UNIVERSITY OF SCIENCE
VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

PROJECT 1

Ares's adventure

CS420 – Artificial Intelligence

Team member

22125010 – Phan Phuc Bao

22125072 – Vo Thinh Phat

22125085 – O Hon Sam

22125103 – Nguyen Vo Hoang Thong

Theoretical Teacher : Nguyen Ngoc Thao

Laboratory Teacher : Nguyen Thanh Tinh

Ho Chi Minh City, 10/11/2024

MEMBERS

Student ID	Full name	Contribution Percentage
22125010	Phan Phuc Bao	25%
22125072	Vo Thinh Phat	25%
22125085	O Hon Sam	25%
22125103	Nguyen Vo Hoang Thong	25%

Work assignment table

Student name	Task	Completion rate
Phan Phuc Bao	Implement A* search algorithm	100%
Vo Thinh Phat	Design and implement UI for visualization	100%
O Hon Sam	Implement BFS and DFS search algorithm	100%
Nguyen Vo Hoang Thong	Implement UCS search algorithm	100%

Demonstration Video

Please click here: https://youtu.be/ogv-GJ0ibL4

Backup link: https://youtu.be/aFlg82uCB0g

Contents

1 Overview of Self-Evaluation

This section provides a reflective analysis of our project outcomes in solving "Ares's Adventure" using various search algorithms. The team's performance is evaluated based on our ability to meet the project requirements, including implementing Breadth-First Search, Depth-First Search, Uniform Cost Search, and A* Search, as well as generating unique test cases. Through this evaluation, we identify both our successes and areas for improvement, particularly regarding algorithm efficiency, code structure, and GUI functionality. Additionally, our report discusses the challenges encountered and insights gained from applying these algorithms to complex scenarios.

2 Define Search Problem

We noticed that Ares's adventure is actually a variation of a famous game called **Sokoban**. A more interesting characteristic of this game compared with Sokoban is stone weight, while boxes in Sokoban has no impact to the action cost. In other words, Sokoban can be seen as Ares's adventure problem with each box's weight set to zero.

2.1 State

In state representation for a search algorithm, it's ideal to include only the variables that actively change throughout the search, as these define the different possible configurations.

- Elements that remain constant across all states like stone weights, the positions of walls, and switches in the maze are parts of the environment.
- Each state consists of:
  - Ares's position: (x, y) coordinates of Ares's location.
  - Stone Positions: A list of (x, y) coordinates for each stone, representing the stone's current locations.
- Note: For consistency, we use Ox to represent the horizontal axis and Oy to represent the vertical axis.

2.2 State Space

The state space in Ares's adventure defines the set of all possible configurations that can be achieved through valid moves within the game environment.

2.3 Initial State

The starting position of Ares, stones, and switches are defined by the input file.

2.4 Actions

Ares can move in four directions (Up, Down, Left, Right) and can push a stone if the next cell of the stone in the same moving direction is *open* (*open* refers to a cell that is not occupied by a wall or a stone).

2.5 Transition Model

A new state resulting from applying only valid moves (actions): Ares can move to adjacent empty cells or push a stone into an open cell, which is neither a wall nor an occupied cell by another stone.

2.6 Goal States

The goal state in this game is to have all the stones placed on switches, regardless of the stones' weights or the specific switch they are on. This means any stone can activate any switch. So, if you have n stones and n switches, there are possible $n!$ goal states.

For example, if there are 3 stones and 3 switches, you could have $3! = 6$ different combinations of which stone is on which switch, and all of those combinations would represent a successfully solved puzzle.

2.7 Action Cost

- Moving without pushing stone costs 1.
- Pushing a stone costs (1 + weight of the stone)

3 Detailed Explanation of Each Search Algorithm

In this section, we analyze four search algorithms used to solve the Ares's Adventure problem: Breadth-First Search (BFS), Depth-First Search (DFS), Uniform Cost Search (UCS), and A* Search. The core principles of each algorithm are discussed individually, while shared utilities and problem-specific challenges related to Ares's Adventure are addressed in a separate section. Overall, the standard versions of these algorithms perform effectively. The main challenge lies in adapting and optimizing them for Ares's Adventure-specific requirements, including detecting valid moves, identifying deadlocks, implementing pruning for performance enhancement, and selecting appropriate cost and heuristic functions for UCS and A*.

3.1 BFS Algorithm

3.1.1 Introduction

- BFS (Breadth-First Search) is an **uninformed search algorithm**, which is given no clue about how close a state is to the goal(s). The agent can just generate successors and distinguish a goal state from a non-goal state [1].

- BFS applies **early goal test** to check whether a node is a solution as soon as it is generated [1].
- **Idea:** Expand the root node first, then all the successors of the root node are next, then their successors, and so on [1].
- **Completeness**: Yes
- **Optimality**: Yes (for problems where all actions have the same cost)
- **Time Complexity** and **Space Complexity** are $\mathcal{O}(b^d)$ where every state has b successors and the solution is at depth d.

3.1.2 Pseudo Code

Algorithm 1 Breadth-First Search (BFS)

```
function BREADTH-FIRST-SEARCH(problem)
    node ← NODE(problem.INITIAL)
    if problem.IS-GOAL(node.STATE) then
        return node
    end if
    frontier ← a FIFO queue with node as an element
    reached ← {problem.INITIAL}
    while not IS-EMPTY(frontier) do
        node ← POP(frontier)                    ▷ Chooses the shallowest node
        for each child in EXPAND(problem, node) do
            s ← child.STATE
            if s is not in reached then
                if problem.IS-GOAL(s) then
                    return child
                end if
                add s to reached
                add child to frontier
            end if
        end for
    end while
    return failure
end function
```

3.1.3 Algorithm Implementation

Initialization

- In the pseudocode, the function initializes the node with the initial problem state. Therefore, our BFS class initializes the algorithm with the input file and retrieves the start state using `self.get_start_state(input_file)`.
- The function `get_start_state()` in the code returns the 2D maze, identifying the positions of Ares, stones, switches and assigns stone weights.

Main Loop: `while not IS-EMPTY(frontier)`

- In the pseudocode, a loop continues as long as there are nodes in the frontier. Each node is dequeued to explore the shallowest node first by using the First-In-First-Out (FIFO) order.
- The code mirrors this by initializing a queue (deque from collections library) with the `start_state` variable including (Ares position, stone positions) and iterates using `queue.popleft()` to process nodes in a BFS manner.

Goal Test: `if problem.IS-GOAL(s)`

- The pseudocode checks if child node of the current state satisfies the goal criteria.
- In the code, this is handled by checking if all stones are on switches, and if so, the `reconstruct_path()` function is called to trace back the solution path and calculate costs.

State Expansion: `for each child in EXPAND(problem, node)`

- The pseudocode expands each node by generating child nodes, adding them to the frontier if they haven't been reached.
- In the code, `get_neighbors()` generates neighboring states, checking valid moves as shown in Listing (e.g., preventing pushing a stone into a wall) and deadlocks then adding them to the `visited` set to avoid revisiting nodes. We implement BFS in graph-search approach.
- To ensure a unique state representation, we apply a **canonical form** by sorting the stone positions in each state. This approach treats any permutation of the same stone configuration (e.g., positions ((1, 4), (2, 3), (2, 5)) vs. ((2, 3), (1, 4), (2, 5))) as identical. Sorting standardizes the order, preventing duplicate explorations of states that are functionally equivalent but arranged differently. This reduces redundancy and optimizes the algorithm's efficiency.

Invalid Moves and Deadlock Detection These are add-on utility functions to improve searching process efficiency by pruning a large number of unnecessary states.

- Function `is_valid_move()` as shown in Listing 1 ensure Ares can move to adjacent empty cells or push a stone into an open cell, which is neither a wall nor an occupied cell by another stone.
- Function `is_deadlock()` indicated in Listing 2 prevents adding states that would be unsolvable (e.g., stones in corners in Listing 3 or against walls without any valid moves in Listing 4).

Path Reconstruction

- `reconstruct_path()` traverses back using `parent_map` and appends each action to the final path.

Tracking Costs and Results

- The code uses `find_cost_each_step()` to calculate the cost, accounting for movement and pushing stones with weight costs. The algorithm also tracks memory and runtime using tracemalloc and psutil libraries.

3.2 DFS Algorithm

3.2.1 Introduction

- DFS (Depth-First Search) is an **uninformed search algorithm**, which is given no clue about how close a state is to the goal(s). The agent can just generate successors and distinguish a goal state from a non-goal state [1].
- We apply **early goal test** for DFS to check whether a node is a solution as soon as it is generated [1].
- **Idea:** Always expand the deepest node in the frontier first [1].
- **Completeness**: No
- **Optimality**: No
- **Time Complexity**: $\mathcal{O}(b^m)$ and **Space Complexity** are $\mathcal{O}(b \cdot m)$ where every state has b successors and the solution found is at depth m.

3.2.2 Pseudo Code

Algorithm 2 Depth-First Search (DFS)

```
1:  function DEPTH-FIRST-SEARCH(problem)
2:      node ← NODE(problem.INITIAL)
3:      if problem.IS-GOAL(node.STATE) then
4:          return node
5:      end if
6:      stack ← a LIFO stack with node as an element
7:      visited ← {problem.INITIAL}
8:      while not IS-EMPTY(stack) do
9:          node ← POP(stack)                    ▷ Chooses the deepest node
10:         for each child in EXPAND(problem, node) do
11:             s ← child.STATE
12:             if s is not in visited then
13:                 if problem.IS-GOAL(s) then
14:                     return child
15:                 end if
16:                 add s to visited
17:                 add child to stack
18:             end if
19:         end for
20:     end while
21:     return failure
22: end function
```

3.2.3 Algorithm Implementation

Initialization

- In the pseudocode, the function initializes the node with the initial problem state. Therefore, our DFS class initializes the algorithm with the input file and retrieves the start state using `self.get_start_state(input_file)`.
- The function `get_start_state()` in the code returns the 2D maze, identifying the positions of Ares, stones, switches and assigns stone weights.

Main Loop: `while not IS-EMPTY(frontier)`

- In the pseudocode, a loop continues as long as there are nodes in the frontier. Each unexplored node is pushed into a stack. The algorithm pops the most recent state from stack to explore the deepest node first by using the Last-In-First-Out (LIFO) order.
- The code mirrors this by initializing a stack (primitive list data structure in Python) with the `start_state` and iterates using `stack.pop()` to process nodes in a DFS manner.

Goal Test: `if problem.IS-GOAL(s)`

- The pseudocode checks if child node of the current state satisfies the goal criteria.
- In the code, this is handled by checking if all stones are on switches, and if so, the `reconstruct_path()` function is called to trace back the solution path and calculate costs.

State Expansion: `for each child in EXPAND(problem, node)`

- The pseudocode expands each node by generating child nodes, adding them to the frontier if they haven't been reached.
- In the code, `get_neighbors()` generates neighboring states, checking valid moves as shown in Listing (e.g., preventing pushing a stone into a wall) and deadlocks then adding them to the 'visited' set to avoid revisiting nodes. We implement DFS in graph-search approach.
- To ensure a unique state representation, we apply a **canonical form** by sorting the stone positions in each state. This approach treats any permutation of the same stone configuration (e.g., positions ((1, 4), (2, 3), (2, 5)) vs. ((2, 3), (1, 4), (2, 5))) as identical. Sorting standardizes the order, preventing duplicate explorations of states that are functionally equivalent but arranged differently. This reduces redundancy and optimizes the algorithm's efficiency.

Invalid Moves and Deadlock Detection

- Function `is_valid_move()` as shown in Listing 1 ensure Ares can move to adjacent empty cells or push a stone into an open cell, which is neither a wall nor an occupied cell by another stone.

- Function `is_deadlock()` indicated in Listing 2 prevents adding states that would be unsolvable (e.g., stones in corners in Listing 3 or against walls without any valid moves in Listing 4).

Path Reconstruction

- `reconstruct_path()` traverses back using `parent_map` and appends each action to the final path.

Tracking Costs and Results

- The code uses `find_cost_each_step()` to calculate the cost, accounting for movement and pushing stones with weight costs. The algorithm also tracks memory and runtime using tracemalloc and psutil. libraries.

3.3 UCS Algorithm

3.3.1 Introduction

- UCS (Uniform Cost Search) is an **uninformed search algorithm** that expands the least-cost path from the root node to the goal state [1].
- UCS takes into account the cost of each action, making it suitable for problems where path costs vary, such as in Ares Adventure, where moving and pushing actions incur different costs.
- **Idea:** Expand nodes in order of their cumulative cost from the root, using a priority queue [1].
- **Completeness**: Yes, if the cost of every action is positive.
- **Optimality**: Yes, as UCS always finds the least-cost path to the goal.
- **Time Complexity** and **Space Complexity** are $\mathcal{O}(b^{\lceil C^*/\epsilon \rceil})$, where C^* is the cost of the optimal solution and ϵ is the minimum cost of any action.

3.3.2 Pseudo Code

Algorithm 3 Uniform Cost Search (UCS)

```
1: function UNIFORM-COST-SEARCH(problem)
2:     node ← NODE(problem.INITIAL, 0)
3:     if problem.IS-GOAL(node.STATE) then
4:         return node
5:     end if
6:     frontier ← a priority queue ordered by path cost with node as an element
7:     reached ← {problem.INITIAL}
8:     while not IS-EMPTY(frontier) do
9:         node ← POP(frontier)                    ▷ Chooses node with the lowest cost
10:        if problem.IS-GOAL(node.STATE) then
11:            return node
12:        end if
13:        for each child in EXPAND(problem, node) do
14:            s ← child.STATE
15:            cost ← node.PATH-COST + ACTION-COST(node, child) ▷ Includes move and
   push costs in Ares Adventure
16:            if s is not in reached or cost < PATH-COST(s) in frontier then
17:                reached[s] ← cost
18:                add child to frontier with priority cost
19:            end if
20:        end for
21:    end while
22:    return failure
23: end function
```

3.3.3 Algorithm Implementation

Initialization

- In the UCS pseudocode, the algorithm initializes the root node with the initial problem state and a path cost of zero. The UCS class initializes by reading the input file, then retrieves the starting positions of Ares and stones through the `get_start_state()` function.
- The function `get_start_state()` provides the initial 2D maze layout, determining the positions of Ares, stones, switches, and assigning weights to each stone.

Main Loop: `while not IS-EMPTY(frontier)`

- The main loop in UCS dequeues the node with the lowest path cost from the priority queue. By doing so, UCS ensures that the algorithm always expands the lowest cumulative cost path first.
- The code mirrors this behavior by using Python's heapq library to manage the priority queue, with states dequeued in order of ascending path cost.

Goal Test: if `problem.IS-GOAL(node.STATE)`

- In the UCS pseudocode, the goal test checks if the current node's state meets the goal conditions.
- The UCS implementation performs this test by verifying if all stones occupy the switch positions. Upon success, `reconstruct_path()` is invoked to generate the solution path and compute the cumulative cost.

State Expansion: `for(each child in EXPAND(problem, node)`

- UCS expands each node by generating its child nodes, computing the path cost for each child. If a child has a new or lower cost than previously recorded, it is added to the priority queue.
- The code's `get_neighbors()` function generates neighboring states based on valid moves. Only feasible moves, such as pushing a stone into an empty cell, are included, and deadlocked states are avoided. This approach ensures efficient exploration by avoiding revisiting or expanding states in deadlock positions.

Cost Calculation

- UCS calculates the cost of each move, adding both movement costs and push costs. In the case of pushing, the cost is computed as the weight of the stone plus one additional cost unit for the move itself.
- This is achieved in the code with a `find_cost_each_step()` function, which determines and returns the appropriate cost based on the type of action (move or push) and the stone's weight.

Duplicate Detection and Path Cost Comparison

- The pseudocode includes a condition to verify if a state should be added to the frontier based on the new path cost. If a state has been reached previously but has a higher cost, the new lower-cost path is used instead.
- The implementation keeps a reached dictionary to store the lowest cost for each state, replacing entries if a new, lower-cost path is found. This strategy prevents redundant expansions of previously reached nodes, improving efficiency.

Invalid Moves and Deadlock Detection

- The way of checking invalid moves and deadlock detection is the same as the above search algorithms.
- Function `is_valid_move` as shown in Listing 1 ensure Ares can move to adjacent empty cells or push a stone into an open cell, which is neither a wall nor an occupied cell by another stone.
- Function `is_deadlock()` indicated in Listing 2 prevents adding states that would be unsolvable (e.g., stones in corners in Listing 3 or against walls without any valid moves in Listing 4).

Path Reconstruction

- The UCS code implementation includes a `reconstruct_path()` function, which backtracks from the goal state to the root, accumulating actions to create the solution path.

Tracking Costs and Results

- Throughout the UCS algorithm, memory and runtime usage are monitored using tracemalloc and psutil. These tools help track the memory footprint and processing time, offering insight into the algorithm's performance.
- The final output includes the total cost of the solution path, comprising all movement and push costs, as well as a detailed list of actions taken to achieve the solution.

3.4 A* Search Algorithm

3.4.1 Introduction

- A* is an **informed search algorithm** that extends UCS by combining the actual cost to reach a node with an estimated cost to reach the goal [1].
- Unlike UCS, A* uses a heuristic function to prioritize nodes that appear closer to the goal, which makes it efficient for solving Sokoban variants like Ares Adventure.
- **Idea:** Expand nodes in order of their total estimated cost $f(n) = g(n) + h(n)$, where $g(n)$ is the path cost and $h(n)$ is the heuristic cost [1].
- **Completeness**: Yes, if the heuristic is admissible and every action cost is positive.
- **Optimality**: Yes, if the heuristic is admissible (never overestimates the cost to the goal).
- **Time Complexity** and **Space Complexity** are $\mathcal{O}(b^{\lceil C^*/\epsilon \rceil})$, where C^* is the optimal solution cost and ϵ is the minimum cost of any action.

3.4.2 Pseudo Code

Algorithm 4 A* Search

```
1: function A-STAR-SEARCH(problem)
2:     node ← NODE(problem.INITIAL, 0)
3:     if problem.IS-GOAL(node.STATE) then
4:         return node
5:     end if
6:     frontier ← a priority queue ordered by total cost f(n) = g(n) + h(n) with node as an element
7:     reached ← {problem.INITIAL : 0}
8:     while not IS-EMPTY(frontier) do
9:         node ← POP(frontier)                    ▷ Chooses node with the lowest total cost f(n)
10:        if problem.IS-GOAL(node.STATE) then
11:            return node
12:        end if
13:        for each child in EXPAND(problem, node) do
14:            s ← child.STATE
15:            cost ← node.PATH-COST + ACTION-COST(node, child)    ▷ Includes movement and pushing costs in Ares Adventure
16:            total-cost ← cost + HEURISTIC(s, problem.GOAL)    ▷ Manhattan Distance in the case of Ares Adventure
17:            if s is not in reached or cost < PATH-COST(s) in frontier then
18:                reached[s] ← total-cost
19:                add child to frontier with priority total-cost
20:            end if
21:        end for
22:    end while
23:    return failure
24: end function
```

3.4.3 Algorithm Implementation

Initialization

- A* initializes similarly to above search algorithms, setting up a root node with an initial path cost of zero and computing the heuristic function. The A* class begins by loading the input data and setting the start state using `get_start_state()`, which identifies the positions of Ares, stones, and switches.
- The `get_start_state()` function provides a 2D maze representation and initializes each stone's weight, which contributes to the overall movement cost.

Main Loop: `while(not IS-EMPTY(frontier)`

- The main loop in A* dequeues nodes based on their combined path and heuristic cost $f(n)$, ensuring that the algorithm prioritizes paths expected to be closer to the goal.

- The code implements this using a priority queue, managed by Python's heapq library, to dequeue states with the lowest total estimated cost, combining actual cost and heuristic value.

Goal Test: `if problem.IS-GOAL(node.STATE)`

- The A* pseudocode checks for the goal state immediately after dequeuing a node, which optimizes goal detection in the search.
- In the implementation, the goal test checks if all stones are on their designated switches. Upon success, `reconstruct_path()` is called to trace back and return the solution path with total costs.

State Expansion: `for each child in EXPAND(problem, node)`

- Each node's state is expanded by generating children, calculating the actual movement cost and adding the heuristic cost to reach the goal from each child node.
- The code's `get_neighbors()` function generates valid moves for Ares, such as pushing stones into open spaces. Deadlocks and invalid moves are filtered out to improve search efficiency, avoiding any unnecessary expansions.

Cost Calculation and Priority Queue Update

- A* calculates the actual movement cost based on whether Ares is moving or pushing a stone. For pushes, the cost is derived from the stone's weight plus an additional cost unit.
- States are added to the priority queue with their total cost $f(n) = g(n) + h(n)$. The code maintains a reached dictionary to store and update the lowest total cost for each state, replacing entries in cases of lower-cost paths.

Invalid Moves and Deadlock Detection

- The way of checking invalid moves and deadlock detection is the same as the above search algorithms.
- Function `is_valid_move()` as shown in Listing 1 ensure Ares can move to adjacent empty cells or push a stone into an open cell, which is neither a wall nor an occupied cell by another stone.
- Function `is_deadlock()` indicated in Listing 2 prevents adding states that would be unsolvable (e.g., stones in corners in Listing 3 or against walls without any valid moves in Listing 4).

Path Reconstruction

- Similar to above search algorithms, the A* implementation reconstructs the solution path by backtracking from the goal state to the start state using a parent map, capturing the sequence of actions required to achieve the solution.

Tracking Costs and Results

- During A* search, memory and processing time are monitored using tracemalloc and psutil libraries, providing a measure of the algorithm's efficiency.
- The final output includes the total cost of the solution path, consisting of all movement and push costs, as well as a list of actions taken to reach the goal.

3.4.4 Heuristic function analysis

In this section, we present an in-depth analysis of the heuristic function, the key component that distinguishes A* from other search algorithms, particularly Uniform Cost Search (UCS). Our analysis focuses on the rationale behind selecting this specific heuristic function for the Ares adventure puzzle, combined with a comprehensive evaluation of its advantages and limitations in practice. Through this analysis, we aim to provide a thorough understanding of how this heuristic mechanism influences the algorithm's performance and solution quality.

Pseudocode

Algorithm 5 Heuristic Function

```
1: function HEURISTIC(state)
2:     ares_position, stone_positions ← state
3:     stone_weights ← START-STATE[stone_weights]
4:     min_total_distance ← 0
       Calculate minimum distance from each stone to any switch
5:     for each stone, weight in stone_positions, stone_weights do
6:         min_switch_distance ← ∞
7:         for each switch in START-STATE[switches] do
8:             distance ← |stone.x − switch.x| + |stone.y − switch.y|
9:             min_switch_distance ← MIN(min_switch_distance, distance)
10:        end for
11:        min_total_distance ← min_total_distance + min_switch_distance × weight
12:    end for
       Calculate minimum distance from Ares to any stone
13:    min_ares_distance ← ∞
14:    for each stone in stone_positions do
15:        distance ← |ares_position.x − stone.x| + |ares_position.y − stone.y|
16:        min_ares_distance ← MIN(min_ares_distance, distance)
17:    end for
18:    return min_total_distance + min_ares_distance
19: end function
```

The heuristic function is composed of 2 keys components:

- **Stone-to-Switch Distance:** each stone is found the Manhattan distance to its nearest switch. Then, the result is multiplied by the stone's weight to account for pushing cost. Finally, getting the total value of results calculated for each stone.
- **Ares-to-Stone Distance:** calculate minimum Manhattan distance from Ares to any stone.

After calculating two factors, we plus it together and finish the heuristic equation.

Benefits

- Stone-centric approach: The heuristic function focuses on the primary goal of getting stones to switches, weights stone movements appropriately based on their costs.
- Player movement integration: the heuristic function includes player position in the estimation, guides player towards useful positions and balances stone movement with player positioning.
- Multi-goal handling: effectively handles multiple stones and switches, allows flexible matching of stones to switches.

Problems

Using Manhattan distance typically ensures a heuristic function never overestimates. However, in Ares adventure, we must also consider stone weights. Adding these weights to our heuristic calculations could risk overestimating true costs, potentially making the function inadmissible.

Conclusion

Despite this concern, our test results in the next part show that our heuristic function works well in practice. It successfully handles both stone weights and distances to switches, helping A* find good solutions efficiently.

4 Utilities functions

4.1 Valid Move Detection

```
1  def is_valid_move(self, position, stone_positions):
2      # 'position' argument can be Ares's position or a
           stone's position
3      x, y = position
4
5      # Prevent Ares from moving into wall
6      if self.start_state['maze'][y][x] == '#':
7          return False
8
9      # Prevent Ares from pushing a stone into another stone
10     if position in stone_positions:
11         return False
12
13     return True
```

Listing 1: Valid move detection function

- Valid moves include:
  - Simple moves with 4 directions (up, left, down , right).
  - Push moves (when Ares is adjacent to a stone).

- Each move will check:
  - Wall collisions.
  - Stone-stone collisions.

4.2 Deadlock Detection

The algorithm's deadlock detection prevents:

```
def is_deadlock(self, stone_positions):
    stones_set = set(stone_positions) # convert to set for
       faster lookup, average time complexity O(1)
    maze = self.start_state['maze']

    for stone in stone_positions:
        x, y = stone

        if (maze[y][x] == '.'):  # Stone is in a switch
           position
            return False

        if self.is_corner_deadlock(x, y, maze):
            return True

        if self.is_wall_deadlock(x, y, maze, stones_set):
            return True

    return False
```

Listing 2: General deadlock detection function

- Corner Deadlocks: if a stone is pushed into a corner without any switches nearby, it's impossible to move it out, making it unsolvable [3] → prune this state.

```
def is_corner_deadlock(self, x, y, maze):
    """Check if stone is stuck in a corner formed by
       walls."""
    corners = [
        ((x-1, y), (x, y-1)),   # Top-left
        ((x+1, y), (x, y-1)),   # Top-right
        ((x-1, y), (x, y+1)),   # Bottom-left
        ((x+1, y), (x, y+1))    # Bottom-right
    ]

    for (x1, y1), (x2, y2) in corners:
        if maze[y1][x1] == '#' and maze[y2][x2] == '#':
            return True
    return False
```

Listing 3: Corner deadlock detection function

- Wall Deadlocks: when a stone is placed against a wall and is blocked by other stones which are also against a wall, it creates a deadlock because the stone can't be moved [3] → prune this state.

```python
def is_wall_deadlock(self, x, y, maze, stones_set):
    """Check if stone is stuck against a wall with no path to any switch."""
    if maze[y][x-1] == '#' or maze[y][x+1] == '#':  # Horizontal wall
        # Check if stone is blocked vertically by other stones
        above_blocked = (x, y-1) in stones_set and (maze[y-1][x-1] == '#' or maze[y-1][x+1] == '#')
        below_blocked = (x, y+1) in stones_set and (maze[y+1][x-1] == '#' or maze[y+1][x+1] == '#')
        if above_blocked or below_blocked:
            return True

    if maze[y-1][x] == '#' or maze[y+1][x] == '#':  # Vertical wall
        # Check if stone is blocked horizontally by other stones
        left_blocked = (x-1, y) in stones_set and (maze[y-1][x-1] == '#' or maze[y+1][x-1] == '#')
        right_blocked = (x+1, y) in stones_set and (maze[y-1][x+1] == '#' or maze[y+1][x+1] == '#')
        if left_blocked or right_blocked:
            return True

    return False
```

Listing 4: Wall deadlock detection function

5 Experimental Result

5.1 Result Summary

5.1.1 Experiment Result of Each Test Case

Table 1: Algorithm Performance on Test Case Input 1

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 1	BFS	**27**	**132**	57,870	966.73	16.21
	DFS	3,069	8,334	83,244	1,290.60	21.81
	UCS	**27**	**132**	162,703	2,914.44	50.48
	A*	**27**	**132**	**7,457**	**268.63**	**2.86**

Table 2: Algorithm Performance on Test Case Input 2

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 2	BFS	**37**	**94**	6,318	129.0095	1.7987
	DFS	149	407	6,564	86.6289	1.8032
	UCS	**37**	**94**	17,748	261.2290	5.0525
	A*	**37**	**94**	**2,509**	**64.5716**	**0.8967**

Table 3: Algorithm Performance on Test Case Input 3

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 3	BFS	**61**	98	99,537	1,519.3130	27.3401
	DFS	107	160	73,004	**990.1822**	**16.6037**
	UCS	63	**92**	218,876	3,817.3518	70.0465
	A*	63	**92**	**69,116**	2,672.473	21.3088

Table 4: Algorithm Performance on Test Case Input 4

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 4	BFS	**76**	244	809,974	11,441.4740	211.5737
	DFS	32,492	61,024	**798,895**	**11,340.4214**	**210.3613**
	UCS	96	**196**	11,271,286	498,597.9238	4,070.1619
	A*	96	**196**	3,855,371	158,110.5044	2,726.7496

Table 5: Algorithm Performance on Test Case Input 5

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 5	BFS	**95**	**294**	21,596	330.2922	6.6210
	DFS	157	409	**16,558**	**250.8879**	**3.8706**
	UCS	**95**	**294**	105,719	1,774.6058	33.6426
	A*	**95**	**294**	91,015	3,146.0078	32.4981

Table 6: Algorithm Performance on Test Case Input 6

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 6	BFS	**120**	**162**	1,656,989	23,340.1673	436.9743
	DFS	158	200	**410**	**4.5600**	**0.0611**
	UCS	**120**	**162**	1,873,435	35,550.1778	550.6296
	A	**120**	**162**	1,651,429	69,041.7771	518.0790

Table 7: Algorithm Performance on Test Case Input 7

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 7	BFS	**282**	**1,857**	27,481	367.2550	7.6326
	DFS	300	1,911	**18,410**	**234.4975**	**3.9673**
	UCS	**282**	**1,857**	30,677	503.5107	10.2070
	A*	**282**	**1,857**	30,431	2414.8247	10.1411

Table 8: Algorithm Performance on Test Case Input 8

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 8	BFS	**79**	**308**	117,047	1,509.3641	29.4232
	DFS	2,050	7,587	92,322	**1,065.8581**	25.9686
	UCS	**79**	**308**	131,247	1,773.7801	37.4528
	A*	**79**	**308**	**79,942**	2,382.5204	**24.9661**

Table 9: Algorithm Performance on Test Case Input 9

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 9	BFS	**25**	69	163,718	3066.6556	45.6771
	DFS	2391	4523	166,412	3001.7211	44.5866
	UCS	27	**57**	247,236	5915.6542	88.2026
	A*	27	**57**	**15,012**	**639.4784**	**6.0918**

Table 10: Algorithm Performance on Test Case Input 10

Test Case	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
Input 10	BFS	**63**	145	8,925	102.3943	2.2673
	DFS	163	341	**5,461**	**62.7699**	**1.7390**
	UCS	69	**128**	19,320	245.8084	5.0489
	A*	69	**128**	15,354	366.1234	4.2981

5.1.2 Experiment Result by Criteria

Table 11: Steps for each Algorithm on Test Cases

Algorithm	Input 1	Input 2	Input 3	Input 4	Input 5
BFS	**27**	**37**	**61**	**76**	**95**
DFS	3,069	149	107	32,492	157
UCS	**27**	**37**	63	96	**95**
A*	**27**	**37**	63	96	**95**
Algorithm	**Input 6**	**Input 7**	**Input 8**	**Input 9**	**Input 10**
BFS	**120**	**282**	**79**	**25**	**63**
DFS	158	300	2,050	2,391	163
UCS	**120**	**282**	**79**	27	69
A*	**120**	**282**	**79**	27	69

Table 12: Nodes for each Algorithm on Test Cases

Algorithm	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**
BFS	57,870	6,318	99,537	809,974	21,596
DFS	83,244	6,564	73,004	**798,895**	**16,558**
UCS	162,703	17,748	218,876	11,271,286	105,719
A*	**7,457**	**2,509**	**69,116**	3,855,371	91,015
Algorithm	**Input 6**	**Input 7**	**Input 8**	**Input 9**	**Input 10**
BFS	1,656,989	27,481	117,047	163,718	8,925
DFS	**410**	**18,410**	92,322	166,412	**5,461**
UCS	1,873,435	30,677	131,247	247,236	19,320
A*	1,651,429	30,431	**79,942**	**15,012**	15,354

Table 13: Time (ms) for each Algorithm on Test Cases

Algorithm	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**
BFS	966.73	129.0095	1,519.3130	11,441.4740	330.2922
DFS	1,290.60	86.6289	**990.1822**	**11,340.4214**	**250.8879**
UCS	2,914.44	261.2290	3,817.3518	498,597.9238	1,774.6058
A*	**268.63**	**64.5716**	2,672.473	158,110.5044	3,146.0078
Algorithm	**Input 6**	**Input 7**	**Input 8**	**Input 9**	**Input 10**
BFS	23,340.1673	367.2550	1,509.3641	3,066.6556	102.3943
DFS	**4.5600**	**234.4975**	**1,065.8581**	3,001.7211	**62.7699**
UCS	35,550.1778	503.5107	1,773.7801	5,915.6542	245.8084
A*	69,041.7771	2,414.8247	2,382.5204	**639.4784**	366.1234

Table 14: Memory (MB) for each Algorithm on Test Cases

Algorithm	**Input 1**	**Input 2**	**Input 3**	**Input 4**	**Input 5**
BFS	16.21	1.7987	27.3401	211.5737	6.6210
DFS	21.81	1.8032	**25.7027**	**207.0738**	**4.8065**
UCS	45.73	5.6643	61.4235	1,486.1637	39.0277
A*	**1.48**	**0.4496**	22.5569	472.2814	27.1638
Algorithm	**Input 6**	**Input 7**	**Input 8**	**Input 9**	**Input 10**
BFS	436.9743	7.6326	29.4232	45.6771	2.2673
DFS	**0.1174**	**5.3774**	**24.4652**	42.6679	**1.4674**
UCS	561.9293	8.7616	31.5037	78.0225	6.3938
A*	514.2785	7.5696	**18.9171**	**4.7584**	4.1688

5.2 Visual Representation of Experimental Results

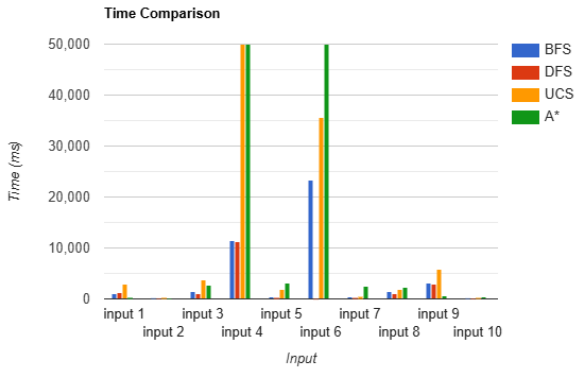

Figure 1: Execution Time Comparison for BFS, DFS, UCS, and A* Across All Test Cases.

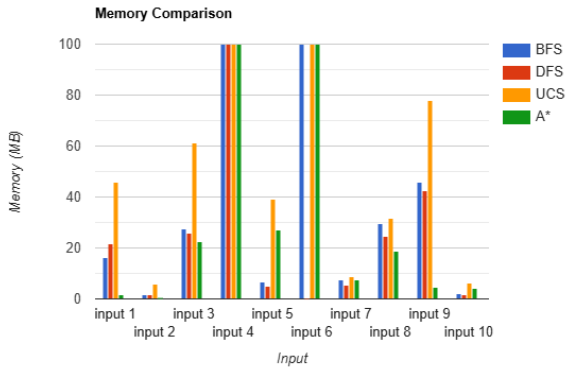

Figure 2: Memory usage Comparison for BFS, DFS, UCS, and A* Across All Test Cases.

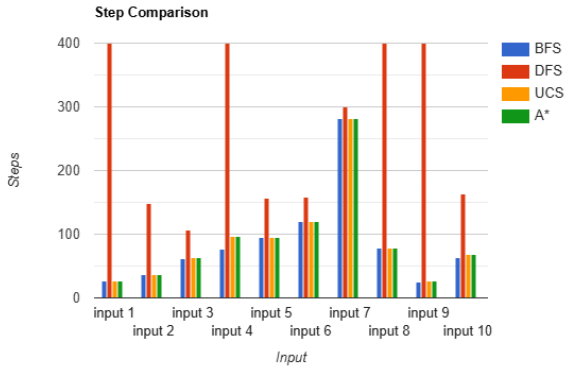

Figure 3: Total Steps Comparison for BFS, DFS, UCS, and A* Across All Test Cases.

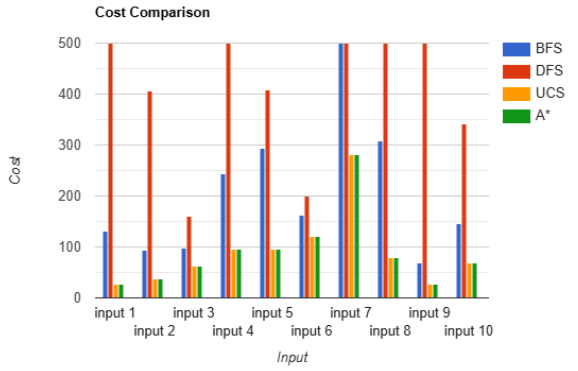

Figure 4: Total Cost Comparison for BFS, DFS, UCS, and A* Across All Test Cases.

In this section, we represent the data with five bar charts, showing each algorithm's performance across five criteria: steps, cost, execution time, and memory used. The goal is to extract meaningful insights from the raw data provided in the tables above. We preprocess the data by setting manual thresholds for each chart to identify and handle outliers. Any outlier values that significantly exceed the threshold will be normalized into threshold value. For instance, a threshold of 500 is set for cost, meaning values above 500 are displayed as 500. This approach reduces the impact of outliers, offering a clearer view of the data while still allowing viewers to recognize outliers if desired.

6 Analysis and Evaluation

6.1 Insights on map types

6.1.1 Non-greedy Maps

- Medium to small-sized maps that are challenging for humans.

- These maps require the agent to "think outside the box" rather than making the most obvious or greedy moves. They test the algorithms' ability to find optimal solutions in scenarios where straightforward strategies may not suffice.
- Test cases: input-01, input-04, input-08, input-09.

6.1.2 High-branching-factor Maps

- These maps are relatively wide and exhibit a high branching factor, leading the Depth-First Search (DFS) algorithm to often explore unpromising paths or "wander" in a non-productive manner. Such configurations are particularly challenging for DFS as it may fail to efficiently prioritize states leading to optimal solutions.
- On the other hand, Breadth-First Search (BFS) tends to produce better results on these maps, as its systematic exploration allows it to maintain near-optimal solutions within the constraints of Uniform Cost Search (UCS) standards.
- Test cases: input-01, input-04, input-08, input-09.

6.1.3 Unified Solution Maps

- This category consists of carefully crafted maps where BFS, UCS, and A* algorithms all produce the same solution, regardless of their inherent differences in goal testing and cost optimization. These maps serve as valuable benchmarks for ensuring the consistency and accuracy of the algorithms across controlled test environments.
- Such maps offer insight into the ideal scenarios where algorithmic results align, suggesting a baseline for expected performance in map layouts without complex trade-offs between solution paths.
- Test cases: input-06, input-08.

6.1.4 Optimality-focused Maps

- Maps within this category are explicitly designed to highlight the efficiency of UCS and A* algorithms in achieving optimal solutions. In these maps, BFS's early goal testing becomes a limiting factor, resulting in a higher-cost solution due to suboptimal path selection, despite potentially reaching the goal in fewer steps.
- DFS generally underperforms, often failing to yield feasible solutions due to its tendency to explore lengthy, costly paths.
- While UCS achieves optimal solutions in these maps, its performance is hindered by significant computational requirements, consuming more time and memory resources. A* demonstrates the best performance, balancing solution optimality with computational constraints, providing the most practical solution under limited resource conditions.
- Test cases: input-04.

6.1.5 Narrow-path Maps

- In one of our curated maps, the layout gives an impression of complexity through a vast structure, yet they are highly solvable due to narrow routes and low branching factors.
- The limited branching factor reduces the decision complexity for algorithms, with BFS, DFS, and UCS able to process paths leading quickly to solutions or dead-ends, depending on the specific path selected. A* excels in this environment by quickly disregarding irrelevant states, efficiently guiding the agent toward the solution.
- These maps are particularly useful for testing each algorithm's handling of perceived complexity versus actual decision path simplicity, as well as their ability to ignore or prioritize states effectively.
- Test cases: input-07.

6.2 Strengths of Each Search Algorithm

6.2.1 BFS

Breadth-First Search (BFS) stands out for its simplicity and guarantees of finding a solution with the fewest moves when all actions have the same cost. Its systematic approach of exploring nodes by levels ensures that solutions are found in minimal depth, which is beneficial in scenarios where step count is more critical than cost minimization. Additionally, BFS is effective in relatively smaller maps, as it doesn't require additional memory for storing complex data structures or heuristic calculations.

6.2.2 DFS

Depth-First Search (DFS) is particularly useful in narrow or constrained maps where paths branch less frequently. Its low memory consumption makes it an attractive choice in memory-constrained environments. Furthermore, its speed in finding a solution quickly on narrow maps allows DFS to sometimes act as an efficient preliminary check for solution feasibility. When paired with optimizations to avoid infinite looping, DFS can quickly explore deep paths, which makes it useful for certain types of preliminary exploratory tasks.

6.2.3 UCS

Uniform Cost Search (UCS) excels in scenarios where all path costs vary and optimal solutions are essential. Its lack of reliance on heuristics makes UCS a strong choice when no reliable heuristic is available, as it only requires accurate cost information. UCS provides an effective balance of optimality and simplicity, making it versatile in applications that require guaranteed least-cost paths, such as logistics and routing in unknown terrains.

6.2.4 A*

A* offers the most practical application potential in real-world scenarios, combining both cost accuracy and efficiency. Its design allows it to leverage problem-specific information

through heuristics, which optimally directs the search towards the goal. When equipped with a well-informed heuristic, A* can outperform UCS in runtime by reducing the number of expanded nodes. Let's take a look at **Input 3**, both A* and UCS found paths with cost 92, which was better than BFS (98) and DFS (160) but but A* was significantly more memory-efficient, using only about half the memory compared to UCS (21.308 compared to 70.0465). This could be one of many major reasons why it is ideal for route-finding in applications like map navigation, robotics, and logistics. Its flexibility allows it to be tailored to different scenarios, achieving optimal performance without sacrificing solution quality. However, this comes at the cost of increased memory usage, as A* must store and manage heuristic information. Nonetheless, with a strong heuristic, A* is frequently the preferred algorithm in scenarios demanding both optimal solutions and efficient performance.

6.3 Weaknesses of Each Search Algorithm

6.3.1 Early Goal Test Weakness in BFS

In some maps (e.g., **Input 4**), BFS's early goal test leads to solutions with fewer steps but higher total cost. This is the algorithm's limitation in scenarios where the cheapest solution is not the one with the least steps.

6.3.2 Wandering Issues in DFS

DFS often fails to find optimal solutions like **Input 1**, especially in maps with high branching factors. It may wander extensively, resulting in solutions with an unacceptable number of steps and high costs.

6.3.3 Performance Issues in UCS

While UCS guarantees optimal solutions, it struggles with performance on larger maps due to the lack of heuristic guidance. The time and memory required can be significantly higher compared to other algorithms (e.g., Input **Input 9**).

6.3.4 Processing Overhead in A*

A*'s perceived limitations are not inherent weaknesses but rather trade-offs that enable a more balanced and generally effective approach in practical problem-solving. While A* is typically slower than BFS and less memory-efficient than DFS, it occasionally lags behind UCS on small maps. However, its ability to balance speed, memory usage, and optimality makes it versatile across a range of scenarios.

A* often consumes more time and memory to find solution in small and tricky maps, compared with BFS and DFS. The reason is that the computational cost of calculating heuristics, maintaining priority queues, and managing open/closed lists can outweigh the benefits in small search spaces. This is clearly shown in **Input 5**, A* took more than 3 seconds, while BFS and DFS ran significantly faster.

6.4 Conclusion

- Breadth-First Search (BFS) is more likely to produce near-optimal solutions compared to Depth-First Search (DFS). However, DFS outperforms all other algorithms in runtime performance on "Narrow-road" maps, with several orders of magnitude improvements in memory and time.
- DFS consistently requires the least memory among our test cases. In contrast, A* search generally visits the fewest nodes, while Uniform Cost Search (UCS) consistently visits the most nodes.
- BFS reliably yields solutions with the smallest number of moves.
- Although both UCS and A* produce optimal solutions, UCS tends to perform significantly slower on larger maps. UCS has a slight advantage on smaller maps, possibly because A* incurs additional computational cost to calculate heuristic values. This additional cost does not compensate for the anticipated node pruning due to the heuristic, resulting in no significant reduction in node expansion for A*.
- The size of the map alone does not necessarily determine the size of its state space. The number of stones plays a more critical role, as the state space grows approximately as:

  $$S = T^{(N+1)}$$

  where:

  - S is the size of the state space,
  - T is the number of accessible tiles on the map, and
  - N is the number of stones.

  This approximation gauges the possible arrangements of stones and the agent. For example, map 7, though large, has a manageable state space due to a smaller exponent (with only two stones), making it solvable. By contrast, map 4, despite its smaller accessible area, contains four stones and switch pairs, significantly increasing its complexity.

7 Graphical User Interface

To clearly show how Ares moves through the maze step-by-step, we used the PyQT6 library [2], which provides a Python interface for the cross-platform GUI toolkit Qt. Using a graphical user interface (GUI) allows users to interact intuitively with the visualization, making it easier to follow and understand the maze-solving process. It transforms complex algorithmic movements into visual, interactive displays, helping to bridge the gap between abstract computational logic and user comprehension.

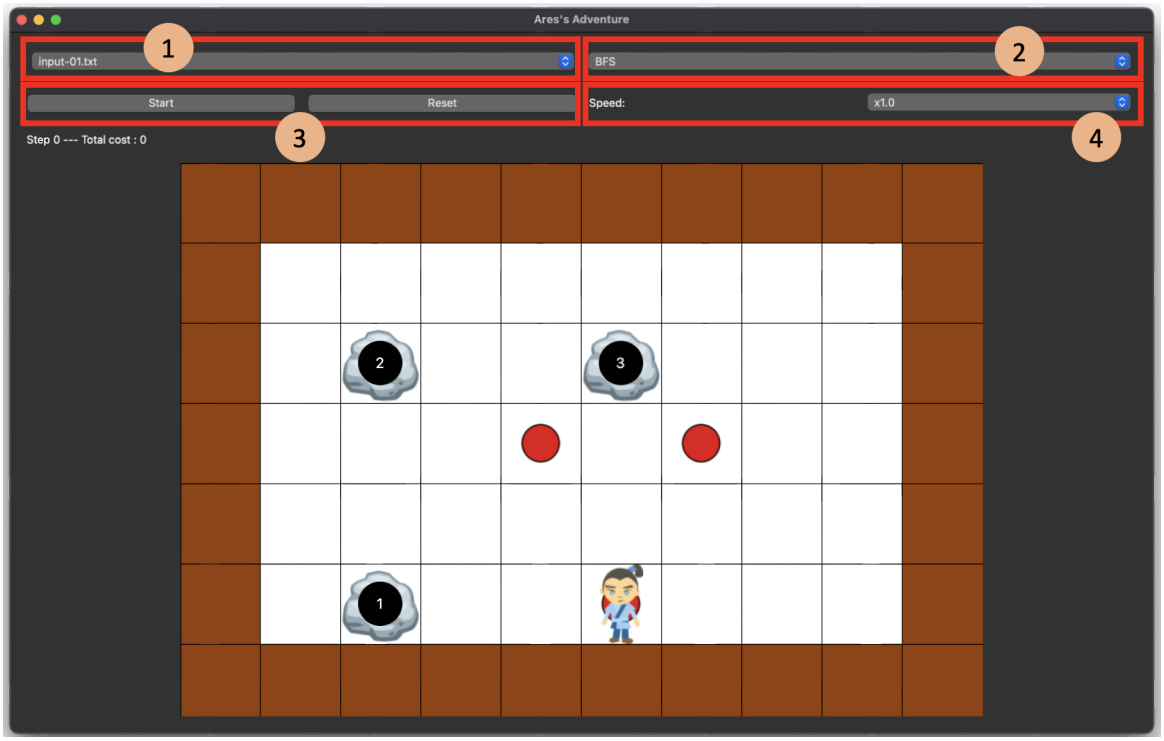

Figure 5: GUI overview

As illustrated in Figure 5, the interface includes key components that allow user interaction:

1. At the top-left corner, there is a dropdown menu for selecting input files, with **input-01.txt** shown as the default option. Users can select from available files such as input-01.txt, input-02.txt, and so on.

2. Beside it, another dropdown menu lets users pick different algorithms. The current selection is "BFS" (Breadth-First Search). Users can also choose between BFS, DFS, UCS, or A*.

3. Below these dropdown menus, there are two buttons: **Start** for beginning the execution and **Reset** for restarting or clearing the current state.

4. On the right side, a slider adjusts the visualization speed of the algorithm, with a default setting of **x1.0**. The speed options range from x1.0 up to x50.0.

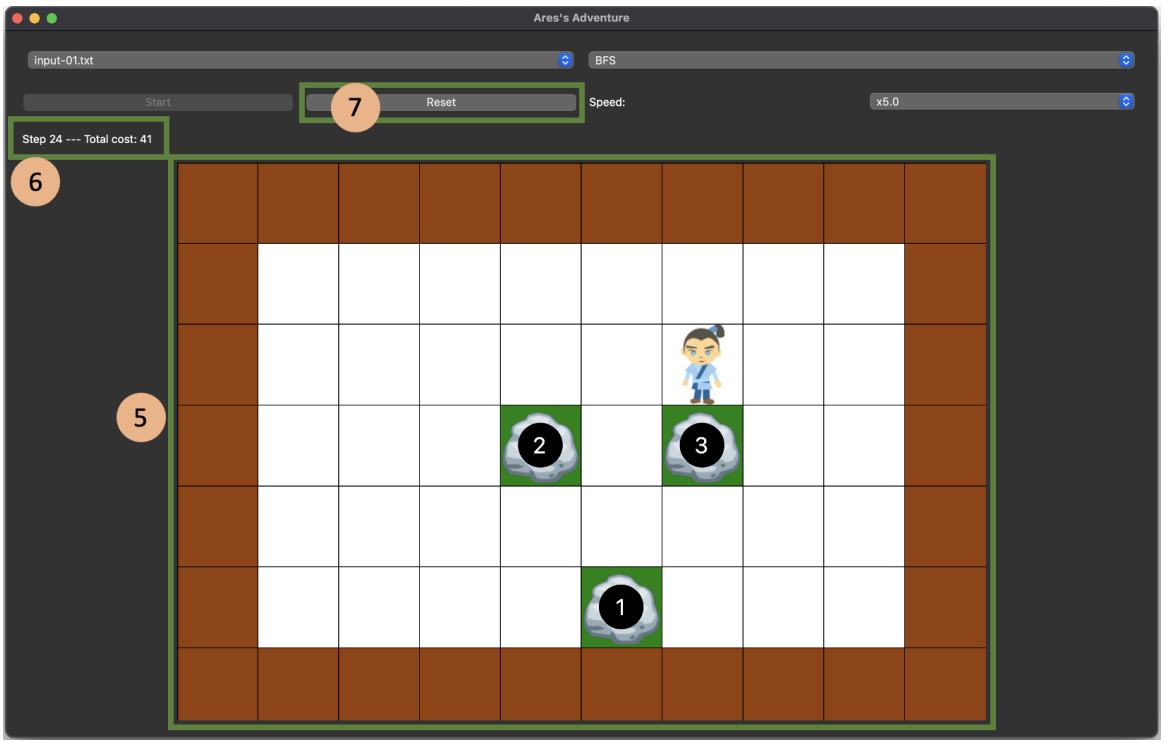

Figure 6: GUI monitor

In Figure 6, the main features displayed include:

5. This area visually tracks Ares' movement through the maze. Users can observe each step as Ares navigates through different paths, providing a clear depiction of how the chosen algorithm works.

6. A text box displays the current step number and the total cost incurred up to that step. This information helps users monitor Ares' progress and understand the efficiency and cost of the path taken.

7. When the Start button is pressed, it becomes disabled until the execution is complete or the Reset button is used. Clicking the Reset button clears the current state and re-enables the Start button, allowing users to run the process again or choose a new configuration.

References

[1] Ngoc-Thao Nguyen, Hai-Minh Nguyen. Global search strategies, 2024.

[2] Riverbank Computing. (2024). PyQt6 (Version 6.7.1) [Software]. Riverbank Computing. https://www.riverbankcomputing.com/software/pyqt/

[3] Sokoban. (n.d.). How to detect deadlocks. Retrieved November 7, 2024, from http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks