CS7IS2: AI Assignment 1

Rokas Paulauskas

March 2, 2025

Abstract

The five maze-solving algorithms—Breadth-First Search (BFS), Depth-First Search (DFS), A* Search, MDP Value Iteration, and MDP Policy Iteration are examined and their performances compared in this assignment. Such as execution time, path length, states investigated, and memory usage, are used to compare algorithms in mazes of different sizes. In addition to presenting the individual algorithm implementations, this report discusses the trade-offs among different search methods, contrasts MDP approaches, and finally compares search-based methods to MDP-based approaches.

1 Introduction

Maze solving is a fundamental problem in artificial intelligence and robotics. In this assignment, five different algorithms are implemented and evaluated:

- **BFS** and **DFS** – classical search algorithms.
- **A* Search** – an informed search algorithm that uses a heuristic (Manhattan distance).
- **MDP Value Iteration** and **MDP Policy Iteration** – methods that address decision making under uncertainty.

The aim is to compare these methods over a range of maze sizes and analyze their performance using multiple metrics.

2 Algorithm Implementations

2.1 Breadth-First Search (BFS)

BFS explores all nodes at the current depth before moving deeper, ensuring that the shortest path is found. A snippet of the implementation is shown below.

Listing 1: BFS Implementation

```
from collections import deque

def bfs_algorithm(m):
    queue = deque([start])
    visited = set([start])
    path = {}
    explored_states = 0
    while queue:
        cur = queue.popleft()
        explored_states += 1
        if cur == destination:
            break
        for d, (dx, dy) in directions.items():
            nxt = (cur[0] + dx, cur[1] + dy)
            if nxt not in visited:
                visited.add(nxt)
```

```
            queue.append(nxt)
            path[nxt] = cur
    return path, explored_states
```

2.2 Depth-First Search (DFS)

DFS explores as far as possible along a branch before backtracking. It is fast but may not find the optimal path.

Listing 2: DFS Implementation

```
def dfs_algorithm(m):
    stack = deque([start])
    visited = {start}
    path = {}
    explored_states = 0
    while stack:
        cur = stack.pop()
        explored_states += 1
        if cur == destination:
            break
        for d, (dx, dy) in directions.items():
            nxt = (cur[0] + dx, cur[1] + dy)
            if nxt not in visited:
                visited.add(nxt)
                stack.append(nxt)
                path[nxt] = cur
    return path, explored_states
```

2.3 A* Search

A* Search combines BFS with a heuristic to efficiently guide the search towards the goal.

Listing 3: A* Implementation

```
def astar_algorithm(m):
    open_set = []
    heapq.heappush(open_set, (f_score[start], heuristic(start, goal), start))
    path, visited = {}, set()
    explored_states = 0
    while open_set:
        _, _, cur = heapq.heappop(open_set)
        explored_states += 1
        if cur == goal:
            break
        for d, (dx, dy) in directions.items():
            nxt = (cur[0] + dx, cur[1] + dy)
            if nxt not in visited:
                visited.add(nxt)
                heapq.heappush(open_set, (f_score[nxt], heuristic(nxt, goal), nxt))
                path[nxt] = cur
    return path, explored_states
```

2.4 MDP Value Iteration

MDP Value Iteration iteratively updates state values until convergence, then derives the optimal policy.

Listing 4: MDP Value Iteration Implementation

```
def mdp_value_iteration(m):
```

```
    explored_states = 0
    while True:
        delta = 0
        new_V = np.copy(V)
        for i in range(rows):
            for j in range(cols):
                if (i, j) == (0, 0):
                    continue
                q_vals = [rewards[next_i, next_j] + gamma * V[next_i, next_j]
                          for next_i, next_j in neighbors]
                best_value = max(q_vals) if q_vals else V[i, j]
                new_V[i, j] = best_value
                delta = max(delta, abs(V[i, j] - best_value))
                explored_states += 1
        V = new_V
        if delta < theta:
            break
    return path, explored_states
```

2.5 MDP Policy Iteration

MDP Policy Iteration alternates between policy evaluation and policy improvement until convergence.

Listing 5: MDP Policy Iteration Implementation

```
def mdp_policy_iteration(m):
    explored_states = 0
    stable_policy = False
    while not stable_policy:
        while True:
            delta = 0
            new_V = np.copy(V)
            for i in range(rows):
                for j in range(cols):
                    if (i, j) == (0, 0):
                        continue
                    u = policy[i, j]
                    next_i, next_j = i + actions[u][0], j + actions[u][1]
                    if condition_met:
                        value = rewards[next_i, next_j] + gamma * V[next_i, next_j]
                    else:
                        value = V[i, j]
                    new_V[i, j] = value
                    delta = max(delta, abs(V[i, j] - value))
                    explored_states += 1
            V = new_V
            if delta < theta:
                break
        stable_policy = True
        for i in range(rows):
            for j in range(cols):
                if (i, j) == (0, 0):
                    continue
                q_vals = []
                for u, d in enumerate(directions):
                    next_i, next_j = i + actions[u][0], j + actions[u][1]
                    if condition_met:
                        q_vals.append((rewards[next_i, next_j] + gamma * V[next_i, next_
                if q_vals:
```

```
                best_value, best_action = max(q_vals)
                if best_action != policy[i, j]:
                    stable_policy = False
                    policy[i, j] = best_action
    return path, explored_states
```

3 Experimental Setup

The experiments were done on different mazes: 10×10, 20×20, 30×30, 50×50, and 100×100. For each maze, the following were measured:

- **Execution Time (s)**: Total time taken to find solution.
- **Path Length**: Number of steps for solution.
- **States Explored**: How many steps were checked for solution.
- **Memory Usage (KB)**: Memory consumption during execution.

The maze was generated with a 40% loop percentage to ensure sufficient complexity while maintaining feasibility in execution time.

4 Results and Analysis

Figures below show the performance trends for each algorithm over different maze sizes.

4.1 Execution Time

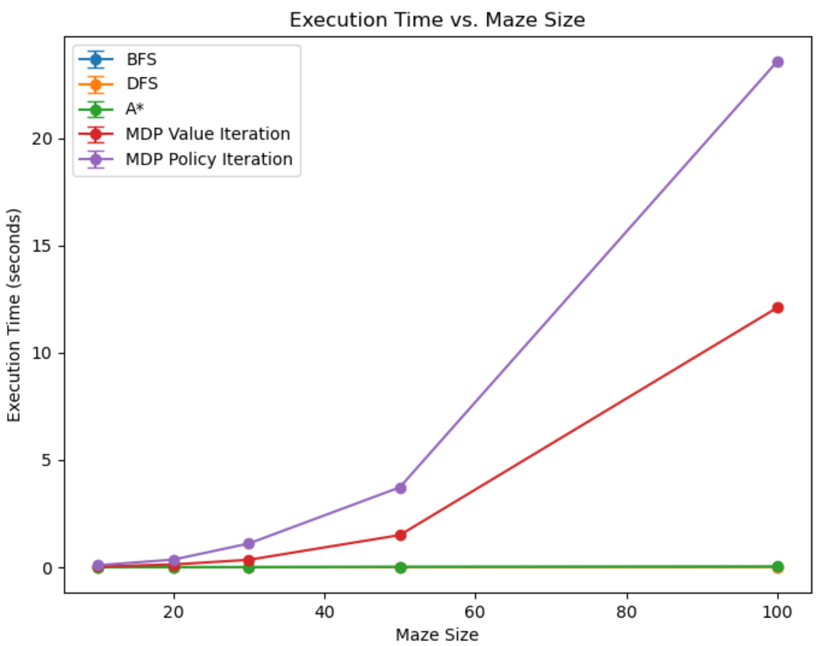

Figure 1: Execution Time vs Maze Size

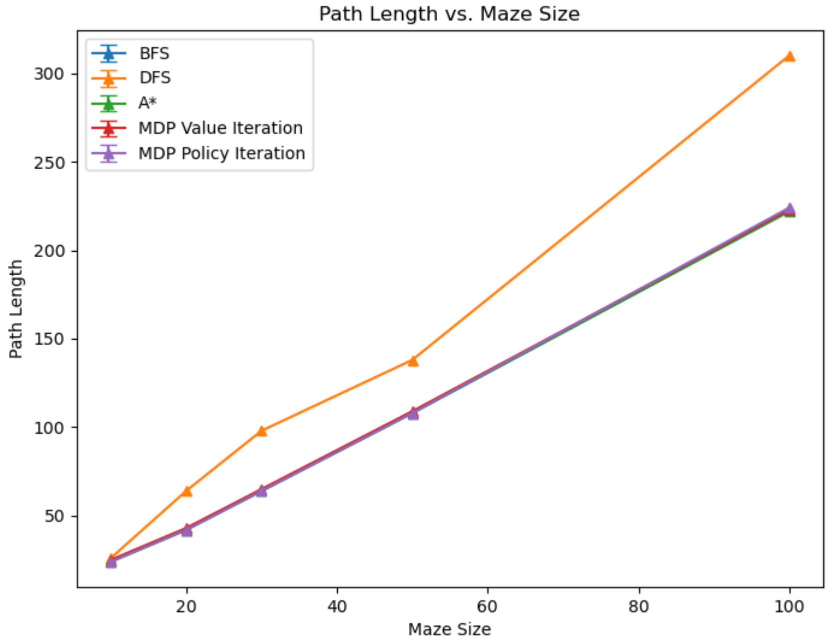

Figure 2: Path Length vs Maze Size

4.2 Path Length

4.3 States Explored

4.4 Summary Statistics

Table 1 shows the aggregated performance metrics.

Table 1: Summary Statistics of Algorithm Performance

Algorithm	Time (s) Mean	Time (s) Std	States Explored Mean	States Explored Std	Path Length
A*	0.0125	0.0173	1608.2	2573.3	94.4
BFS	0.0039	0.0060	2776.8	4141.7	94.4
DFS	0.0016	0.0012	217.8	198.3	134.4
MDP Policy Iteration	5.6615	9.7971	1647088.0	2769879.1	94.4
MDP Value Iteration	2.7214	4.9994	525545.2	976236.2	95.4

5 Comparative Analysis

5.1 Search Algorithms (BFS, DFS, A*)

Among the search algorithms:

- **BFS** guarantees the shortest path but explores many states.
- **DFS** is the fastest in terms of execution time but often results in suboptimal paths.
- **A*** offers a good balance between exploration and optimality due to the use of a heuristic.

The experimental results indicate that while DFS is fastest, its path quality suffers compared to BFS and A*.

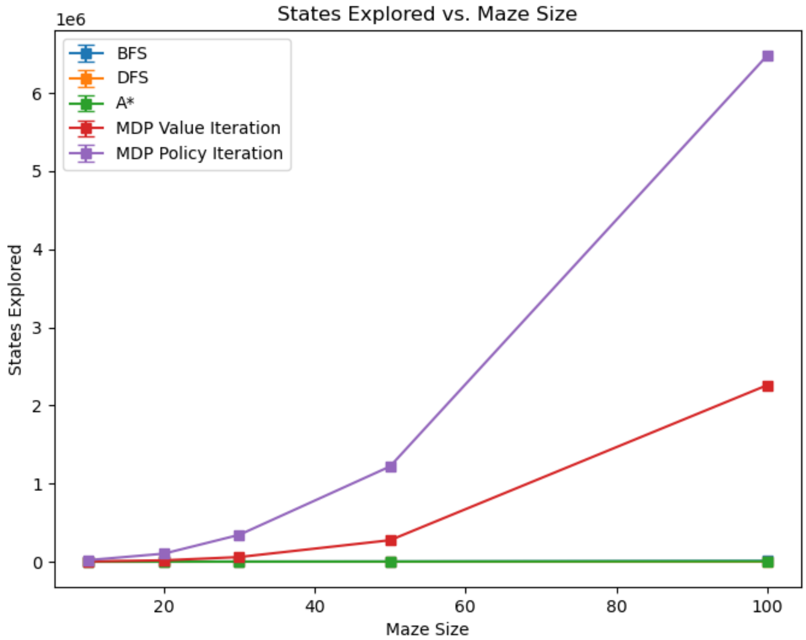

Figure 3: States Explored vs Maze Size

5.2 MDP Algorithms (Value vs Policy Iteration)

The two MDP methods differ significantly:

- **MDP Value Iteration** converges faster in some cases, but its performance is sensitive to the chosen parameters (e.g., gamma, theta).
- **MDP Policy Iteration** generally achieves a stable policy, albeit with a higher computational cost.

Both methods are computationally more expensive compared to search algorithms, but they provide a robust framework in uncertain environments.

5.3 Search vs MDP Algorithms

When comparing search-based methods to MDP-based methods:

- **Search Algorithms** are more efficient in deterministic maze environments.
- **MDP Methods** have advantage in scenarios where the environment has uncertainty or varying rewards, despite their higher computational demands.

The advantages and disadvantages have to be chosen between execution time and robustness.

6 Discussion and Design Choices

Key design choices include:

- **Heuristic for A***: The Manhattan distance was chosen for its simplicity and effectiveness in grid-based mazes.
- **MDP Parameters**: The values of gamma and theta were selected based on preliminary experiments to balance convergence speed and solution quality.
- **Maze Generator**: A loop percentage of 40% was used to ensure a reasonable challenge without making the maze unsolvable.

These choices are confirmed by the experimental outcomes, which show clear performance trends that align with theoretical expectations.

7 Conclusion

This report has presented an in-depth analysis of five maze solving algorithms. While search algorithms (BFS, DFS, A*, etc.) are well-suited for deterministic settings, MDP methods offer a viable alternative when dealing with uncertain or dynamic environments. The comprehensive performance analysis using multiple metrics demonstrates the trade-offs between execution speed, optimality, and computational resources.

Appendices

Appendix A: BFS Implementation

```
import sys
import time
from collections import deque
from pyamaze import maze, agent, textLabel, COLOR


def bfs_algorithm(m):
    start_time = time.time()
    directions = {'E': (0, 1), 'S': (1, 0), 'W': (0, -1), 'N': (-1, 0)}
    destination = (1, 1)
    start = (m.rows, m.cols)

    queue = deque([start])
    visited = set([start])
    path = {}
    explored_states = 0

    while queue:
        current = queue.popleft()
        explored_states += 1
        if current == destination:
            break
        for d, (dx, dy) in directions.items():
            if m.maze_map[current][d]:
                nxt = (current[0] + dx, current[1] + dy)
                if nxt not in visited:
                    visited.add(nxt)
                    queue.append(nxt)
                    path[nxt] = current

    bfs_path = {}
    current = destination
    while current in path:
        bfs_path[path[current]] = current
        current = path[current]

    return bfs_path, round(time.time() - start_time, 6), explored_states


if __name__ == '__main__':
    if len(sys.argv) == 3:
        try:
```

```python
            rows = int(sys.argv[1])
            cols = int(sys.argv[2])
        except ValueError:
            print("Invalid input. Provide integers for rows and columns.")
            sys.exit(1)
    else:
        rows, cols = 30, 30

    print(f"Generating {rows}×{cols} maze...")
    m = maze(rows, cols)
    m.CreateMaze(loopPercent=40, theme=COLOR.light)

    path, exec_time, explored = bfs_algorithm(m)
    print(f"Path found: {path}")
    print(f"Execution time: {exec_time} seconds")
    print(f"States explored: {explored}")

    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
    textLabel(m, 'Path Length', len(path) + 1)
    textLabel(m, 'Execution Time', exec_time)
    textLabel(m, 'States Explored', explored)
    m.run()
```

Appendix B: DFS Implementation

```python
import sys
import time
from collections import deque
from pyamaze import maze, agent, textLabel, COLOR


def dfs_algorithm(m):
    start_time = time.time()
    directions = {'E': (0, 1), 'S': (1, 0), 'W': (0, -1), 'N': (-1, 0)}
    destination, start = (1, 1), (m.rows, m.cols)
    stack = deque([start])
    visited = {start}
    path = {}
    explored_states = 0

    while stack:
        current = stack.pop()
        explored_states += 1
        if current == destination:
            break
        for d, (dx, dy) in directions.items():
            if m.maze_map[current][d]:
                nxt = (current[0] + dx, current[1] + dy)
                if nxt not in visited:
                    visited.add(nxt)
                    stack.append(nxt)
                    path[nxt] = current

    dfs_path = {}
    current = destination
    while current in path:
        dfs_path[path[current]] = current
        current = path[current]
```

```python
    return dfs_path, round(time.time() - start_time, 6), explored_states


if __name__ == '__main__':
    if len(sys.argv) == 3:
        try:
            rows, cols = int(sys.argv[1]), int(sys.argv[2])
        except ValueError:
            print("Invalid input. Please provide integers for maze dimensions.")
            sys.exit(1)
    else:
        rows, cols = 30, 30

    print(f"Generating {rows}×{cols} maze...")
    m = maze(rows, cols)
    m.CreateMaze(loopPercent=40, theme=COLOR.light)

    path, exec_time, explored = dfs_algorithm(m)
    print(f"Path found: {path}")
    print(f"Execution time: {exec_time} seconds")
    print(f"States explored: {explored}")

    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
    textLabel(m, 'Path Length', len(path) + 1)
    textLabel(m, 'Execution Time', exec_time)
    textLabel(m, 'States Explored', explored)
    m.run()
```

Appendix C: A* Implementation

```python
import sys
import time
import heapq
from collections import defaultdict
from pyamaze import maze, agent, textLabel, COLOR


def heuristic(cur, goal):
    return abs(cur[0] - goal[0]) + abs(cur[1] - goal[1])


def astar_algorithm(m):
    start_time = time.time()
    directions = {'E': (0, 1), 'S': (1, 0), 'W': (0, -1), 'N': (-1, 0)}
    start, goal = (m.rows, m.cols), (1, 1)

    g_score = defaultdict(lambda: float('inf'))
    f_score = defaultdict(lambda: float('inf'))
    g_score[start], f_score[start] = 0, heuristic(start, goal)

    open_set = []
    heapq.heappush(open_set, (f_score[start], heuristic(start, goal), start))
    path, visited = {}, set()
    explored_states = 0

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

```
        if current in visited:
            continue
        visited.add(current)
        explored_states += 1
        if current == goal:
            break
        for d, (dx, dy) in directions.items():
            if m.maze_map[current][d]:
                nxt = (current[0] + dx, current[1] + dy)
                if nxt in visited:
                    continue
                tentative_g = g_score[current] + 1
                if tentative_g < g_score[nxt]:
                    g_score[nxt] = tentative_g
                    f_score[nxt] = tentative_g + heuristic(nxt, goal)
                    heapq.heappush(
                        open_set, (f_score[nxt], heuristic(nxt, goal), nxt))
                    path[nxt] = current

    astar_path = {}
    current = goal
    while current in path:
        astar_path[path[current]] = current
        current = path[current]

    return astar_path, round(time.time() - start_time, 6), explored_states


if __name__ == '__main__':
    if len(sys.argv) == 3:
        try:
            rows, cols = int(sys.argv[1]), int(sys.argv[2])
        except ValueError:
            print("Invalid input. Please provide integers for maze dimensions.")
            sys.exit(1)
    else:
        rows, cols = 30, 30

    print(f"Generating {rows}×{cols} maze...")
    m = maze(rows, cols)
    m.CreateMaze(loopPercent=40, theme=COLOR.light)

    path, exec_time, explored = astar_algorithm(m)
    print(f"Path found: {path}")
    print(f"Execution time: {exec_time} seconds")
    print(f"States explored: {explored}")

    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
    textLabel(m, 'A* Path Length', len(path) + 1)
    textLabel(m, 'Execution Time', exec_time)
    textLabel(m, 'States Explored', explored)
    m.run()
```

Appendix D: MDP Value Iteration Implementation

```
import numpy as np
import sys
import time
```

```python
from pyamaze import maze, agent, textLabel, COLOR


def mdp_value_iteration(m):
    start_time = time.time()
    actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
    directions = 'ESWN'
    rows, cols = m.rows, m.cols
    gamma = 0.95
    theta = max(0.001, 1 / (rows * cols))

    V = np.zeros((rows, cols))
    rewards = np.full((rows, cols), -0.1)
    rewards[0, 0] = 100
    explored_states = 0

    while True:
        delta = 0
        new_V = np.copy(V)
        for i in range(rows):
            for j in range(cols):
                if (i, j) == (0, 0):
                    continue
                q_vals = []
                for u, d in enumerate(directions):
                    next_i, next_j = i + actions[u][0], j + actions[u][1]
                    if 0 <= next_i < rows and 0 <= next_j < cols and m.maze_map[(i+1, j+
                        q_vals.append(
                            rewards[next_i, next_j] + gamma * V[next_i, next_j])
                best_value = max(q_vals) if q_vals else V[i, j]
                new_V[i, j] = best_value
                delta = max(delta, abs(V[i, j] - best_value))
                explored_states += 1
        V = new_V
        if delta < theta:
            break

    policy = np.full((rows, cols), -1, dtype=int)
    for i in range(rows):
        for j in range(cols):
            if (i, j) == (0, 0):
                continue
            q_vals = []
            for u, d in enumerate(directions):
                next_i, next_j = i + actions[u][0], j + actions[u][1]
                if 0 <= next_i < rows and 0 <= next_j < cols and m.maze_map[(i+1, j+1)][
                    q_vals.append(
                        (rewards[next_i, next_j] + gamma * V[next_i, next_j], u))
            if q_vals:
                _, best_action = max(q_vals)
                policy[i, j] = best_action
            else:
                policy[i, j] = -1

    path = {}
    i, j = rows - 1, cols - 1
    max_steps = rows * cols
    steps = 0
```

```
    while (i, j) != (0, 0):
        if not (0 <= i < rows and 0 <= j < cols):
            break
        idx = policy[i, j]
        if idx == -1:
            print(f"Stuck at ({i+1},{j+1}), adjusting policy...")
            break
        next_i, next_j = i + actions[idx][0], j + actions[idx][1]
        steps += 1
        if steps > max_steps:
            print("Warning: MDP value iteration path construction took too long.")
            break
        path[(i+1, j+1)] = (next_i+1, next_j+1)
        i, j = next_i, next_j

    if (1, 1) not in path:
        print("Warning: Goal was not reached, adjusting policy...")
        path[(1, 1)] = (2, 1) if (2, 1) in path else (1, 2)

    return path, round(time.time() - start_time, 6), explored_states


if __name__ == '__main__':
    if len(sys.argv) == 3:
        try:
            rows, cols = int(sys.argv[1]), int(sys.argv[2])
        except ValueError:
            print("Invalid input. Provide integers for maze dimensions.")
            sys.exit(1)
    else:
        rows, cols = 30, 30

    print(f"Generating {rows}x{cols} maze...")
    m = maze(rows, cols)
    m.CreateMaze(loopPercent=40, theme=COLOR.light)

    path, exec_time, explored = mdp_value_iteration(m)
    print(f"Path found: {path}")
    print(f"Execution time: {exec_time} seconds")
    print(f"States explored: {explored}")

    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
    textLabel(m, 'MDP Path Length', len(path))
    textLabel(m, 'Execution Time', exec_time)
    textLabel(m, 'Explored States', explored)
    m.run()
```

Appendix E: MDP Policy Iteration Implementation

```
from pyamaze import maze, agent, textLabel, COLOR
import numpy as np
import time
import sys


def mdp_policy_iteration(m):
    start_time = time.time()
    directions = 'ESWN'
```

```python
actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
rows, cols = m.rows, m.cols
gamma = 0.9
theta = 0.001

V = np.zeros((rows, cols))
policy = np.random.choice(len(actions), size=(rows, cols))
rewards = np.full((rows, cols), -1.0)
rewards[0, 0] = 100

explored_states = 0
stable_policy = False
while not stable_policy:
    while True:
        delta = 0
        new_V = np.copy(V)
        for i in range(rows):
            for j in range(cols):
                if (i, j) == (0, 0):
                    continue
                u = policy[i, j]
                next_i, next_j = i + actions[u][0], j + actions[u][1]
                if 0 <= next_i < rows and 0 <= next_j < cols and m.maze_map[(i+1, j
                    value = rewards[next_i, next_j] + \
                        gamma * V[next_i, next_j]
                else:
                    value = V[i, j]
                new_V[i, j] = value
                delta = max(delta, abs(V[i, j] - value))
                explored_states += 1
        V = new_V
        if delta < theta:
            break

    stable_policy = True
    for i in range(rows):
        for j in range(cols):
            if (i, j) == (0, 0):
                continue
            q_vals = []
            for u, d in enumerate(directions):
                next_i, next_j = i + actions[u][0], j + actions[u][1]
                if 0 <= next_i < rows and 0 <= next_j < cols and m.maze_map[(i+1, j
                    q_vals.append(
                        (rewards[next_i, next_j] + gamma * V[next_i, next_j], u))
            if q_vals:
                best_value, best_action = max(q_vals)
                if best_action != policy[i, j]:
                    stable_policy = False
                    policy[i, j] = best_action

path = {}
i, j = rows - 1, cols - 1
max_steps = rows * cols
steps = 0
while (i, j) != (0, 0):
    if not (0 <= i < rows and 0 <= j < cols):
        break
```

```
        idx = policy[i, j]
        next_i, next_j = i + actions[idx][0], j + actions[idx][1]
        steps += 1
        if steps > max_steps:
            print(
                "Warning: MDP policy iteration path construction exceeded maximum steps.
            break
        path[(i+1, j+1)] = (next_i+1, next_j+1)
        i, j = next_i, next_j

    return path, round(time.time() - start_time, 6), explored_states


if __name__ == '__main__':
    if len(sys.argv) == 3:
        try:
            rows, cols = int(sys.argv[1]), int(sys.argv[2])
        except ValueError:
            print("Invalid input. Provide integers for maze size.")
            sys.exit(1)
    else:
        rows, cols = 20, 20

    print(f"Generating {rows}×{cols} maze...")
    m = maze(rows, cols)
    m.CreateMaze(loopPercent=40, theme=COLOR.light)

    path, exec_time, explored = mdp_policy_iteration(m)
    print(f"Path found: {path}")
    print(f"Execution time: {exec_time} seconds")
    print(f"States explored: {explored}")

    a = agent(m, footprints=True, filled=True)
    m.tracePath({a: path})
    textLabel(m, 'MDP Path Length', len(path))
    textLabel(m, 'Execution Time', exec_time)
    textLabel(m, 'States Explored', explored)
    m.run()
```