TRINITY COLLEGE DUBLIN

School of Computer Science and Statistics

Abhishek Zade

StudentId: 24332461

Assignment 1 **CS7IS2 Artificial Intelligence**

Introduction

In artificial intelligence, maze solving is a typical task that entails negotiating a maze of obstacles in order to arrive to a predetermined destination. This assignment involves solving mazes of different difficulty using both conventional search algorithms and Markov Decision Process (MDP) approaches. In order to assess the performance of five algorithms—Depth-First Search (DFS), Breadth-First Search (BFS), A* Search, MDP Value Iteration, and MDP Policy Iteration—in terms of runtime, memory use, and the caliber of the solution route that is produced, I want to implement and compare them.

Additionally, I created a custom maze generator that can generate mazes according to various difficulty settings and algorithms (DFS, Prim's, and Aldous-Broder). By enforcing walls on the borders and designated access and departure locations, the generator guarantees a consistent structure. The trade-offs between algorithmic efficiency and solution optimality are further demonstrated by visual aids like graphs and solution diagrams.

Methodology

The design and implementation of the maze generation, solution algorithms, and performance analysis assessment system are described in this part.

Maze Generation

Maze Structure

Every cell in the grid-based maze is either a wall (1) or navigable (0). While some cells are specified as the entry (usually at `(1, 0)`) and exit (usually at `(maze.shape[0]-2, maze.shape[1]-1)`), the outer borders are tightly enforced as walls.

Generation Techniques

A DFS-based method is used to create a "perfect" maze at lower difficulty levels (up to 3), when there is only one distinct path connecting any two points. Prim's algorithm is used with purposeful extra holes to generate loops for moderate difficulty levels (4–6), offering more complexity and various possible pathways. The Aldous-Broder method is used to create mazes with many loops and a high degree of randomization at higher difficulty levels (7 and above), providing a more complex and unpredictable maze structure.

A* Diagram for Maze Generation

Figure 1 illustrates an A* solution on a medium-sized maze. A* leverages the Manhattan distance heuristic to balance the cost-so-far with the estimated remaining cost, demonstrating how heuristic information can improve search efficiency.

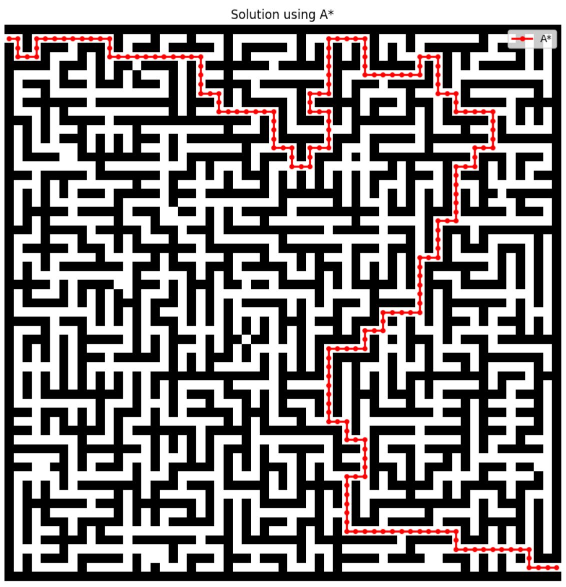


Figure 1: An A* solution on a medium-sized maze, demonstrating heuristic-driven pathfinding.

Maze Solving Algorithms

Five algorithms are implemented to solve the maze:

- **Depth-First Search (DFS):** Uses a stack-based recursive approach, exploring deeply before backtracking. This method is simple but may generate longer paths.
- **Breadth-First Search (BFS):** Uses a queue for level-by-level exploration, guaranteeing the shortest path in terms of the number of steps.
- **A* Search:** A heuristic estimate of the cost to reach the target, usually based on the Manhattan distance, is combined with the actual cost from the start node in the A* solver code's informed search algorithm. The next node to explore is dynamically chosen using a priority queue based on the lowest combined cost (typically represented as f(n) = g(n) + h(n)), guaranteeing effective labyrinth navigation and speedy path discovery. The approach balances path optimality and computational economy by constantly updating cost values and keeping track of visited nodes to prevent needless re-exploration.
- **MDP Value Iteration:** The Value Iteration approach is used by the MDP Value solver code to solve the labyrinth by using the Bellman update equation to directly update the value function for each state. Using this method, the algorithm iteratively improves the value of each state by taking into account both the immediate benefit and the discounted future rewards from later stages. This process continues until the value changes are negligible, or below a predetermined convergence

threshold. Following convergence, a greedy policy is extracted by selecting the course of action that results in the state with the highest value. This effectively creates an optimal maze-navigating strategy based on the value function that has been learned.

- **MDP Policy Iteration:** The MDP Policy solver code models the maze as a Markov Decision Process where each cell represents a state with associated actions and rewards. It implements the Policy Iteration algorithm by first evaluating the current policy—calculating the expected rewards for each state—and then improving the policy by selecting actions that maximize the long-term reward. This iterative process of alternating between policy evaluation and policy improvement continues until the policy converges to a stable solution, ensuring that the derived policy yields the best expected cumulative reward based on the defined reward structure and transition probabilities.

MDP Parameters and Equations

For the MDP-based methods, two key parameters are the discount factor (γ) and the convergence threshold (θ). The discount factor determines the weight given to future rewards. The Bellman update equation for Value Iteration is:

$$V(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a)V(s') \right]$$

where $V(s)$ is the value of state s, $R(s,a)$ is the immediate reward for taking action a in state s, and $P(s'|s,a)$ is the transition probability to state s'. A higher discount factor (e.g., 0.998 or 0.9998) emphasizes future rewards but increases the number of iterations required for convergence, thereby impacting overall runtime.

Performance Evaluation Framework

The performance of each algorithm is evaluated by varying both maze size and difficulty. Key performance metrics include:

- **Path Length:** Number of steps in the solution path.
- **Runtime:** Time required to compute the solution.
- **Memory Usage:** Estimated via the difference between pre- and post-execution measurements.

Composite graphs are used to collect and display data by comparing these metrics across various configurations. Sometimes the operating system's trash collection or memory cleanup causes the final measured memory to be lower than the initial measurement, which results in negative readings.

Results and Analysis

This section provides a detailed analysis based on the experimental results obtained from running the program with various maze dimensions and difficulty levels.

Traditional Search Algorithms

DFS, BFS, and A* are examples of conventional search algorithms that work quite well for short mazes (e.g., 10×10 with difficulty 3). With runtimes in the low milliseconds (e.g., BFS at 0.00016 sec and A* at 0.00018 sec), the data demonstrates that all three algorithms find paths of comparable length (e.g., 63 cells for a 10×10 maze with difficulty 3). The absolute runtime of these methods increases slightly (DFS: 0.0549 sec, BFS: 0.09579 sec, A*: 0.17272 sec) as the labyrinth size increases (e.g., 100×100) while still achieving satisfactory performance. The solution pathways are shown in Figures 3 and 4, which emphasize that although DFS, while sometimes faster, tends to generate longer paths since it is depth-first. BFS, on the other hand, ensures the shortest path.

MDP-Based Algorithms

The computing cost of the MDP-based techniques (Policy Iteration and Value Iteration) increases significantly, especially when maze size and complexity rise. For example, MDP Policy Iteration required 72 iterations (runtime of about 0.03034 seconds) to converge in a 10×10 maze of difficulty 3, while Value Iteration took 0.01085 seconds. However, the runtime for Value Iteration takes about 80 seconds, while the runtime for Policy Iteration soars to over 21 seconds (with nearly 3909 iterations) as the maze size climbs to 100×100 with difficulty 3. At higher difficulty levels, this pattern becomes increasingly more noticeable. The high discount factor directly leads to the longer runtime and more iterations, even though it is advantageous for long-term reward accuracy. Additionally, system-level memory cleanup techniques are responsible for the negative memory use values seen in some MDP results (e.g., -44176 kB for Policy Iteration in a 100×100 maze).

Comparative Analysis of Traditional vs. MDP-Based Methods

A direct comparison reveals that traditional search algorithms are far more efficient in terms of both runtime and memory usage. For example:

- In a 10×10 maze, traditional methods consistently solve the maze in under 0.003 seconds, whereas MDP-based methods, although still fast, take noticeably longer (approximately 0.030 sec for Policy Iteration).
- In a 100×100 maze, traditional search methods complete in less than 0.2 seconds at most, while MDP-based approaches require orders of magnitude more time (21 sec for Policy Iteration and 79 sec for Value Iteration).

Figure 2 shows a composite graph comparing these metrics. The steep increase in runtime and the occurrence of negative memory usage values in MDP methods clearly indicate that, despite their sophisticated reward-based decision-making framework, they are less scalable compared to traditional methods when applied to larger, more complex mazes.

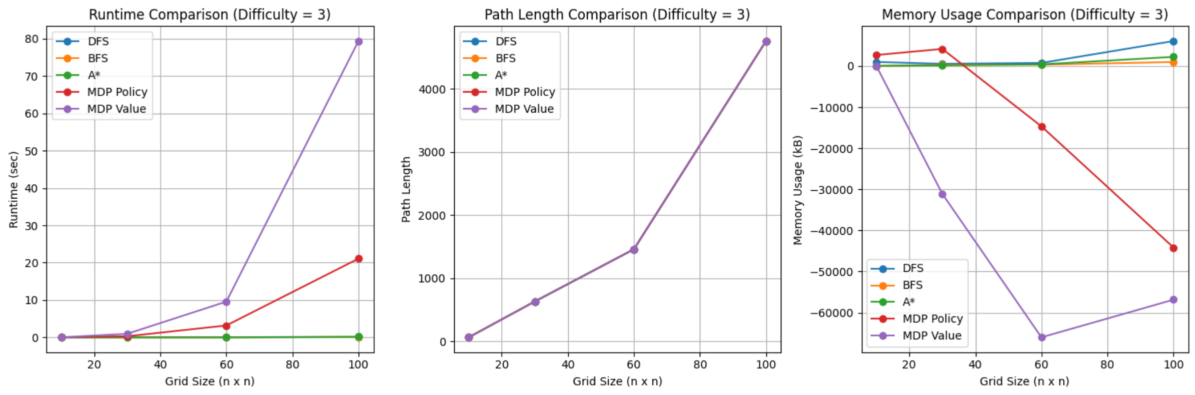
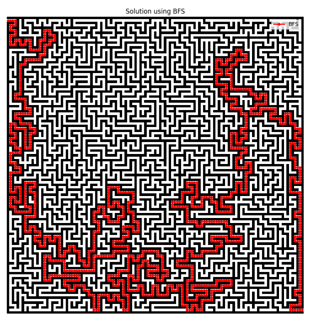


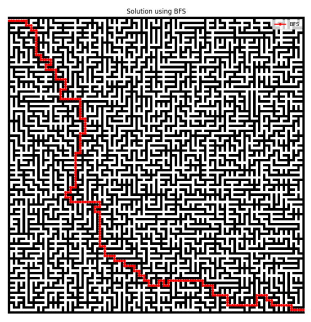
Figure 2: Composite Performance Comparison of Maze Solving Algorithms. Negative memory usage values indicate that the final measurement was lower than the initial measurement due to system-level memory cleanup.

Additional Diagram References

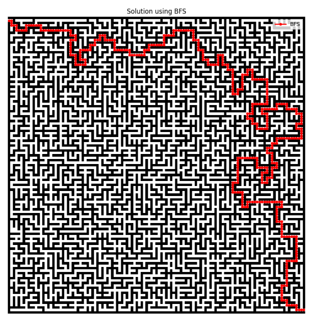
Figures 3 and 4 show solution paths for BFS and DFS on a 60×60 maze, respectively. Figures 5 and 6 depict solution paths for MDP Policy Iteration and MDP Value Iteration. These diagrams visually reinforce the analysis by demonstrating differences in path quality and computational efficiency between the algorithms.



BFS Solution Easy

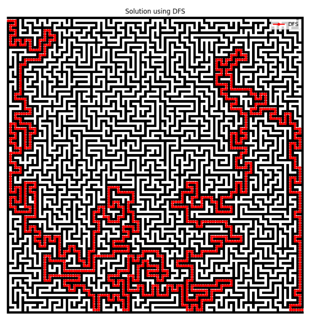


BFS Solution Medium

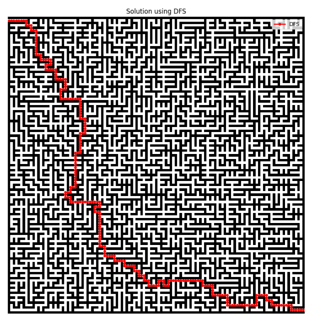


BFS Solution Hard

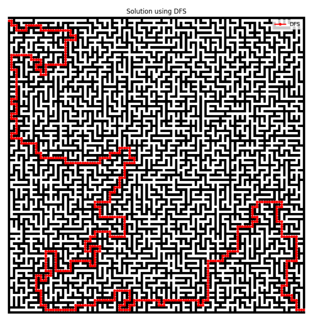
Figure 3: 60×60 BFS Maze Diagrams



DFS Solution Easy

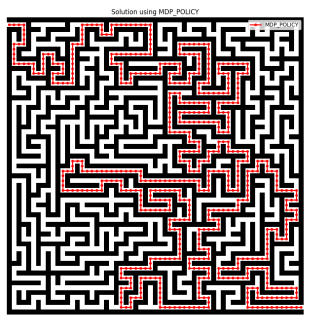


DFS Solution Medium

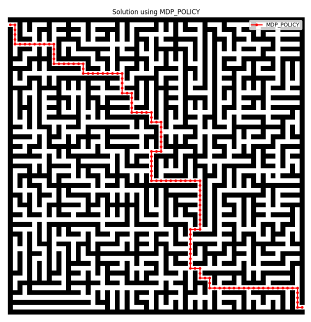


DFS Solution Hard

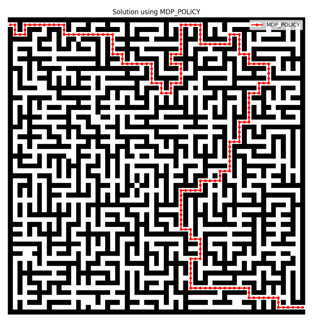
Figure 4: 60×60 DFS Maze Diagrams



MDP Policy Easy

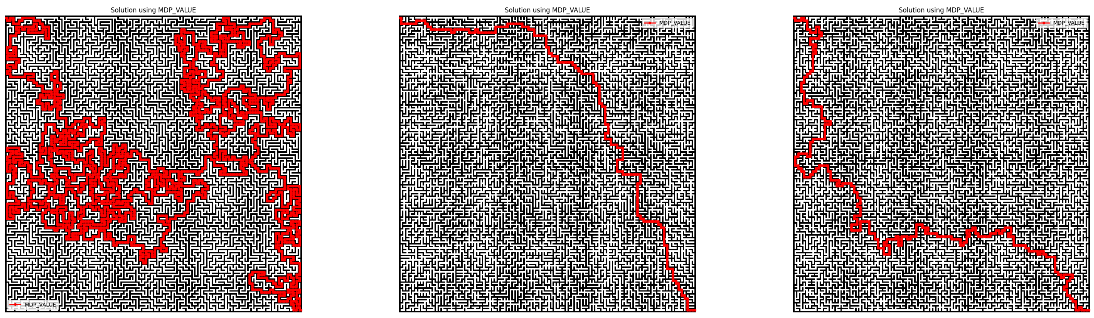


MDP Policy Medium



MDP Policy Hard

Figure 5: 30×30 Maze Solutions: MDP Policy Iteration



MDP Value Easy MDP Value Medium MDP Value Hard

Figure 6: 100×100 Maze Solutions: MDP Value Iteration

Conclusion

According to the thorough examination, conventional search algorithms like DFS, BFS, and A* continuously perform better in terms of scalability and efficiency than MDP-based techniques, especially for real-time applications and bigger labyrinth configurations. Even as maze size and complexity increase, traditional approaches' relatively simple exploration mechanisms—DFS's recursive depth-first traversal, BFS's level-by-level expansion guaranteeing optimal path length, and A*'s heuristic-driven search that efficiently condenses the search space—lead to significantly lower runtime and memory usage. While conceptually appealing due to their capacity to incorporate long-term planning and nuanced reward structures, MDP-based techniques such as Policy Iteration and Value Iteration, on the other hand, require significantly more computational resources; their performance is highly sensitive to the selected discount factor and convergence threshold, resulting in significant increases in the number of iterations and runtime when applied to larger state spaces. Furthermore, their practical implementation is made more difficult by sporadic negative memory usage measurements that come from the measurement process and the ensuing system-level memory cleanup. MDP-based methods might be more appropriate for complex decision-making tasks where thorough long-term reward optimization is necessary and adequate computational resources are available, but traditional search algorithms are generally the best option for situations where quick response and low resource consumption are crucial.

Appendix

```python
import os
import csv
import datetime
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path
from dfs_solver import solve_dfs
from bfs_solver import solve_bfs
from astar_solver import solve_astar
from mdp_policy_solver import solve_mdp_policy_iteration
from mdp_value_solver import solve_mdp_value_iteration
from maze_generator import generate_maze


def create_results_directory():
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    results_dir = Path(f"results_{timestamp}")
    results_dir.mkdir(parents=True, exist_ok=True)
    return results_dir

def save_maze_solution(maze, path, algorithm, results_dir):
    plt.figure(figsize=(10, 10))
    plt.imshow(maze, cmap="gray_r")

    if path:
        path_x, path_y = zip(*path)
        plt.plot(path_y, path_x, marker="o", color="red", markersize=4, linewidth=2, label=algorithm)

    plt.title(f"Solution␣using␣{algorithm}")
    plt.legend()
    plt.axis("off")

    image_path = results_dir / f"{algorithm}_solution.png"
    plt.savefig(image_path)
    plt.close()

def save_results_to_csv(results, results_dir):
    csv_filename = results_dir / "algorithm_performance.csv"
    fieldnames = ["Algorithm", "Path␣Length", "Runtime␣(sec)", "Memory␣(kB)", "Discount", "Theta"]

    with open(csv_filename, "w", newline="") as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for alg, data in results.items():
            writer.writerow({
                "Algorithm": alg,
                "Path␣Length": data["Path␣Length"],
                "Runtime␣(sec)": data["Runtime"],
                "Memory␣(kB)": data["Memory"],
                "Discount": data.get("Discount", "N/A"),
                "Theta": data.get("Theta", "N/A"),
            })
    print(f"Performance␣data␣saved␣in␣{csv_filename}")

def plot_performance_comparison(results, results_dir):
    algorithms = list(results.keys())
    runtimes = [results[alg]["Runtime"] for alg in algorithms]
    path_lengths = [results[alg]["Path␣Length"] for alg in algorithms]
    memories = [results[alg]["Memory"] for alg in algorithms]

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.bar(algorithms, runtimes, color="skyblue")
    plt.ylabel("Runtime␣(seconds)")
    plt.title("Algorithm␣Runtime␣Comparison")

    plt.subplot(1, 3, 2)
    plt.bar(algorithms, path_lengths, color="salmon")
    plt.ylabel("Path␣Length␣(steps)")
    plt.title("Algorithm␣Path␣Length␣Comparison")

    plt.subplot(1, 3, 3)
    plt.bar(algorithms, memories, color="lightgreen")
    plt.ylabel("Memory␣(kB)")
    plt.title("Algorithm␣Memory␣Usage␣Comparison")

    plt.suptitle("Maze␣Solver␣Performance␣Comparison")
    performance_image_path = results_dir / "performance_comparison.png"
    plt.savefig(performance_image_path)
    plt.close()
    print(f"Performance␣comparison␣graph␣saved␣at␣{performance_image_path}")


def analyze_algorithms(maze, start, goal, algorithms, solve_functions, params):
    results_dir = create_results_directory()
    results = {}

    for algorithm, solve_func in solve_functions.items():
        if algorithm in algorithms:
            print(f"Running␣{algorithm}...")

            mem_before = os.popen("ps␣-o␣rss=␣-p␣" + str(os.getpid())).read().
                strip()
            path, steps, runtime = solve_func(maze, start, goal, **params.get(
                algorithm, {}))
            mem_after = os.popen("ps␣-o␣rss=␣-p␣" + str(os.getpid())).read().
                strip()

            mem_usage = int(mem_after) - int(mem_before)
            results[algorithm] = {
                "Path␣Length": steps,
                "Runtime": runtime,
                "Memory": mem_usage,
                **params.get(algorithm, {}),
            }

            save_maze_solution(maze, path, algorithm, results_dir)

    save_results_to_csv(results, results_dir)
    plot_performance_comparison(results, results_dir)

    return results

def main():
    dim = int(input("Enter␣maze␣dimension␣(number␣of␣cells␣per␣side):␣"))
    difficulty = int(input("Enter␣maze␣difficulty␣(1-10):␣"))

    maze = generate_maze(difficulty=difficulty, dim=dim)
    start = (1, 0)
    goal = (maze.shape[0]-2, maze.shape[1]-1)

    print("\nSelect␣algorithms␣to␣run␣(separate␣by␣commas):")
    print("Options:␣DFS,␣BFS,␣A*,␣MDP_POLICY,␣MDP_VALUE")
    selected_algorithms = input("Enter␣choices:␣").upper().split(",")

    selected_algorithms = [alg.strip() for alg in selected_algorithms]
    valid_algorithms = {"DFS", "BFS", "A*", "MDP_POLICY", "MDP_VALUE"}
    selected_algorithms = [alg for alg in selected_algorithms if alg in
       valid_algorithms]

    if not selected_algorithms:
        print("No␣valid␣algorithms␣selected.␣Exiting.")
        return

    solve_functions = {
        "DFS": solve_dfs,
        "BFS": solve_bfs,
        "A*": solve_astar,
        "MDP_POLICY": solve_mdp_policy_iteration,
        "MDP_VALUE": solve_mdp_value_iteration,
    }

    mdp_params = {}
    if "MDP_POLICY" in selected_algorithms:
        mdp_params["MDP_POLICY"] = {
            "discount": float(input("Enter␣discount␣factor␣for␣MDP␣Policy␣
               Iteration␣(e.g.,␣0.9):␣")),
            "theta": float(input("Enter␣convergence␣threshold␣for␣MDP␣Policy␣
                Iteration␣(e.g.,␣0.001):␣")),
        }
    if "MDP_VALUE" in selected_algorithms:
        mdp_params["MDP_VALUE"] = {
            "discount": float(input("Enter␣discount␣factor␣for␣MDP␣Value␣
                Iteration␣(e.g.,␣0.9):␣")),
            "theta": float(input("Enter␣convergence␣threshold␣for␣MDP␣Value␣
                Iteration␣(e.g.,␣0.001):␣")),
        }

    analyze_algorithms(maze, start, goal, selected_algorithms, solve_functions,
         mdp_params)

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

Listing 1: driver code

```
import matplotlib.pyplot as plt

def get_neighbors(cell, maze):
    r, c = cell
    neighbors = []
    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        nr, nc = r + dr, c + dc
        if 0 <= nr < maze.shape[0] and 0 <= nc < maze.shape[1] and maze[nr, nc]
             == 0:
            neighbors.append((nr, nc))
    return neighbors

def overlay_path_on_maze(maze, path, algorithm_name, steps, runtime, filename):
    plt.figure(figsize=(8, 8))
    plt.imshow(maze, cmap='binary')
    if path:
        rows = [p[0] for p in path]
        cols = [p[1] for p in path]
        plt.plot(cols, rows, color='red', linewidth=2, label='Solution␣Path')
    plt.title(f"{algorithm_name}\nSteps:␣{steps}␣|␣Runtime:␣{runtime:.4f}␣sec")
    plt.axis('off')
    plt.legend()
    plt.savefig(filename)
    plt.close()
```

Listing 2: common

```
# (Include your Python code here)
```

Listing 3: Python Code for Maze Solvers l

```
import time
from collections import deque
```

```python
from common import get_neighbors

def solve_bfs(maze, start, goal):
    start_time = time.time()
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        current, path = queue.popleft()
        if current == goal:
            runtime = time.time() - start_time
            return path, len(path), runtime
        if current in visited:
            continue
        visited.add(current)
        for neighbor in get_neighbors(current, maze):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
    return None, 0, time.time() - start_time

if __name__ == "__main__":
    pass
```

Listing 4: bfs solver

```python
import time
from common import get_neighbors

def solve_dfs(maze, start, goal):
    start_time = time.time()
    stack = [(start, [start])]
    visited = set()
    while stack:
        current, path = stack.pop()
        if current == goal:
            runtime = time.time() - start_time
            return path, len(path), runtime
        if current in visited:
            continue
        visited.add(current)
        for neighbor in get_neighbors(current, maze):
            if neighbor not in visited:
                stack.append((neighbor, path + [neighbor]))
    return None, 0, time.time() - start_time
```

Listing 5: dfs solver

```python
import time
import heapq
from common import get_neighbors

def solve_astar(maze, start, goal):
    start_time = time.time()
    def heuristic(a, b):
        return abs(a[0]-b[0]) + abs(a[1]-b[1])
    open_set = []
    heapq.heappush(open_set, (heuristic(start, goal), 0, start, [start]))
    visited = {}
    while open_set:
        f, g, current, path = heapq.heappop(open_set)
        if current == goal:
            runtime = time.time() - start_time
            return path, len(path), runtime
        if current in visited and visited[current] <= g:
            continue
        visited[current] = g
        for neighbor in get_neighbors(current, maze):
            new_cost = g + 1
            heapq.heappush(open_set, (new_cost + heuristic(neighbor, goal),
                new_cost, neighbor, path + [neighbor]))
    return None, 0, time.time() - start_time
```

Listing 6: a* solver

```
import time
import numpy as np

def solve_mdp_policy_iteration(maze, start, goal, discount=0.9, theta=0.001):
    start_time = time.time()
    rows, cols = maze.shape
    passable = (maze == 0)

    r_idx, c_idx = np.indices((rows, cols))

    actions = np.array([[-1, 0],
                        [ 1, 0],
                        [ 0, -1],
                        [ 0,  1]])
    num_actions = actions.shape[0]

    V = np.zeros((rows, cols))
    policy = -1 * np.ones((rows, cols), dtype=int)
    random_policy = np.random.randint(0, num_actions, size=(rows, cols))
    mask = passable.copy()
    mask[goal] = False
    policy[mask] = random_policy[mask]

    def safe_next_state(a_idx):
        dr, dc = actions[a_idx]
        cand_r = r_idx + dr
        cand_c = c_idx + dc
        in_bounds = (cand_r >= 0) & (cand_r < rows) & (cand_c >= 0) & (cand_c <
            cols)
        valid = np.zeros_like(in_bounds, dtype=bool)
```

```python
        valid[in_bounds] = passable[cand_r[in_bounds], cand_c[in_bounds]]
        next_r = np.where(valid, cand_r, r_idx)
        next_c = np.where(valid, cand_c, c_idx)
        return next_r, next_c

    cand_r = np.empty((num_actions, rows, cols), dtype=int)
    cand_c = np.empty((num_actions, rows, cols), dtype=int)
    for a in range(num_actions):
        cand_r[a], cand_c[a] = safe_next_state(a)

    def compute_candidate_values():
        reward = np.where((cand_r == goal[0]) & (cand_c == goal[1]), 0, -1)
        candidate_vals = reward + discount * V[cand_r, cand_c]
        return candidate_vals

    policy_stable = False
    iteration = 0
    while not policy_stable:
        iteration += 1
        while True:
            candidate_vals = compute_candidate_values()
            update_mask = (policy != -1)
            V_new = V.copy()
            idx = np.where(update_mask)
            V_new[idx] = candidate_vals[policy[idx], idx[0], idx[1]]
            delta = np.max(np.abs(V_new - V))
            V = V_new
            if delta < theta:
                break

        candidate_vals = compute_candidate_values()
        best_actions = np.argmax(candidate_vals, axis=0)
        new_policy = policy.copy()
        new_policy[update_mask] = best_actions[update_mask]
        if np.all(new_policy[update_mask] == policy[update_mask]):
            policy_stable = True
        else:
            policy = new_policy

    print(f"Policy␣Iteration␣converged␣after␣{iteration}␣iterations.")

    path = [start]
    current = start
    for _ in range(10000):
        if current == goal:
            break
        r, c = current
        a = policy[r, c]
        if a == -1:
            print("No␣valid␣action␣found␣at␣state", current, "stopping␣path␣
                extraction.")
            break
        next_r, next_c = safe_next_state(a)
        next_state = (next_r[r, c], next_c[r, c])
        if next_state == current:
            print("Stuck in local optimum at", current, "stopping path extraction.")
            break
        current = next_state
        path.append(current)

    runtime = time.time() - start_time
    return path, len(path), runtime
```

Listing 7: mdp policy solver

```python
import time

def solve_mdp_value_iteration(maze, start, goal, discount=0.99, theta=0.001):
    start_time = time.time()

    start = (int(start[0]), int(start[1]))
    goal = (int(goal[0]), int(goal[1]))

    rows, cols = maze.shape
    states = {(r, c) for r in range(rows) for c in range(cols) if maze[r, c] == 0}

    if start not in states or goal not in states:
        raise ValueError("Start or Goal state is not passable!")

    actions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    transitions = {}
    for s in states:
        r, c = s
        transitions[s] = {}
        for a in actions:
            ns = (r + a[0], c + a[1])
            if ns not in states:
                ns = s
            transitions[s][a] = ns

    V = {s: 0.0 for s in states}

    while True:
        delta = 0.0
        newV = {}
        for s in states:
            if s == goal:
                newV[s] = 0.0
                continue

            best_val = float('-inf')
            for a in actions:
                ns = transitions[s][a]
                rwd = 1.0 if ns == goal else -0.01
                val = rwd + discount * V[ns]
                best_val = max(best_val, val)

            newV[s] = best_val
            delta = max(delta, abs(newV[s] - V[s]))
        V = newV
        if delta < theta:
            break

    s = start
    path = [s]
    while s != goal:
        best_val = float('-inf')
        best_ns = s
        for a in actions:
            ns = transitions[s][a]
            rwd = 1.0 if ns == goal else -0.01
            val = rwd + discount * V[ns]
            if val > best_val:
                best_val = val
                best_ns = ns
        if best_ns == s:
            break
        s = best_ns
        path.append(s)

    runtime = time.time() - start_time
    return path, len(path), runtime
```
Listing 8: mdp value solver