Vietnam National University

University of Science

Faculty of Information Technology

Project Report

Topic: Search Algorithm - Ares's Adventure

Course: CS426 - Introduction to Artificial Intelligence

Group members:
Le Phan Minh Khoa (22125039)
Le Thanh Lam (22125046)
Doan Cong Thanh(22125096)
Nguyen Chinh Thong(22125102)

Contents

1 Work Assignment

Task	Name	Percentage
Implement BFS	Nguyen Chinh Thong	12.5%
Implement DFS	Doan Cong Thanh	12.5%
Implement UCS	Le Phan Minh Khoa	12.5%
Implement A*	Le Thanh Lam	12.5%
Design UI	Le Thanh Lam	12.5%
Testing	Nguyen Chinh Thong	12.5%
Report	Doan Cong Thanh, Le Phan Minh Khoa	25%

Table 1: Task Assignments

2 Self Evaluation

Table 2: Grading Criteria

No.	Details	Score
1	Implement BFS correctly.	10%
2	Implement DFS correctly.	10%
3	Implement UCS correctly.	10%
4	Implement A* correctly.	10%
5	Generate at least 10 test cases for each level with different attributes.	10%
6	Result (output file and GUI).	15%
7	Videos to demonstrate all algorithms for some test case.	10%
8	Report your algorithm, experiment with some reflection or comments.	25%
Total		**100%**

3 Problem Formulation

Problem Statement We are solving the Sokoban puzzle, in which the main character navigates a maze, trying to push all the boxes to the target position. Additionally, each box has its own weight, which affects the cost incurred to solve this problem. Here are basic rules of Sokoban;

- Movement: The player can only push boxes; they can't pull them.
- Single Box Pushing: Only one box can be pushed at a time.
- Goal: All boxes must be placed on the designated storage spots to win the level.

- Walls and Obstacles: The player must navigate around walls or barriers, which adds complexity to finding the solution.

Search problem Let the main character be the problem-solving agent which simulates sequences of actions in its model, searching until it finds a solution. Whenever the agent takes an action, this leads the environment into a state.

Environment The Environment is the surrounding world around the agent which is not part of the agent itself. In this problem, we define environment as a bag of immutable features of a specific sokoban configuration. For example, the position of wall or goals inside the sokoban maze will remain unchanged as the agent solves this problem, hence we will save it inside the environment. Our environment consists of these constants:

- Maze: The list of strings stores necessary information about the map. In particular, only consisting of the position of walls and goals, which is not changed during the search.
- Goals list: A list of integers pairs referencing to the position of goals.

State space A state space contains a set of possible states in which an environment can be.

State Our state consists of the following variables.

- Player position: The coordination of the player on a two-dimensional map represented by 2 integers $x\_coordination$ and $y\_coordination$.
- A map of boxes: tracking the position of each box is associated with its weight.

Initial State The valid state in which the agent starts as in the original Sokoban game.

Goal State A state that meets the objective of this problem: All boxes are placed in the target places.

Actions The agent can execute these four actions: MOVE UP, MOVE LEFT, MOVE DOWN, MOVE RIGHT. These actions could be valid or invalid at specific situations .

Transition Model A state resulted after applying a valid action on the previous one. This could change the value of two variables in the state:

- Player position: As the player moved.
- The map of boxes: The player could move the box to another position.

The valid and invalid actions are defined as below:

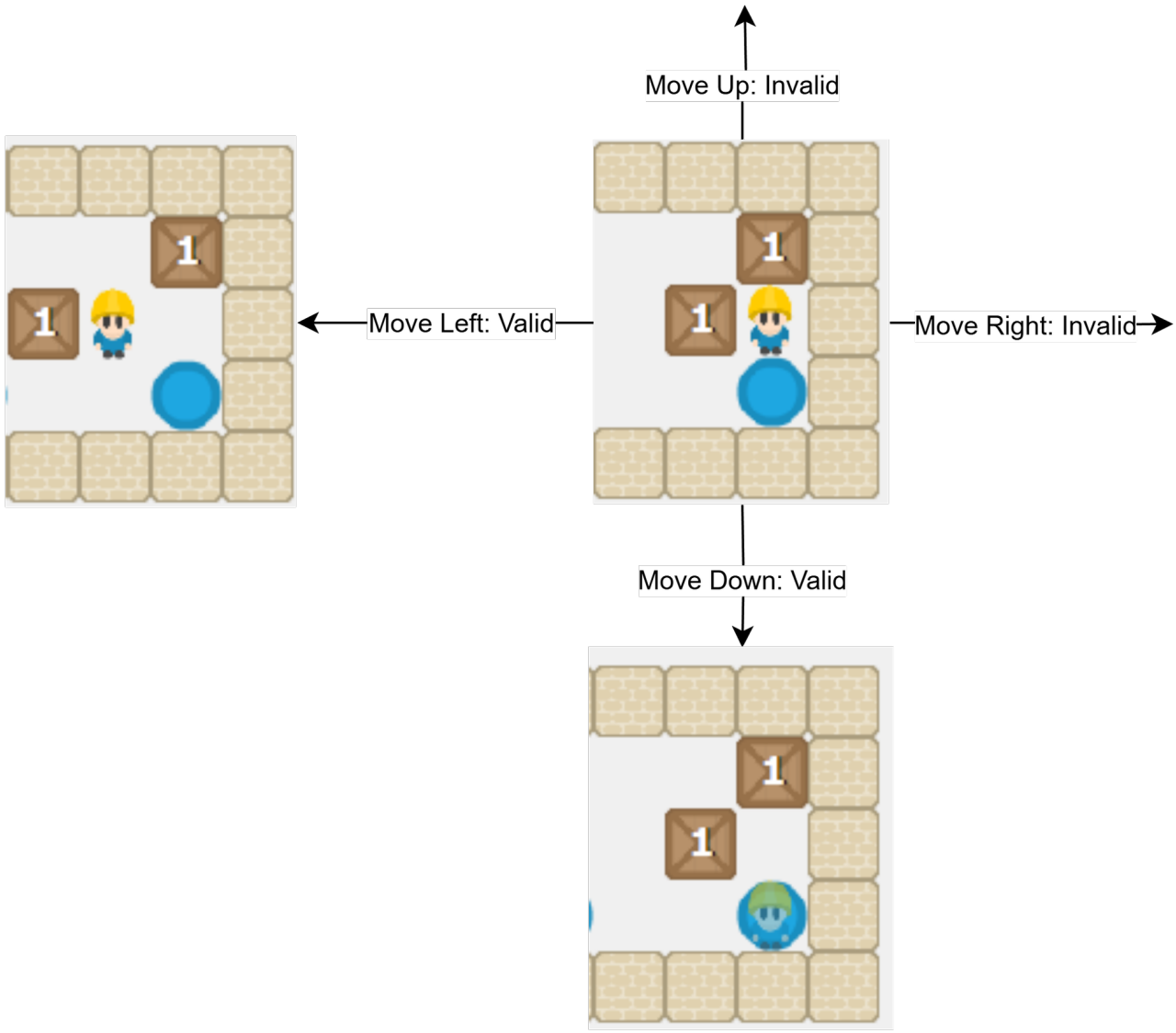

Figure 1: Example for valid and invalid moves

- A valid action: occurs when the player attempts to either move to an adjacent empty block or push a box into an empty block or goal position.
- An invalid action occurs when the player's intended movement is blocked. This happens in two main cases:
  - The player attempts to move into the wall.
  - The player push the box, but the box is blocked either by the wall or another box in the direction of the push.

Action cost The effort consumed to perform the action depends on the type of it:

- Moving character: 1 (unit cost)
- Pushing box: 1 + weight of box (unit cost)

4 Implementation Detail of Algorithms

4.1 State, Node, Environment, Problem

According to the problem formulation given, we create 3 basic classes to represent the problem, which are State, Problem and Node.

4.1.1 State

This class stores the basic information of a state (those were mentioned in *3 Problem Formulation*). The corresponding class diagram is like below (we only mention some important attributes and methods of this class):

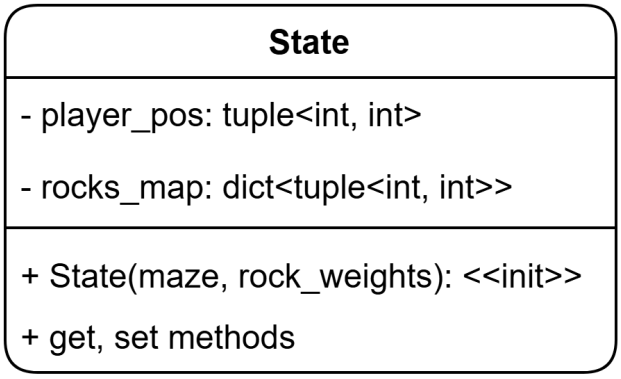

Figure 2: State Class

4.1.2 Search Data Structure - Node

In order to perform searching, we need a data structure designated for searching, that is the Node class. It consists of:

- A state
- Its parent state
- The action that causes the transition from parent state to the state this node contains.
- The total cost of traversing from initial state to its state.

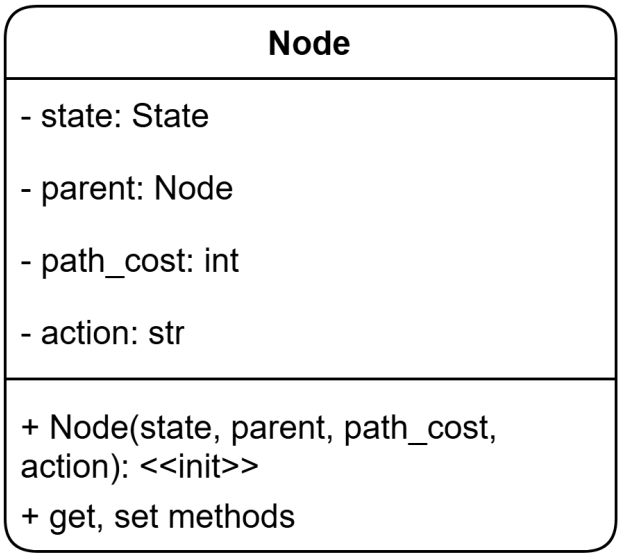

Figure 3: Node Class

4.1.3 Environment

The environment only stores unchanged information of the problem we are solving. Aside from basic information of environment mentioned in *3. Problem Formulation*, we also add some constants for convenience during searching. Those are:

- `unreachable_positions`: a set of positions such that when we push a rock into, it would be impossible to bring that rock to any goals.
- `unreachable_pairs`: a map of goal and its corresponding unreachable positions (once you push a rock into these positions, it is impossible to move the rock to that goal).
- `initial_player_pos`: original player position to generate the initial state (if needed).
- `initial_rocks_map`: original rocks map to generate the initial state (if needed).

Additionally, there are also methods for getting attributes and the initial state of the problem. The corresponding class diagram is provided in Figure 4.

4.1.4 Problem

We define the problem through this class, hence it contains:

- A list of actions a player can use (MOVE UP, MOVE LEFT, MOVE DOWN, MOVE RIGHT).
- The corresponding base environment.
- Methods for goal-state checking and initial-state checking.

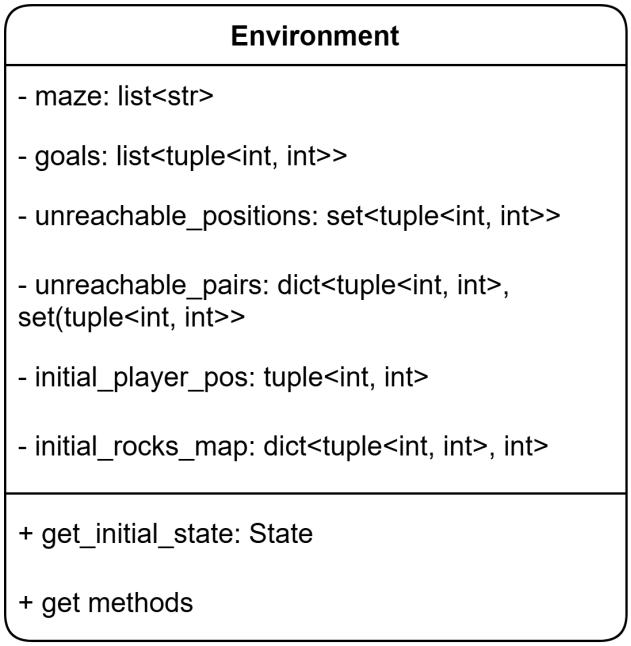

Figure 4: Environment Class

- A method that applies an action into a state and returns:
    1. Resulting state, a boolean showing whether there is a rock moved, and the cost of moving (if the action is valid, defined in *3 Problem Formulation*).
    2. None if the action is invalid in *3 Problem Formulation*.

The overall view of our search structure is demonstrated in the example of Figure 5. We can see that node A as the node containing the initial state, then it has no parent, no action and no path cost. Meanwhile, node B is a successor of node A, representing the action MOVING UP ("u") and the action cost is 1.

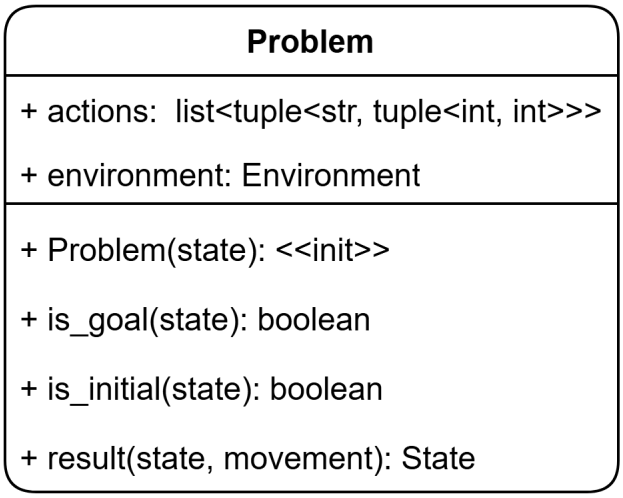

Figure 5: Problems Class

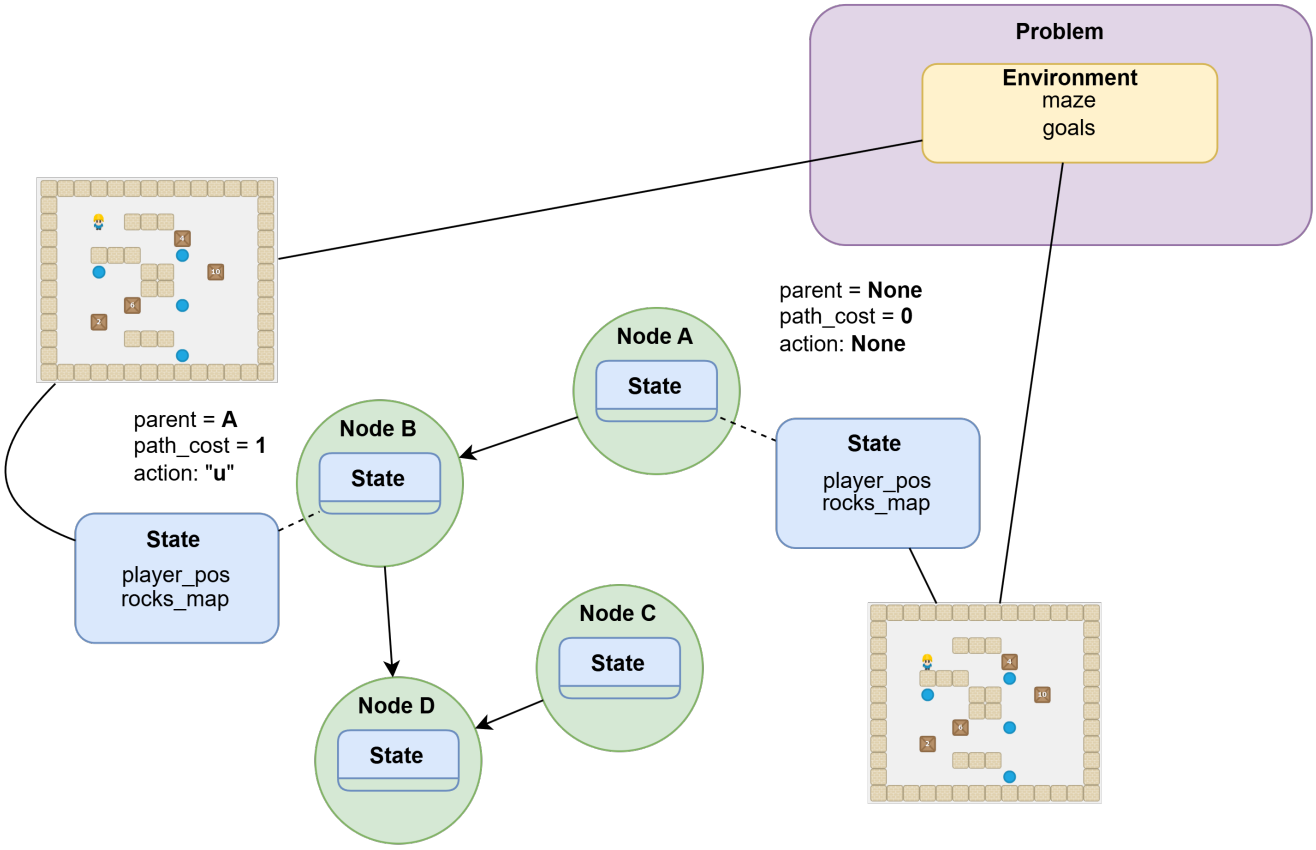

Figure 6: Search Structure for this problem

4.2 DFS

4.2.1 Implementation Idea

The Depth-First Search algorithm follows with a straight forward principle: expand the deepest node in the frontier, prioritizing the most recently node that have been discovered. This method ensure exploring as far down each branch as possible before backtracking to chose the alternative path.

4.2.2 Detail

The pseudo-code which illustrates how Depth-First search algorithm works is described in algorithm 1.

Algorithm 1 Depth-First Search

```
1: function DepthFirstSearch(problem)
2:     Initialize node ← Node(problem.initial, None, None, 0)
3:     if problem.is_goal(node.state) then
4:         return trace_path(node)                    ▷ Return the path to the solution
5:     end if
6:     Initialize frontier ← Stack([node])            ▷ Using a stack structure for DFS
7:     Initialize reached ← Set([node.state])
8:     nodes_generated ← 1
9:     while frontier is not empty do
10:        node ← frontier.pop()
11:        for all (action, movement) in problem.actions do
12:            (child_state, rock_moved, moving_cost) ← problem.result(node.state, movement)
13:            if child_state is None then
14:                continue
15:            end if
16:            if child_state not in reached then
17:                child_node ← Node(
                       state=child_state, parent=node,
                       action=(action.upper() if rock_moved else action),
                       path_cost=node.path_cost + moving_cost)
18:                if problem.is_goal(child_state) then
19:                    return trace_path(child_node)          ▷ Return the solution path
20:                end if
21:                reached.add(child_state)
22:                frontier.append(child_node)
23:                nodes_generated ← nodes_generated +1
24:            end if
25:        end for
26:    end while
27:    return None                                    ▷ No solution exists
28: end function
```

Explanation

1. Function Initialization (Line 2)
   - The function initializes the starting node (`node`) using `problem.initial`.
   - This node contains:
     - the initial state,
     - no parent (since it's the root),
     - no action,
     - no movement
     - a path cost of 0.
2. Goal Check for Initial Node (Lines 3-5)
   - Immediately checks if the initial node itself is a goal.
   - If it is, the function returns the path leading to this node using `trace_path(node)`.
3. Data Structure Initialization (Lines 6-8)
   - `frontier` is initialized as a stack containing only the initial node (`Stack([node])`).
     - In DFS, a stack is used to track nodes to be explored in a Last-In-First-Out (LIFO) manner.
   - `reached` is a set that keeps track of states already explored to avoid revisiting nodes. This implementation looks quite different to the DFS algorithm application on search tree due to the graph space of the states. In other words, the same states can be regenerated on other paths.
     - This set is initialized with the initial node's state.
   - `nodes_generated` is a counter that keeps track of the total number of nodes generated during the search.
     - It is initialized to 1 since we start with the initial node.
4. Main Loop (Lines 9-25)
   - The `while` loop runs as long as there are nodes in the frontier.
   - If the frontier is empty and no solution is found, the function will return `None` at the end, indicating no solution.
5. Node Expansion (Line 10)
   - The algorithm takes the last node from the frontier (pop operation), making it the current node to explore.
6. Exploring Actions (Lines 11-12)
   - The algorithm iterates over all possible actions from the current node's state using `problem.actions`.

- For each action, it calculates:
  - `child_state`: The resulting state after performing the action.
  - `rock_moved`: A flag indicating if a specific condition (like moving a rock) occurred.
  - `moving_cost`: The cost associated with this action.
- `problem.result(node.state, movement)` defines how the problem transitions from the current state to a new state.

7. Handling Null States (Lines 13-15)
   - If `child_state` is `None`, the action is ignored (`continue` statement).
   - This can occur when the action is invalid or leads outside the problem's boundaries.
8. Cycle Avoidance Check (Line 16)
   - The algorithm checks if `child_state` has already been reached (i.e., it is in `reached`).
   - This prevents revisiting and creating cycles in the search.
9. Creating a New Node (Lines 17-18)
   - If the state has not been visited, a new node (`child_node`) is created with:
     - `state`: The new state after the action.
     - `parent`: The current node (for tracing back the path).
     - `action`: The action taken to reach this state. The action is capitalized if `rock_moved` is true.
     - `path_cost`: The total path cost, updated by adding `moving_cost` to the current node's path cost.
10. Goal Check for Child Node (Lines 19-20)
    - After creating the child node, it checks if this new node is a goal state.
    - If `problem.is_goal(child_state)` is `True`, it returns the path to the solution using `trace_path(child_node)`.
11. Adding to Reached and Frontier (Lines 21-23)
    - If the child node is not a goal, `child_state` is added to `reached`, ensuring it won't be revisited.
    - The child node is added to the `frontier` stack, and `nodes_generated` is incremented to reflect the new node creation.
12. End of Loop and Return (Lines 24-28)
    - After all possible actions for the current node have been explored, the algorithm returns to the `while` loop to continue with the next node in the frontier.
    - If the frontier is emptied without finding a solution, `None` is returned, indicating no solution exists.

4.2.3 Evaluation

- Completeness: In this scenario, the Sokoban puzzel has finite state spaces and thus, it will eventually find the solution if it exists.
- Optimality: The solution returned using DFS might not be the shortest or least-cost path if there exists multiple paths to the goal state since it try to expand as far possible as the current branch. Thus, it is not optimal.
- Time Complexity: The time complexity of DFS depends on the branching factor b [1] and the maximum depth m of the search tree. In the worst case, DFS explores all nodes in the search tree, leading to a time complexity of $O(b^m)$.
- Space Complexity: As we discussed earlier, Depth-first search apply on this problem have to track not only the current path (list of nodes contain corresponding state) it follows but also the state have reached. Therefore, the space complexity is $O(b^m)$.

4.3 BFS

4.3.1 Implementation Idea

The Breadth-First Search (BFS) algorithm follows a different principle: it expands the shallowest node in the frontier, prioritizing nodes that were discovered earliest. BFS explores all nodes at the present depth level before moving on to nodes at the next depth level. This approach ensures that all nodes closer to the root are visited before any nodes deeper in the structure, making BFS an ideal choice for finding the shortest path in an unweighted graph.

4.3.2 Detail

The pseudo-code in Algorithm 2 bellow illustrates how Breadth-First search algorithm works.

Explanation The evaluation of the Breadth-First Search (BFS) algorithm is similar to that of the Depth-First Search (DFS) algorithm. The primary difference lies in the data structure used: BFS uses a queue to track nodes in a First-In-First-Out (FIFO) order, while DFS uses a stack. This difference affects the order in which nodes are explored, with BFS exploring nodes level by level and DFS diving as deeply as possible along each branch before backtracking.

4.3.3 Evaluation

- **Completeness:** BFS is complete, meaning that if there is a solution to the problem, BFS will always find it. This is particularly advantageous in puzzles like Sokoban with finite state spaces.
- **Optimality:** In this problem, we try to solve Sokoban with weighted rocks. Thus, BFS in this case is not optimal.

[1] The average number of successors per node

Algorithm 2 Breadth-First Search

```
function BreadthFirstSearch(problem)
    Initialize node ← Node(problem.initial, None, None, 0)
    if problem.is_goal(node.state) then
        return trace_path(node)                          ▷ Return the path to the solution
    end if
    Initialize frontier ← Queue([node])                  ▷ Using a queue structure for BFS
    Initialize reached ← Set([node.state])
    nodes_generated ← 1
    while frontier is not empty do
        node ← frontier.pop()
        for all (action, movement) in problem.actions do
            (child_state, rock_moved, moving_cost) ← problem.result(node.state, movement)
            if child_state is None then
                continue
            end if
            if child_state not in reached then
                child_node ← Node(
                    state=child_state, parent=node,
                    action=(action.upper() if rock_moved else action),
                    path_cost=node.path_cost + moving_cost)
                if problem.is_goal(child_state) then
                    return trace_path(child_node)        ▷ Return the solution path
                end if
                reached.add(child_state)
                frontier.append(child_node)
                nodes_generated ← nodes_generated +1
            end if
        end for
    end while
    return None                                          ▷ No solution exists
end function
```

- **Time Complexity:** The time complexity of BFS is $O(b^d)$ where d is the depth of the shallowest solution. In the worst case, BFS may need to explore all nodes up to the depth d of the solution, leading to exponential growth in time for large trees or high branching factors.
- **Space Complexity:** BFS has a space complexity of $O(b^d)$, as it must store all nodes at the current level before moving to the next level. However, it is essential for applications requiring optimality, as BFS ensures the shortest path is found.

4.4 UCS

4.4.1 Implementation Idea

UCS (Uniform-cost search) resembles the mechanism of Dijkstra's algorithm which keeps track and updates the total cost of the path from the initial state to the current state. Before taking an action, the agent chose the node with the lowest path cost lead to that state in the frontier. This ensures the optimal action lead to the unique state.

4.4.2 Detail

Algorithm 3 describes how UCS works.

Explanation

1. Function Initialization (Line 2)
   - A node is initialized using `problem.initial`. This node represents the starting point with:
     - `state`: The initial state.
     - `parent`: `None` (it has no parent as it is the root node).
     - `action`: `None` (no action led to this state).
     - `path_cost`: 0 (the starting node has no cost to reach).
2. Data Structure Initialization (Lines 3-5)
   - `frontier`: Initialized as a min-heap (priority queue).
     - The min-heap ensures that nodes are popped in increasing order of their path cost.
   - `reached`: A dictionary that records the minimum cost to reach each state.
     - This helps in avoiding unnecessary exploration of paths with higher costs.
   - `nodes_generated` is a counter that tracks the number of nodes generated during the search, starting from 1 (for the initial node).
3. Adding Initial Node to Frontier (Line 6)
   - The initial node is pushed into the `frontier` with a priority value of 0 (its path cost).
4. Main Loop (Lines 7-26)
   - The `while` loop continues as long as there are nodes in the `frontier`.

Algorithm 3 Uniform Cost Search (UCS)

```
function UniformCostSearch(problem)
    Initialize node ← Node(problem.initial, None, None, 0)
    Initialize frontier as an empty min-heap (priority queue)
    Initialize reached as an empty dictionary  ▷ Stores the minimum cost to reach each state
    nodes_generated ← 1
    heapq.heappush(frontier, (0, node))                    ▷ Insert initial node with cost 0
    while frontier is not empty do
        _, node ← heapq.heappop(frontier)          ▷ Pop the node with the lowest path cost
        state ← node.state
        if problem.is_goal(state) then
            return trace_path(node)                        ▷ Return the path to the solution
        end if
        for all (action, movement) in problem.actions do
            (child_state, rock_moved, moving_cost) ← problem.result(node.state, movement)
            if child_state is None then
                continue
            end if
            child_cost ← node.path_cost + moving_cost
            if child_state not in reached or reached[child_state] > child_cost then
                child_node ← Node(
                    state=child_state, parent=node,
                    action=(action.upper() if rock_moved else action),
                    path_cost=child_cost)
                reached[child_state] ← child_cost
                heapq.heappush(frontier, (child_cost, child_node))
                nodes_generated ← nodes_generated + 1
            end if
        end for
    end while
    return None                                                ▷ No solution exists
end function
```

- If the `frontier` becomes empty without finding a solution, it returns `None` to indicate failure.

5. Expanding the Least-Cost Node (Line 8)
   - The algorithm removes the node with the lowest path cost from `frontier` (using `heappop` from the priority queue).
   - The state of this node is stored for further operations.
6. Goal Check (Lines 9-11)
   - It checks if the current state is a goal state (`problem.is_goal(state)`).
   - If the goal is found, the function immediately returns the solution path to this node using `trace_path(node)`.
7. Exploring Actions (Lines 12-13)
   - The algorithm iterates over all actions available from the current node's state.
   - For each action, it calculates:
     - `child_state`: The new state reached by applying the action.
     - `rock_moved`: A flag indicating specific conditions (like moving a rock, which might modify the path cost).
     - `moving_cost`: The cost associated with this action.
8. Handling Null States (Lines 14-16)
   - If `child_state` is `None`, this action is ignored (via `continue`).
   - This can happen if an action is invalid or goes outside the problem's boundaries.
9. Calculating Child Path Cost (Line 18)
   - The `child_cost` is calculated by adding `moving_cost` to the `path_cost` of the current node.
10. Cycle Avoidance and Cost Comparison (Lines 19-20)
    - The algorithm checks if `child_state` is either not in the `reached` dictionary or if the recorded cost for `child_state` in `reached` is higher than `child_cost`.
    - This ensures that only the least-cost path to each state is kept.
11. Creating and Adding Child Node (Lines 21-23)
    - If the above condition is satisfied, a new node (`child_node`) is created with:
      - `state`: The `child_state`.
      - `parent`: The current node (for path tracing).
      - `action`: The action that led to this state, capitalized if `rock_moved` is true.
      - `path_cost`: The cumulative cost of the path to this node (`child_cost`).

- The `reached` dictionary is updated with the new minimum cost for `child_state`.
- The child node is then pushed into the `frontier` with a priority equal to `child_cost` to maintain the min-heap property.
- `nodes_generated` is incremented to reflect the creation of a new node.

12. End of Loop and Return (Lines 24-28)
    - After all possible actions for the current node are explored, the algorithm returns to the `while` loop to continue with the next node in the `frontier`.
    - If the `frontier` is exhausted without finding a solution, the function returns `None`, indicating that no solution exists.

4.4.3 Evaluation

- Completeness: In the finite state spaces, UCS will eventually find the solution if it exists.
- Optimality: The late-goal test which is applied on this algorithm ensures the least-cost path to the state first examined. Thus, the first solution found will have a cost that is at least as low as the cost of any other node in the frontier.
- Time complexity: In general, UCS explores all paths with cost $\leq C^*$, leading to a complexity of $O(b^{1+\lfloor C^*/\epsilon \rfloor})$.
  - C^*: the cost of the optimal solution.
  - ϵ: the minimum positive cost per action, where $\epsilon > 0$.
- Space Complexity: The space complexity is also $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, as UCS stores all generated nodes in memory.

4.5 A*

4.5.1 Implementation Idea

A* search improves UCS by combining the actual path cost so far (g) with a heuristic estimate (h) of the remaining cost to the goal and expanding nodes based on this combined cost. If the heuristic is suitable, we will be better directed to the goal state.

4.5.2 Detail

The pseudo-code showing how A* works is in Algorithm 4 below.

A* search is implemented in almost the same way with UCS, the only difference is that A* uses a heuristic function to evaluate each generated state and expands node based on the combined cost (f(n) = g(n) + h(n)) instead of actual cost (g(n)) alone. As the implementation of A* and UCS are mostly the same, I will focus on how the heuristic function works.

Algorithm 4 A* Search

```
function AStarSearch(problem)
    Initialize node ← Node(problem.initial, None, None, 0)
    Initialize frontier ← PriorityQueue([(0, node)])    ▷ Using a priority queue for A*
    Initialize reached ← Dictionary{node.state : heuristic(node.state)}    ▷ Tracks the combined cost (g + h) for each state
    Initialize heuristic ← Dictionary{node.state : heuristic(node.state)} ▷ Stores heuristic values
    nodes_generated ← 1
    while frontier is not empty do
        (cost, node) ← frontier.pop()
        state ← node.state
        if problem.is_goal(state) then
            return trace_path(node)                                    ▷ Return the solution path
        end if
        for all (action, movement) in problem.actions do
            child_state, rock_moved, moving_cost ← problem.result(node.state, movement)
            if child_state is None then
                continue
            end if
            if child_state not in reached then
                heuristic[child_state] ← heuristic_cost(child_state)        ▷ Compute heuristic if not previously reached
            end if
            child_combined_cost ← node.path_cost + moving_cost + heuristic[child_state]
            if child_state not in reached or reached[child_state] > child_combined_cost then
                child_node ← Node(
                    state=child_state, parent=node,
                    action=(action.upper() if rock_moved else action),
                    path_cost=child_combined_cost - heuristic[child_state])
                reached[child_state] ← child_combined_cost
                frontier.push((child_combined_cost, child_node))
                nodes_generated ← nodes_generated +1
            end if
        end for
    end while
    return None                                                        ▷ No solution exists
end function
```

Explanation for heuristic function This heuristic function estimates the cost from the current state to a goal state. And we have the general formula for this heuristic for state n:

$$h(n) = \text{MinimumCostBipartiteMatching}(\mathcal{G})$$

where: $\mathcal{G} = (\mathcal{R}, \mathcal{G}, \mathcal{E})$ is a complete bipartite graph consisting of:

- The set of rock vertices $\mathcal{R}$.
- The set of goal vertices $\mathcal{G}$.
- The set of edges $\mathcal{E}$ having an endpoint in $\mathcal{R}$ and an endpoint in $\mathcal{G}$, with the weight is the estimated Manhattan distance between the corresponding rock and goal multiplying by the rock weight:
$$E_{ij} = E_{ji} = \text{EstimatedManhattanDistance}(R_i, G_j) * W_i$$

The intuition of our function is actually pretty simple: we anticipate that, to reach the goal state, a player needs to move all rocks to the corresponding goals, one by one. Then, each rock would be assigned to a goal, one by one, leading to a **minimum cost bipartite matching** problem.

In order to solve this problem, we need the cost matrix showing the weight of all edges in this perfect bipartite graph. However, in a complex problem like Sokoban, calculating this cost is very difficult, as the maze structure is hard to generalize, and rocks often stand in others way.

Hence, to find the heuristic value, we try to relax the problem a little bit by dividing the problem into n smaller problem (with n be the number of goals). Each smaller problem will removes all $n-1$ rocks from the original problem, so there is only one rock left. Up to now, subproblem contains only one rock, multiple fixed goals, fixed wall, hence we can calculate the cost from the rock to each goal easily! After finishing cost calculation, we create the cost matrix and use Hungarian algorithm to find the minimum cost matching of the mentioned graph $\mathcal{G}$, the result is also the heuristic value we need to find.

It is also important to note that, when we relax this problem to n subproblem, we can precalculate the minimum distance from a goal to all position in the maze using simple BFS n times. Hence, when computing the heuristic value, we only need to look up the minimum distance, multiplying it by (weight + 1) to get the estimated minimum cost from a rock to a goal.

Our heuristic function for A* search is shown in the Algorithm 5. A brief explanation for this pseudo-code is given below:

1. Collect rock positions with weights, goal positions from the state, and the dictionary of distance matrix for each goal.
2. Calculate the cost matrix for Hungarian algorithm: we already have the distance matrix for each goal, so we only need to look up the estimated distance calculated in the subproblem and then multiplying by (weight + 1) to get the estimated cost.
3. Use the Hungarian (Linear Sum Assignment) algorithm to find the minimum-cost assignment between rocks and goals.
4. Return the sum of these optimal pair costs, that is the heuristic value we need to find.

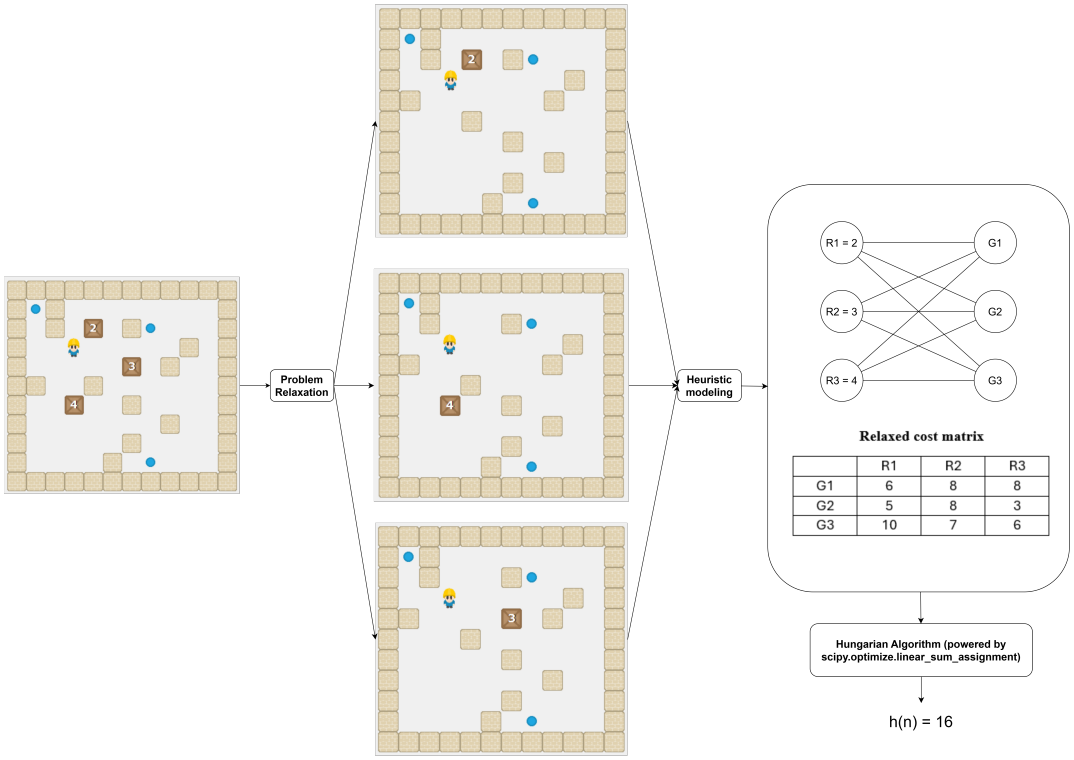

Figure 7: Heuristic calculation procedure

Algorithm 5 Heuristic Cost Calculation

function HEURISTICCOST(`state`)
  `rocks_list` ← list of (`rock_pos`, `weight`) from `state.rocks_map`
  `goal_positions` ← `self.problem.environment.goals`
  `distance_matrices` ← `self.problem.environment.distance_matrices`
  Initialize `cost_matrix` ← empty list
  for each `goal_pos` **in** `goal_positions` **do**
    Initialize `cost_row` ← empty list
    for each (`rock_pos`, `weight`) **in** `rocks_list` **do**
      `distance` ← `distance_matrices[goal_pos][rock_pos[0]][rock_pos[1]]`
      `cost` ← `distance` × (`weight` + 1)
      Append `cost` to `cost_row`
    end for
    Append `cost_row` to `cost_matrix`
  end for
  (`row_ind`, `col_ind`) ← `LinearSumAssignment(cost_matrix)`
  return $\sum_{i=1}^{|\texttt{row\_ind}|}$ `cost_matrix[row_ind[`i`]][col_ind[`i`]]`
end function

4.5.3 Evaluation

Before going into evaluation of the algorithm, we will have a brief review of the heuristic used. After some analysis, we have some properties like below:

- We used minimum cost bipartite matching for matching each rock to each goal, this ensures the cost we achieve is the smallest cost possible. Also, the problem would require more cost as in reality, rock can stand in the way of others. Hence, the heuristic function never overestimates the true cost to the goal.
- The heuristic does not account for free move (moving without pushing rock) of the player, hence we have the properties:

  $$h(n) - h(n^{'}) = 0 \leq c(n, n^{'})$$

  where transition from n to n' does not involve pushing rock. This ensures the consistency of heuristic function when player perform free move.
- In the case of pushing move, when we push a rock closer to a goal, the heuristic is ensured to decrease (at most) by the weight of that rock + 1. Hence we also have:

  $$h(n) - h(n^{'}) \leq c(n, n^{'}) = weight + 1$$

Then, this heuristic is consistent.
Now we come to the evaluation of corresponding A* algorithm.

- **Completeness:** A* search is complete, meaning it is guaranteed to find a solution if one exists in this problem where state space is finite.
- **Optimality:** A* is cost-optimal as it is using a consistent heuristic.
- **Time Complexity:** The time complexity of A* depends on the branching factor b, the depth d of the optimal solution, and the quality of the heuristic $h(n)$. In the worst case, A* may explore all nodes with $f(n) \leq C^*$ (where C^* is the cost of the optimal solution), leading to an exponential time complexity of $O(b^d)$ when the heuristic provides little guidance.
- **Space Complexity:** A* search has a high space complexity because it stores all generated states in memory. In the worst case, the space complexity is also $O(b^d)$, as all nodes with $f(n) \leq C^*$ may need to be retained in memory.

5 Testing Methodology

In this section, we will describe how we applied four search algorithms to the Sokoban problem by designing three different levels, each with a unique challenge. These levels were set up to explore different aspects of algorithm performance, allowing us to observe how they handle simple, moderate, and complex situations in the Sokoban environment.

5.1 Simple Case

This level is intended to let the algorithms focus on straightforward moves without much complexity.

- **Rocks and Targets:** Minimal, with only 1 to 2 rocks and targets. This reduces the complexity and allows us to test each algorithm's core movements without additional distractions.
- **Obstacles:** Rare; most paths are open, so the algorithms do not need to navigate around barriers.
- **Maze size:** Sufficiently large to allow easy movement.

5.2 Medium-Hard Case

This setup introduces several factors that increase the difficulty level, testing the efficiency of each algorithm in a more constrained space.

- **Rocks and Targets:** Increased to 3-5 rocks and targets, which demands a deeper look-ahead and more strategic planning from the algorithms.
- **Obstacles:** More densely placed, requiring the algorithms to navigate through a congested maze, which tests their path finding abilities in tighter spaces.
- **Maze Size:** Varies from small to large. This setup demonstrates each algorithm's adaptability to different maze sizes, challenging them with both dense and open layouts.

5.3 Special Case

In this level, additional factors are introduced to test how each algorithm performs in terms of speed, memory usage, and nodes explored.

- **Rock Weight:** Rocks have variable weights, making them more responsive to different pushes. We designed two similar levels with different rock weights to reveal how each algorithm's solution efficiency is affected by small changes in rock weight.
- **Redundant Paths:** The maze includes narrow paths and limited movement options, forcing the algorithms to choose a specific path. This configuration highlights each algorithm's efficiency in terms of nodes explored and memory consumption, given a constrained number of possible moves.

5.4 Experimental Approach

For each of these level types, we conducted an analysis of the algorithms' performance using the following metrics:

- **Average Runtime:** The time each algorithm takes to complete the search.
- **Nodes Expanded:** The total number of nodes expanded by each algorithm, which provides insight into their efficiency in reaching the solution.

- **Memory Used:** The maximum amount of memory each algorithm requires at a particular runtime, indicating resource requirements across different settings.
- **Solution Path Cost:** The total cost of the solution path, which depends on rock weights and maze density. The number of steps each algorithm takes to solve a level reflects the complexity of the solution path.
- **Steps Used**: the number of actions used in the solution, an additional metric (often used in traditional Sokoban problem).

After conducting these experiments, we summarize the results in tables to provide a clear comparison of each algorithm's performance across different levels. Line graphs are also used to illustrate how each algorithm performs under various conditions, providing a straightforward comparison of their strengths and limitations in solving the weighted Sokoban problem.

6 Testing

6.1 Testing Configuration

We conduct these tests on a Lenovo Legion 5 laptop with the following specifications:

- **CPU:** AMD Ryzen 7 5800H
- **GPU:** NVIDIA GeForce RTX 3060
- **RAM:** 16 GB (13.9 GB usable)
- **System Type:** 64-bit operating system, x64-based processor

All tests are performed with the laptop plugged in and running Windows 11 Home, version 23H2.

6.2 Testing Results and Insights

6.2.1 Simple case

We can see that, A* and DFS dominates the simple test case in run time, memory and nodes generated. However, A* shows its power in the optimality (as it utilizes consistent heuristic), and we can see that the total cost of A*, UCS are optimized while DFS wastes a lot of steps by trying random moves. Furthermore, UCS and BFS are much slower in time as they try to cover all cases without any knowledge of the problem like A*, they are also slower than DFS as DFS can go really deep in the search tree and eventually find the goal if the map is not too large. The memory usage of each algorithm aligns well with the theory, where DFS uses the least, while UCS and BFS needs a lot of memory to solve the sparse map. The exceptional one in memory is A*, because we have implemented deadlock detection and consistent heuristic to prun a lot of nodes.

Map	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
input-01.txt	DFS	403	2303	2282	60.77	1.27
	BFS	23	623	2411	66.34	1.25
	UCS	23	623	3539	118.37	1.73
	A*	23	623	669	28.26	0.35
input-02.txt	DFS	359	4228	6974	215.03	3.74
	BFS	27	826	131666	4757.60	74.09
	UCS	28	729	190505	9020.03	104.21
	A*	28	729	7354	365.85	4.03
input-03.txt	DFS	12503	38479	61213	1762.82	33.46
	BFS	29	831	932067	49357.91	514.24
	UCS	29	831	1320707	74407.78	694.09
	A*	29	831	64622	3438.43	34.72
input-04.txt	DFS	122	140	432	7.48	0.22
	BFS	26	42	2065	45.20	1.03
	UCS	26	42	3303	98.45	1.62
	A*	26	42	578	20.75	0.27

Table 3: Performance Comparison for Simple Case

Map	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
input-05.txt	DFS	32492	468092	203941	12157.14	110.08
	BFS	76	2650	248749	18232.03	125.97
	UCS	96	1680	249287	22221.53	133.89
	A*	96	1680	106699	16973.23	59.68
input-06.txt	DFS	31	159	55	3.00	0.02
	BFS	23	151	542	18.22	0.25
	UCS	23	151	702	26.92	0.31
	A*	23	151	368	27.41	0.16
input-07.txt	DFS	2379	7343	180489	11168.16	92.99
	BFS	146	629	671572	48801.33	359.41
	UCS	146	607	835735	85246.87	464.49
	A*	146	607	127522	20466.74	70.01
input-08.txt	DFS	1160	1345	77298	4144.21	36.10
	BFS	164	205	298122	20541.59	140.17
	UCS	164	205	298366	25639.76	142.07
	A*	164	205	323075	53729.20	143.86

Table 4: Performance Comparison for Medium-Hard

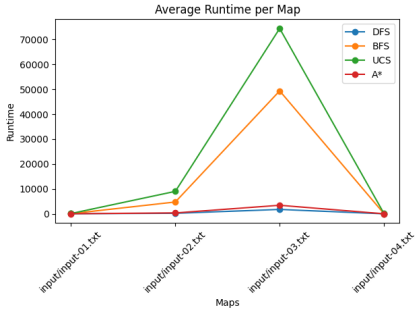

Figure 8: Run time of each algorithm on simple test case

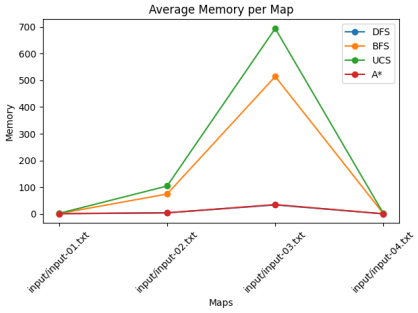

Figure 9: Memory of each algorithm on simple test case

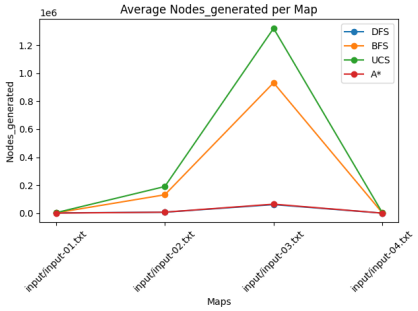

Figure 10: Nodes generated of each algorithm on simple test case

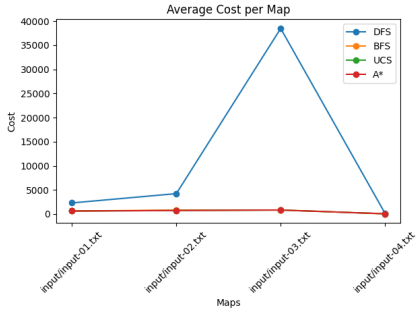

Figure 11: Cost of each algorithm on simple test case

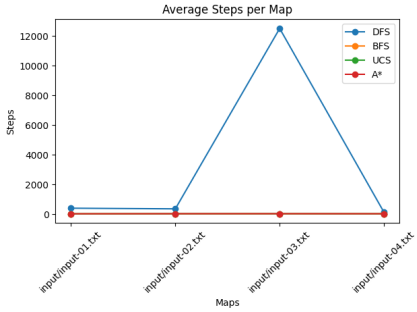

Figure 12: Steps of each algorithm on simple test case

6.2.2 Medium-Hard case

In the case of medium-hard scenarios, where the maze contains more obstacles and includes 3-5 rocks, UCS and A* clearly outperform the other algorithms. Their preference for low-cost paths and heuristics in the case of A* was important in finding better paths in the complex environment. In contrast, DFS explores far too many redundant paths. With no optimization or cost-awareness, the solution costs come out much larger and the number of steps higher in general. BFS reaches medium performance due to the fact that it systematically goes through all nodes at each level. As compared to UCS and A*, lacking the strategy to minimize cost, it is a bit slow for this dense and difficult maze setting.

6.2.3 Special case

Weight rocks For problems with two or more rocks of different weights, the existence of pairs with different weights severely impacts algorithm speed. As the number of such pairs increases, the algorithms have to consider more unique states because each combination with a unique rock weight increases the positions in state space. This therefore innate higher complexity slows the performance of the algorithms on the basis of all measures.
In particular, for test cases like input-09.txt, input-10.txt, and input-11.txt, we reduced the number of rock pairs with different weights. This reduction led to fewer permutations of the states and thus to a far simpler search space, for which all metrics improved considerably. With these settings,

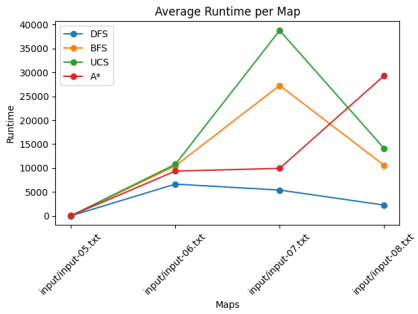

Figure 13: Run time of each algorithm on medium test case

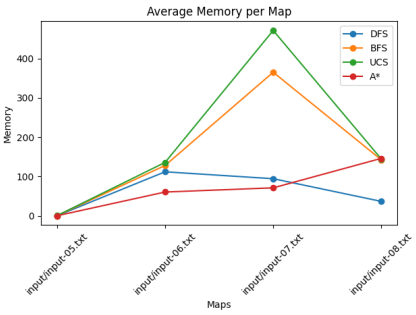

Figure 14: Memory of each algorithm on medium test case

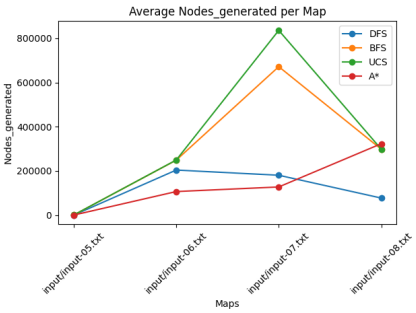

Figure 15: Nodes generated of each algorithm on medium test case

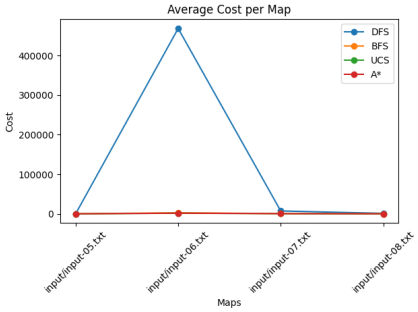

Figure 16: Cost of each algorithm on medium test case

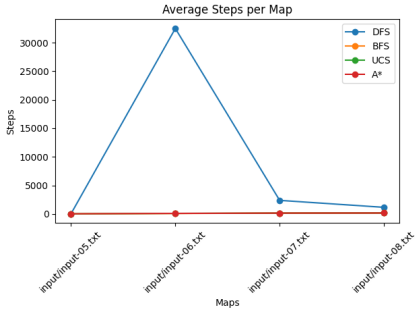

Figure 17: Steps of each algorithm on medium test case

Map	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
input-09.txt	DFS	302	1547	3898	205.05	2.01
	BFS	56	267	5743	306.47	3.22
	UCS	56	267	8021	493.16	4.12
	A*	56	267	2714	375.68	1.34
input-10.txt	DFS	302	822	3898	175.08	1.90
	BFS	56	186	5675	308.26	3.18
	UCS	56	186	8816	592.36	4.48
	A*	56	186	3537	409.13	1.75
input-11.txt	DFS	270	645	3090	147.71	1.51
	BFS	56	171	4366	202.18	2.17
	UCS	56	171	6944	518.21	3.57
	A*	56	171	3244	354.55	1.53

Table 5: Performance Comparison for Special Case (Weight Rocks)

reducing the complexity introduced by the presence of differences in the weights of the rocks allowed each algorithm to reach an optimal solution in terms of run time, memory consumption, and node expansion.

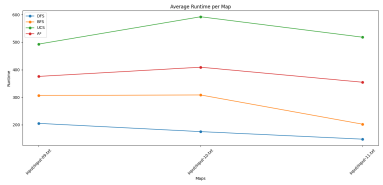

Figure 18: Run time of each algorithm on weight rock case

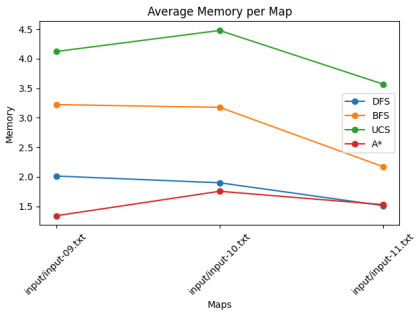

Figure 19: Memory of each algorithm on weight rock test case

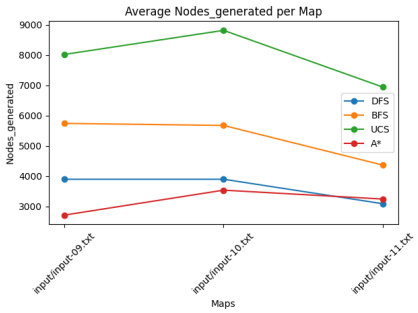

Figure 20: Nodes generated of each algorithm on weight rock test case

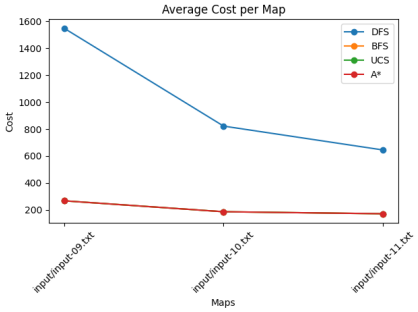

Figure 21: Cost of each algorithm on weight rock test case

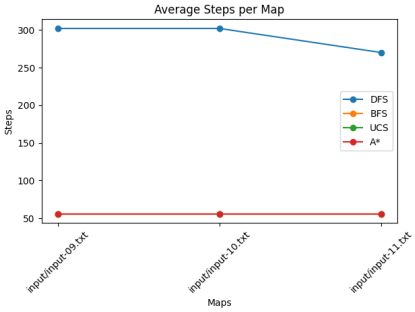

Figure 22: Steps of each algorithm on weight rock test case

Map	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
input-12.txt	DFS	4056	4336	80022	2325.42	43.22
	BFS	1782	2052	105889	3352.94	53.72
	UCS	1782	2052	107615	3731.61	58.44
	A*	1782	2052	127906	6946.44	65.12

Table 6: Performance Comparison for Special Case (Narrow Map)

Narrow map On narrow maps, where there is usually only one action to be taken by the player in each state, DFS executes very well. The limited action choices guide DFS through a single path by not allowing it to branch redundantly and hence reach the goal with very little backtracking. The path-following nature inherent in DFS goes very well with its depth-first manner of execution and hence allows it to perform the best overall in the narrow map configurations.

Non heuristic Bellow is the result of running test We noticed that each of these algorithms runs differently in terms of approaching the Sokoban problem, since their respective search strategies differ. On one hand, BFS expands nodes level by level, trying to explore large areas without a directed way to reach the goal. On the other hand, UCS and A* are very dependent on cost minimization; hence, their searches will always be directed toward lower heuristic values, which tends to push rocks as close to their respective goals as possible. However, this maze configuration

Map	Algorithm	Steps	Cost	Nodes	Time (ms)	Memory (MB)
input-13.txt	DFS	1935	6405	507370	15620.87	253.65
	BFS	x	x	x	x	x
	UCS	x	x	x	x	x
	A*	x	x	x	x	x

Table 7: Performance Comparison for Special Case (Non heuristic)

often requires a good solution, moving rocks to seemingly "worse" positions first in order to enable other rocks into place, creating temporary setbacks that UCS and A* are not designed to handle properly, since their strategy aims at paths that would seem closer to an immediate solution.
It was, however, under the in-depth search time limit of 2 minutes that DFS yielded the best performance. Unlike both algorithms, it explores the maze in a completely blind manner, without taking into consideration the cost of a path or the possibility of encountering worse obstacles later on. Because of the lack of filtering, DFS was able to continue onward and eventually attain a solution within the given time limit, even when it did not understand the concept of optimal or "worse" states of the maze.

7 Link youtube

https://www.youtube.com/watch?v=vbgIRIp_vPY&feature=youtu.be