Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Assignment I
CS7IS2 Artificial Intelligence

Ujjayant Kadian (22330954)

February 28, 2025

Introduction

In this project, we implemented and evaluated five different algorithms for solving mazes: Depth-First Search (DFS), Breadth-First Search (BFS), A* search, Policy Iteration, and Value Iteration. These algorithms represent both classical graph search techniques (DFS, BFS, A*) and Markov Decision Process (MDP) solution methods (Policy and Value Iteration), providing a broad perspective on approaches to navigate through a maze.

A systematic evaluation of different maze-solving approaches is important to understand their trade-offs in terms of efficiency, optimality, and resource usage. Different algorithms may excel under different conditions: for example, some guarantee the shortest path while others use less memory or time. By comparing multiple strategies on the same set of maze problems, we can identify which algorithms are best suited for particular scenarios and why.

Methodology

Search Algorithms (DFS, BFS, A*)

We implemented three search algorithms that explore the maze as a graph of states:

Depth-First Search (DFS)

DFS explores the maze by going as deep as possible along one path before backtracking. It uses a stack (LIFO) structure or recursion to prioritize depth, which can quickly find a solution in practice but does not guarantee that the found path is the shortest.

Breadth-First Search (BFS)

BFS explores layer by layer outward from the start. It uses a queue (FIFO) to ensure that the first time the goal is reached, it is via the shortest possible path (in an unweighted grid). BFS guarantees an optimal solution but often at the cost of exploring many nodes and using more memory.

A* Search

A* is an informed search algorithm that uses a heuristic to guide exploration. We chose the Manhattan distance heuristic for A*, which estimates the number of moves to reach the goal by summing horizontal and vertical distances. This heuristic is admissible (it never overestimates the true distance on a grid with only orthogonal moves) and consistent, ensuring A* finds an optimal path while usually expanding far fewer nodes than BFS. A* maintains a priority queue of frontier nodes ordered by the estimated total path cost:

$$f(n) = g(n) + h(n) \tag{1}$$

where:

- $g(n)$ is the cost from the start.
- $h(n)$ is the heuristic (Manhattan distance).

The design choice of Manhattan distance was made because it closely reflects actual path cost in a grid maze and significantly speeds up search by directing it toward the goal.

MDP Algorithms (Policy Iteration, Value Iteration)

We also modeled the maze as a Markov Decision Process (MDP) to apply dynamic programming solutions. In this formulation, each cell of the maze is a state, and moving in one of the four directions (up, down, left, right) is an action.

We assigned a small negative reward for each move (e.g., -1 per step) to encourage shorter solutions, and reaching the goal yields a reward of 0 to represent completion. A discount factor γ (e.g., 0.9) was used to ensure the convergence of iterative algorithms.

Policy Iteration

Policy Iteration consists of two steps:

- **Policy Evaluation:** We compute the value function $V(s)$ for the current policy by iteratively updating state values until they converge within a small threshold θ.
- **Policy Improvement:** The policy is greedily updated at each state to the action that maximizes the expected value (looking one step ahead using the current V).

These steps repeat until the policy stabilizes (no change in the policy in an iteration), guaranteeing an optimal policy.

Value Iteration

Value Iteration directly updates the value function by repeatedly applying the Bellman optimality update:

$$V_{\text{new}}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V(s') \right] \tag{2}$$

In our deterministic maze, this simplifies to:

$$V_{\text{new}}(s) = \max_{a \in \text{actions}} [-1 + \gamma V(s_{\text{next}})] \tag{3}$$

for non-goal states (with 0 reward at the goal). We iterated this update for all states simultaneously until values converged, then derived the optimal policy by choosing the best action for each state from the final value function.

Both MDP approaches guarantee an optimal solution (shortest path in terms of cumulative negative reward) to the maze. The design choice for MDP included setting an appropriate discount ($\gamma \approx 0.9$) to prioritize shorter paths while still considering future rewards and a convergence threshold θ (we used a very small value like 10^{-4} to ensure accurate convergence).

Maze Generation and State Representation

The mazes used in this project were generated using a randomized maze generator to provide a variety of test cases. We used a depth-first maze generation algorithm to carve out random paths, ensuring each generated maze has a unique solution path with some branching dead-ends. Maze sizes were varied (e.g., 10×10, 15×15, 18×18 grids) to test the algorithms under different scales.

The maze is represented as a grid of cells, each cell being either open or a wall. The state representation for the search and MDP algorithms is a coordinate pair (x, y) corresponding to a cell in the grid. From each state, an agent can move into one of up to four neighboring states (up, down, left, right) if that neighbor is within bounds and not a wall.

In the implementation, we encapsulated neighbor expansion logic in a function (e.g., `get_neighbors(state)`) that returns valid adjacent states. This abstraction allowed all algorithms to use the same state space consistently. We also ensured that the start and goal positions are placed in open cells of the maze and that the goal is reachable from the start in every generated maze.

Design Considerations

Throughout the implementation, we made careful choices to ensure fairness and correctness in comparisons. For example, in A* search, we used Manhattan distance due to its efficiency and admissibility. We also paid attention to data structures:

- BFS was implemented with a queue (`collections.deque`) for efficiency.
- DFS was implemented with a stack or recursion.
- A* was implemented with a min-heap (`heapq`) for frontier prioritization.

For the MDP algorithms, we had to choose a stopping criterion for iterations (a small threshold on value change for convergence). We decided to treat reaching the goal as a terminal state that ends an episode (no further reward after reaching the goal), which simplifies the reward structure.

One important practical consideration was to separate visualization from logic: while we used Pygame to animate the maze and the search progress, the performance measurements were done with rendering turned off to measure pure algorithmic efficiency. This avoided the overhead of graphical updates skewing the results.

All algorithms were integrated into a single interface so that they could run on the same mazes and be timed and measured in a uniform way.

Test Cases

We evaluated the algorithms on a set of mazes of different sizes and complexities. Maze dimensions of 10×10, 15×15, and 18×18 were chosen to represent small, medium, and moderately larger mazes. For each maze size, multiple random mazes were generated to ensure that results are not particular to one maze layout. Each algorithm was run on each maze instance with a fixed start and goal (e.g., top-left corner as start, bottom-right corner as goal), which is typical for maze puzzles. By using multiple maze instances for each size, we could observe average performance and also note any variability in outcomes.

The random generation ensures that some mazes have straightforward short paths while others force the algorithms to explore extensively, which is useful for testing the robustness and worst-case behavior of the algorithms.

Metrics for Comparison

We collected several metrics to compare performance: execution time, memory usage, nodes expanded, maximum frontier size, and iteration counts (for the MDP methods).

- **Execution Time:** Measured in seconds using wall-clock time for each algorithm's run, providing an indication of speed.
- **Memory Usage:** Tracked by monitoring the peak memory allocated during the algorithm's execution, particularly focused on the data structures for the search frontier or value tables for MDP. Memory was measured in megabytes.
- **Nodes Expanded:** Counts how many states were removed from the frontier and fully processed (i.e., for search algorithms, how many distinct cells were explored). This reflects the work done by the algorithm in searching the space.
- **Max Frontier Size:** Records the largest number of states in the frontier (open set) at any point during the search, which is an indicator of the algorithm's space complexity and memory demand. For DFS, this is effectively the deepest recursion stack size; for BFS and A*, this can be much larger.
- **Iteration Counts (for MDP methods):** Instead of nodes expanded, we consider iteration counts:
  - For Value Iteration, the number of iterations it took for the value function to converge.
  - For Policy Iteration, the number of policy improvement steps and the total number of value evaluation iterations performed.
  - We logged the iteration counts for policy iteration (how many times the policy was updated before reaching optimal) as well as how many sweeps of value updates were done in total during policy evaluation phases.

All these metrics together give a comprehensive picture of algorithm performance beyond just whether it found the path.

Procedure

Each algorithm was run in a controlled environment where the start, goal, and maze configuration were the same, allowing for fair comparison. We disabled Pygame rendering during timed runs to ensure that visualization delays did not affect the measured execution time.

Before timing, we also "warmed up" Python's interpreter for fairness by running a quick dummy search so that any one-time overhead (such as Python's first-time function compilation) would not skew the first measurement.

For memory measurement, we used Python's `tracemalloc` and recorded the peak memory usage of the algorithm's data structures. Time and memory data, as well as node counts and iterations, were logged to a CSV file for analysis. We plotted graphs from this data to visualize the comparisons.

Challenges and Considerations

During experimentation, we encountered several challenges. One notable observation was that DFS occasionally outperformed BFS in execution time for some maze instances. At first, this seems counterintuitive since BFS is guaranteed to find the shortest path with minimal exploration. However, this occurred in mazes where the goal was located deep in one particular branch of the search tree:

- DFS, by luck or the nature of the maze structure, would dive straight toward the goal without exploring much else.
- BFS, on the other hand, systematically explored many other branches of the same depth before eventually reaching the goal.

In such cases, DFS found a (non-optimal) solution faster in raw time because it did less work than BFS (which was busy ensuring no shorter path was missed). This highlights that optimality (BFS's strength) can come at the cost of extra exploration.

We also had to address Pygame limitations when scaling up the experiments. Rendering a maze and the search step-by-step in Pygame is slow for larger mazes or repeated runs, so we had to decouple the performance runs from the visualization.

Another challenge was ensuring that the termination conditions for the iterative algorithms (Policy and Value Iteration) were tuned correctly. If the threshold θ is too tight, the algorithm might take a very long time to converge; if too loose, the policy might not be fully optimal. We chose a sufficiently small θ that balanced accuracy and runtime.

Overall, careful setup and a few adjustments were needed to ensure we could collect accurate performance data for all five algorithms under comparable conditions.

Results

Execution Time Comparison

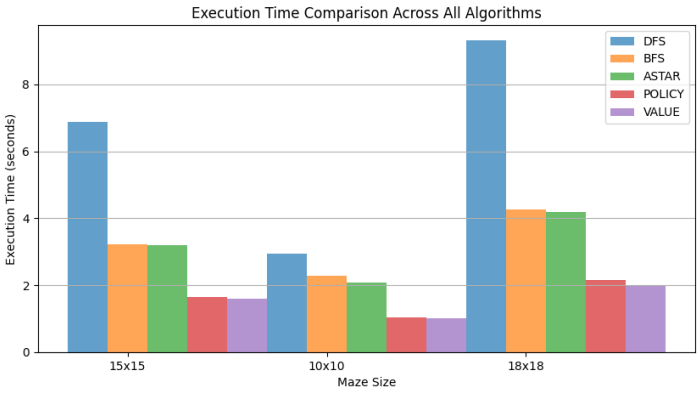

Figure 1: Execution time comparison for DFS, BFS, A*, Policy Iteration, and Value Iteration on various maze sizes.

The execution time results show clear differences in efficiency between the algorithms. In general, BFS and A* were faster than DFS on most tested mazes, as expected, since DFS can waste time exploring long dead-ends

while BFS and A* systematically find shorter paths. A* tended to be the fastest of the search algorithms, leveraging the Manhattan heuristic to expand significantly fewer nodes on average (thereby reducing runtime).

Interestingly, the MDP-based algorithms (Policy Iteration and Value Iteration) performed competitively in these tests. Often their execution times were on par with, or even lower than, the search algorithms for the same maze. For example, in larger 18×18 mazes, BFS and DFS took on the order of 8–12 seconds to find the goal, whereas Value Iteration solved the maze in around 2–3 seconds in our implementation. This counter-intuitive result stems from the efficiency of the dynamic programming approach in exploring the state space: Value Iteration quickly propagates value information from the goal to all states, effectively solving multiple paths in parallel, whereas BFS/DFS explore one path at a time.

Policy Iteration also performed well, though it has some overhead per iteration; its total time was similar to Value Iteration in our data. It's worth noting that DFS occasionally had lower times than BFS in some cases, illustrating the scenario where DFS lucked into a direct path to the goal. However, those cases are not consistent, and DFS's time skyrocketed on other mazes where it explored extensively. Overall, Figure 1 demonstrates that A* is the most time-efficient for single-path finding due to heuristic guidance, and the MDP approaches are surprisingly efficient given they compute a full policy (with Value Iteration marginally faster than Policy Iteration in our results).

Memory Usage Comparison

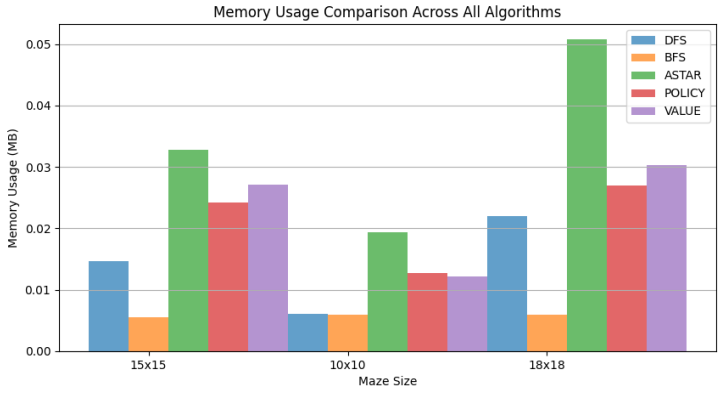

Figure 2: Peak memory usage of each algorithm during maze solving.

Memory usage patterns reflect the expected characteristics of each algorithm, with some notable deviations from theoretical expectations.

DFS, which typically requires minimal memory by storing only the current path and a stack of unexpanded nodes, demonstrates higher memory usage than BFS in some maze sizes. The graph shows that DFS consumed more memory than BFS in the 10×10 and 18×18 mazes, which is surprising since DFS is generally expected to be more memory-efficient. This suggests that in our implementation, DFS may have maintained additional state information or was affected by recursion depth.

BFS, which traditionally requires more memory due to storing a large frontier, actually had the lowest memory usage among search algorithms in most cases, especially in 10×10 and 18×18 mazes. This contradicts the general expectation that BFS would have the highest memory footprint. One possible explanation is that in smaller mazes, BFS efficiently prunes explored nodes, while DFS may encounter high recursion depth and memory overhead.

A* consumed the most memory overall. This is likely due to the cost of maintaining heuristic-related data structures, such as the priority queue (open set) and cost mappings.

The MDP algorithms (Policy and Value Iteration) require memory to store the value function (and policy), essentially an array of size equal to the number of states. For a maze of size $N \times N$, this is $O(N^2)$ space. In our largest case (18×18, 324 states), the value table is very small, and thus Policy and Value Iteration had memory usage on the lower side when compared to A*.

Number of Nodes Expanded

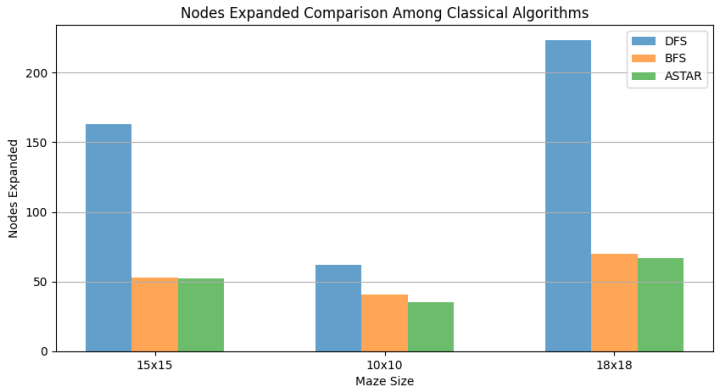

Figure 3: Number of nodes expanded by each search algorithm

This metric counts how many states each algorithm had to process, which is directly tied to the work done by the algorithm. BFS by design explores every reachable node up to the depth of the solution, so its node expansions tend to be high. DFS can have either low or high node expansions depending on the maze layout. Our data reflected this variability: in some 15×15 maze instances, DFS expanded significantly larger nodes than BFS.

A* consistently expanded the fewest nodes among the search algorithms in all our tests. The Manhattan heuristic effectively directs A* toward the goal, so it ignores large portions of the maze that lie far from the direct path.

Maximum Frontier Size During Search

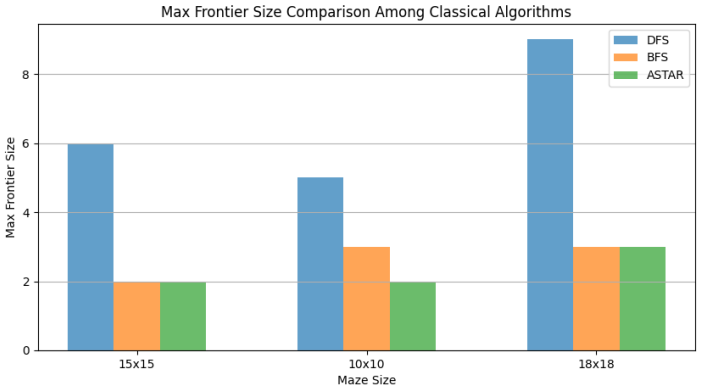

Figure 4: Maximum frontier size during search (BFS, DFS, A)

DFS had the largest frontier sizes among the search methods. The maximum frontier size of BFS was generally equal to that of A*.

Comparison of Policy Iteration and Value Iteration

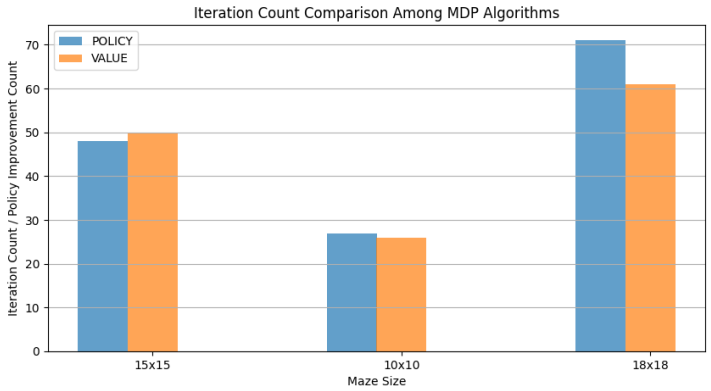

Figure 5: Comparison of Policy Iteration and Value Iteration in terms of convergence.

Value Iteration typically required on the order of tens of iterations to converge to an optimal value function for the maze, from around 25 in smaller mazes up to around 60 in the largest. Policy Iteration took fewer iterations in terms of policy changes but each policy evaluation phase involved iterating over all states until convergence.

In summary, Policy and Value Iteration both successfully computed optimal policies for the mazes, with Value Iteration being somewhat more direct (fewer total iterations) and Policy Iteration providing a more stable convergence in fewer, larger steps.

Conclusion

In analyzing these results, we can draw distinctions between the algorithm categories. The search algorithms (DFS, BFS, A*) focus on finding a single path from start to goal. Among them, BFS and A* guarantee optimality (with A* being generally faster thanks to the heuristic), whereas DFS is non-optimal but sometimes lucky.

The MDP algorithms (Policy Iteration and Value Iteration) on the other hand compute a solution for the entire state space, effectively giving not just one path but an optimal action for every possible state in the maze. The results show that for small mazes, this global approach is quite feasible and can even outrun traditional search.

A Code

Maze Generator

```
import pygame
import random
from queue import PriorityQueue
from collections import deque

# ----------------------------- #
#       Color Definitions       #
# ----------------------------- #
WHITE   = (255, 255, 255)  # Maze walls
BLACK   = (0, 0, 0)        # Background
BLUE    = (0, 0, 255)      # Visited cells during search
ORANGE  = (255, 165, 0)    # Frontier cells (cells to be explored)
GREEN   = (0, 255, 0)      # Final solution path
PURPLE  = (128, 0, 128)    # Currently processing cell

DELAY   = 30             # Delay in milliseconds for animation speed

# ----------------------------- #
#       Maze Cell Class         #
# ----------------------------- #
class Cell:
    def __init__(self, row, col):
        self.row = row
```

```
        self.col = col
        # Each cell has four walls in order: [top, right, bottom, left]
        self.walls = [True, True, True, True]
        self.visited = False   # Used during maze generation

    def draw(self, win, cell_size):
        """Draw the cell walls on the window."""
        x = self.col * cell_size
        y = self.row * cell_size

        # Draw the top wall
        if self.walls[0]:
            pygame.draw.line(win, WHITE, (x, y), (x + cell_size, y), 2)
        # Draw the right wall
        if self.walls[1]:
            pygame.draw.line(win, WHITE, (x + cell_size, y), (x + cell_size, y +
                cell_size), 2)
        # Draw the bottom wall
        if self.walls[2]:
            pygame.draw.line(win, WHITE, (x + cell_size, y + cell_size), (x, y +
                cell_size), 2)
        # Draw the left wall
        if self.walls[3]:
            pygame.draw.line(win, WHITE, (x, y + cell_size), (x, y), 2)

# ---------------------------- #
#          Maze Class          #
# ---------------------------- #
class Maze:
    def __init__(self, rows, cols, cell_size):
        self.rows = rows
        self.cols = cols
        self.cell_size = cell_size
        # Create a 2D grid of cells
        self.grid = [[Cell(r, c) for c in range(cols)] for r in range(rows)]
        self.stack = []  # Stack used for the recursive backtracking algorithm

    def index(self, row, col):
        """Return the cell at (row, col) if within bounds; otherwise return None."""
        if row < 0 or col < 0 or row >= self.rows or col >= self.cols:
            return None
        return self.grid[row][col]

    def get_unvisited_neighbors(self, cell):
        """Return a list of unvisited neighbor cells (top, right, bottom, left)."""
        neighbors = []
        row, col = cell.row, cell.col

        # Check the four directions
        top    = self.index(row - 1, col)
        right  = self.index(row, col + 1)
        bottom = self.index(row + 1, col)
        left   = self.index(row, col - 1)

        if top and not top.visited:
            neighbors.append(top)
        if right and not right.visited:
            neighbors.append(right)
        if bottom and not bottom.visited:
            neighbors.append(bottom)
        if left and not left.visited:
            neighbors.append(left)
        return neighbors

    def remove_walls(self, current, next_cell):
        """Remove the walls between the current cell and the next cell."""
        dx = current.col - next_cell.col
        dy = current.row - next_cell.row
        if dx == 1:  # next_cell is to the left of current
            current.walls[3] = False
            next_cell.walls[1] = False
        elif dx == -1:  # next_cell is to the right of current
            current.walls[1] = False
            next_cell.walls[3] = False
        if dy == 1:  # next_cell is above current
            current.walls[0] = False
```

```
            next_cell.walls[2] = False
        elif dy == -1:  # next_cell is below current
            current.walls[2] = False
            next_cell.walls[0] = False

    def generate_maze(self, win):
        """Generate the maze using recursive backtracking."""
        current = self.grid[0][0]
        current.visited = True
        self.stack.append(current)

        while self.stack:
            current = self.stack[-1]
            neighbors = self.get_unvisited_neighbors(current)

            if neighbors:
                # Choose a random unvisited neighbor
                next_cell = random.choice(neighbors)
                next_cell.visited = True
                # Remove the wall between current and next_cell
                self.remove_walls(current, next_cell)
                self.stack.append(next_cell)
            else:
                self.stack.pop()

            # Optional: Animate the maze generation process
            self.draw(win)
            highlight_cell(win, (current.row, current.col), PURPLE, self.cell_size)
            pygame.display.update()
            pygame.time.delay(DELAY)

        # Reset visited flags so they can be used in the search visualizations
        for row in self.grid:
            for cell in row:
                cell.visited = False

    def draw(self, win):
        """Draw the complete maze (all cells and their walls)."""
        win.fill(BLACK)
        for row in self.grid:
            for cell in row:
                cell.draw(win, self.cell_size)

# ---------------------------- #
#       Helper Functions       #
# ---------------------------- #
def highlight_cell(win, cell_coord, color, cell_size):
    """
    Highlight a cell at the given coordinate with a colored rectangle.
    A small margin is added so the underlying walls are still visible.
    """
    row, col = cell_coord
    x = col * cell_size
    y = row * cell_size
    pygame.draw.rect(win, color, (x + 4, y + 4, cell_size - 8, cell_size - 8))

def get_neighbors_coord(cell_coord, maze):
    """
    Given a cell coordinate (row, col), return a list of neighboring cell coordinates
    that are accessible (i.e. where the wall between them has been removed).
    """
    row, col = cell_coord
    neighbors = []
    cell = maze.grid[row][col]

    # Check top neighbor
    if not cell.walls[0] and row > 0:
        neighbors.append((row - 1, col))
    # Check right neighbor
    if not cell.walls[1] and col < maze.cols - 1:
        neighbors.append((row, col + 1))
    # Check bottom neighbor
    if not cell.walls[2] and row < maze.rows - 1:
        neighbors.append((row + 1, col))
    # Check left neighbor
    if not cell.walls[3] and col > 0:
```

```
        neighbors.append((row, col - 1))
    return neighbors

def reconstruct_path(came_from, start, end):
    """
    Reconstruct the path from start to end using the dictionary that maps each cell
    to the cell it came from.
    """
    path = []
    current = end
    while current is not None:
        path.append(current)
        current = came_from.get(current)
    path.reverse()
    return path
```

DFS

```
import pygame
from maze_generator import highlight_cell, get_neighbors_coord, reconstruct_path, DELAY,
    BLUE, ORANGE, PURPLE, GREEN

def solve_dfs(maze, win):
    """
    Solve the maze using Depth-First Search (DFS) and animate the search.

    Parameters:
        maze : Maze object.
        win  : Pygame window

    Returns a tuple: (steps_taken, nodes_expanded, max_frontier_size), where:
        - steps_taken: number of cells in the final optimal path.
        - nodes_expanded: number of nodes expanded during the search.
        - max_frontier_size: maximum number of nodes in the frontier at any point during
            the search.

    """
    start = (0, 0)
    end = (maze.rows - 1, maze.cols - 1)
    stack = [start]
    came_from = {start: None}
    visited = set()

    nodes_expanded = 0
    max_frontier_size = len(stack)

    while stack:
        current = stack.pop()
        nodes_expanded += 1
        max_frontier_size = max(max_frontier_size, len(stack))
        visited.add(current)

        # Visualize the current state of the search
        maze.draw(win)
        for cell in visited:
            highlight_cell(win, cell, BLUE, maze.cell_size)
        for cell in stack:
            highlight_cell(win, cell, ORANGE, maze.cell_size)
        highlight_cell(win, current, PURPLE, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

        if current == end:
            break

        for neighbor in get_neighbors_coord(current, maze):
            if neighbor not in visited and neighbor not in stack:
                came_from[neighbor] = current
                stack.append(neighbor)
                max_frontier_size = max(max_frontier_size, len(stack))

    # Reconstruct and animate the final solution path
    path = reconstruct_path(came_from, start, end)
    for cell in path:
```

```
        maze.draw(win)
        for path_cell in path:
            highlight_cell(win, path_cell, GREEN, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

    return (len(path), nodes_expanded, max_frontier_size)
```

BFS

```
import pygame
from collections import deque
from maze_generator import highlight_cell, get_neighbors_coord, reconstruct_path, DELAY,
    BLUE, ORANGE, PURPLE, GREEN

def solve_bfs(maze, win):
    """
    Solve the maze using Breadth-First Search (BFS) and animate the search.
    Parameters:
        maze : Maze object.
        win  : Pygame window

    Returns a tuple: (steps_taken, nodes_expanded, max_frontier_size), where:
        - steps_taken: number of cells in the final optimal path.
        - nodes_expanded: number of nodes expanded during the search.
        - max_frontier_size: maximum number of nodes in the frontier at any point during
          the search.
    """
    start = (0, 0)
    end = (maze.rows - 1, maze.cols - 1)
    queue = deque([start])
    came_from = {start: None}
    visited = {start}

    nodes_expanded = 0
    max_frontier_size = len(queue)

    while queue:
        current = queue.popleft()
        nodes_expanded += 1
        max_frontier_size = max(max_frontier_size, len(queue))

        # Visualize the current search state
        maze.draw(win)
        for cell in visited:
            highlight_cell(win, cell, BLUE, maze.cell_size)
        for cell in queue:
            highlight_cell(win, cell, ORANGE, maze.cell_size)
        highlight_cell(win, current, PURPLE, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

        if current == end:
            break

        for neighbor in get_neighbors_coord(current, maze):
            if neighbor not in visited:
                visited.add(neighbor)
                came_from[neighbor] = current
                queue.append(neighbor)
                max_frontier_size = max(max_frontier_size, len(queue))

    # Reconstruct and animate the final solution path
    path = reconstruct_path(came_from, start, end)
    for cell in path:
        maze.draw(win)
        for path_cell in path:
            highlight_cell(win, path_cell, GREEN, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

    return (len(path), nodes_expanded, max_frontier_size)
```

A*

```
import pygame
from queue import PriorityQueue
from maze_generator import BLUE, ORANGE, PURPLE, GREEN, DELAY, highlight_cell,
    get_neighbors_coord, reconstruct_path

def heuristic(a, b):
    """Manhattan distance heuristic for A* search."""
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

def solve_astar(maze, win):
    """
    Solve the maze using the A* search algorithm and animate the process.
    Uses the Manhattan distance as the heuristic.

    Parameters:
        maze : Maze object.
        win  : Pygame window

    Returns a tuple: (steps_taken, nodes_expanded, max_frontier_size), where:
        - steps_taken: number of cells in the final optimal path.
        - nodes_expanded: number of nodes expanded during the search.
        - max_frontier_size: maximum number of nodes in the frontier at any point during
            the search.
    """
    start = (0, 0)
    end = (maze.rows - 1, maze.cols - 1)
    open_set = PriorityQueue()
    open_set.put((0, start))
    came_from = {}

    # Initialize g_score and f_score dictionaries
    g_score = { (r, c): float('inf') for r in range(maze.rows) for c in range(maze.cols)
        }
    g_score[start] = 0
    f_score = { (r, c): float('inf') for r in range(maze.rows) for c in range(maze.cols)
        }
    f_score[start] = heuristic(start, end)

    open_set_hash = {start}
    visited = set()

    nodes_expanded = 0
    max_frontier_size = len(open_set_hash)

    while not open_set.empty():
        current = open_set.get()[1]
        open_set_hash.remove(current)
        nodes_expanded += 1
        max_frontier_size = max(max_frontier_size, len(open_set_hash))
        visited.add(current)

        # Visualize the A* search state
        maze.draw(win)
        for cell in visited:
            highlight_cell(win, cell, BLUE, maze.cell_size)
        for cell in open_set_hash:
            highlight_cell(win, cell, ORANGE, maze.cell_size)
        highlight_cell(win, current, PURPLE, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

        if current == end:
            break

        for neighbor in get_neighbors_coord(current, maze):
            tentative_g = g_score[current] + 1  # Cost of 1 for each move
            if tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic(neighbor, end)
                if neighbor not in open_set_hash:
                    open_set.put((f_score[neighbor], neighbor))
```

```
                    open_set_hash.add(neighbor)
                    max_frontier_size = max(max_frontier_size, len(open_set_hash))

    # Reconstruct and animate the final solution path
    path = reconstruct_path(came_from, start, end)
    for cell in path:
        maze.draw(win)
        for path_cell in path:
            highlight_cell(win, path_cell, GREEN, maze.cell_size)
        pygame.display.update()
        pygame.time.delay(DELAY)

    return (len(path), nodes_expanded, max_frontier_size)
```

Value Iteration

```
import math
import pygame
from maze_generator import highlight_cell, DELAY, GREEN
from utils import get_possible_actions, extract_policy_path

def value_iteration(maze, win, gamma=0.9, theta=1e-4):
    """
    Solve the maze using Value Iteration.

    Parameters:
      maze   : Maze object (with attributes rows, cols, and grid where each cell has
          .walls)
      gamma  : Discount factor.
      theta  : Convergence threshold.
      win    : Pygame window

    Returns:
      A tuple (steps_taken, iter_count, 0) where:
        - steps_taken: number of cells in the final optimal path.
        - iter_count: number of full state sweeps (iterations) until convergence.
        - The third metric is 0 (not used for value iteration).

    Reward for each move is -1; terminal state is (maze.rows-1, maze.cols-1).
    """
    # Initialize V(s)=0 for all states.
    V = {}
    states = []
    for r in range(maze.rows):
        for c in range(maze.cols):
            state = (r, c)
            V[state] = 0.0
            states.append(state)
    terminal = (maze.rows - 1, maze.cols - 1)

    iter_count = 0
    while True:
        iter_count += 1
        delta = 0
        for state in states:
            if state == terminal:
                continue
            actions = get_possible_actions(maze, state)
            if not actions:
                continue
            v = V[state]
            Q_values = []
            for (a, next_state) in actions:
                Q_values.append(-1 + gamma * V[next_state])
            V[state] = max(Q_values)
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break

    # Derive optimal policy.
    policy = {}
    for state in states:
        if state == terminal:
            policy[state] = None
```

```
            continue
        actions = get_possible_actions(maze, state)
        if not actions:
            policy[state] = None
            continue
        best_action = None
        best_value = -math.inf
        for (a, next_state) in actions:
            q = -1 + gamma * V[next_state]
            if q > best_value:
                best_value = q
                best_action = a
        policy[state] = best_action

    # Animate the optimal path if a window is provided.
    path = extract_policy_path(policy, maze)
    if win is not None:
        for state in path:
            maze.draw(win)
            highlight_cell(win, state, GREEN, maze.cell_size)
            pygame.display.update()
            pygame.time.delay(DELAY)
    return len(path), iter_count, 0
```

Policy Iteration

```
import math
import pygame
from maze_generator import highlight_cell, DELAY, GREEN
from utils import get_possible_actions, extract_policy_path

def policy_iteration(maze, win, gamma=0.9, theta=1e-4):
    """
    Solve the maze using Policy Iteration.

    Parameters:
      maze   : Maze object.
      gamma  : Discount factor.
      theta  : Convergence threshold for policy evaluation.
      win    : Pygame window

    Returns:
      A tuple (steps_taken, policy_improvement_count, total_evaluation_iterations) where:
        - steps_taken: number of cells in the final optimal path.
        - policy_improvement_count: number of times the policy was improved.
        - total_evaluation_iterations: total sweeps performed during all policy
            evaluations.

    Reward for each move is -1; terminal state is (maze.rows-1, maze.cols-1).
    """
    states = []
    for r in range(maze.rows):
        for c in range(maze.cols):
            states.append((r, c))
    # Initialize arbitrary policy and value function.
    policy = {}
    V = {}
    for state in states:
        V[state] = 0.0
        if state == (maze.rows - 1, maze.cols - 1):
            policy[state] = None
        else:
            actions = get_possible_actions(maze, state)
            policy[state] = actions[0][0] if actions else None
    terminal = (maze.rows - 1, maze.cols - 1)

    total_evaluation_iterations = 0
    policy_improvement_count = 0

    def policy_evaluation(policy, V):
        eval_iterations = 0
        while True:
            eval_iterations += 1
            delta = 0
```

```
            for state in states:
                if state == terminal:
                    continue
                a = policy[state]
                actions = get_possible_actions(maze, state)
                next_state = None
                for (act, ns) in actions:
                    if act == a:
                        next_state = ns
                        break
                if next_state is None:
                    continue
                v = V[state]
                V[state] = -1 + gamma * V[next_state]
                delta = max(delta, abs(v - V[state]))
            if delta < theta:
                break
        return eval_iterations

    stable = False
    while not stable:
        eval_iters = policy_evaluation(policy, V)
        total_evaluation_iterations += eval_iters
        policy_improvement_count += 1
        stable = True
        for state in states:
            if state == terminal:
                continue
            actions = get_possible_actions(maze, state)
            if not actions:
                continue
            old_action = policy[state]
            best_action = None
            best_value = -math.inf
            for (a, next_state) in actions:
                q = -1 + gamma * V[next_state]
                if q > best_value:
                    best_value = q
                    best_action = a
            if best_action != old_action:
                policy[state] = best_action
                stable = False

    # Animate the optimal path.
    path = extract_policy_path(policy, maze)
    if win is not None:
        for state in path:
            maze.draw(win)
            highlight_cell(win, state, GREEN, maze.cell_size)
            pygame.display.update()
            pygame.time.delay(DELAY)
    return len(path), policy_improvement_count, total_evaluation_iterations
```