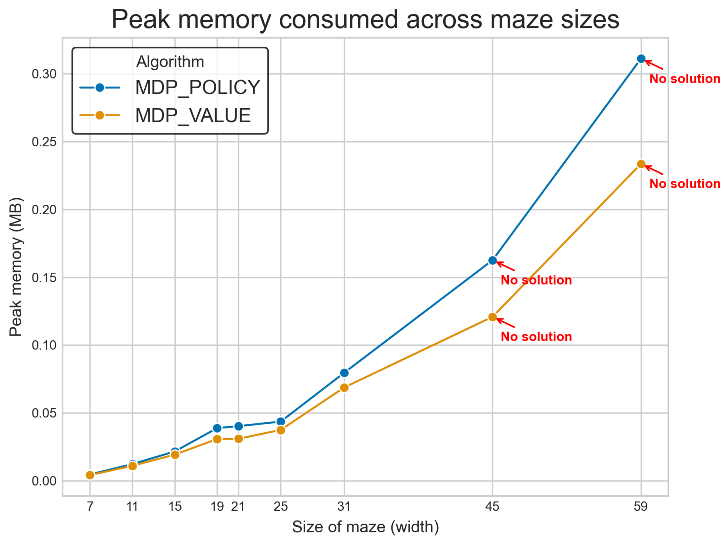


Figure 8: Comparison of memory consumed across different maze sizes for MDP algorithms.

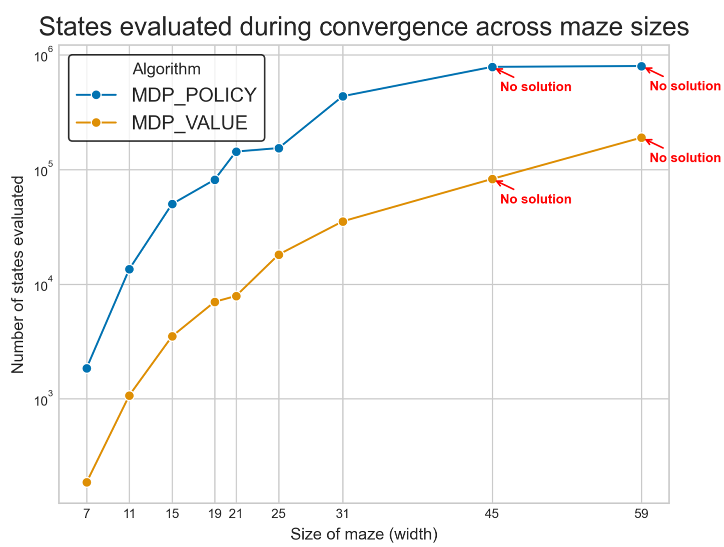


Figure 9: Comparison of states evaluated across different maze sizes for MDP algorithms, showing the huge increase in evaluation stage as maze sizes increase.

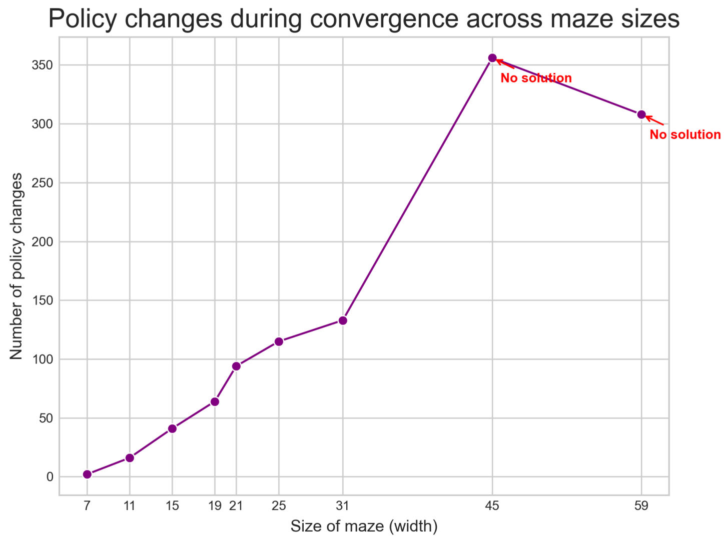


Figure 10: Plot of amount of policy changes executed during maze solving with MDP Policy algorithm.

- **State space explosion**: MDP algorithms suffer from the "curse of dimensionality" where computational requirements grow exponentially with the number of states. For this maze problem, it translated to quadratic growth in states with respect to maze dimensions (width×height).
- **Computation time growth**: While search algorithms showed moderate growth in execution time (from 0.1ms to 10ms) as maze size increased from 7×7 to 101×101, MDP algorithms exhibited dramatic increases from milliseconds for small mazes to several minutes or even hours for mazes beyond 45×45. Several larger maze configurations were unable to find solution and/or complete within practical time constraints.
- **Value Iteration vs Policy Iteration**: Value Iteration performed marginally better scalability than Policy Iteration for larger mazes, likely due to Policy Iteration's need to solve systems of equations during policy evaluation. However, both algorithms implementation became impractical beyond certain maze dimensions.

This analysis reveals that while MDP methods provide a theoretically sound approach to maze-solving with guaranteed optimality, their practical application is limited to smaller maze instances due to computational constraints. For large-scale maze solving, search algorithms like A* offer a significantly more efficient alternative while still guaranteeing optimal paths.

3.3 Search vs. MDP Algorithm Comparison

4 Conclusion

5 References

Appendices

A DFS Implementation Code

```
class DFSSolver(MazeSolverBase):
    """
    Maze solver using Depth-First Search.
    """
    def __init__(self, title, maze):
        """
        Initialize the maze solver with a given maze array.

        Args:
        maze (numpy.ndarray): 2D array representing the maze
        ('#' for walls, '.' for paths, 'S' for start, 'G' for goal).
        """
        super().__init__(title, maze)

        self.nodes_explored = 0
        self.nodes_available = len(list(zip(*np.where(maze == '.'))))
        self.execution_time = None

        self.start = tuple(zip(*np.where(maze == 'S')))[0]
        self.goal = tuple(zip(*np.where(maze == 'G')))[0]

        self.visited = np.zeros_like(maze, dtype=bool)
        self.solution_path = []

    def is_valid_move(self, x, y):
        """
        Check if the move is valid (within bounds and not a wall).

        Args:
        x (int): x-coordinate
        y (int): y-coordinate

        Returns:
        bool: True if move is valid, False otherwise.
        """
        return (0 <= x < self.width and
                0 <= y < self.height and
                self.maze[y, x] != '#' and
                not self.visited[y, x])

    def solve(self):
        """
        Solve the maze using Iterative Depth-First Search.

        Returns:
        list: Solution path if found, empty list otherwise.
        """
        # reset nodes explored, visited array, and solution path
        self.nodes_explored = 0
        self.visited = np.zeros_like(self.maze, dtype=bool)
        self.solution_path = []

        # define start point
        start_x, start_y = self.start

        # moves for dfs: down, right, up, left
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        stack = [(start_x, start_y, [])]

        while stack:
            current_x, current_y, path = stack.pop()
```

```python
            if self.visited[current_y, current_x]:
                continue

            self.visited[current_y, current_x] = True
            self.nodes_explored += 1

            current_path = path + [(current_x, current_y)]

            if (current_x, current_y) == self.goal:
                self.solution_path = current_path
                return current_path

            for dx, dy in moves:
                next_x, next_y = current_x + dx, current_y + dy

                if self.is_valid_move(next_x, next_y):
                    stack.append((next_x, next_y, current_path))

        # return empty array when no solution found
        return []

    def get_performance_metrics(self):
        """
        Return performance of the DFS algorithm for the maze solution.
        """
        return {
            'path_length': len(self.solution_path),
            'nodes_explored': self.nodes_explored,
            'nodes_available': self.nodes_available,
            'execution_time': self.execution_time,
            'is_solution_found': bool(self.solution_path)
        }
```

B BFS Implementation Code

```python
class BFSSolver(MazeSolverBase):
    """
    Maze solver using Breadth-First Search.
    """
    def __init__(self, title, maze):
        """
        Initialize the maze solver with a given maze array.

        Args:
        maze (numpy.ndarray): 2D array representing the maze
        ('#' for walls, '.' for paths, 'S' for start, 'G' for goal).
        """
        super().__init__(title, maze)

        self.nodes_explored = 0
        self.nodes_available = len(list(zip(*np.where(maze == '.'))))
        self.execution_time = None

        # find start and goal positions
        self.start = tuple(zip(*np.where(maze == 'S')))[0]
        self.goal = tuple(zip(*np.where(maze == 'G')))[0]

        self.visited = np.zeros_like(maze, dtype=bool)
        self.solution_path = []

    def is_valid_move(self, x, y):
        """
        Check if the move is valid (within bounds and not a wall).

        Args:
        x (int): x-coordinate
```

```python
        y (int): y-coordinate

        Returns:
        bool: True if move is valid, False otherwise.
        """
        return (0 <= x < self.width and
                0 <= y < self.height and
                self.maze[y, x] != '#' and
                not self.visited[y, x])

    def solve(self):
        """
        Solve the maze using Breadth-First Search.

        Returns:
        list: Solution path if found, empty list otherwise.
        """
        # reset nodes explored, visited array, and solution path
        self.nodes_explored = 0
        self.visited = np.zeros_like(self.maze, dtype=bool)
        self.solution_path = []

        # define start point
        start_x, start_y = self.start

        # moves for bfs: down, right, up, left
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        # use a queue for BFS (First-In-First-Out)
        queue = deque([(start_x, start_y, [])])
        self.visited[start_y, start_x] = True  # Mark start as visited
        self.nodes_explored += 1

        while queue:
            current_x, current_y, path = queue.popleft()  # Get the oldest element (FIFO)

            current_path = path + [(current_x, current_y)]

            # Check if goal is reached
            if (current_x, current_y) == self.goal:
                self.solution_path = current_path
                return current_path

            # Try all four directions
            for dx, dy in moves:
                next_x, next_y = current_x + dx, current_y + dy

                # If move is valid and cell not visited
                if self.is_valid_move(next_x, next_y):
                    queue.append((next_x, next_y, current_path))
                    self.visited[next_y, next_x] = True  # Mark as visited when added to queue
                    self.nodes_explored += 1

        # return empty array when no solution found
        return []

    def get_performance_metrics(self):
        """
        Return performance of the BFS algorithm for the maze solution.
        """
        return {
            'path_length': len(self.solution_path),
            'nodes_explored': self.nodes_explored,
            'nodes_available': self.nodes_available,
            'execution_time': self.execution_time,
            'is_solution_found': bool(self.solution_path)
        }
```

C A* Implementation Code

```
class AStarSolver(MazeSolverBase):
    """
    Maze solver using A* Search.
    """
    def __init__(self, title, maze):
        """
        Initialize the maze solver with a given maze array.

        Args:
        maze (numpy.ndarray): 2D array representing the maze
        ('#' for walls, '.' for paths, 'S' for start, 'G' for goal).
        """
        super().__init__(title, maze)

        self.nodes_explored = 0
        self.nodes_available = len(list(zip(*np.where(maze == '.'))))
        self.execution_time = None

        self.counter = 0

        self.start = tuple(zip(*np.where(maze == 'S')))[0]
        self.goal = tuple(zip(*np.where(maze == 'G')))[0]

        self.visited = np.zeros_like(maze, dtype=bool)
        self.solution_path = []

    def manhattan_distance(self, pos1, pos2):
        """
        Calculate Manhattan distance heuristic.

        Args:
        pos1 (tuple): First position (x, y)
        pos2 (tuple): Second position (x, y)

        Returns:
        int: Manhattan distance between positions
        """
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

    def is_valid_move(self, x, y):
        """
        Check if the move is valid (within bounds and not a wall).

        Args:
        x (int): x-coordinate
        y (int): y-coordinate

        Returns:
        bool: True if move is valid, False otherwise.
        """
        return (0 <= x < self.width and
                0 <= y < self.height and
                self.maze[y, x] != '#' and
                not self.visited[y, x])

    def solve(self):
        """
        Solve the maze using A* Search.

        Returns:
        list: Solution path if found, empty list otherwise.
        """
        self.nodes_explored = 0
        self.visited = np.zeros_like(self.maze, dtype=bool)
        self.solution_path = []
```

```python

        # track maximum fringe size for metrics
        max_fringe_size = 0

        start_pos = self.start
        goal_pos = self.goal

        parent = {}

        g_score = {start_pos: 0}

        # initialize priority queue for open set
        # with format: (f_score, counter, position)
        open_set = [(self.manhattan_distance(start_pos, goal_pos), self.counter, start_pos)]
        self.counter += 1

        while open_set:
            # update max fringe size
            max_fringe_size = max(max_fringe_size, len(open_set))

            # get node with lowest f_score
            _, _, current = heapq.heappop(open_set)
            current_x, current_y = current

            # skip if already visited
            if self.visited[current_y, current_x]:
                continue

            # mark as visited and update explored nodes
            self.visited[current_y, current_x] = True
            self.nodes_explored += 1

            if current == goal_pos:
                # reconstruct path if goal reached
                self.solution_path = self._reconstruct_path(parent, current)
                return self.solution_path

            # explore neighbors in the order: down, right, up, left
            moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

            for dx, dy in moves:
                next_x, next_y = current_x + dx, current_y + dy
                next_pos = (next_x, next_y)

                if self.is_valid_move(next_x, next_y):
                    # calculate tentative g-score
                    tentative_g_score = g_score[current] + 1

                    # when we find a better path to this neighbor
                    if next_pos not in g_score or tentative_g_score < g_score[next_pos]:
                        # update the path
                        parent[next_pos] = current
                        g_score[next_pos] = tentative_g_score

                        # calculate f_score = g_score + heuristic
                        f_score = tentative_g_score + self.manhattan_distance(next_pos,
    goal_pos)
                        heapq.heappush(open_set, (f_score, self.counter, next_pos))
                        self.counter += 1

        # return empty array when no path found
        return []

    def _reconstruct_path(self, parent, current):
        """
        Reconstruct the path from start to goal.

        Args:
```

```python
        parent (dict): Dictionary mapping node to its parent
        current (tuple): Current node (the goal)

        Returns:
        list: List of positions from start to goal
        """
        path = [current]
        while current in parent:
            current = parent[current]
            path.insert(0, current)
        return path

    def get_performance_metrics(self):
        """
        Return performance of the A* algorithm for the maze solution.
        """
        return {
            'path_length': len(self.solution_path),
            'nodes_explored': self.nodes_explored,
            'nodes_available': self.nodes_available,
            'execution_time': self.execution_time,
            'is_solution_found': bool(self.solution_path)
        }
```

D MDP Value Iteration Implementation Code

```python
class MDPValueIterationSolver(MazeSolverBase):
    """
    Maze solver using MDP Value Iteration.
    """
    def __init__(self, title, maze, discount_factor=0.9, theta=0.001, max_iterations=1000):
        """
        Initialize the MDP Value Iteration solver.

        Args:
            title: Name/title of the maze
            maze: The maze to solve (NumPy array)
            discount_factor: Discount factor for future rewards (gamma)
            theta: Threshold for determining value convergence
            max_iterations: Maximum number of iterations to perform
        """
        super().__init__(title, maze)

        self.nodes_available = len(list(zip(*np.where(maze == '.'))))

        self.execution_time = None

        self.start = tuple(zip(*np.where(maze == 'S')))[0]
        self.goal = tuple(zip(*np.where(maze == 'G')))[0]

        self.discount_factor = discount_factor
        self.theta = theta
        self.max_iterations = max_iterations
        self.values = {}
        self.policy = {}
        self.iterations = 0
        self.states_evaluated = 0

        self.solution_path = []

    def is_wall(self, x, y):
        """Check if a cell is a wall."""
        # check if coordinates are within bounds
        if x < 0 or x >= self.width or y < 0 or y >= self.height:
            return True
        # check if the cell is a wall (represented by '#')
        return self.maze[y, x] == '#'
```

```python

    def get_states(self):
        """Get all valid states (positions) in the maze."""
        states = []
        for y in range(self.height):
            for x in range(self.width):
                # include only non-wall cells as valid states
                if not self.is_wall(x, y):
                    states.append((x, y))
        return states

    def get_actions(self):
        """Define possible actions as cardinal directions."""
        return [
            (-1, 0),  # up
            (1, 0),   # down
            (0, -1),  # left
            (0, 1)    # right
        ]

    def get_reward(self, state, next_state):
        """
        Define rewards for transitions.

        Args:
            state: Current state (row, col)
            next_state: Next state (row, col)

        Returns:
            reward: Reward for this transition
        """
        # set reward = 100 for reaching the goal
        if next_state == self.goal:
            return 100

        # set penalty = -100 for hitting a wall
        if self.is_wall(*next_state):
            return -100

        # calculate Manhattan distance to goal for getting gradient reward
        goal_x, goal_y = self.goal
        next_x, next_y = next_state
        curr_x, curr_y = state

        curr_dist = abs(curr_x - goal_x) + abs(curr_y - goal_y)
        next_dist = abs(next_x - goal_x) + abs(next_y - goal_y)

        # if the next state is closer to goal, give bonus
        if next_dist < curr_dist:
            return -0.5

        # set small penalty for each step to encourage shorter paths
        return -1

    def get_transition_prob(self, state, action, next_state):
        """
        Get transition probability P(next_state | state, action).
        For a deterministic environment, this is 1 if next_state is the result of
        applying action to state, and 0 otherwise.
        """
        x, y = state
        dx, dy = action
        expected_next_state = (x + dx, y + dy)

        # if next_state is the expected result of the action, probability is 1
        if expected_next_state == next_state:
            # check if the move is valid (i.e. it doesn't hit a wall)
            if not self.is_wall(*expected_next_state):
```

```
                return 1.0

        # if we're expecting to hit a wall, we stay in the same place
        if self.is_wall(*expected_next_state) and state == next_state:
            return 1.0

        # otherwise probability is 0
        return 0.0

    def solve(self):
        """
        Solve the maze using Value Iteration.

        Returns:
            path: List of positions forming the path from start to goal
        """
        print(f"Start position: {self.start}")
        print(f"Goal position: {self.goal}")

        states = self.get_states()
        actions = self.get_actions()

        # initialize value function with goal having high value
        self.values = {state: 0 for state in states}
        self.values[self.goal] = 100

        # set properties for Value Iteration
        self.iterations = 0
        self.states_evaluated = 0

        for i in range(self.max_iterations):
            self.iterations += 1
            delta = 0

            # update values for all states
            for state in states:
                self.states_evaluated += 1

                # skip updating value if goal state
                if state == self.goal:
                    continue

                old_value = self.values[state]

                # calculate new value using Bellman equation
                new_value = float('-inf')

                for action in actions:
                    action_value = 0
                    for next_state in states:
                        prob = self.get_transition_prob(state, action, next_state)

                        if prob > 0:
                            reward = self.get_reward(state, next_state)
                            action_value += prob * \
                                (reward + self.discount_factor * self.values[next_state])
                    if action_value > new_value:
                        new_value = action_value

                self.values[state] = new_value
                delta = max(delta, abs(old_value - new_value))

            if delta < self.theta:
                print(f"Value Iteration converged after {i+1} iterations")
                break

        # extract policy from value function
        self.policy = {}
```

```
        for state in states:
            if state == self.goal:
                self.policy[state] = None
                continue

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

            for action in actions:
                action_value = 0

                for next_state in states:
                    prob = self.get_transition_prob(state, action, next_state)
                    if prob > 0:
                        reward = self.get_reward(state, next_state)
                        action_value += prob * \
                            (reward + self.discount_factor * self.values[next_state])

                if action_value > best_value:
                    best_value = action_value
                    best_action = action

            self.policy[state] = best_action

        path = self.extract_path()
        self.solution_path = path
        return path

    def extract_path(self):
        """
        Extract the path from start to goal using the computed policy.

        Returns:
            path: List of positions from start to goal
        """
        path = [self.start]
        current = self.start
        visited = {self.start}  # track visited states to prevent loops

        max_path_length = self.width * self.height

        while current != self.goal and len(path) < max_path_length:
            action = self.policy[current]

            if action is None:
                break

            x, y = current
            dx, dy = action
            next_state = (x + dx, y + dy)

            if self.is_wall(*next_state):
                break

            # Check for loops
            if next_state in visited:
                print(f"Warning: Loop detected in path at {next_state}")
                break

            visited.add(next_state)
            path.append(next_state)
            current = next_state

        return path

    def get_performance_metrics(self):
        """
        Return performance metrics for the solver.
```

```

        Returns:
            metrics: Dictionary of performance metrics
        """
        path = self.extract_path()
        is_solution_found = len(path) > 1 and path[-1] == self.goal

        return {
            'path_length': len(path) if is_solution_found else 0,
            'nodes_available': self.nodes_available,
            'iterations': self.iterations,
            'states_evaluated': self.states_evaluated,
            'execution_time': self.execution_time,
            'is_solution_found': is_solution_found
        }
```

E MDP Policy Iteration Implementation Code

```
class MDPPolicyIterationSolver(MazeSolverBase):
    """
    Maze solver using MDP Policy Iteration solver.
    """
    def __init__(self, title, maze, discount_factor=0.999,
        theta=0.0001, max_iterations=500, policy_eval_iterations=50):
        """
        Initialize the MDP Policy Iteration solver.

        Args:
            title: Name/title of the maze
            maze: The maze to solve (NumPy array)
            discount_factor: Discount factor for future rewards (gamma)
            theta: Threshold for determining value convergence
            max_iterations: Maximum number of policy iterations to perform
            policy_eval_iterations: Number of iterations for policy evaluation step
        """
        super().__init__(title, maze)

        self.nodes_available = len(list(zip(*np.where(maze == '.'))))

        self.execution_time = None

        self.start = tuple(zip(*np.where(maze == 'S')))[0]
        self.goal = tuple(zip(*np.where(maze == 'G')))[0]

        self.discount_factor = discount_factor
        self.theta = theta
        self.max_iterations = max_iterations
        self.policy_eval_iterations = policy_eval_iterations
        self.values = {}
        self.policy = {}
        self.iterations = 0
        self.policy_changes = 0
        self.states_evaluated = 0

        self.solution_path = []

    def is_wall(self, x, y):
        """Check if a cell is a wall."""
        # check if coordinates are within bounds
        if x < 0 or x >= self.width or y < 0 or y >= self.height:
            return True
        # check if the cell is a wall (typically represented by '#')
        return self.maze[y, x] == '#'

    def get_states(self):
        """Get all valid states (positions) in the maze."""
        states = []
```

```
        for y in range(self.height):
            for x in range(self.width):
                # include only non-wall cells as valid states
                if not self.is_wall(x, y):
                    states.append((x, y))
        return states

    def get_actions(self):
        """Define possible actions as cardinal directions."""
        return [
            (-1, 0),  # up
            (1, 0),   # down
            (0, -1),  # left
            (0, 1)    # right
        ]

    def get_reward(self, state, next_state):
        """
        Enhanced reward structure with goal-directed gradient.

        Args:
            state: Current state (x, y)
            next_state: Next state (x, y)

        Returns:
            reward: Reward for this transition
        """
        # set high reward for reaching the goal
        if next_state == self.goal:
            return 100

        # set penalty for hitting a wall
        if self.is_wall(*next_state):
            return -100

        # calculate Manhattan distance to goal for gradient reward
        goal_x, goal_y = self.goal
        next_x, next_y = next_state
        curr_x, curr_y = state

        # get distances to goal
        curr_dist = abs(curr_x - goal_x) + abs(curr_y - goal_y)
        next_dist = abs(next_x - goal_x) + abs(next_y - goal_y)

        # if next state is closer to goal, give bonus
        if next_dist < curr_dist:
            return -0.5  # Small step penalty but better than standard step

        # standard step penalty
        return -1

    def get_transition_prob(self, state, action, next_state):
        """
        Get transition probability P(next_state | state, action).
        For a deterministic environment, this is 1 if next_state is the result of
        applying action to state, and 0 otherwise.
        """
        x, y = state
        dx, dy = action
        expected_next_state = (x + dx, y + dy)

        # if next_state is the expected result of the action, probability is 1
        if expected_next_state == next_state:
            # check if the move is valid (i.e. it doesn't hit a wall)
            if not self.is_wall(*expected_next_state):
                return 1.0

        # if we're expecting to hit a wall, we stay in the same place
```

```python
        if self.is_wall(*expected_next_state) and state == next_state:
            return 1.0

        # otherwise probability is 0
        return 0.0

    def policy_evaluation(self, policy, states, actions):
        """
        Evaluate policy with convergence check.

        Args:
            policy: Current policy mapping states to actions
            states: List of all states
            actions: List of possible actions

        Returns:
            values: Dictionary mapping states to their values
        """
        values = {state: 0 for state in states}

        # set goal state value higher to create gradient
        values[self.goal] = 100

        for _ in range(self.policy_eval_iterations):
            delta = 0
            for state in states:
                self.states_evaluated += 1

                # skip evaluating if reached goal state
                if state == self.goal:
                    continue

                old_value = values[state]
                action = policy[state]

                # if no action is defined for this state, skip it
                if action is None:
                    continue

                new_value = 0
                for next_state in states:
                    prob = self.get_transition_prob(state, action, next_state)

                    if prob > 0:
                        reward = self.get_reward(state, next_state)
                        new_value += prob * (reward + self.discount_factor * values[
    next_state])

                values[state] = new_value
                delta = max(delta, abs(old_value - new_value))

            # check for convergence
            if delta < self.theta:
                break

        return values

    def policy_improvement(self, values, states, actions):
        """
        Improve policy based on value function.

        Args:
            values: Current value function
            states: List of all states
            actions: List of possible actions

        Returns:
            policy: Improved policy
```

```python
            is_stable: Whether the policy has stabilized
        """
        policy = {}
        is_stable = True

        for state in states:
            if state == self.goal:
                policy[state] = None
                continue

            old_action = self.policy.get(state)

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

            for action in actions:
                action_value = 0

                for next_state in states:
                    prob = self.get_transition_prob(state, action, next_state)
                    if prob > 0:
                        reward = self.get_reward(state, next_state)
                        action_value += prob * (reward + self.discount_factor * values[
    next_state])

                if action_value > best_value:
                    best_value = action_value
                    best_action = action

            policy[state] = best_action

            # check if policy has changed
            if old_action != best_action:
                is_stable = False
                self.policy_changes += 1

        return policy, is_stable

    def solve(self):
        """
        Solve the maze using Policy Iteration.

        Returns:
            path: List of positions forming the path from start to goal
        """
        print(f"Start position: {self.start}")
        print(f"Goal position: {self.goal}")

        states = self.get_states()
        actions = self.get_actions()

        # initialize policy with goal-directed actions
        self.policy = {}
        for state in states:
            if state == self.goal:
                self.policy[state] = None
            else:
                # get all valid actions
                valid_actions = []
                for action in actions:
                    x, y = state
                    dx, dy = action
                    next_state = (x + dx, y + dy)
                    if not self.is_wall(*next_state):
                        valid_actions.append(action)

                if valid_actions:
                    # use action that gets closest to goal if possible
```

```
                    goal_x, goal_y = self.goal
                    best_action = None
                    min_distance = float('inf')

                    for action in valid_actions:
                        x, y = state
                        dx, dy = action
                        next_x, next_y = x + dx, y + dy
                        dist = abs(next_x - goal_x) + abs(next_y - goal_y)

                        if dist < min_distance:
                            min_distance = dist
                            best_action = action

                    self.policy[state] = best_action
                else:
                    self.policy[state] = None

        # set properties for policy iteration
        self.iterations = 0
        self.policy_changes = 0
        self.states_evaluated = 0

        for i in range(self.max_iterations):
            self.iterations += 1

            # one, do Policy Evaluation
            self.values = self.policy_evaluation(self.policy, states, actions)

            # two, do Policy Improvement
            new_policy, is_stable = self.policy_improvement(self.values, states, actions)
            self.policy = new_policy

            # now check if policy has stabilized
            if is_stable:
                print(f"Policy Iteration converged after {i+1} iterations")
                break

        path = self.extract_path()
        self.solution_path = path
        return path

    def extract_path(self):
        """
        Extract the path from start to goal using the computed policy.

        Returns:
            path: List of positions from start to goal
        """
        path = [self.start]
        current = self.start
        visited = {self.start}  # track visited states to prevent loops

        max_path_length = self.width * self.height

        while current != self.goal and len(path) < max_path_length:
            action = self.policy[current]

            if action is None:
                break

            x, y = current
            dx, dy = action
            next_state = (x + dx, y + dy)

            # check if the next state is valid
            if self.is_wall(*next_state):
                # this shouldn't happen with a valid policy
```

```
                break

            # Check for loops
            if next_state in visited:
                print(f"Warning: Loop detected in path at {next_state}")
                break

            visited.add(next_state)
            path.append(next_state)
            current = next_state

        return path

    def get_performance_metrics(self):
        """
        Return performance metrics for the solver.

        Returns:
            metrics: Dictionary of performance metrics
        """
        path = self.extract_path()
        is_solution_found = len(path) > 1 and path[-1] == self.goal

        return {
            'path_length': len(path) if is_solution_found else 0,
            'nodes_available': self.nodes_available,
            'iterations': self.iterations,
            'policy_changes': self.policy_changes,
            'states_evaluated': self.states_evaluated,
            'execution_time': self.execution_time,
            'is_solution_found': is_solution_found
        }
```