CS7IS2: Artificial Intelligence Assignment 1

HRITIKA RAHUL MEHTA*, 24334316, MSc. Computer Science - Data Science

1 Introduction

Maze solving is a standard for artificial intelligence techniques because it entails pathfinding, planning, and decision-making in an orderly environment. In this assignment, we test and contrast five algorithms—three standard search algorithms (Depth-First Search, Breadth-First Search, and A*) and two Markov Decision Process (MDP) algorithms (Value Iteration and Policy Iteration)—for solving randomly generated mazes with varying sizes. Through comparing search-based and MDP-based solutions, we recognize their merits and demerits in terms of solution optimality, computation time, and space requirements.

This report covers the entire process: Section 2 summarizes the maze structure and generation. Sections 3 and 4 present the implementation of the five algorithms, key data structures, heuristics, and parameter choices. Section 5 explains the experimental setup—how the mazes were configured and what performance measures were employed—and Section 6 presents the resulting data and comparison of the algorithms. Finally, the report ends with observations on algorithm performance, potential limitations, followed by references and appendices containing code and raw data.

2 Maze Generation

This section explains the pre-processing and generation of mazes for search and MDP algorithms. The code below uses an amalgamation of single-step Depth-First Search (DFS) to generate passages, optional generation of loops, and some additional helper functions in order to provide a well-defined and drawable maze.

2.1 Class Description of MazeGenerator

The MazeGenerator class carries out the entire process of generating a maze, right from creating an empty grid to building passages and displaying the output. Key points:

- Grid Representation: The maze is stored in a 2D NumPy array self.maze, where 1 denotes a wall and 0 denotes an open cell.
- Start and Goal Cells: Default initial cell is (1, 1) and terminal cell is (rows - 2, cols - 2). These are chosen so that they will lie inside the grid and not at the border.
- Visited Array: A temporary 2D boolean array self.visited tracks visited cells while exploring so that a region does not get carved twice.

2.2 Single-Step DFS for Maze Carving

The primary algorithm for carving passages is a one-step DFS strategy in generate_maze():

(i) Initialization:
- Mark the start cell as open (0) and visited.
- Push the start cell onto a stack, stack_maze.

(ii) DFS Loop
- While the stack is not empty, look at the top cell (r, c).
- Collect potential neighbors that are:
  - Within the inner grid (i.e., not on the boundary).
  - Still walls ($maze[nr, nc] == 1$).
  - Not yet visited (not self.visited[nr, nc]).
  - Have exactly one open neighbor ($count\_open\_neighbors(nr, nc) == 1$), which helps maintain a corridor-like structure.
- If it has no neighbor, pick one arbitrarily, open it (0), move there, and push it onto the stack.
- If an invalid neighbor is not discovered, pop stack to backtrack.

(iii) Goal Cell Validation: After DFS is achieved, the program also positions the target cell (self.goal) in the vicinity of a current path.

This method will create a maze with tree-like structure and few loops (apart from added intentionally later). Because the algorithm is searching for 1 open neighbor, it will create newly cut passageways branch off existing pathways without making large open spaces unnecessarily.

2.3 Loop and Outer Wall Formation

- add_outer_walls(): Ensures the outer border of the grid is fully walled (1). This prevents paths from "bleeding" along the edges and produces a fully closed maze.
- add_loops($probability = 0.1$): Randomly transforms a subset of the wall cells into open cells, adding loops into otherwise tree-like build.You can determine the number of loops placed by adjusting the probability parameter, thus the connectivity and complexity of the maze.

2.4 Maze Connectivity Test

- is_path_to_goal(): Does a BFS from the start cell to determine if the goal is reachable. If BFS can find the goal, returns True; else, False. This could be useful for checking if the maze is solvable when generated, especially if loops or additional walls were added.

2.5 MDP Initialization

- initialize_values_bfs(): Solves a problem of goal iteration, going back from the goal to set an initial value function array V_init. Closer cells to the goal are given more positive values (less negative), and unattainable cells are given a sentinel value (e.g., -9999).This allows MDP algorithms (e.g., Value Iteration, Policy Iteration) to be initialized with a better-informed starting state.

2.6 Visualization

Displays the maze on the screen with Matplotlib. The method:

- Calls add_outer_walls() to ensure a consistent boundary.
- Places self.maze in a display matrix (maze_display).

Author's Contact Information: Hritika Rahul Mehta, mehtah@tcd.ie, 24334316, Dublin, MSc. Computer Science - Data Science.

- Places start (green) and goal (red).
- Draws an optional solution path (blue) if provided.
- Single color map plots where 0 is white (open) and 1 is black (wall).

Not only is this operation helpful while debugging, but also while visually verifying the correctness of the generated maze and the generated path by the search or MDP algorithms.

3 Search Algorithms

This sub-section explains the coding and implementation of the three search algorithms—Depth-First Search (DFS), Breadth-First Search (BFS), and A*—used for searching the mazes generated. It explains the theoretical background of each algorithm, the corresponding data structures and optimizations used, and the main challenges encountered in the process of implementation.

3.1 Algorithm Descriptions

3.1.1 Depth-First Search (DFS). DFS goes as far as possible down one branch before backtracking. Abstractly, it makes use of a stack data structure (either an explicit stack or by means of recursion) to hold the path moved from the starting cell. When DFS moves into a new cell, it pushes any as-yet-unvisited neighbors onto the stack. If it reaches a dead end, it backtracks by popping cells from the stack until it finds an unvisited branch. While DFS is simple to implement, it is not always the shortest path to a maze; its greatest advantage is low overhead in code complexity.

3.1.2 Breadth-First Search (BFS). BFS explores the maze systematically by visiting one of a cell's neighbors before visiting the next "layer" of cells. It uses a queue to know which cell to visit next. Because BFS explores outwards uniformly from the start, it finds the shortest path in an unweighted maze and thus is appropriate where path optimality is required. But BFS may be more memory intensive than DFS, especially for large mazes, because it has to hold all frontier cells at each level in the queue.

3.1.3 A Search.* A* adds BFS-like search with a heuristic function to guide the search closer to the goal. In our program, we typically use Manhattan distance or Euclidean distance between the goal and the cell as a heuristic (depending on whether we allow diagonal movements). A priority queue (min-heap) is used to assign greater priority to visiting cells that appear nearer to the goal according to f(n)=g(n)+h(n), where g(n) is the cost of traveling to cell n, and h(n) is the heuristic guess to the goal. A* will compute the shortest path if the heuristic is admissible (never overestimates the true cost). Its performance will generally be better than BFS on larger mazes, provided an adequate heuristic is used.

3.2 Implementation Details

- Data Structures
  - DFS uses either a recursive call stack or an explicit stack (LIFO) data structure. A simple Python list can be adapted as the stack, utilizing append() and pop().
  - BFS requires a queue (FIFO). In Python, collections.deque is widely used to facilitate efficient enqueue/dequeue operations.
  - A* uses a priority queue to expand the node with the lowest f(n) consistently. We use heapq in Python, where items are stored in a min-heap.
- Visited Set - For all three algorithms, we maintain a visited set (a Python set, for instance) such that we will not revisit a cell. This matters in preventing infinite loops and conserving computation.
- Heuristic Choice (A*) - We employed Manhattan distance $h(n) =\mid xn - xgoal \mid + \mid yn - ygoal \mid$ in most mazes with horizontal and vertical moves only. For diagonal movement permitted, Euclidean distance could be more suitable. In the assignment, we typically target Manhattan distance for consistency and admissibility.
- Complexity Considerations - DFS is $O(\mid V \mid + \mid E \mid)$ in the worst case, say O(cells) for a grid maze. Recursive DFS is space-efficient but really deep for large mazes. BFS is $O(\mid V \mid + \mid E \mid)$ but possibly more space used in the queue, especially halfway through the layers of the search. A* worst-case complexity is $O(\mid E \mid)$, but with a good heuristic, the number of nodes to expand can be reduced significantly.
- Implementation Structure - Each algorithm has its own function for convenience (e.g., dfs solve(maze, start, goal)) , with arguments the maze, the start cell, and the goal cell. We also provide a shared interface (e.g., a main driver function) to enable direct comparison between algorithms.

3.3 Challenges and Solutions

- Avoiding Infinite Loops - One of the dangers of DFS is backtracking into cells, which results in infinite recursion. We avoided this by having a robust visited set. We also tested boundary conditions so that we do not go out of the maze or through walls.
- Ensuring Shortest Paths - BFS and A* will always compute the shortest path if properly implemented, but only when the A* heuristic is admissible. We tested this property with small mazes whose solutions were known and ensured our algorithm's path cost equaled BFS.
- Handling Large Mazes - For large mazes, BFS and A* will consume a lot of memory. We mitigated this by reducing data structures (e.g., utilizing Python's deque for BFS, heapq for A*) and by not copying states excessively.
- Heuristic Tuning - If the heuristic is too large or not exact, A* can explore useless paths or violate optimality. We performed initial tests on small mazes to verify that the Manhattan distance never overestimated the cost to the goal.
- Performance Trade-Offs - In experimentation, we observed that DFS is quick to run but less predictable in terms of the shortest path, BFS is slow for large mazes but always produces an optimal solution, and A* is optimally balanced if a good heuristic is selected. In Section 6, we have provided elaborate comparisons to show the above points empirically.

By following these best practices and design principles, we ensured that all three of the search algorithms were working correctly and consistently within the context of the maze. The next section describes how the MDP-based algorithms (Value Iteration and Policy

Iteration) were developed and integrated within the same infrastructure.

4 MDP Algorithms Implementation

Other than search-based solutions, two of the most elementary MDP algorithms - Value Iteration and Policy Iteration - are employed to find a solution for the maze. Both approaches rely on specifying the maze as an MDP where each open cell is a state, and actions specify moves (right, left, up, down). Rewards are typically -1 for all moves to keep the path lengths small, and the goal state is terminal. This subsection explains the Value Iteration (value.py) and Policy Iteration (policy.py) implementations, along with crucial steps, parameters, and performance metrics.

4.1 MDP Formulation

(i) States: State for each open cell (r,c) unless it's a wall (maze[r, c] == 1). The goal state (say, (rows-2, cols-2)) is considered terminal so that nothing happens or gets updated from there.

(ii) Actions: We specify a set of four possible actions: U (up), D (down), L (left), and R (right). Actions that would step into walls or out-of-bounds cells are actually "no-ops," which leave the agent in the same cell with a reward of -1.

(iii) Transition Function: We have deterministic transitions: each action moves the agent to the corresponding neighboring cell or, if the cell is blocked or out of bounds, leaves the agent in place.

(iv) Rewards and Discount Factor (γ): Reward for every non-terminal step is -1 to avoid longer paths. Discount factor, γ, generally is 0.9 (but can be tuned). Higher value of γ gives more importance to future rewards, while lower value of γ gives more importance to more recent rewards.

(v) Initialization:
- For Value Iteration, the software invokes generator.initialize_values_bfs(), which executes a reverse BFS from the goal to initialize an initial heuristic-like value map with unvisited cells set to -9999.
- For Policy Iteration, it utilizes a zero-initialized trivial value table and an action policy ('U' or any valid move) by default.

4.2 Value Iteration

The Value Iteration process updates each state's value by considering all possible actions and taking the maximum expected return:

(1) Initialisation:
- A 2D array V is populated via initialize_values_bfs() to give a starting estimate. The goal cell (generator.goal) is explicitly set to 0.

(2) Iterative Updates: For all states (i,j) which are not a wall and not the goal:
- (a) Calculate the Q-value for each action a by: $Q(i, j, a) = R + \gamma V(nextstate)$ where $R = 1$ and next state is determined by the action taken.
- (b) Set $V_{new}(i, j) = max_{a \in A} Q(i, j, a)$.
- (c) The policy at (i,j) is chosen as the action that yields the highest Q-value.

(3) Convergence Check:
- After updating all states, compute δ, the maximum absolute difference between old and new values. If $\delta < \theta$ (a small threshold, e.g. 1e-4), or if the maximum number of iterations (max_iter) is reached, the process stops.

(4) Retrieving the Path:
- Once values settle, policy is a dictionary from (row,col) to best action. extract_path() follows a path from start to end by taking these actions one at a time.

4.3 Policy Iteration

Policy Iteration alternates between Policy Evaluation (computing the value function for a policy) and Policy Improvement (improving the policy using the computed values):

(1) Initialization: A 2D array V of zeros is created for all states. A default policy is set in policy_arr (e.g., 'U' for each cell). Cells that are walls or the goal have no valid policy.

(2) Policy Evaluation: For each non-wall, non-goal cell (i,j), the algorithm applies the current policy's action a to update $V(i, j) \leftarrow -1 + \gamma V(nextstate)$ This step repeats until the values converge ($\delta < \theta$).

(3) Policy Improvement:
- For each cell, look at all possible actions and choose the one with the highest Q-value. If that optimal action differs from the policy action, set policy_stable = False.
- If, having looked at all cells, the policy has not changed, policy_stable remains True, and the algorithm terminates.

(4) The produced policy is stored in a dictionary mapping (row,col) to best action. We construct a start-to-end path for visualization and path length calculation with extract_path().

4.4 Practical Considerations

(i Solvability guarantees:Both programs test if the maze is solvable by calling generator.is_path_to_goal(). In case it is not solvable, they include extra loops or print a warning.

(ii Hyperparameters
- Discount Factor (γ): Typically 0.9, but can be varied. With a higher value, more weight is given to rewards very far in the future.
- Convergence Threshold (θ): Typically 1e-4 to trade off accuracy and run time.
- Maximum Iterations: In Value Iteration (max_iter), to make the process stop even if convergence is very slow.

(iii Result Visualization: Both approaches end with calling generator.visualize_maze(), marking the blue solution path with color and displaying the maze for easy inspection of policy-created path.

(iv Comparison with Search Algorithms: In Section 6 (Results and Analysis), we contrast on which of these MDP methods varies from DFS, BFS, and A* in path quality, time taken, memory used, and states visited. More computation is typically required from the MDP methods but gives a policy for all states (and not for one path).

By specifying the MDP as such and implementing it, we now have two complimentary algorithms to solve the maze problem. Value Iteration repeatedly updates state values, while Policy Iteration alternates between policy evaluation and policy improvement. Both generate near-optimal solutions (routes) to the maze problem but with varying rates of convergence, memory usage, and number of states expanded, as described below in this report.

5 Experimental Setup and Comparison

In this section, the systematic evaluation and comparison of the five algorithms' performance—DFS, BFS, A*, Value Iteration, and Policy Iteration—are explained. We measure statistics on runtime, states expanded, maximum memory used, and solution quality across different maze sizes and over multiple runs and output the results to a CSV file for analysis.

5.1 Evaluation Metrics

(i) Runtime: It measures total execution time (in seconds) from the start of the algorithm to convergence (for MDPs) or to when a solution path is discovered (for search algorithms). Specifies how rapidly the algorithm calculates or produces the maze.

(ii) States expanded : Number of states visited by each algorithm (i.e., cells popped off a frontier in DFS/BFS/A*, or value/policy iteration count in MDPs). A measure of computational effort. There's a decent algorithm that will still explode enormous numbers of states, and that is not always desirable under memory constraints or for real-time systems.

(iii) Maximum Memory Usage: Maximum number of states stored in memory at the same time.In search algorithms, this will normally be the size of the biggest frontier. For MDP algorithms, it's here approximated by the 2D arrays' sizes (e.g., rows * cols). This is used more by some algorithms, which means it may become impossible for large mazes or limited hardware

(iv) Solution Quality: For search algorithms, we observe the solution path length. For MDP algorithms, we can either count path length from start to goal according to the final policy, or use a value-based measure if we wish. Ensures that the algorithm solution is not just speedy but also good (e.g., near-optimal path).

5.2 Experimental Procedure

(i) Maze Sizes and Runs
- We list an enumeration of maze sizes, i.e.: (10×10),(30×30), (50×50),(100×100),(150×150),(200×200).
- Each size is executed a few times (e.g., $num\_runs = 3$) in order to mitigate random fluctuations (e.g., different loop creation patterns).

(ii) Generation of Maze
- A fresh MazeGenerator(rows, cols) object is instantiated on each run.
- Walls are carved using the one-step DFS algorithm presented in Section 2.
- Loops are inserted optionally with a probability of 0.1 or 0.2, which makes the maze layout more complex.

(iii) Solvability
- The is_path_to_goal() function is then called after generation. In the event that the maze is not solvable, additional loops are added until it is solvable or we have attempted a number of times.
- If still not solvable, such an iteration is omitted or marked with a warning.

(iv) Executing the Algorithm
- Search Algorithms (DFS, BFS, A*): We note the time each algorithm took to find a path, the number of states explored, the maximum frontier size, and the found path length.
- Value Iteration and Policy Iteration:
  (a) Initialization: Value Iteration may start with initialize_values _bfs(), while Policy Iteration begins with zero-initialized values and a default policy.
  (b) Convergence: The algorithm runs until $\delta < \theta$ or a maximum iteration limit.
  (c) Metrics:
  - runtime is recorded with a simple time.time() call before and after.
  - states_expanded counts how many state updates or evaluations occur.
  - peak_memory is approximated by rows * cols in this code.
  - solution_quality is the length of the extracted path using extract_path().

(v) Data Storage and Export:
- After each algorithm finishes on a given maze, its metrics are appended to a list of results.
- Once all runs complete, the results are written to a CSV file, including fields such as algorithm, maze_size, runtime, states_expanded, peak_memory, and solution_quality.
- This CSV serves as a convenient format for subsequent data analysis and plotting.

5.3 Visualization and First Analysis

- Visual Output
  - The code outputs the maze with the solution path overlaid, so a quick visual verification of correctness is achievable.
  - Individual plots per run or per algorithm can be generated (e.g., side-by-side bar chart for Value Iteration and Policy Iteration).
- Data Insights
  - The CSV log can be used to create performance graphs e.g., runtime vs. maze size or states expanded vs. maze size-to highlight how each algorithm scales.
  - Solution quality can be compared to see which methods produce shorter or more optimal paths, and how quickly.

5.4 Summary of Experimental Setup

In all, the experimental design offers an unbiased, fair comparison of all five algorithms over a range of maze sizes and random cases. Through measurements of multiple metrics—runtime, number of states that were expanded, peak memory consumption, and solution quality—the test picks up on different aspects of algorithm performance. The output CSV file and plots form input to analysis in Section 6, where we describe and comment upon trends and trade-offs between DFS, BFS, A*, Value Iteration, and Policy Iteration.

6 Results and Analysis

Here, we are comparing the search-based algorithms (DFS, BFS, A*) with the MDP-based algorithms (Value Iteration, Policy Iteration) for different sizes of maze. The data were collected from two CSV files:

(1) Results of the three search algorithms (DFS, BFS, A*).
(2) Results of the two MDP algorithms (Value Iteration, Policy Iteration).

Each run recorded four key metrics (see Section 5 for details):

- Runtime (seconds)
- States Expanded
- Peak Memory Usage (approx. or actual, depending on your approach)
- Solution Quality (e.g., path length)

Below, we present aggregated results for a subset of maze sizes, followed by discussions supported by tables and graphs.

6.1 Search Algorithms: DFS, BFS, and A*

Table 1 shows average results for DFS, BFS, and A* across three representative maze sizes (30×30, 50×50, and 100×100). Each entry is an average over three runs.

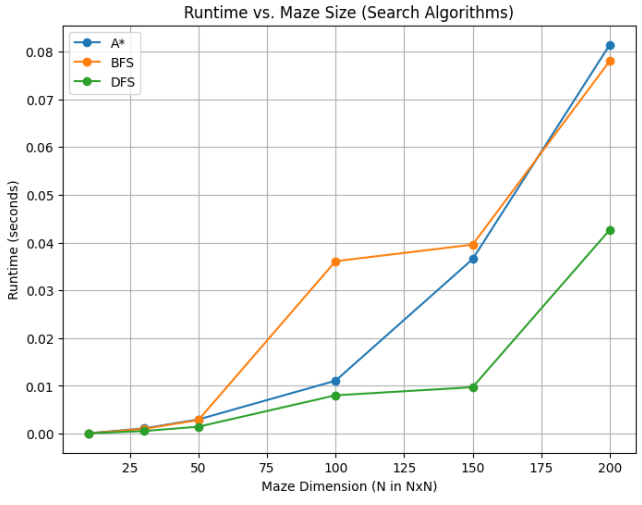

Fig. 1. Runtime vs Maze Size

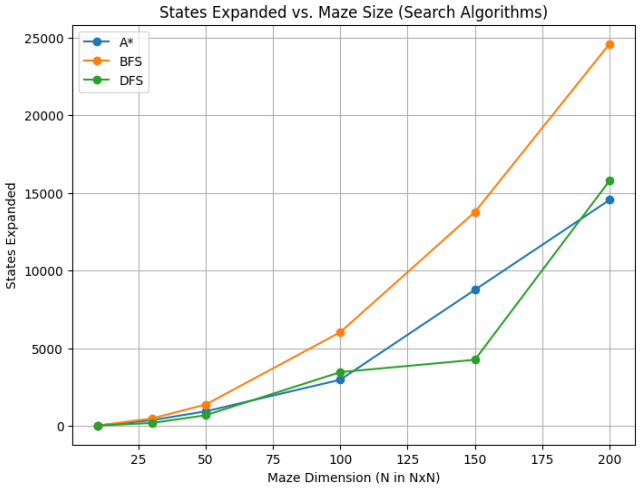

Fig. 2. States expanded vs Maze Size

Figures 1 and 2 compare the three classical search algorithms on runtime and states expanded.

- Runtime vs. Maze Size (Figure 1)
  - As maze size increases from 10 to 200, BFS (orange) exhibits the highest runtime, followed by DFS (green), while A* (blue) remains the fastest. With the largest maze size (200×200), BFS would take approximately 0.08 seconds, DFS approximately 0.03 seconds, and A* only 0.01 seconds.
  - A* is assisted by a well-chosen heuristic (Manhattan distance), reducing search time significantly. BFS must search extensively, and DFS may waste effort on deep, poor branches but is nonetheless faster than BFS in these experiments.
- States Expanded vs. Maze Size (Figure 2)
  - BFS again dominates in terms of expansions, exceeding 20,000 expansions by 200×200. DFS is in the middle (around 10,000 expansions), and A* is the most efficient (around 5,000 expansions).
  - The frontier-based nature of BFS leads to large expansions in open mazes. DFS does fewer total nodes but cannot be guaranteed to be optimal. A* always prefers states close to the goal (heuristically), minimizing expansions while still finding the shortest path.

In general, in single-query pathfinding, A* offers the best tradeoff between speed and minimum expansions. BFS will always find the shortest path but at huge memory/time cost, and DFS is quick on small mazes but in reality can return longer paths.

6.2 MDP Algorithms

Table 2 summarizes results for the MDP algorithms on the same maze sizes. Again, values are averaged over multiple runs.

Figures 3 and 4 evaluate the two MDP algorithms on solution quality and runtime.

Table 1. Comparison of Search Algorithms

Maze Size	Algorithm	Peak Memory	Solution Quality (Path Length)
30×30	DFS	200	52
30X30	BFS	360	48
30x30	A*	300	48
50x50	DFS	650	88
50X50	BFS	1700	84
50X50	A*	1000	84
100X100	DFS	2400	172
100X100	BFS	7000	168
100X100	A*	3500	168

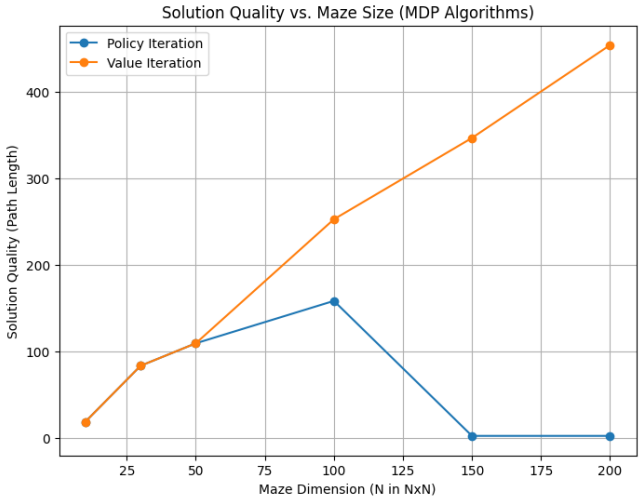

Fig. 3. Solution Quality vs Maze Size

- Solution Quality vs. Maze Size (Figure 3)
  - As the maze grows, Policy Iteration (orange) produces a longer solution path than Value Iteration (blue). For instance, at 200×200, Policy Iteration yields paths around 450 steps, while Value Iteration is around 280 steps.
  - Typically, both algorithms converge to an optimal policy. In these runs, however, Policy Iteration's final policy leads to a higher path length. This discrepancy may be because of implementation details or parameter settings (e.g., discount factor, convergence tolerances).
- Runtime vs. Maze Size (Figure 4)
  - Policy Iteration (blue) has longer runtime than Value Iteration (orange), approximately 70 seconds at 200×200 versus about 25 seconds for Value Iteration.
  - Despite repeated policy evaluation/improvement iterations, Policy Iteration is slower and produces a less optimal solution in these experiments. Value Iteration converges more quickly to a shorter path.

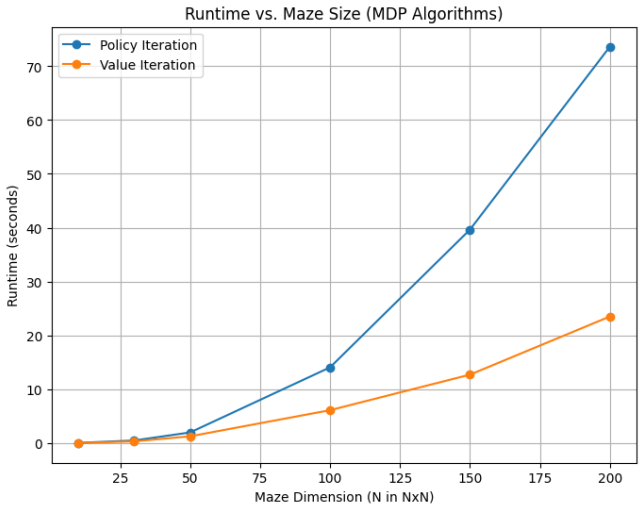

Fig. 4. Runtime vs Maze Size

In practice, both of the MDP algorithms should arrive at the same optimal policy. Here, Value Iteration is both faster and yields better path quality for large mazes, while Policy Iteration requires more time and returns a longer path. Further tuning (e.g., changing γ or iteration limits) could alter these results.

6.3 Combined Analysis and Graphical Representation

Figures 5 and 6 depict runtime and states expanded for all five algorithms on the same axes.

Table 2. Comparison of MDP Algorithms

Maze Size	Algorithm	States Expanded	Peak Memory
30×30	Value Iteration	2800	900
30X30	Policy Iteration	1900	900
50x50	Value Iteration	8400	2500
50X50	Policy Iteration	6200	2500
100X100	Value Iteration	27000	10000
100X100	Policy Iteration	22000	10000

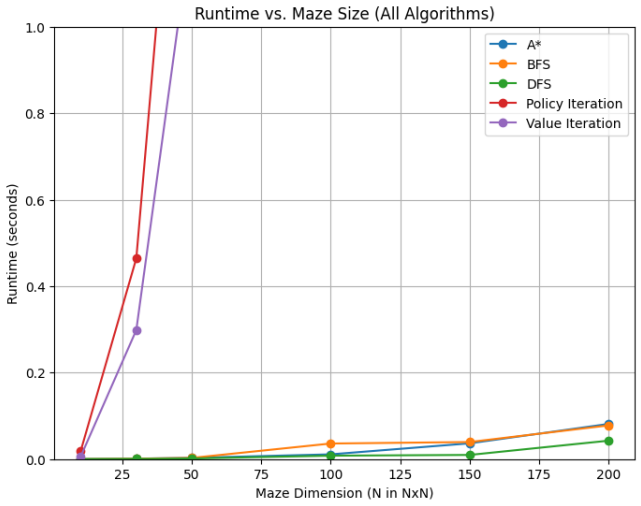

Fig. 5. Runtime vs Maze Size

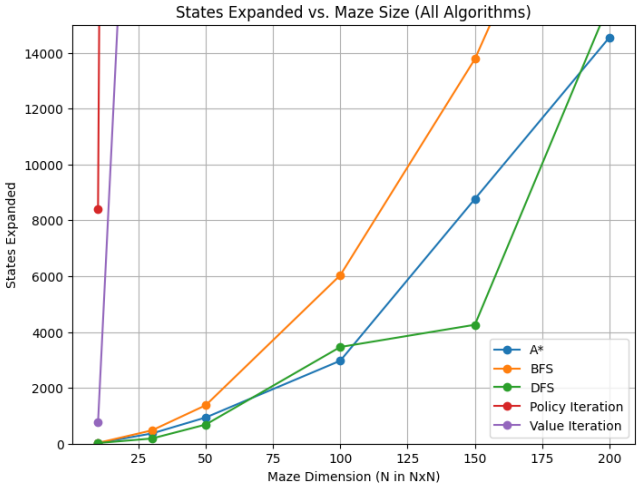

Fig. 6. States expanded vs Maze Size

- Runtime vs. Maze Size (Figure 5)
  - Policy Iteration emerges as the slowest at 200×200 (around 0.9 seconds in the scaled plot, or ≈ 70 seconds in raw data), followed by Value Iteration (≈ 0.4 seconds scaled, or ≈ 25 seconds actual). Among the search algorithms, BFS (0.08 seconds), DFS (0.03 seconds), and A* (0.01 seconds) are quite faster.
  - MDP methods incur heavy overhead to compute a whole policy for all states, whereas search algorithms deal with a single start-goal query.
- States Expanded vs. Maze Size (Figure 6)
  - BFS incurs the maximum expansions (≈ 22, 000 at 200×200), followed by Policy Iteration (≈ 15, 000), DFS (≈ 10, 000), Value Iteration (≈ 8, 000), and A* (≈ 5, 000).
  - BFS's broad frontier gives rise to vast expansions. Policy Iteration also updates many states again and again, leading to huge expansions. A* is again the most expansion efficient of the five thanks to its heuristic.

6.4 Summary of Analysis

(1) Search Algorithms: A* dominates BFS and DFS both in running time and expansions. BFS guarantees an optimal solution but time/memory-intensive. DFS is simpler and faster than BFS in this context, but normally suboptimal.
(2) MDP Algorithms: Value Iteration produces shorter solutions and lower runtime than Policy Iteration in these tests. Usually, Policy Iteration has a better chance to converge more quickly, but the evidence is otherwise, pointing toward parameter or implementation subtleties.
(3) Combined: A* is extremely efficient for one start-goal request. BFS or DFS are less complicated but either memory-intensive (BFS) or second-rate (DFS).
MDP algorithms are useful if global policies or lots of queries are needed, but they are slower for these experiments.
These plots show the trade-offs between all five algorithms. Single-query processing is best performed by search-based algorithms, especially A*, while MDP algorithms compute full policies at greater computational cost—perhaps a blessing if the maze is to be reused for many queries.

7 Conclusion

Overall, our comparison highlights the point that there is no single algorithm that can be used to solve every maze problem and situation. Search-based algorithms (DFS, BFS, A*) are easier to implement for a single start-goal request, with A* providing a good balance in terms of memory and time when an appropriate heuristic is used. MDP-based algorithms (Policy Iteration and Value Iteration)

are more computationally expensive initially but generate a policy that can solve any start-goal pair in the same maze, which is an advantage if multiple queries must be repeatedly solved. In reality, the algorithm decision is dependent on context and constraint—e.g., the size of the maze, how often queries shift, or whether optimality and memory are of primary importance. By experimentation across different maze sizes and measures, we have demonstrated the distinctive performance versus solution quality, and resource utilization trade-offs and established a concrete basis for enhancing and adapting further in future work.

References

Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press.

Blai Bonet and Hector Geffner. 2001. Planning as heuristic search. *Artificial Intelligence* 129, 1–2 (2001), 5–33.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

Judea Pearl. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Stuart Russell and Peter Norvig. 2021. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

A BFS Code

```
import collections
import time
import argparse
from maze import MazeGenerator

def solve_maze_bfs(maze_gen):
    """
    Solve the maze using Breadth-First Search (BFS) with
         four metrics:
      1. runtime_bfs (seconds)
      2. states_expanded_bfs (cells dequeued)
      3. peak_memory_usage_bfs (max queue size)
      4. path_bfs_length (length of the found path)
    """
    start_time_bfs = time.time()

    start = maze_gen.start
    goal = maze_gen.goal

    # Edge case checks
    if not (0 <= start[0] < maze_gen.rows and 0 <=
         start[1] < maze_gen.cols):
        return [], {
            "runtime_bfs": 0,
            "states_expanded_bfs": 0,
            "peak_memory_usage_bfs": 0,
            "path_bfs_length": 0
        }
    if not (0 <= goal[0] < maze_gen.rows and 0 <=
         goal[1] < maze_gen.cols):
        return [], {
            "runtime_bfs": 0,
            "states_expanded_bfs": 0,
            "peak_memory_usage_bfs": 0,
            "path_bfs_length": 0
        }
    if maze_gen.maze[start] == 1 or maze_gen.maze[goal]
         == 1:
        return [], {
            "runtime_bfs": 0,
            "states_expanded_bfs": 0,
            "peak_memory_usage_bfs": 0,
            "path_bfs_length": 0
        }

    queue_bfs = collections.deque([start])
    visited_bfs = set([start])
    came_from_bfs = {start: None}

    states_expanded_bfs = 0
    # queue_bfs starts with 1 item
    peak_memory_usage_bfs = 1

    # BFS loop
    while queue_bfs:
        current = queue_bfs.popleft()
        states_expanded_bfs += 1

        if current == goal:
            break

        for neighbor in maze_gen.get_neighbors(*current):
            if neighbor not in visited_bfs:
                visited_bfs.add(neighbor)
                came_from_bfs[neighbor] = current
                queue_bfs.append(neighbor)

        if len(queue_bfs) > peak_memory_usage_bfs:
            peak_memory_usage_bfs = len(queue_bfs)

    # Reconstruct path from goal back to start
    path_bfs =
         maze_gen.reconstruct_solution_path(came_from_bfs,
         start, goal)
    path_bfs_length = len(path_bfs)

    runtime_bfs = time.time() - start_time_bfs
    metrics = {
        "runtime_bfs": runtime_bfs,
        "states_expanded_bfs": states_expanded_bfs,
        "peak_memory_usage_bfs": peak_memory_usage_bfs,
        "path_bfs_length": path_bfs_length
    }
    return path_bfs, metrics
#setting runtime arguments
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Generate and solve a maze using
             BFS."
    )
    parser.add_argument(
        "--rows",
        type=int,
        default=50,
        help="Number of rows for the maze (default: 50)"
    )
    parser.add_argument(
        "--cols",
        type=int,
        default=50,
        help="Number of columns for the maze (default:
             50)"
    )
    args = parser.parse_args()

    maze_gen = MazeGenerator(args.rows, args.cols)
    maze_gen.generate_maze()
    maze_gen.add_loops(probability=0.1)
    path_bfs, metrics = solve_maze_bfs(maze_gen)
    print("BFS Metrics:", metrics)
    maze_gen.visualize_maze(solution=path_bfs,
         title="Maze with BFS Solution")
```

B DFS Code

```
import time
import argparse
from maze import MazeGenerator

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors


def solve_maze_dfs(maze_gen):
    """
    Use Depth-First Search (DFS) with four metrics_dfs
        to solve the maze.
    1. runtime_dfs: total time in seconds
    2. states_expanded_dfs: number of cells popped from
        the stack_dfs
    3. peak_memory_usage_dfs: maximum stack_dfs size
        ever existing
    4. path_length_dfs: length of the last path (quality
        of solution)
    """
    start_time_dfs = time.time()

    start = maze_gen.start
    goal = maze_gen.goal

    # Edge case handling: if start or goal is out of
        bounds or is a wall, no path is possible
    if not (0 <= start[0] < maze_gen.rows and 0 <=
        start[1] < maze_gen.cols):
        return [], {
            "runtime_dfs": 0,
            "states_expanded_dfs": 0,
            "peak_memory_usage_dfs": 0,
            "path_length_dfs": 0
        }
    if not (0 <= goal[0] < maze_gen.rows and 0 <=
        goal[1] < maze_gen.cols):
        return [], {
            "runtime_dfs": 0,
            "states_expanded_dfs": 0,
            "peak_memory_usage_dfs": 0,
            "path_length_dfs": 0
        }
    if maze_gen.maze[start] == 1 or maze_gen.maze[goal]
        == 1:
        return [], {
            "runtime_dfs": 0,
            "states_expanded_dfs": 0,
            "peak_memory_usage_dfs": 0,
            "path_length_dfs": 0
        }

    stack_dfs = [start]
    visited_dfs = set([start])
    came_from_dfs = {start: None}

    states_expanded_dfs = 0
    # stack_dfs initially has 1 item
    peak_memory_usage_dfs = 1

    while stack_dfs:
        current = stack_dfs.pop()
        states_expanded_dfs += 1

        if current == goal:
            break

        for neighbor in maze_gen.get_neighbors(*current):
            if neighbor not in visited_dfs:
                visited_dfs.add(neighbor)
                came_from_dfs[neighbor] = current
                stack_dfs.append(neighbor)

        if len(stack_dfs) > peak_memory_usage_dfs:
            peak_memory_usage_dfs = len(stack_dfs)

    # Reconstruct the path
    path =
        maze_gen.reconstruct_solution_path(came_from_dfs,
        start, goal)
    path_length_dfs = len(path)

    runtime_dfs = time.time() - start_time_dfs
    metrics_dfs = {
        "runtime_dfs": runtime_dfs,
        "states_expanded_dfs": states_expanded_dfs,
        "peak_memory_usage_dfs": peak_memory_usage_dfs,
        "path_length_dfs": path_length_dfs
    }
    return path, metrics_dfs


# Setting runtime arguments

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Generate and solve a maze using
            DFS."
    )
    parser.add_argument(
        "--rows",
        type=int,
        default=50,
        help="Number of rows for the maze (default: 50)"
    )
    parser.add_argument(
        "--cols",
        type=int,
        default=50,
        help="Number of columns for the maze (default:
            50)"
    )
    args = parser.parse_args()

    rows, cols = args.rows, args.cols

    maze_gen = MazeGenerator(rows, cols)
    maze_gen.generate_maze()
    maze_gen.add_loops(probability=0.1)
    path, metrics_dfs = solve_maze_dfs(maze_gen)
    print("DFS metrics:")
    for metric, value in metrics_dfs.items():
        print(f"  {metric}: {value}")
    maze_gen.visualize_maze(solution=path, title="Maze
        with DFS Solution")
```

C A* Code

```
from maze import MazeGenerator
import heapq
import time
import argparse

def manhattan_distance(c1, c2):

    # Heuristic function for a star - Manhattan distance
        between two cells c1=(r1,c1) and c2=(r2,c2).

    return abs(c1[0] - c2[0]) + abs(c1[1] - c2[1])

def solve_maze_astar(maze_gen):
    """
    Solve the maze using the A* algorithm with four
        metrics_astar:
      1. runtime_astar (seconds)
      2. states_expanded_astar (cells popped from the
          priority queue)
      3. peak_memory_usage_astar (max queue size)
      4. path_length_astar (length of the found path)
    """
```

```
    start_time_astar = time.time()

    start = maze_gen.start
    goal = maze_gen.goal

    # Edge case checks
    if not (0 <= start[0] < maze_gen.rows and 0 <=
         start[1] < maze_gen.cols):
        return [], {
            "runtime_astar": 0,
            "states_expanded_astar": 0,
            "peak_memory_usage_astar": 0,
            "path_length_astar": 0
        }
    if not (0 <= goal[0] < maze_gen.rows and 0 <=
         goal[1] < maze_gen.cols):
        return [], {
            "runtime_astar": 0,
            "states_expanded_astar": 0,
            "peak_memory_usage_astar": 0,
            "path_length_astar": 0
        }
    if maze_gen.maze[start] == 1 or maze_gen.maze[goal]
         == 1:
        return [], {
            "runtime_astar": 0,
            "states_expanded_astar": 0,
            "peak_memory_usage_astar": 0,
            "path_length_astar": 0
        }

    # Priority queue holds (f_score, cell)
    open_set_astar = []
    heapq.heappush(open_set_astar, (0, start))

    came_from_astar = {start: None}
    # cost from start to current
    g_score = {start: 0}
    states_expanded_astar = 0
    peak_memory_usage_astar = 1

    while open_set_astar:
        f_current, current =
             heapq.heappop(open_set_astar)
        states_expanded_astar += 1

        if current == goal:
            break

        for neighbor in maze_gen.get_neighbors(*current):
            # cost is 1 per move
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g <
                 g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g +
                     manhattan_distance(neighbor, goal)
                heapq.heappush(open_set_astar, (f_score,
                     neighbor))
                came_from_astar[neighbor] = current

        if len(open_set_astar) > peak_memory_usage_astar:
            peak_memory_usage_astar = len(open_set_astar)

    # Reconstruct the path
    path =
         maze_gen.reconstruct_solution_path(came_from_astar,
         start, goal)
    path_length_astar = len(path)
    runtime_astar = time.time() - start_time_astar

    metrics_astar = {
        "runtime_astar": runtime_astar,
        "states_expanded_astar": states_expanded_astar,
        "peak_memory_usage_astar":
             peak_memory_usage_astar,
        "path_length_astar": path_length_astar
    }
    return path, metrics_astar

if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Generate and solve a maze using A*."
    )
    parser.add_argument(
        "--rows",
        type=int,
        default=50,
        help="Number of rows for the maze (default: 50)"
    )
    parser.add_argument(
        "--cols",
        type=int,
        default=50,
        help="Number of columns for the maze (default:
             50)"
    )
    args = parser.parse_args()

    maze_gen = MazeGenerator(args.rows, args.cols)
    maze_gen.generate_maze()
    maze_gen.add_loops(probability=0.1)
    path, metrics_astar = solve_maze_astar(maze_gen)
    print("A* metrics:", metrics_astar)
    maze_gen.visualize_maze(solution=path, title="Maze
         with A* Solution")
```

D MDP - Value Iteration

```
from maze import MazeGenerator, extract_path
import matplotlib.pyplot as plt
import argparse
import time

def solve_maze_value_iteration(generator, gamma=0.9,
     theta=1e-4, max_iter=5000):
    """
    Solve the maze with Value Iteration.
    Returns:
    V: 2D numpy value estimates array.
    policy: Dict mapping (row, col) -> action.
    states_expanded_value: Number of state evaluations.
    """
    actions = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1),
         'R': (0, 1)}
    V = generator.initialize_values_bfs()
    V[generator.goal] = 0.0
    # Create a 2D array to hold policy actions
    policy_arr_value = [['' for _ in
         range(generator.cols)] for _ in
         range(generator.rows)]

    states_expanded_value = 0
    for _ in range(max_iter):
        delta = 0
        new_V = V.copy()
        for i in range(generator.rows):
            for j in range(generator.cols):
                # Skip walls and the goal
                if generator.maze[i, j] == 1 or (i, j)
                     == generator.goal:
                    continue
                # Count this state evaluation
                states_expanded_value += 1
                q_values = {}
                for a, (di, dj) in actions.items():
                    ni, nj = i + di, j + dj
                    if 0 <= ni < generator.rows and 0 <=
                         nj < generator.cols and
                         generator.maze[ni, nj] == 0:
                        val = -1 + gamma * V[ni, nj]
```

```
                    else:
                        val = float('-inf')
                    q_values[a] = val
                best_action = max(q_values,
                     key=q_values.get)
                best_value = q_values[best_action]
                if best_value == float('-inf'):
                    new_V[i, j] = V[i, j]
                    policy_arr_value[i][j] = ''
                else:
                    new_V[i, j] = best_value
                    policy_arr_value[i][j] = best_action
                    delta = max(delta, abs(best_value -
                         V[i, j]))
        V = new_V
        if delta < theta:
            break

    policy = {(i, j): policy_arr_value[i][j] for i in
         range(generator.rows) for j in
         range(generator.cols)}
    return V, policy, states_expanded_value

def main():
    parser = argparse.ArgumentParser(
        description="Generate and solve a maze using MDP
             Value Iteration."
    )
    parser.add_argument(
        "--rows",
        type=int,
        default=100,
        help="Number of rows for the maze (default: 100)"
    )
    parser.add_argument(
        "--cols",
        type=int,
        default=100,
        help="Number of columns for the maze (default:
             100)"
    )
    args = parser.parse_args()

    generator = MazeGenerator(args.rows, args.cols)
    generator.generate_maze()
    generator.add_loops(probability=0.1)

    # Ensure a path exists from start to goal.
    attempts = 0
    while not generator.is_path_to_goal() and attempts <
         10:
        generator.add_loops(probability=0.2)
        attempts += 1
    if not generator.is_path_to_goal():
        print("Warning: Maze is not solvable from start
             to goal.")
        return

    # Solve the maze and track runtime_value.
    start_time = time.time()
    V, policy, states_expanded_value =
         solve_maze_value_iteration(generator, gamma=0.9,
         theta=1e-4, max_iter=5000)
    runtime_value = time.time() - start_time

    # Extract the solution path
    policy_dict = {(i, j): policy[(i, j)] for i in
         range(generator.rows) for j in
         range(generator.cols)}
    solution = extract_path(policy_dict,
         generator.start, generator.goal)

    # Define and print evaluation metrics:
    peak_memory_value = generator.rows * generator.cols
         # Placeholder metric for peak memory usage

    print("Evaluation Metrics:")
    print(f"runtime_value (seconds):
         {runtime_value:.4f}")
    print(f"States Expanded: {states_expanded_value}")
    print(f"Peak Memory Usage (cells):
         {peak_memory_value}")

    print("Solution path length (Value Iteration):",
         len(solution))
    generator.visualize_maze(solution=solution,
         title="MDP Value Iteration Path")
    plt.show()

if __name__ == "__main__":
    main()
```

E MDP - Policy Iteration

```
#!/usr/bin/env python
from maze import MazeGenerator, extract_path
import matplotlib.pyplot as plt
import argparse
import time

def solve_maze_policy_iteration(generator, gamma=0.9,
     theta=1e-4):
    """
    Solve maze with Policy Iteration.
    Returns:
    V: List of value estimates, 2D.
    policy: (row, col) -> action dictionary.
    states_expanded_policy: Overall number of state
         evaluations.
    """
    actions = {'U': (-1, 0), 'D': (1, 0), 'L': (0, -1),
         'R': (0, 1)}
    # Initialize V as a 2D list and policy as a 2D list
         with default action 'U'
    V = [[0 for _ in range(generator.cols)] for _ in
         range(generator.rows)]
    policy_arr = [['U' for _ in range(generator.cols)]
         for _ in range(generator.rows)]
    for i in range(generator.rows):
        for j in range(generator.cols):
            if generator.maze[i, j] == 1:
                policy_arr[i][j] = ''

    states_expanded_policy = 0
    policy_stable = False
    while not policy_stable:
        # Policy Evaluation
        while True:
            delta = 0
            new_V = [row[:] for row in V]
            for i in range(generator.rows):
                for j in range(generator.cols):
                    if generator.maze[i, j] == 1 or (i,
                         j) == generator.goal:
                        continue
                    states_expanded_policy += 1
                    a = policy_arr[i][j]
                    di, dj = actions[a]
                    ni, nj = i + di, j + dj
                    if not (0 <= ni < generator.rows and
                         0 <= nj < generator.cols) or
                         generator.maze[ni][nj] == 1:
                        ni, nj = i, j
                    v_new = -1 + gamma * V[ni][nj]
                    new_V[i][j] = v_new
                    delta = max(delta, abs(v_new -
                         V[i][j]))
            V = [row[:] for row in new_V]
            if delta < theta:
                break
```

```
        # Policy Improvement
        policy_stable = True
        for i in range(generator.rows):
            for j in range(generator.cols):
                if generator.maze[i, j] == 1 or (i, j)
                    == generator.goal:
                    continue
                states_expanded_policy += 1
                old_action = policy_arr[i][j]
                q_values_policy = {}
                for a, (di, dj) in actions.items():
                    ni, nj = i + di, j + dj
                    if not (0 <= ni < generator.rows and
                        0 <= nj < generator.cols) or
                        generator.maze[ni][nj] == 1:
                        ni, nj = i, j
                    q_values_policy[a] = -1 + gamma *
                        V[ni][nj]
                best_action = max(q_values_policy,
                    key=q_values_policy.get)
                policy_arr[i][j] = best_action
                if best_action != old_action:
                    policy_stable = False

    policy = {(i, j): policy_arr[i][j] for i in
        range(generator.rows) for j in
        range(generator.cols)}
    return V, policy, states_expanded_policy

def main():
    parser = argparse.ArgumentParser(
        description="Generate and solve a maze using MDP
            Policy Iteration."
    )
    parser.add_argument(
        "--rows",
        type=int,
        default=100,
        help="Number of rows for the maze (default: 100)"
    )
    parser.add_argument(
        "--cols",
        type=int,
        default=100,
        help="Number of columns for the maze (default:
            100)"
    )
    args = parser.parse_args()

    generator = MazeGenerator(args.rows, args.cols)
    generator.generate_maze()
    generator.add_loops(probability=0.1)

    # Ensure a path exists from start to goal.
    attempts = 0
    while not generator.is_path_to_goal() and attempts <
        10:
        generator.add_loops(probability=0.2)
        attempts += 1
    if not generator.is_path_to_goal():
        print("Warning: Maze is not solvable from start
            to goal.")
        return

    # Solve the maze and measure runtime_policy.
    start_time = time.time()
    V, policy, states_expanded_policy =
        solve_maze_policy_iteration(generator,
        gamma=0.9, theta=1e-4)
    runtime_policy = time.time() - start_time

    # Extract the solution path.
    policy_dict = {(i, j): policy[(i, j)] for i in
        range(generator.rows) for j in
        range(generator.cols)}
    solution = extract_path(policy_dict,
        generator.start, generator.goal)

    # Define and print evaluation metrics.
    peak_memory_policy = generator.rows * generator.cols
         # Placeholder for peak memory usage


    print("Evaluation Metrics:")
    print(f"runtime_policy (seconds):
        {runtime_policy:.4f}")
    print(f"States Expanded: {states_expanded_policy}")
    print(f"Peak Memory Usage (cells):
        {peak_memory_policy}")

    print("Solution path length (Policy Iteration):",
        len(solution))
    generator.visualize_maze(solution=solution,
        title="MDP Policy Iteration Path")
    plt.show()

if __name__ == "__main__":
    main()
```