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

2 Project Information

- **Name:** Ares's Adventure.
- **Developing Environment:** Visual Studio Code (Windows).
- **Programming Language:** Python.
- **Libraries and Tools:**
  - **Libraries:**
    - **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.
  - **Tools:**
    - **Git**, **GitHub**: Source code version control.
    - **CapCut**: Video editing.
    - **ChatGPT**, **DeepSeek**: Optimize algorithm and GUI building assistant.
    - **Visual Studio Code**: Code editor for Python, Latex.

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%

4 Self-evaluation

No.	Criteria	Rate
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) is an algorithm used for traversing or searching graph structures. It explores all neighbors of a node before moving to the next level, ensuring that it finds the shortest path in an unweighted graph. BFS operates using a queue, processing nodes in a first-in, first-out (FIFO) order. This guarantees that the shallowest (least depth) solution is found first, making it particularly useful in scenarios where all edges have equal cost.

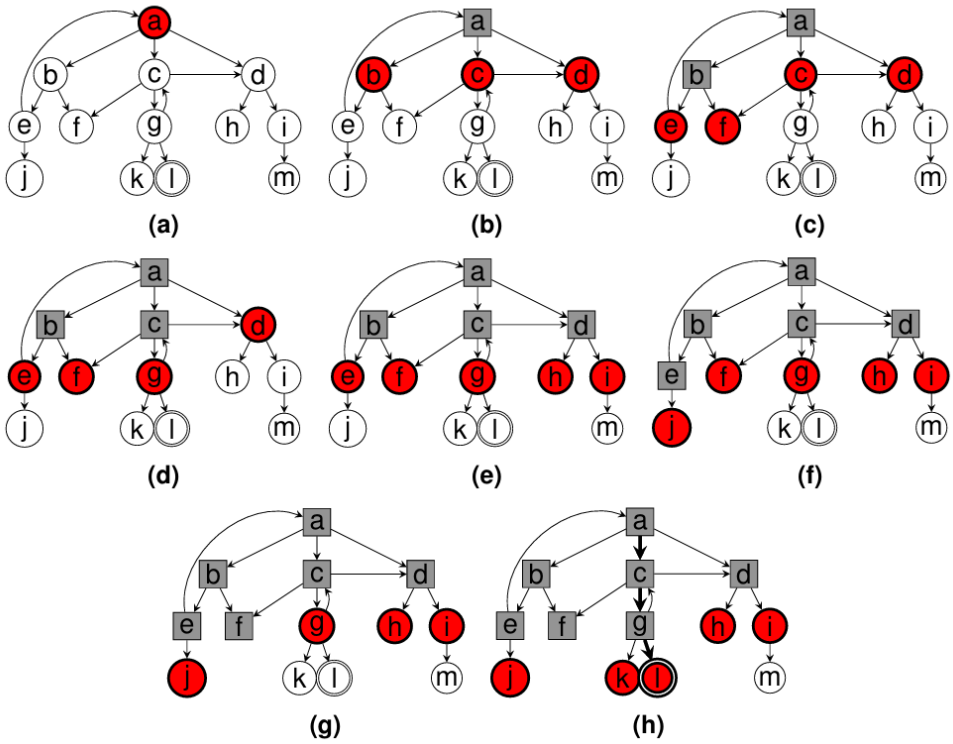


Figure 1: Breadth First Search (BFS)

Pseudocode

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

```
queue ← [start]
while queue is not empty do
    node ← dequeue(queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        if neighbor not visited then
            mark neighbor as visited
            enqueue(queue, neighbor)
        end if
    end for
end while
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 and deadlock cells.
- **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 graph traversal algorithm that explores as deeply as possible along each branch before backtracking. It uses a stack (either explicit or through recursion) to keep track of visited nodes. Unlike BFS, DFS does not guarantee the shortest path in an unweighted graph, but it is often preferred when searching for any solution rather than the

optimal one. DFS is especially useful in problems related to connectivity, cycle detection, and topological sorting.

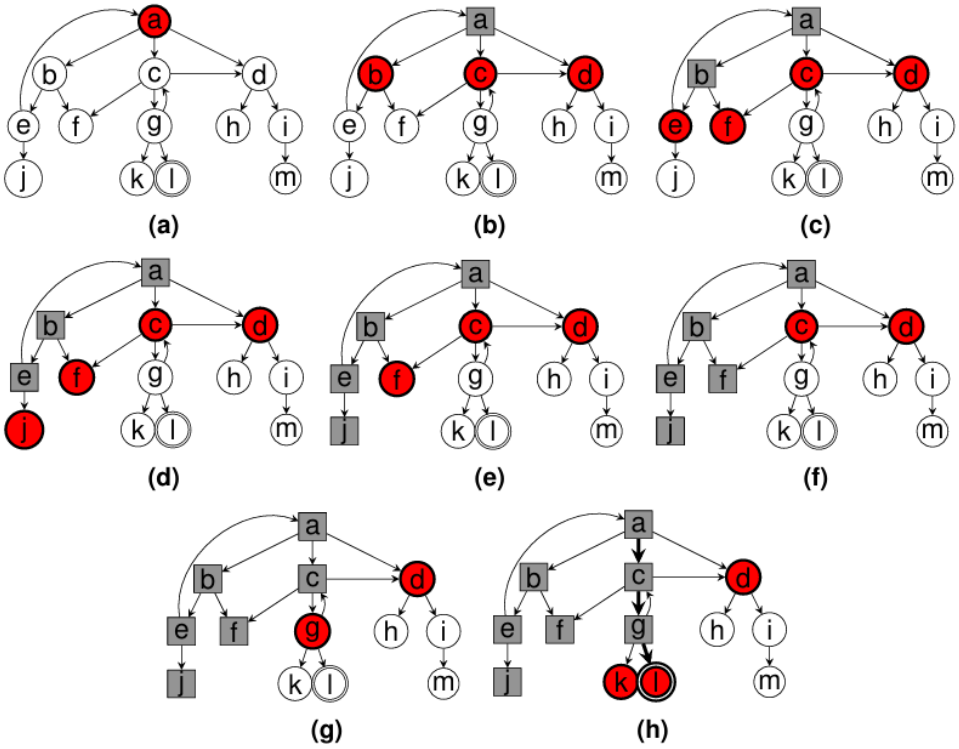


Figure 2: Depth First Search

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 and deadlock cells.
- **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

Uniform Cost Search (UCS) is a graph traversal algorithm that expands the least-cost node first.

Unlike BFS and DFS, UCS considers edge weights, ensuring that the path found is the shortest in terms of total cost. It is functionally equivalent to Dijkstra's algorithm when used for single-source shortest path problems.

UCS operates using a priority queue, where nodes are processed based on their cumulative path cost, making it ideal for finding optimal solutions in weighted graphs.

Pseudocode

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

```
priority queue ← [(start, cost = 0)]
while priority queue is not empty do
    (node, cost) ← dequeue(priority queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        new cost ← cost + move cost
        if neighbor not visited or new cost < previous cost then
            mark neighbor as visited
            enqueue(priority queue, (neighbor, new cost))
        end if
    end for
end while
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 and deadlock cells.
- **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 Dijkstra's Algorithm

Dijkstra's algorithm is a classic shortest path algorithm that guarantees finding the minimum-cost path from a starting node to all other nodes in a weighted graph.

It operates by iteratively selecting the node with the lowest cumulative cost and updating the distances of its neighbors.

Like UCS, Dijkstra's algorithm uses a priority queue, but it is typically employed in broader applications such as routing and network optimization.

Pseudocode

Algorithm 4 Dijkstra's Algorithm (*start, goal*)

```
priority queue ← [(start, cost = 0)]
distances[start] ← 0
while priority queue is not empty do
    (node, cost) ← dequeue(priority queue)
    if node = goal then
        return distances
    end if
    for all neighbor in valid moves do
        new cost ← cost + move cost
        if new cost < distances[neighbor] then
            distances[neighbor] ← new cost
            enqueue(priority queue, (neighbor, new cost))
        end if
    end for
end while
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 and deadlock cells.

- **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.

5.5 A* Search with heuristic

The A* algorithm is a widely used technique for pathfinding and graph traversal. It relies heavily on heuristics to estimate the future cost of reaching the goal. Essentially, A* extends uniform cost search by incorporating a heuristic function to guide its path selection, making it more efficient in many cases. The choice of heuristic depends on the specific problem domain, highlighting the importance of domain knowledge. For A* to be consistent, the modified cost function must always remain greater than zero.

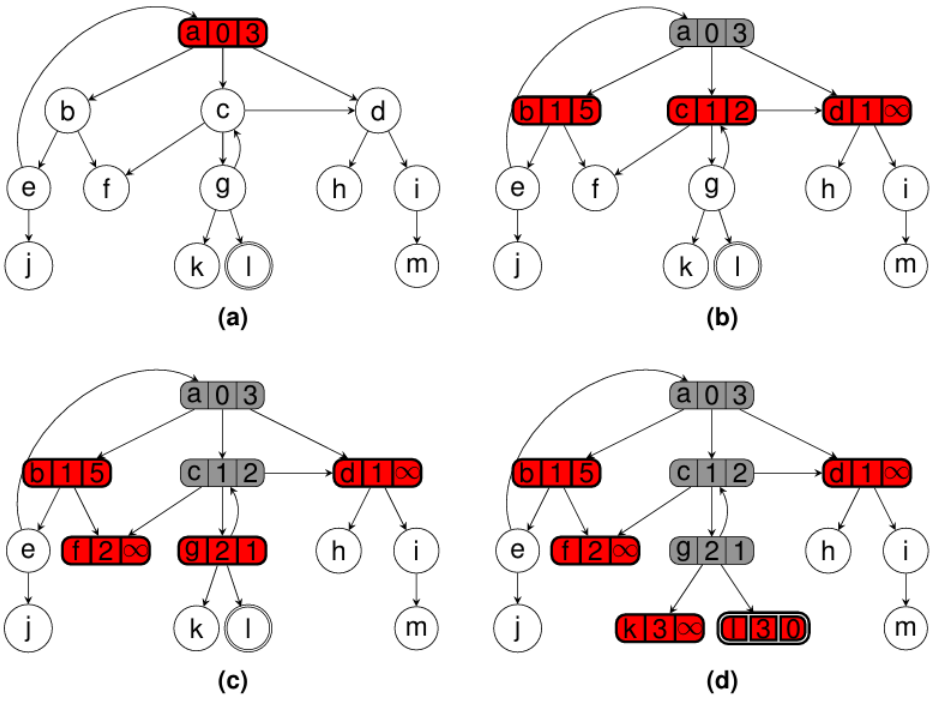


Figure 3: A* Algorithm

Pseudocode

Algorithm 5 A* Search (*start, goal, heuristic*)

```
priority queue ← [(start, cost = 0, estimated total cost = heuristic(start))]
while priority queue is not empty do
    (node, cost) ← dequeue(priority queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        new cost ← cost + move cost
        estimated total cost ← new cost + heuristic(neighbor)
        if neighbor not visited or new cost < previous cost then
            mark neighbor as visited
            enqueue(priority queue, (neighbor, new cost, estimated total cost))
        end if
    end for
end while
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 and deadlock cells.

- **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.

Heuristics in A* Algorithm

A* search uses a heuristic function $h(n)$ to estimate the cost from a given state to the goal. The total cost function is defined as:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the actual cost from the start state to the current state n.
- $h(n)$ is the estimated cost from n to the goal state.

For Sokoban, common heuristic choices include:

- **Manhattan Distance:** The sum of the absolute differences between the x and y coordinates of each stone and its nearest goal position.
- **Hungarian Algorithm:** Computes an optimal assignment of stones to switches, minimizing the total weighted Manhattan distance.

A* is optimal if $h(n)$ is *admissible* (never overestimates the true cost) and *consistent* (satisfies the triangle inequality).

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.

5.6 Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that selects the next node based on a heuristic function. Unlike A*, it considers only the estimated cost to the goal and ignores the cost incurred so far. This makes it faster but less optimal, as it can get trapped in local optima. GBFS is often used when a fast, approximate solution is needed rather than the absolute shortest path.

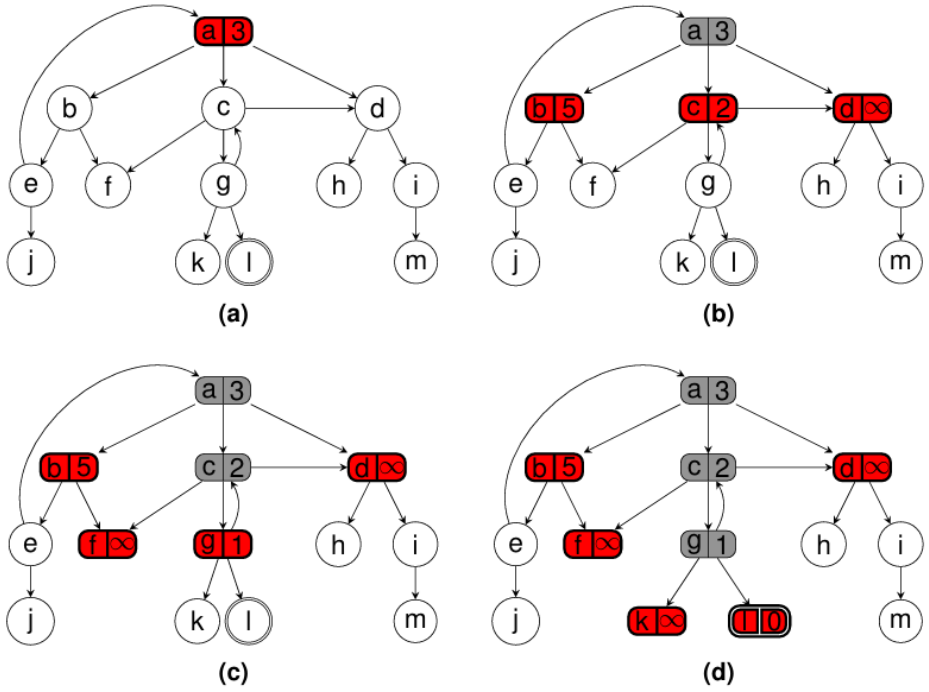


Figure 4: Greedy Best-First Search

Pseudocode

Algorithm 6 Greedy Best-First Search (*start, goal, heuristic*)

```
priority queue ← [(start, heuristic(start))]
while priority queue is not empty do
    node ← dequeue(priority queue)
    if node = goal then
        return path
    end if
    for all neighbor in valid moves do
        if neighbor not visited then
            mark neighbor as visited
            enqueue(priority queue, (neighbor, heuristic(neighbor)))
        end if
    end for
end while
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 and deadlock cells.
- **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.

5.7 Swarm Algorithm and Variants

Swarm-based algorithms are inspired by the collective behavior of biological swarms, such as flocks of birds or colonies of ants. These algorithms use multiple agents working in parallel to explore the search space efficiently. Instead of relying on a single expanding frontier like traditional search algorithms, swarm algorithms distribute the search effort across multiple agents, enabling faster convergence in large or complex environments. They are particularly useful in pathfinding, optimization, and multi-agent coordination problems.

Convergent Swarm Algorithm

The Convergent Swarm Algorithm enhances the basic swarm approach by ensuring that multiple search agents progressively converge toward an optimal or near-optimal solution. It balances exploration and exploitation, reducing redundant searches while maintaining diversity in the search space. By dynamically adjusting the movement and evaluation strategies of agents, the convergent swarm improves solution quality and efficiency over time.

Bidirectional Swarm Algorithm

The Bidirectional Swarm Algorithm further optimizes search efficiency by initiating two simultaneous search processes—one from the start position and another from the goal. These two search fronts progress independently until they meet, significantly reducing the search time compared to unidirectional approaches. This variant is particularly effective in large graphs, where expanding from both ends minimizes unnecessary exploration and accelerates pathfinding.

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.
- **SwarmBidirectional.search()** Implements a bidirectional search variant of Swarm, exploring the state space from both the start and goal positions simultaneously.

Time and Space Complexity

Swarm Algorithm (General Case)

- **Time Complexity**: $O(k \cdot b^d)$

  where:

  - k is the number of agents.
  - b is the branching factor.
  - d is the depth of the solution.

- **Space Complexity**: $O(k \cdot S)$

  where S is the size of the state space.

Convergent Swarm Algorithm

- **Time Complexity**: $O(k \cdot b^{d'})$

  where $d' < d$, since the algorithm reduces redundant exploration.

- **Space Complexity**: $O(k \cdot S)$

Bidirectional Swarm Algorithm

- **Time Complexity**: $O(k \cdot b^{d/2})$

  where the depth of the search is approximately halved due to the bidirectional approach.

- **Space Complexity**: $O(k \cdot S)$

Comparison Table

Algorithm	Time Complexity	Space Complexity
Swarm	$O(k \cdot b^d)$	$O(k \cdot S)$
Convergent Swarm	$O(k \cdot b^{d'})$	$O(k \cdot S)$
Bidirectional Swarm	$O(k \cdot b^{d/2})$	$O(k \cdot S)$

5.8 Comparison of Pathfinding Algorithms

Algorithm	Steps	Weight	Nodes Explored	Time (ms)	Memory
BFS	**16**	695	5327	480.75	3.01 MB
DFS	85	707	444	57.66	222.34 KB
UCS	24	**405**	20404	1560.00	13.04 MB
A*	24	**405**	2146	382.55	1.92 MB
GBFS	26	**405**	**99**	**16.15**	**82.50 KB**
Dijkstra	24	**405**	29450	2040.00	18.28 MB

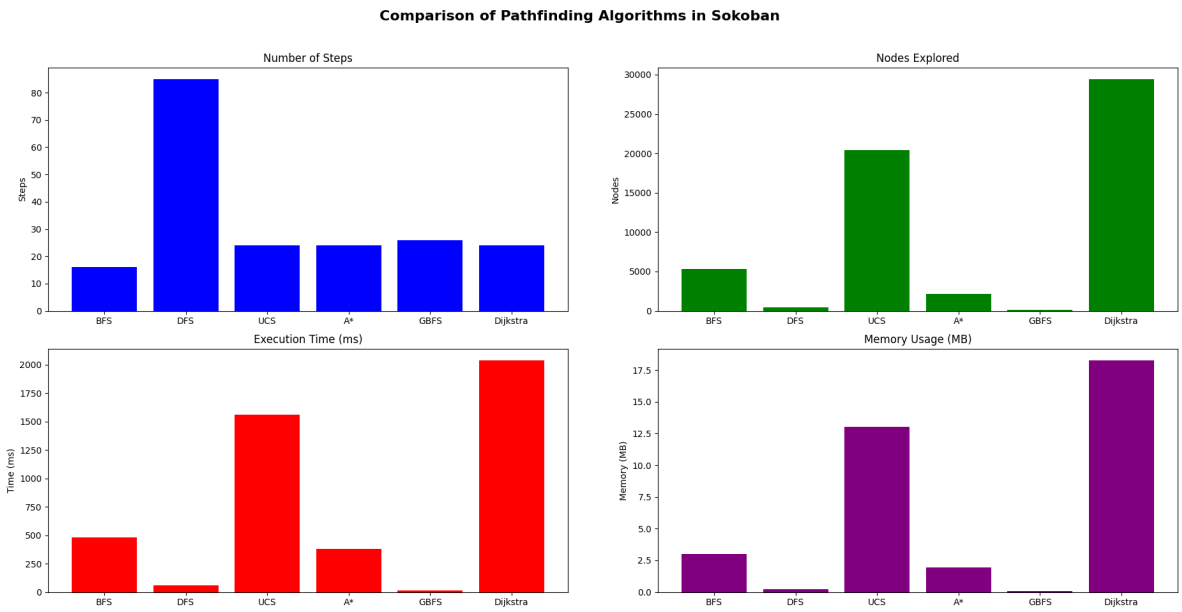


Figure 5: Comparison of Pathfinding Algorithms in Sokoban

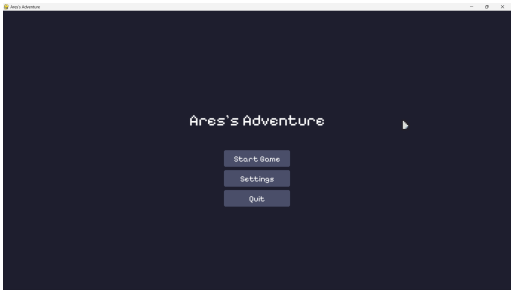
Conclusion

This report analyzed various search algorithms applied to the Sokoban problem, comparing their efficiency in terms of steps taken, nodes expanded, time, and memory usage. The results indicate that:

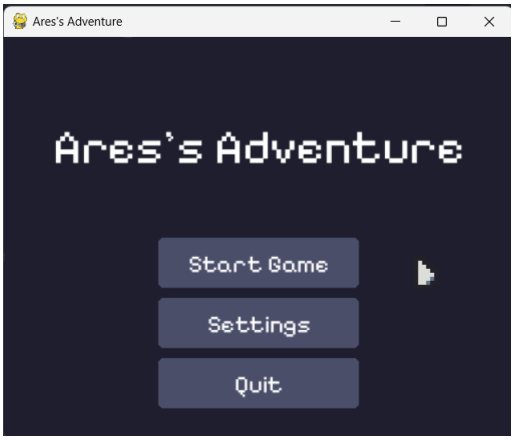
- **A*** and **UCS** produced optimal solutions but required significant memory and processing time.
- **BFS** efficiently found the shortest path in unweighted space but struggled in large graphs.
- **DFS** explored deeply but was inefficient for finding optimal paths.
- **GBFS** was the fastest algorithm but sometimes produced suboptimal solutions due to its greedy nature.
- **Dijkstra's** algorithm performed similarly to UCS but expanded more nodes due to its exhaustive nature.
- **Swarm-based approaches** showed potential in parallelized exploration, though their performance depends heavily on heuristic design.

Future improvements could involve enhanced heuristics, hybrid approaches, or reinforcement learning techniques to further optimize search performance in Sokoban and similar problems.

6 App Screenshots



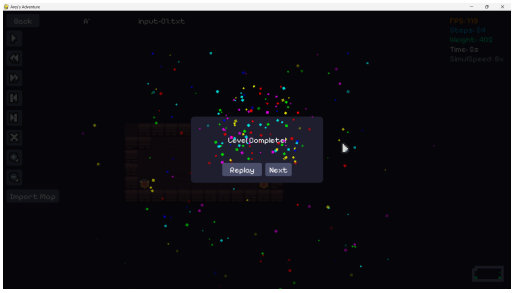
(a) Home Screen



(b) Home Responsive Screen

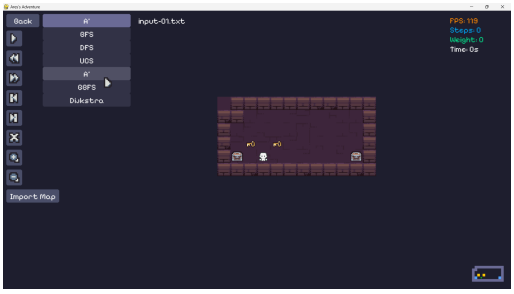
Figure 6: Home and Home (Responsive) Screens

(a) Game Screen

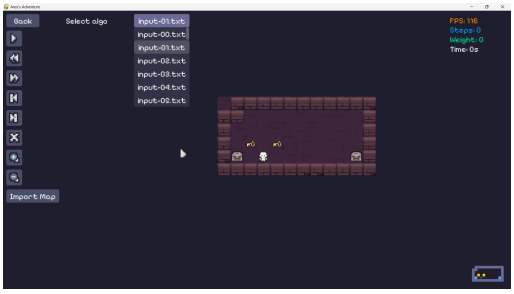


(b) Win Screen

Figure 7: Game and Win Screens

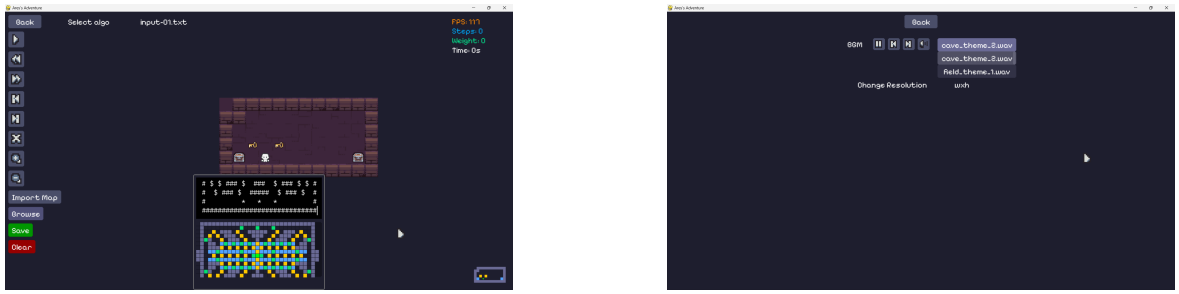


(a) Algorithm Picking Screen



(b) Map Picking Screen

Figure 8: Game and Win Screens



(a) Import Map Screen (b) Settings Screen

Figure 9: Import Map and Settings Screens

7 References

1. Rye documentation
2. pygame documentation
3. Sokoban AI Game repo: heavy-inspired by this repo for algorithms' structure and some test cases
4. 'AI in Game Playing: Sokoban Solver' - CS 221 Project Progress Report, by Anand Venkatesan, Atishay Jain, Rakesh Grewal
5. 'Using an Algorithm Portfolio to Solve Sokoban', by Nils Froleyks
6. Swarm Intelligence Algorithms