Vietnam National University,
Ho Chi Minh City

University of Science

Faculty of Information Technology

Project 01: Ares's Adventure Report

CS14003 – Introduction to Artificial Intelligence

Ngo Nguyen The Khoa	23127065
Bui Minh Duy	23127040
Nguyen Le Ho Anh Khoa	23127211

March 3, 2025

Contents

1 Group Information

- **Subject:** Introduction to Artificial Intelligence.
- **Class:** 23CLC09.
- **Lecturer:** Bui Duy Dang, Le Nhut Nam.
- **Team members:**

No.	Fullname	Student ID	Email
1	Ngo Nguyen The Khoa	23127065	nntkhoa23@clc.fitus.edu.vn
2	Bui Minh Duy	23127040	bmduy23@clc.fitus.edu.vn
3	Nguyen Le Ho Anh Khoa	23127211	nlhakhoa23@clc.fitus.edu.vn

- **Tools:**
  - **Git**, **GitHub**: Source code version control.
  - **CapCut**: Video editing.
  - **ChatGPT**, **Gemini** and **DeepSeek**.
  - **Visual Studio Code**: Code editor (Python, Latex).

2 Project Information

- **Name:** Ares's Adventure.
- **Developing Environment:** Visual Studio Code (Windows).
- **Programming Language:** Python.
- **Libraries and Tools:**
  - **rye:** A comprehensive project and package management solution for Python.
  - **PyGame:** A Python game maker library.
  - `numpy`: used for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
  - `scipy`: pre-implementation of the Hungarian Min-matching algorithm.

3 Work assignment table

No.	**Task Description**	**Assigned to**	**Rate**
1	Implement BFS, DFS	Minh Duy	100%
2	Implement UCS, Dijkstra	Anh Khoa	100%
3	Implement A*, GBFS	The Khoa	100%
4	Implement Swarm	Anh Khoa	100%
5	Optimize heuristic function using Hungarian for min-matching	The Khoa, Minh Duy	100%
6	Optimize the number of expanded nodes by using deadlock detection	The Khoa, Anh Khoa	100%
7	Video Editing	Minh Duy	100%
8	Report	All members	100%
All requirements are completed! No errors occur while running the program			

4 Self-evaluation

No.	**Criteria**	**Score**
1	Implement BFS correctly.	100%
2	Implement DFS correctly.	100%
3	Implement UCS correctly.	100%
4	Implement A* correctly.	100%
5	Implement GBFS correctly.	100%
6	Generate at least 10 test cases for each level with diffenrent attributes.	100%
7	Result (output file and GUI).	100%
8	Video to demonstrate all algorithms for some test cases.	100%
9	Report.	100%
10	Implement, written report for Dijkstra's Algorithm and Swarm Algorithm.	100%

5 Algorithms' Implementations

5.1 Breadth First Search

Breadth First Search (BFS), as the name says, explores the search space in the increasing order of the depth and the costs of traveling from one state to another is assumed to be a positive number. Typically, this algorithm is often associated with the concept of stack and queue and pushing and popping from the stack.

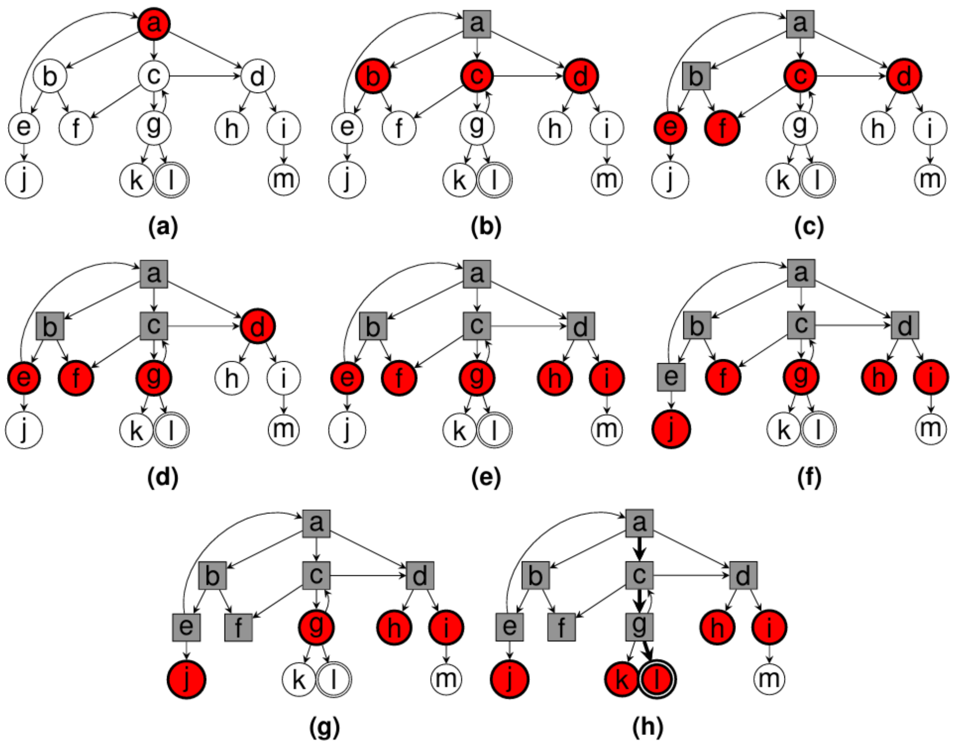

Figure 1: Breadth First Search (BFS)

Pseudocode

Algorithm 1 Breadth First Search (*start, goal*)

```
 1: queue gets [start]
 2: while queue is not empty do
 3:     node gets dequeue(queue)
 4:     if node = goal then
 5:         return path
 6:     end if
 7:     for all neighbor in valid moves do
 8:         if neighbor not visited then
 9:             mark neighbor as visited
10:             enqueue(queue, neighbor)
11:         end if
12:     end for
13: end while
14: return failure
```

Implementation

- **__init__(...)** Initializes the BFS algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes an optional deadlock detection flag.
- **search()** Implements the BFS algorithm using a queue (FIFO). The search begins with the initial state in the frontier. The function explores states by dequeuing from the front, checking for the goal condition, and enqueuing valid new states.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir)** Generates a new state by moving the player in the specified direction, updating the relevant positions.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution. It explores all nodes at the current depth before moving deeper.

Space Complexity: $O(b^d)$, as it stores all nodes at the current depth in memory.

5.2 Depth First Search

Depth First Search (DFS) is a special case of backtracking search algorithm. The search starts from the root and proceeds to the farthest node before backtracking. The difference between this and the backtracking is that this stops the search once a goal is reached and does not care if it is not minimum.

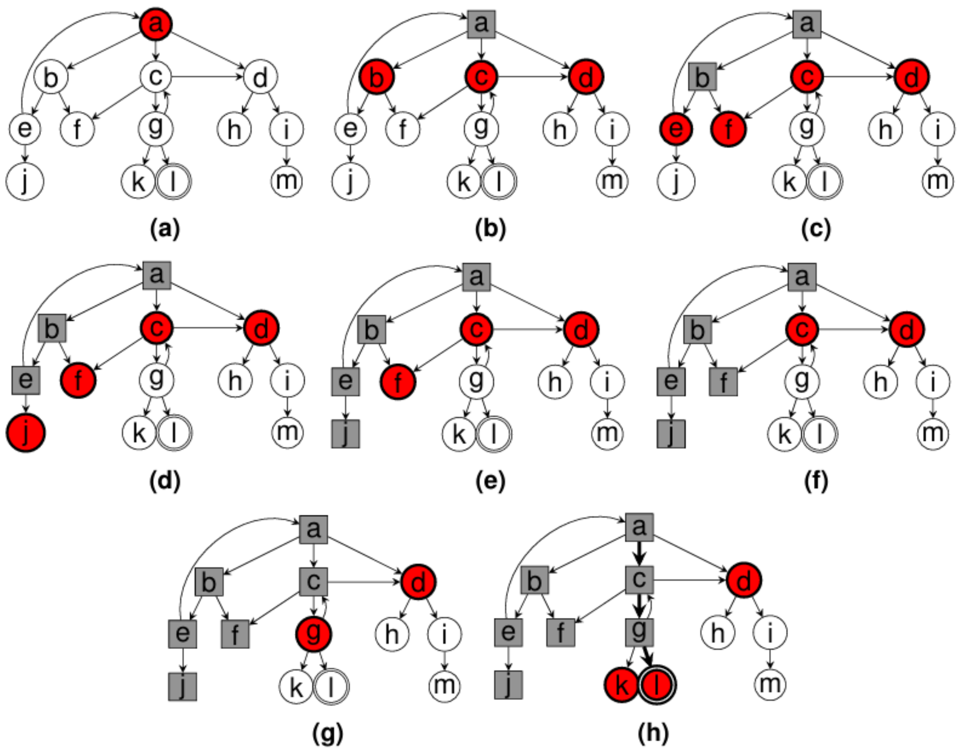

Figure 2: Depth First Search

Pseudocode

Algorithm 2 Depth First Search (*start, goal*)

```
1:  stack ← [start]
2:  while stack is not empty do
3:      node ← pop(stack)
4:      if node = goal then
5:          return path
6:      end if
7:      for all neighbor in valid moves do
8:          if neighbor not visited then
9:              mark neighbor as visited
10:             push(stack, neighbor)
11:         end if
12:     end for
13: end while
14: return failure
```

Implementation

- **__init__(...)** Initializes the DFS algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes an optional deadlock detection flag.

- **search()** Implements the DFS algorithm using a stack (LIFO). The search begins with the initial state in the frontier. The function explores states by popping from the stack and checking for the goal condition. If a new valid state is discovered, it is marked as visited and pushed onto the stack for further exploration.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir)** Generates a new state by moving the player in the specified direction, updating the relevant positions.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O(b^m)$, where m is the maximum depth of the search tree. In the worst case, DFS may explore an entire path before backtracking.

Space Complexity: $O(m)$, since DFS only needs to store nodes along the current path.

5.3 Uniform Cost Search

For any search problem, Uniform Cost Search (UCS) is the better algorithm than the previous ones. The search algorithm explores in branches with more or less same cost. This consist of a priority queue where the path from the root to the node is the stored element and the depth to a particular node acts as the priority. UCS assumes all the costs to be non negative. While the DFS algorithm gives maximum priority to maximum depth, this gives maximum priority to the minimum cumulative cost.

Pseudocode

Algorithm 3 Uniform Cost Search (*start, goal*)

```
1: priority queue ← [(start, cost = 0)]
2: while priority queue is not empty do
3:     (node, cost) ← dequeue(priority queue)
4:     if node = goal then
5:         return path
6:     end if
7:     for all neighbor in valid moves do
8:         new cost ← cost + move cost
9:         if neighbor not visited or new cost < previous cost then
10:            mark neighbor as visited
11:            enqueue(priority queue, (neighbor, new cost))
12:        end if
13:    end for
14: end while
15: return failure
```

Implementation

- **__init__(...)** Initializes the Uniform Cost Search (UCS) algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes an option for deadlock detection. The initial state's cost g is set to zero.
- **search()** Implements the UCS algorithm using a priority queue (min-heap). The function expands the node with the lowest accumulated cost g at each step. It explores all possible states, updating costs and storing them in a hash table for efficient lookup.
- **handle(new_state, closed, frontier, state_hash_table)** Manages newly generated states, adding them to the frontier if they have not been visited or updating their cost if a lower-cost path is found.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir)** Generates a new state by moving the player in the specified direction, updating positions, and recalculating cost values.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O(b^C)$, where C is the cost of the optimal solution. In the worst case, UCS expands all nodes up to the goal depth.

Space Complexity: $O(b^C)$, as it stores all expanded nodes in memory.

5.4 A* Search with heuristic

A* algorithm is one of the popular technique used in path finding and graph traversals. This algorithm completely relies on heuristics for computing the future cost of a problem. This algorithm is equivalent to the uniform cost search with modified edge cost. This heuristics is chosen according to the case where the algorithm is implemented, thus emphasizing the importance of domain knowledge. This algorithm is consistent if the modified cost is greater than zero.

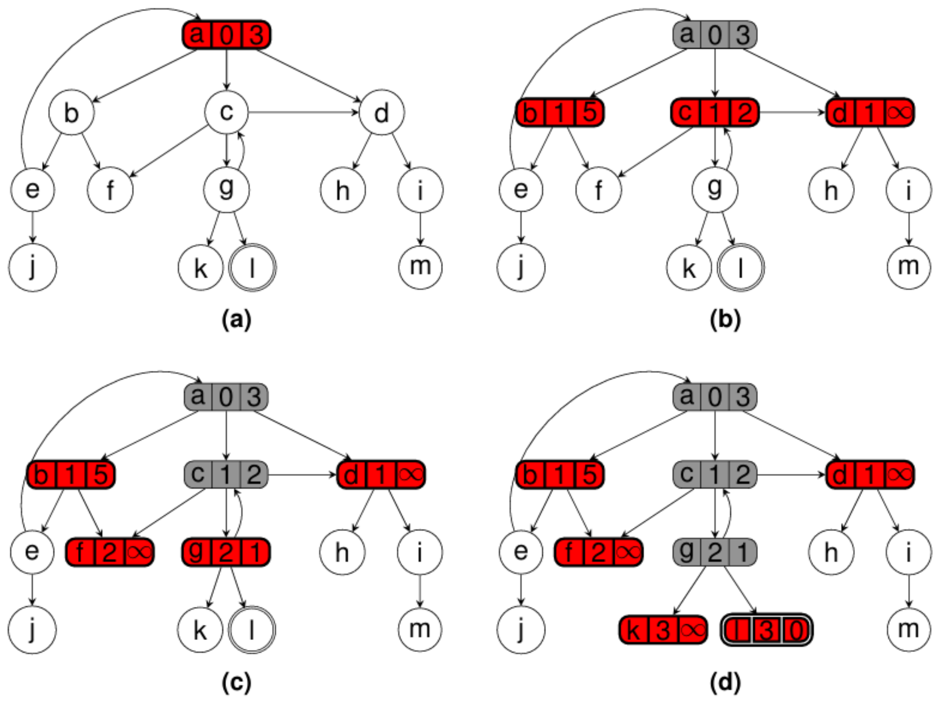

Figure 3: A* Algorithm

Pseudocode

Algorithm 4 A* Search (*start, goal, heuristic*)

```
1:  priority queue ← [(start, cost = 0, estimated total cost = heuristic(start))]
2:  while priority queue is not empty do
3:      (node, cost) ← dequeue(priority queue)
4:      if node = goal then
5:          return path
6:      end if
7:      for all neighbor in valid moves do
8:          new cost ← cost + move cost
9:          estimated total cost ← new cost + heuristic(neighbor)
10:         if neighbor not visited or new cost < previous cost then
11:             mark neighbor as visited
12:             enqueue(priority queue, (neighbor, new cost, estimated total cost))
13:         end if
14:     end for
15: end while
16: return failure
```

Implementation

- **__init__(...)** Initializes the A* search algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes options for deadlock detection and heuristic optimization.

- **search()** Implements the A* search algorithm using a priority queue (min-heap). The function explores states by selecting the one with the lowest cost $f = g + h$. It expands nodes by generating successors, updating costs, and maintaining a hash table for efficient state lookup.
- **handle(new_state, closed, frontier, state_hash_table)** Manages newly generated states, checking if they should be added to the frontier or updated in the hash table based on their cost values.
- **heuristic(stones_pos, switches_pos)** Computes the heuristic function to estimate the cost to reach the goal. It selects between the Hungarian heuristic and Manhattan heuristic based on the optimization flag.
- **mahattan_heuristic(stones_pos, switches_pos)** Calculates the heuristic using the Manhattan distance, summing up the minimum distances from each stone to a switch.
- **hungarian_heuristic(stones_pos, switches_pos)** Uses the Hungarian algorithm to optimally assign stones to switches, minimizing the total weighted Manhattan distance.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir, heuristic)** Generates a new state by moving the player in the specified direction, updating positions and recalculating heuristic values.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O(b^d)$ in the worst case, but with a good heuristic, it can be significantly reduced. If the heuristic is admissible and consistent, A* is optimal and complete.

Space Complexity: $O(b^d)$, as it keeps all generated nodes in memory.

5.5 Greedy Best-First Search

Greedy Best-First Search is similar to A* algorithm. The difference is that the vertices in the priority queue are ordered only by the estimated remaining distance to the solution. It has to be noted that complete best first search is not optimal.

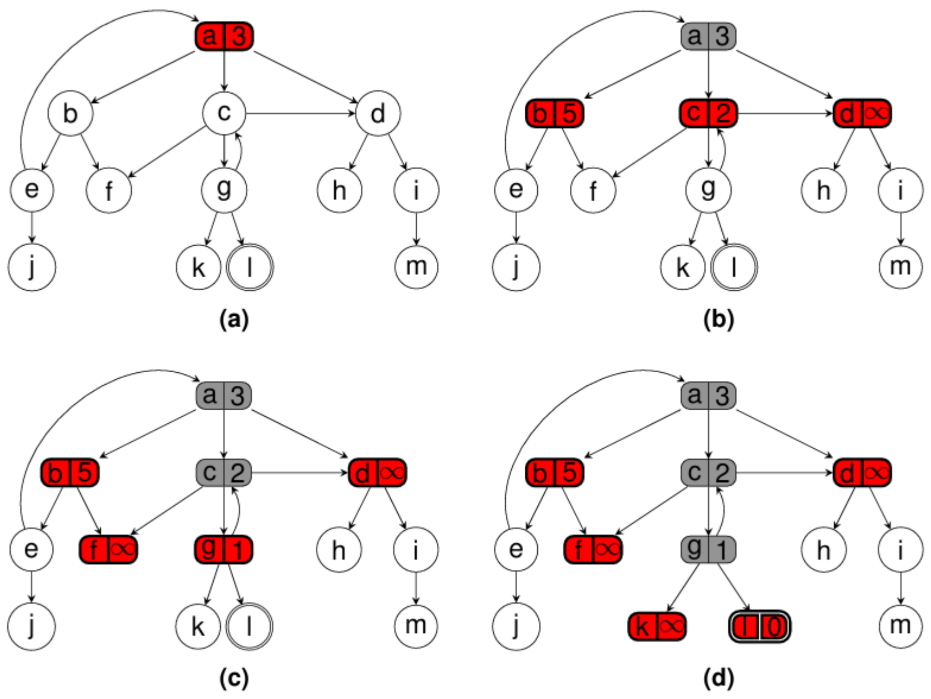

Figure 4: Greedy Best-First Search

Pseudocode

Algorithm 5 Greedy Best-First Search (*start, goal, heuristic*)

```
1: priority queue ← [(start, heuristic(start))]
2: while priority queue is not empty do
3:     node ← dequeue(priority queue)
4:     if node = goal then
5:         return path
6:     end if
7:     for all neighbor in valid moves do
8:         if neighbor not visited then
9:             mark neighbor as visited
10:            enqueue(priority queue, (neighbor, heuristic(neighbor)))
11:        end if
12:    end for
13: end while
14: return failure
```

Implementation

- **__init__(...)** Initializes the Greedy Best-First Search (GBFS) algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes options for deadlock detection and heuristic optimization.
- **search()** Implements the GBFS algorithm using a priority queue (min-heap). The function explores states by selecting the one with the lowest heuristic h, without

considering the cost g. It expands nodes by generating successors, updating costs, and maintaining a hash table for efficient state lookup.

- **handle(new_state, closed, frontier, state_hash_table)** Manages newly generated states, checking if they should be added to the frontier or updated in the hash table based on their heuristic values.
- **heuristic(stones_pos, switches_pos)** Computes the heuristic function to estimate the cost to reach the goal. It selects between the Hungarian heuristic and Manhattan heuristic based on the optimization flag.
- **mahattan_heuristic(stones_pos, switches_pos)** Calculates the heuristic using the Manhattan distance, summing up the minimum distances from each stone to a switch.
- **hungarian_heuristic(stones_pos, switches_pos)** Uses the Hungarian algorithm to optimally assign stones to switches, minimizing the total weighted Manhattan distance.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir, heuristic)** Generates a new state by moving the player in the specified direction, updating positions and recalculating heuristic values.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O(b^d)$, where d is the depth of the solution. GBFS does not guarantee optimality and may explore irrelevant paths.

Space Complexity: $O(b^d)$, as it stores all visited states.

5.6 Dijkstra's Algorithm

Pseudocode

Algorithm 6 Dijkstra's Algorithm (*start, goal*)

```
 1: priority queue ← [(start, cost = 0)]
 2: distances[start] ← 0
 3: while priority queue is not empty do
 4:     (node, cost) ← dequeue(priority queue)
 5:     if node = goal then
 6:         return distances
 7:     end if
 8:     for all neighbor in valid moves do
 9:         new cost ← cost + move cost
10:         if new cost < distances[neighbor] then
11:             distances[neighbor] ← new cost
12:             enqueue(priority queue, (neighbor, new cost))
13:         end if
14:     end for
15: end while
16: return distances
```

Implementation

- **__init__(...)** Initializes the Dijkstra search algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes an option for deadlock detection. The initial state's cost g is set to zero.
- **search()** Implements Dijkstra's algorithm using a priority queue (min-heap). The function explores states based on the lowest accumulated cost g. It expands nodes by generating successors, updating costs, and maintaining a hash table for efficient state lookup.
- **handle(new_state, closed, frontier, state_hash_table)** Manages newly generated states, checking if they should be added to the frontier or updated in the hash table based on their cost values.
- **can_go(current_state, dir)** Checks whether the player can move in a given direction from the current state without encountering obstacles.
- **go(current_state, dir)** Generates a new state by moving the player in the specified direction, updating positions, and recalculating cost values.
- **construct_path(final_state)** Reconstructs the sequence of moves leading to the goal state by backtracking from the final state.

Time and Space Complexity

Time Complexity: $O((V + E)\log V)$, where V is the number of vertices and E is the number of edges. Using a priority queue (min-heap) allows efficient updates.

Space Complexity: $O(V + E)$, as it stores all nodes and edges in the graph.

5.7 Swarm Algorithm

Pseudocode

Implementation

- **__init__(...)** Initializes the Swarm search algorithm with grid dimensions, matrix representation, initial player position, stone positions, and switch positions. It also includes options for deadlock detection and heuristic optimization.
- **search()** Defines the base search function for Swarm, Swarm Convergent, and Swarm Bidirectional approaches. Currently, the function tracks expanded nodes but does not implement a complete pathfinding process.
- **heuristic(stones_pos, switches_pos)** Computes the heuristic function to estimate the cost to reach the goal. The function dynamically selects between the Hungarian heuristic and Manhattan heuristic based on the optimization flag.
- **mahattan_heuristic(stones_pos, switches_pos)** Uses the Manhattan distance to estimate the cost, summing the minimum weighted distances from each stone to a switch.
- **hungarian_heuristic(stones_pos, switches_pos)** Utilizes the Hungarian algorithm to optimally assign stones to switches, minimizing the total weighted Manhattan distance.
- **SwarmConvergent.search()** Implements a variant of Swarm where multiple paths converge towards a solution. Currently, it maintains a set of explored states without a full implementation.
- **SwarmBidirectional.search()** Implements a bidirectional search variant of Swarm, exploring the state space from both the start and goal positions simultaneously. It maintains a set of explored states but lacks a fully developed pathfinding process.
- **AntColonyOptimization.__init__(...)** Initializes the Ant Colony Optimization (ACO) search algorithm with the same parameters as Swarm, allowing for heuristic-based optimizations.
- **AntColonyOptimization.search()** Implements the base ACO search function, tracking expanded states but lacking full functionality.

Time and Space Complexity

Time Complexity: Highly dependent on the heuristic used. The Hungarian heuristic runs in $O(n^3)$, while Manhattan distance runs in $O(n)$. Overall, complexity varies between $O(n^3)$ and $O(b^d)$.

Space Complexity: $O(b^d)$, as it stores visited states.

6 App Screenshots

7 References

1. Rye documentation