VNUHCM

University of Science

Department of Information Technology

Project 1: Report

Topic: Search

Subject: Artificial Intelligence

Students:

Trần Phương Anh (22125004)

Lê Quốc Huy (22125031)

Nguyễn Vĩnh Khang (22125035)

Huỳnh Tuấn Minh (22125055)

November 10th, 2024

Contents

1 Work Assignment

Task Description	Assigned to	Completion Rate
UI/UX Design	Nguyễn Vĩnh Khang	100%
GUI Implementation	Huỳnh Tuấn Minh, Nguyễn Vĩnh Khang	100%
BFS Algorithm Implementation	Huỳnh Tuấn Minh	100%
DFS Algorithm Implementation	Nguyễn Vĩnh Khang	100%
UCS Algorithm Implementation	Lê Quốc Huy	100%
A* Algorithm Implementation	Trần Phương Anh	100%
Testing	Trần Phương Anh	100%
Demonstration Video	Lê Quốc Huy	100%
Writing Report	Nguyễn Vĩnh Khang, Lê Quốc Huy, Huỳnh Tuấn Minh, Trần Phương Anh	100%

Table 1: Work assignment table

2 Self Evaluation

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

Table 2: Self-evaluation of the project requirements.

3 Problem Formulation

3.1 Problem Description

In this Sokoban-inspired game, we have a grid-based maze of size n×m, where each cell can be one of the following:

- **Free space:** Ares, the agent, can freely move into these cells.
- **Wall:** Cells that Ares cannot enter or move stones.
- **Stone:** Objects that Ares can push, one at a time, onto specific cells (switches) but cannot pull.
- **Switch:** Target cells that need to be occupied by stones to complete the objective.

3.2 Objective:

The goal is to push all stones onto switches while minimizing the movement cost for Ares. The number of stones equals the number of switches, and any stone can activate any switch.

3.3 Game Mechanics and Constraints:

- **Movement:**
  - Ares can move one cell at a time in four directions (Up, Down, Left, Right).
  - Moving into a cell without pushing a stone incurs a base movement cost of 1.
  - Ares cannot move through walls or stones.
  - If a stone is in the next cell in the direction of movement and the cell beyond the stone is free, Ares can push the stone into that cell.
  - Stones cannot be pushed into walls, other stones, or out of bounds.
- **Stone Weights and Movement Costs:**
  - Each stone has a unique weight. Pushing a stone incurs an additional movement cost based on the stone's weight.

- The cost of pushing a stone is equal to the base movement cost (1) plus the weight of the stone.

3.4 Search Algorithms to Implement:

Students will use the following search algorithms to find an optimal or feasible path to achieve the objective:

1. **Breadth-First Search (BFS):** Explore the grid in a level-wise manner, with no weight considerations, making it suitable for unweighted movements but suboptimal for this weighted scenario.
2. **Depth-First Search (DFS):** Explore deeper paths before others, which may help in some grid layouts, but is generally not optimal for finding the shortest path with weights.
3. **Uniform Cost Search (UCS):** A cost-aware search algorithm that will explore paths with the least accumulated movement cost, making it suitable for handling weighted stone movements.
4. **A* Search with Heuristic:** Use a heuristic function to estimate the cost from the current state to the goal, aiming to achieve an optimal solution faster. A heuristic could be the Manhattan distance from each stone to the nearest switch, weighted by the stone weight.

3.5 Assumptions and Additional Notes:

- Each switch can be activated by any stone, regardless of weight.
- The game ends when all switches are activated.
- The initial positions of Ares, stones, walls, and switches are provided.

The implementation should define clear state representations, cost calculations for movements, and transitions for each search algorithm, incorporating the constraints and objectives of the problem as described.

4 Implement Detail of Algorithms

4.1 Breadth-first search (BFS)

4.1.1 Implementation Idea

From what we discussed about the problem, we will explore all states from the first position of Ares the player. Ares will explore all the adjacent positions and remember all the states. He will then move to one of the states he remembered and continue exploring. The strategy follows **breadth-first search (BFS)**, letting Ares explore the state level by level, making sure that all possible states are explored before moving to other states. Hence, we will guarantee to find the **shortest path** from Ares to the goal state, where all stones are on the switches.

4.1.2 Detail

In detail, we will keep track of each state with Ares' position and all the stones' positions. A global data are saved including walls' positions and switches' positions.

1. Starting from Ares' initial position, we will use a queue to keep track of all the states that Ares has explored. We also use a list named visited to keep all expanded states, so that Ares will not move to revisited states, resulting in increasing performance.

2. Ares starts to explore all possible adjacent states, which are reachable from Ares. Once he has finished, he will remember all the states in the queue. he continues exploring by popping out of the queue states, ensuring that the earlier explored states will be moved first. When exploring, Ares also checked if that new state was the goal state or not. If yes, then Ares will stop exploring and return to the movements that he has made, which are represented as a string.

3. When adding states to the queue, Ares will also mark those states to be visited and will not revisit that state (no more exploration when visiting this state again).

```python
def bfs(self, data, visited, shared_stop_event, pool, pool_size=1):
        """
        Breadth-first search algorithm to find the shortest path to the goal state
        """
        queue = deque([self])
        # Use the shared list
        visited.append((self.player_pos, tuple(self.boxes)))

        while not shared_stop_event.is_set() and queue:
            batch_size = min(len(queue), pool_size)
            current_states = [queue.popleft() for _ in range(batch_size)]

            # Check if in the batch, there is a goal state or not
            for state in current_states:
                if state.is_goal_state(data.goal_state):
                    print("done")
                    queue.clear()
                    return state, data.node_count

            results = pool.starmap(self.generate_state, [(state, data) for state in current_states])

            for neighbors in results:
                for neighbor in neighbors:
                    if neighbor is None: continue
                    # Only add the neighbor to the visited list if it hasn't been added yet
                    visited_list = [neighbor.player_pos, neighbor.boxes]
                    if visited_list not in visited:
                        visited.append(visited_list)
                        queue.append(neighbor)
                        data.node_count += 1
        return None, data.node_count
```

Listing 1: Codes for BFS's Algorithm

From the code, we use concurrency to speed up the process, by letting each processor take out

a state and generate neighbors (adjacent states) then return to the result. We also implement a shared visited list which can be shared through many processors.

However, as this concurrency is heavily CPU-bound, which means that any actions read/write file will make the process slow and result in downgrade performance. Also, the shared visited list is an issue if many processors try to access the same time, which means there is a waiting time hence, resulting in downgraded performance.

This figure illustrates a mapped-out process of Breadth-First Search (BFS) as it explores and queues states:

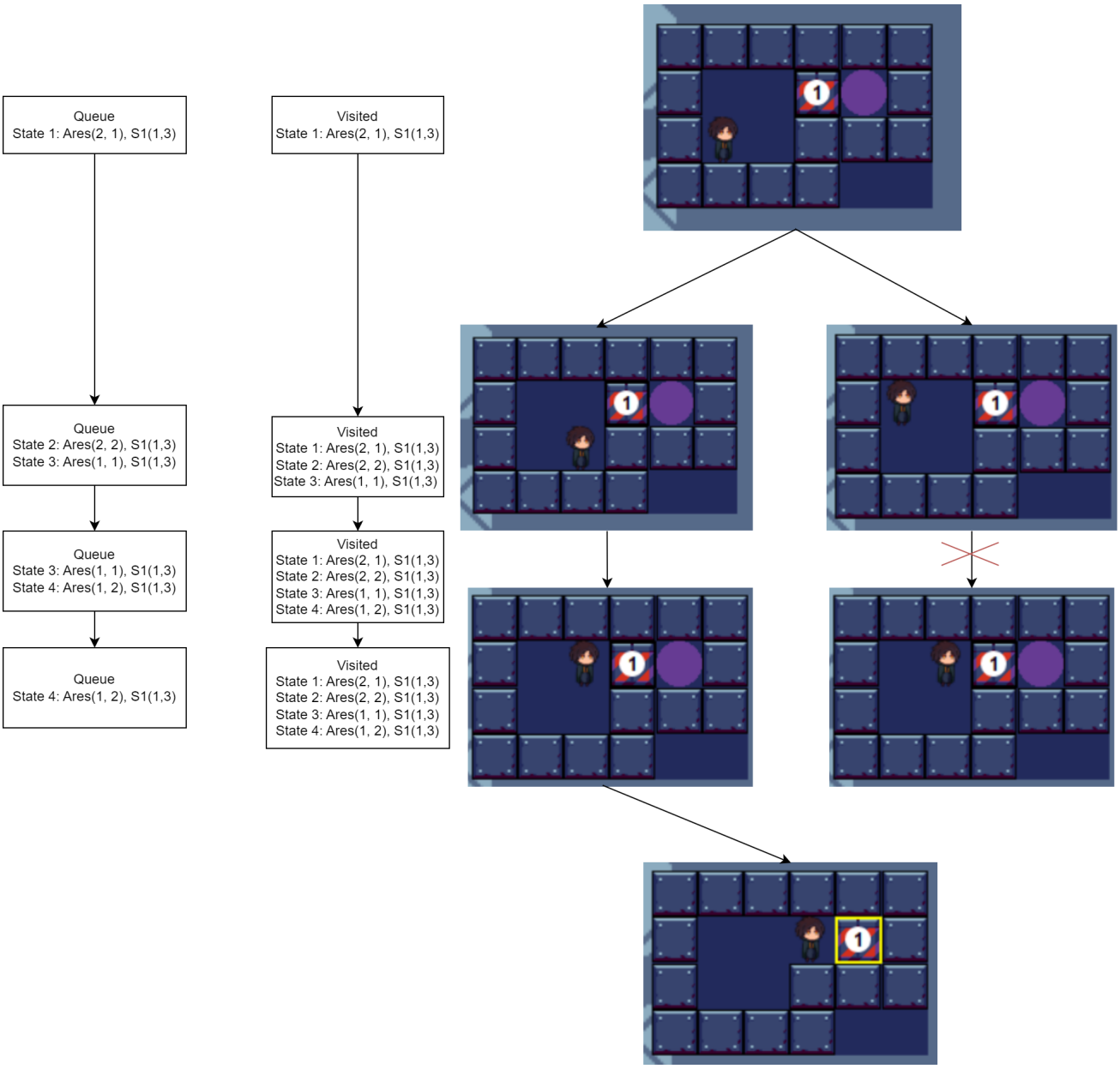

Figure 1: BFS Process Map with State Exploration and Queue Management

In Figure 1, we observe BFS progressing level by level. Starting from the top, BFS explores two neighboring states, adding them to the queue and marking them as visited. At the second level, BFS encounters a state that has already been visited (illustrated in the right-hand branch), leading it to prune that state to avoid redundancy.

4.1.3 Evaluation

- **Completeness**: BFS guarantees if a solution exists, it will always find the path to the goal state.
- **Optimality**: Since BFS explores the state level by level, it will always find the shortest path to the goal state.
- **Time complexity**: Suppose we have a solution that is found at depth **d**. Possible states that are generated by a state are 4, excluding invalid moves. Then BFS will find a solution in $\mathbf{O}(4^d)$
- **Space complexity**: Suppose we have a solution that is found at depth **d**. Possible states that are generated by a state are 4, excluding invalid moves. Then BFS will store $\sum_{i=0}^{d} 4^i$ states, resulting in $\mathbf{O}(4^d)$
- **Limitation**: In advance, when a rock is assigned a weight, then BFS will ignore it and hence, BFS may find a suboptimal solution (a state where the cost to complete is the smallest)

4.2 Depth-first Search (DFS)

4.2.1 Implementation Idea

From the initial position of Ares, he continues to explore all the children nodes and put them into the stack data structure (Last In First Out). Therefore, the next expanded node will always be the children of the previous node. This strategy follows depth-first search (DFS), allowing Ares to explore in a straight way to the leaf node before considering other nodes at the same level.

4.2.2 Detail

1. Similar to the strategy of BFS, we will monitor the positions of all the stones and Ares in each state. The locations of switches and walls are among the global data that is saved.

2. To begin, we'll use a stack to track Ares' progress from his starting position. We use a list named visited to keep all expanded states, preventing Ares from moving to revisited states and improving performance.

3. Next, Ares begins to explore all possible adjacent states that are accessible from Ares. After finishing, he will remember all the states in the stack data structure. He continues exploring by popping out of the stack the last element and moving the previously explored states first and so on. During exploration, Ares checked if the new state was the goal state. If yes, Ares will stop exploring and return to the movements he made, represented by a string.

4. Finally, when adding states to the queue, Ares will mark them as visited for later checking and ensure that the algorithm will be completed.

From the code, we use concurrency in order to speed up the process, by letting each processor take out a state and generating neighbors (adjacent states) then return back to the result. We also implement a shared visited list which can be shared through many processors.
However, as this concurrency is heavily CPU-bound, which means that any actions read/write file will make the process slow and result in downgrade performance. Also, the shared visited list is also an issue if many processors try to access the same time, which means there is a waiting time hence, resulting in downgraded performance.

Code for DFS's Algorithm:

```
def dfs(self, data, visited, shared_stop_event, pool, pool_size=1):
        """
        Depth-first search algorithm to find the path to the goal state
        """
        stack = deque([self])
        # Use the shared list
        visited.append((self.player_pos, tuple(self.boxes)))

        while not shared_stop_event.is_set() and stack:
            batch_size = min(len(stack), pool_size)
            current_states = [stack.pop() for _ in range(batch_size)]

            # Check if in the batch, there is a goal state or not
            for state in current_states:
                if state.is_goal_state(data.goal_state):
                    print("done")
                    stack.clear()
                    return state, data.node_count

            results = pool.starmap(self.generate_state, [(state, data) for state
    in current_states])

            for neighbors in results:
                for neighbor in neighbors:
                    if neighbor is None: continue
                    # Only add the neighbor to the visited list if it hasn't
   been added yet
                    visited_list = [neighbor.player_pos, neighbor.boxes]
                    if visited_list not in visited:
                        visited.append(visited_list)
                        stack.append(neighbor)
                        data.node_count += 1
        return None, data.node_count
```

Listing 2: Codes for DFS's Algorithm

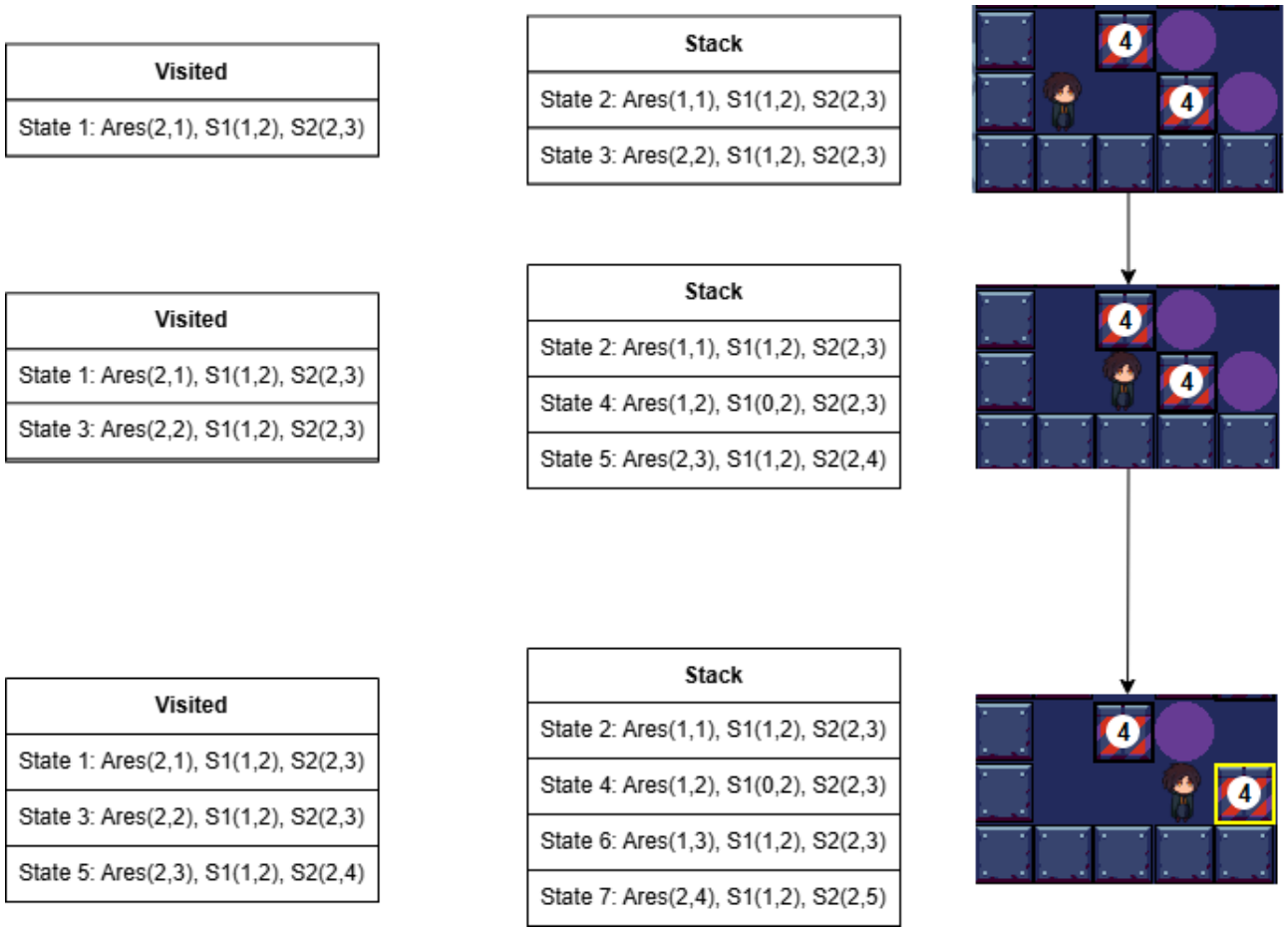

Figure 2: Illustrative DFS process with a map

4.2.3 Evaluation

- **Completeness**: Because the game states are in finite tree-shaped state spaces, visited nodes and deadlocks are checked so DFS is complete.
- **Optimality**: DFS is not guided by proximity to the target or by minimal cost. It only follows a path to its end, regardless of how close that path is to the destination, which can lead to longer or suboptimal paths. DFS is not designed to account for path costs, even if edges have weights. It explores as far as possible along each branch before backtracking, which means it could potentially miss a shorter path that lies along a different branch.
- **Time complexity**: Suppose we have a solution that is found at depth **d**. Possible states that are generated by a state are 4, excluding invalid moves. Then the time complexity of DFS is $\mathbf{O}(4^d)$

- **Space complexity**: Suppose we have a solution that is found at depth **d**. Possible states that are generated by a state are 4, excluding invalid moves. Then the space complexity of DFS is $\mathbf{O}(4d)$

4.3 Uniform Cost Search (UCS)

4.3.1 Implementation Idea

The Uniform Cost Search (UCS) algorithm finds the least-cost path from Ares' starting position to the goal state. UCS operates similarly to BFS but considers the cost of each move when selecting the next state to explore. Since the cost in this problem depends on the weight of each box, Ares will prioritize moves that minimize the cumulative cost to reach the goal.

UCS maintains a **priority queue** to store states, where each state is prioritized by its cumulative cost from the starting position. Ares explores the lowest-cost state first, ensuring that UCS will find the **least-cost path** to the goal state, where all stones are positioned on the switches.

4.3.2 Detail

To implement UCS, we track each state by storing Ares' position, all stone positions, and the cumulative cost from the start. A priority queue is used to store states with their costs, ensuring that lower-cost states are dequeued before higher-cost ones.

```
class UCS_GameState:
    """
    This class implements the game state using Uniform Cost Search (UCS).
    UCS considers the cost to reach each node, expanding the least-cost path
   first.
    """
    def __init__(self, player_pos, boxes, string_move="", g_cost=0, parent=None)
   :
        """
        Initialize the game state with player's position, boxes, g-cost, and
   parent reference.
        """
        self.player_pos = player_pos
```

```
        self.boxes = boxes
        self.string_move = string_move  # Move string (e.g., "RURU")
        self.g_cost = g_cost
        self.parent = parent
```

Listing 3: UCS GameState Class and UCS Search Function

1. **Initialize**: Begin at Ares' starting position with zero cost. Insert this initial state into the priority queue.

2. **State Exploration**: Pop the state with the lowest cost from the priority queue. For each adjacent position of Ares, calculate the move cost (including box weights if a box is pushed). Add this new state to the priority queue with the updated cumulative cost.

3. **Goal Check**: Each time a state is dequeued, check if it is the goal state. If it is, UCS terminates and returns the path represented as a string of moves along with the total cost.

4. **Marking Visited States**: To avoid cycles, each state is marked as visited when it is dequeued. If a state is revisited with a lower cost, update its cost and path in the priority queue, ensuring that UCS always explores the least-cost path.

```
class UCS_GameState:
    @staticmethod
    def ucs(initial_state, goal_state, data):
        priority_queue = []
        heapq.heappush(priority_queue, (initial_state.g_cost, initial_state))
        visited = {}

        while priority_queue:
            current_cost, current_state = heapq.heappop(priority_queue)

            if current_state.is_goal_state(goal_state):
                return current_state, data.node_count

            # Track the state by its position and boxes, and only expand if we find a lower cost
            state_key = (tuple(current_state.player_pos), tuple(map(tuple, current_state.boxes)))

```

```
            if state_key in visited and visited[state_key] <= current_cost:
                continue

            # Mark this state with the current cost
            visited[state_key] = current_cost

            # Expand neighbors
            neighbors = current_state.get_neighbors(data)
            data.node_count += len(neighbors)
            for neighbor in neighbors:
                neighbor_key = (tuple(neighbor.player_pos), tuple(map(tuple,
neighbor.boxes)))
                if neighbor_key not in visited or visited[neighbor_key] >
neighbor.g_cost:
                    heapq.heappush(priority_queue, (neighbor.g_cost, neighbor))

        return None, data.node_count
```

Listing 4: UCS GameState Class and UCS Search Function

4.3.3 Evaluation

- **Completeness**: UCS is complete, as it explores all possible paths if a solution exists, ensuring that it will find the goal state.
- **Optimality**: UCS is optimal; it always finds the least-cost path to the goal state because it explores states in order of cumulative cost.
- **Time complexity**: Suppose there are **n** states and each state generates up to 4 possible moves. The time complexity of UCS is $\mathbf{O}(n \log n)$ due to the priority queue operations for each state.
- **Space complexity**: UCS stores all states in the priority queue and visited list, resulting in a space complexity of $\mathbf{O}(n)$, where **n** is the number of possible states.

4.4 A* Search with Heuristic

4.4.1 Implementation Idea

The A* search algorithm finds the least-cost path from Ares' starting position to the goal state, similar to UCS, but it guides the search in a more informed way using a heuristic. The evaluation function $f(n)$ combines the actual cost from the start to a state and an estimated cost to the goal, allowing it to prioritize states that are both low-cost and likely closer to the goal. In this problem, we will use a heuristic that estimates the cost of pushing the stones toward the switches. This approach ensures that A* searches in the direction of the goal while minimizing the cumulative cost.

A* also uses a **priority queue**, where states are prioritized by their total estimated cost (actual cost + heuristic cost). This guides Ares to find the **least-cost path** to the goal state, where all stones are on switches while reducing unnecessary exploration.

4.4.2 `AStar_GameState` Class

The core class `AStar_GameState` represents a game state with methods for calculating movement costs, generating neighbors, and computing the heuristic.

```
class AStar_GameState:
    def __init__(self, player_pos, boxes, string_move="", g_cost=0, data=None):
        """
        Initialize the game state with player's position, boxes, g-cost and data
    of the map.
        """
        self.player_pos = player_pos
        self.boxes = boxes
        self.string_move = string_move  # Move string (e.g., "RURU")
        self.data = data  # Reference to Initialized_data instance

        # Cost to reach this state from the start state
        self.g_cost = g_cost

        # Heuristic cost
        self.h_cost = self.calculate_heuristic(self, self.data.goal_state)

```

```
        # Total cost (f_cost = g_cost + h_cost)
        self.f_cost = self.g_cost + self.h_cost
```

Listing 5: Class Definition for `AStar_GameState`

The constructor given in Listing 5 initializes each game state with the player's position, box positions, and cost parameters:

- `g_cost`: the cost from the start to this state.
- `h_cost`: estimated cost to the goal, calculated by `calculate_heuristic`.
- `f_cost`: total estimated cost ($f(n) = g(n) + h(n)$).

4.4.3 `AStar` Class Initialization

To implement A*, we track each state by storing Ares' position, all stone positions, the cumulative cost from the start, and an estimated cost to the goal. A priority queue is used to store states, prioritizing them by their total estimated cost $f(n)$.

```
class AStar:
    def __init__(self, start_player_pos, start_boxes, data, max_moves=None):
        self.start_player_pos = start_player_pos
        self.start_boxes = start_boxes
        self.data = data
        self.open_list = []  # Min-heap priority queue
        self.open_dict = {}  # Dictionary to track open states by their (player_pos, boxes) key
        self.closed_set = set()  # Explored states
        self.node_count = 0
        self.max_moves = max_moves

        start_node = AStar_GameState(start_player_pos, start_boxes, "", 0, None, data)
        heapq.heappush(self.open_list, (start_node.f_cost, start_node))
        self.open_dict[(tuple(start_node.player_pos), tuple(tuple(box) for box in start_node.boxes))] = start_node
```

Listing 6: Initializing A* with Priority Queue and Closed Set

The `AStar` class initializer shown in Listing 6 sets up the search algorithm with:

- A min-heap priority queue (`open_list`) to manage open nodes.
- A dictionary (`open_dict`) for tracking states and their costs.
- A `closed_set` to store explored states.
- The `node_count` counter to keep track of expanded nodes.
- The start state is inserted into the priority queue to initiate the search.

```
def search(self, shared_stop_event):
    while self.open_list and not shared_stop_event.is_set():
        f_cost, current_state = heapq.heappop(self.open_list)
        key = (tuple(current_state.player_pos), tuple(tuple(box) for box in
   current_state.boxes))

        # If the popped state is no longer in open_dict, skip it
        if key not in self.open_dict or self.open_dict[key].f_cost != f_cost:
            continue
        del self.open_dict[key]

        if current_state.is_goal_state(self.data.goal_state):
            print(f"Solution path: {current_state.string_move}")
            return current_state, self.node_count

        self.closed_set.add(key)
        neighbors = current_state.get_neighbors(self.data, self.closed_set)
        self.node_count += len(neighbors)

        for _, neighbor in neighbors:
            neighbor_key = (tuple(neighbor.player_pos), tuple(tuple(box) for box
    in neighbor.boxes))

            if neighbor_key in self.closed_set:
                continue

```

```
25            if neighbor_key not in self.open_dict or neighbor.f_cost < self.
    open_dict[neighbor_key].f_cost:
26                heapq.heappush(self.open_list, (neighbor.f_cost, neighbor))
27                self.open_dict[neighbor_key] = neighbor
28
29     print("No solution found.")
30     return None, self.node_count
```

Listing 7: Search Method of `AStar` for Goal Pathfinding

The 'search' method (Listing 7) explores nodes in order of increasing cost $f(n) = g(n) + h(n)$.

- It continues searching while there are unvisited nodes in `open_list` and no shared stop event is triggered.
- Each iteration pops the state with the lowest `f_cost` from `open_list` as the current state.
- If the `current_state` matches the goal state (verified via `is_goal_state`), the method prints the solution path and returns the current state and the `node_count`.
- Otherwise, the `current_state` is added to `closed_set` to mark it as explored.
- For each neighboring state, the method calculates the `f_cost` and checks if it should be added to the open list.
  - If a neighbor has a better `f_cost` than any existing record in `open_dict`, it replaces the old record and is added to the open list.

If no solution is found, the method outputs a message and returns `None` along with the `node_count`.

4.4.4 `calculate_heuristic` Method

The `calculate_heuristic` method combines several heuristic elements to provide an admissible and consistent estimate for the cost:

1. **Box-Goal Weighted Distance**: For each box, we calculate the Manhattan distance to each goal position and apply a specific weight according to the box's individual weight (if provided). This accounts for the increased difficulty of moving heavier boxes. Manhattan distance is preferred in grid-based environments like this Ares problem, where movement is

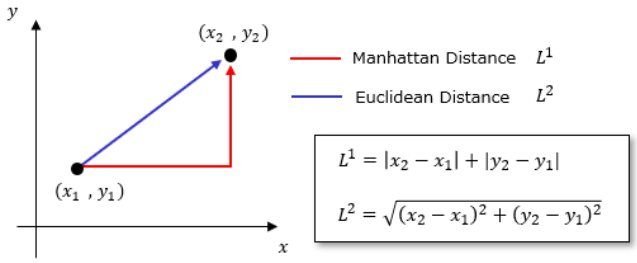

Figure 3: Difference between Manhattan distance and Euclidean distance (source: [1])

restricted to vertical and horizontal steps for its accurate reflection of the movement cost in a grid compared to Euclidean distance (Figure 3). The result is a distance matrix, where each cell represents the weighted Manhattan distance from a box to a goal.

2. **Optimal Assignment using the Hungarian Algorithm**: Given the matrix of weighted distances, we use the Hungarian algorithm to match each box to its optimal goal. This algorithm ensures each box is assigned to the goal that minimizes the total weighted distance, leading to a more precise and efficient heuristic for each box-goal configuration. The method `linear_sum_assignment` from `scipy.optimize` returns the optimal indices for matching boxes to goals.

3. **Ares's Proximity to Unsolved Boxes**: To account for the movement cost of the Ares, we add the minimum Manhattan distance between the Ares's current position and the nearest unsolved box. Including this cost enables the heuristic to consider the player's proximity to boxes that need to be moved, thus improving the search process's efficiency.

Combining these elements yields a heuristic estimate calculated as follows:

$$\text{Heuristic} = \text{Total Weighted Box-Goal Distance} + \text{Minimum Ares-Box Distance}$$

```
class AStar_GameState:
    def calculate_heuristic(self, state, goal_state):
        """
        Calculate an admissible and consistent heuristic for the puzzle.
        This heuristic considers:
        - The weighted distances between each box and each goal.
        - The minimum distance between the player and each unsolved box.
```

```
        """
        distance_matrix = []

        # Compute weighted Manhattan distances between each box and each goal
        for i, box in enumerate(state.boxes):
            row = []
            for goal in goal_state:
                # Calculate Manhattan distance and apply weight
                manhattan_distance = abs(box[0] - goal[0]) + abs(box[1] - goal
[1])
                weight = self.data.stone_weights[i] if i < len(self.data.
stone_weights) else 1  # Default weight is 1
                weighted_distance = manhattan_distance * weight
                row.append(weighted_distance)
            distance_matrix.append(row)

        # Use the Hungarian algorithm for optimal box-goal assignments
        box_indices, goal_indices = linear_sum_assignment(distance_matrix)
        total_weighted_distance = sum(distance_matrix[box][goal] for box, goal
in zip(box_indices, goal_indices))

        # Calculate minimum distance between player and each unsolved box
        unsolved_boxes = [box for box in state.boxes if box not in goal_state]
        player_box_distances = [
            abs(state.player_pos[0] - box[0]) + abs(state.player_pos[1] - box
[1])
            for box in unsolved_boxes
        ]
        min_player_box_distance = min(player_box_distances) if
player_box_distances else 0

        return total_weighted_distance + min_player_box_distance
```

Listing 8: Calculating Heuristic Using Weighted Manhattan Distances

4.4.5 Reflection on the Heuristic Design

While this design ensures an admissible and consistent heuristic, it is worth reflecting on whether the added complexity is always justified.

- Requires calculating weighted Manhattan distances between each box and goal.
- Builds a distance matrix and applies the Hungarian algorithm for optimal assignments.
- Computes the minimum player-box distance for unsolved boxes.

This precision may be excessive for simpler levels, which could be solved by balancing Precision and Efficiency:

- **Basic Manhattan Distance**: Suitable for levels with minimal box weight variation.
- **Adaptive Heuristic Application**: Apply detailed heuristic selectively for complex configurations.
- **Heuristic Caching**: Store heuristic values for common states to avoid redundant calculations.

4.4.6 Evaluation

- **Completeness**: A* is complete and guarantees a solution if one exists, as it systematically explores states based on both cost and heuristic.
- **Optimality**: A* is optimal when the heuristic used is admissible. This ensures that A* finds the least-cost path to the goal state without overestimating costs.
- **Time Complexity**: Suppose there are **n** possible states, each generating up to 4 moves. With the priority queue operations, the time complexity is generally $\mathbf{O}(n \log n)$.
- **Space Complexity**: A* maintains a priority queue and visited list, resulting in a space complexity of $\mathbf{O}(n)$ where **n** is the number of possible states.

5 Testing

5.1 Description of the test cases and experiment results

5.2 Test cases

5.2.1 Test case 1

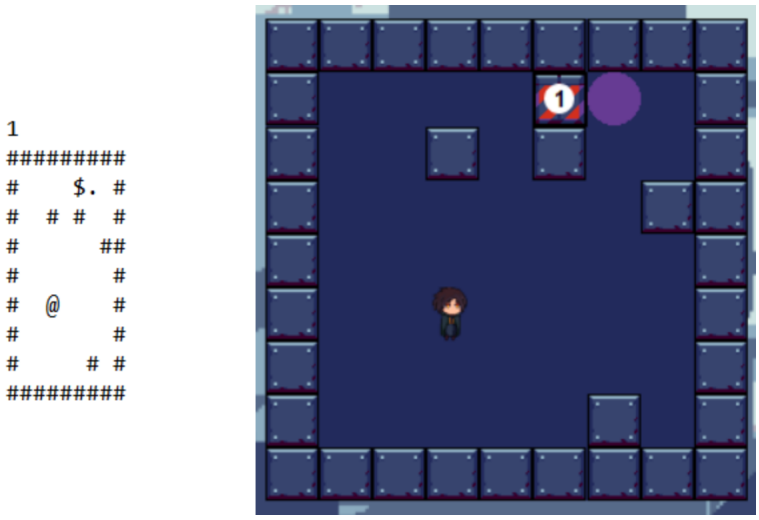

Figure 4: Test case 1

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	6	7	115	869.5023059844971	71.61
DFS	6	7	42	964.4215106964111	71.68
UCS	6	7	136	375.11587142944336	71.64
A*	6	7	25	389.7511959075928	70.98

Table 3: Test cases 1 - Experiment results

5.2.2 Test case 2

This test case shows how UCS and A* manage weighted boxes, pushing the heavier box toward the closer goal to minimize total cost for an efficient solution.

```
1 99
 ############
##          #
#           #
# $ $       #
#. @       .#
#############
```

Figure 5: Test case 2

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	16	711	25481	10270.39647102356	71.45
DFS	125	2026	7758	12002.42018699646	72.27
UCS	23	428	118552	863.7204170227051	85.04
A*	23	428	10741	564.011812210083	85.55

Table 4: Test cases 2 - Experiment results

5.2.3 Test case 3

```
1 2 3
  ######
  #    #
### $  #
#@ $.. #
# $ . ##
####  #
   ####
```

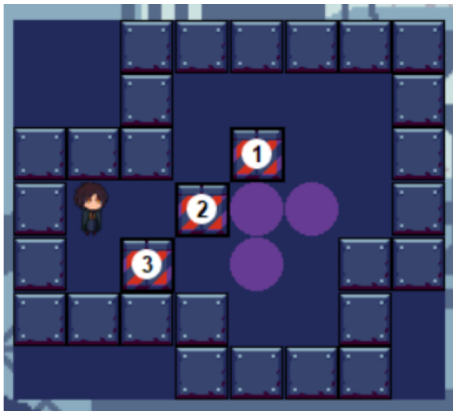

Figure 6: Test case 3

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	11	21	1405	1631.885290145874	77.64
DFS	231	351	25507	144821.46620750427	84.62
UCS	11	21	7882	402.9698371887207	81.14
A*	11	21	226	374.1445541381836	81.15

Table 5: Test cases 3 - Experiment results

5.2.4 Test case 4

In this test case, the configuration makes it impossible for Ares to reach the goal, resulting in no solution.

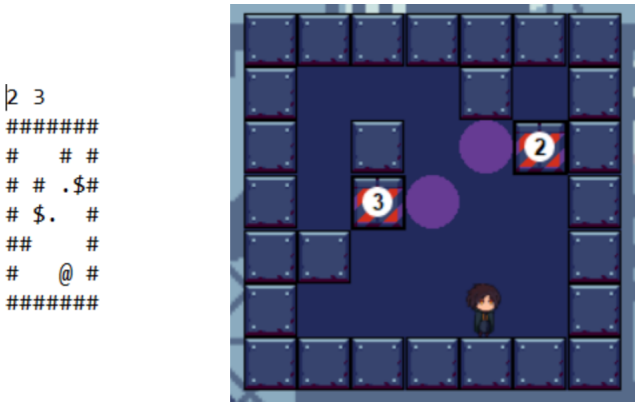

Figure 7: Test case 4

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	0	0	246	1450.8264064788818	77.13
DFS	0	0	246	1431.0891628265387	77.17
UCS	0	0	1840	379.7571659088135	77.30
A*	0	0	325	381.67619705200195	76.91

Table 6: Test cases 4 - Experiment results

5.2.5 Test case 5

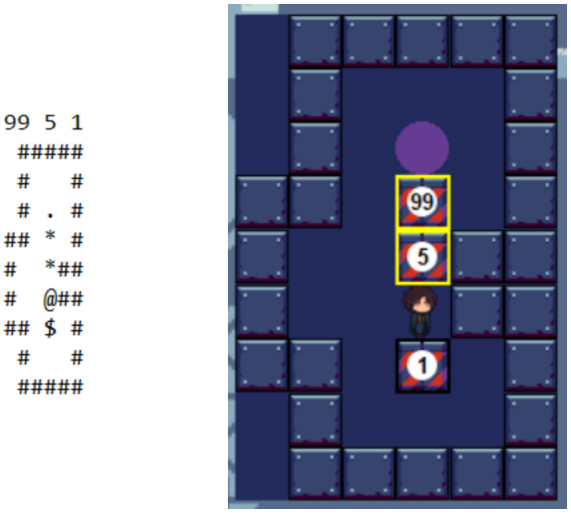

Figure 8: Test case 5

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	46	466	54596	53531.582832336426	81.27
DFS	172	2116	12650	12893.92066001892	78.12
UCS	43	252	106189	858.710527420044	93.00
A*	43	252	5271	485.7180118560791	92.50

Table 7: Test cases 5 - Experiment results

5.2.6 Test case 6

In this test case, BFS outperforms both A* and UCS in terms of time. The key reason is that BFS ignores the weight of the stones, whereas both A* and UCS take into account the costs.

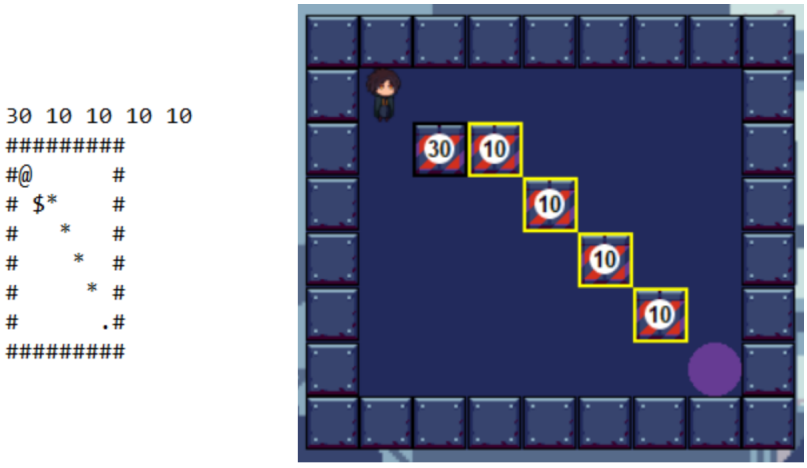

Figure 9: Test case 6

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	12	282	31106	32984.07697677612	329.49
DFS	0	0	29938	Timeout	330.95
UCS	0	0	9491597	Timeout	330.30
A*	18	128	1835363	59588.17148208618	731.09

Table 8: Test cases 6 - Experiment results

5.2.7 Test case 7

```
2 3 4
########
#  $.  #
#   #  #
# $. $ #
# # .  #
#@##  ##
########
```

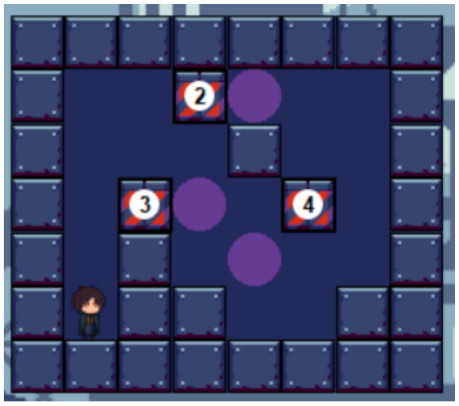

Figure 10: Test case 7

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	21	37	7828	2501.8279552459717	91.00
DFS	23	42	474	1326.2391090393066	91.00
UCS	24	37	22781	493.55530738830566	87.41
A*	24	37	1048	410.94183921813965	86.91

Table 9: Test cases 7 - Experiment results

5.2.8 Test case 8

```
8 1
#########
# #   $.#
## #    #
#  #  . #
#   $ # #
# #     #
#  # #  #
#    # @#
#########
```

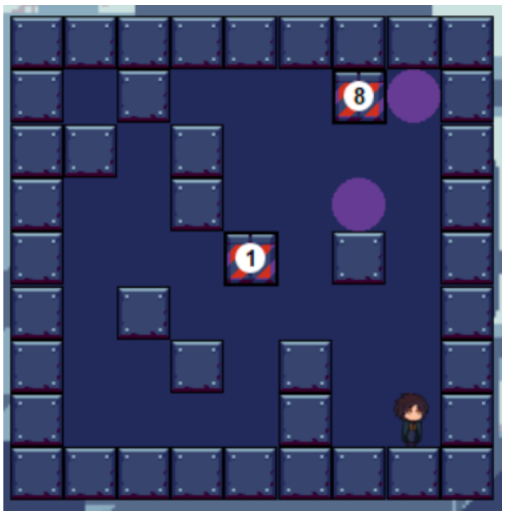

Figure 11: Test case 8

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	17	28	1432	7097.216844558716	86.77
DFS	123	238	4040	4649.962425231934	86.77
UCS	17	28	4393	433.3758354187012	86.42
A*	17	28	440	397.9365825653076	86.43

Table 10: Test cases 8 - Experiment results

5.2.9 Test case 9

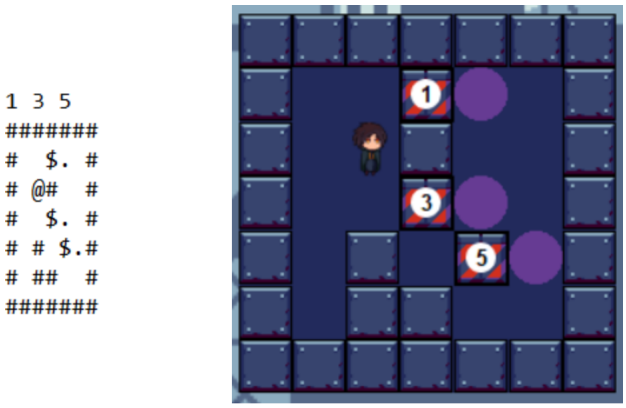

Figure 12: Test case 9

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	8	17	130	847.597599029541	71.27
DFS	10	19	646	1358.954906463623	71.50
UCS	8	17	377	384.9062919616699	71.46
A*	8	17	41	383.9735984802246	71.50

Table 11: Test cases 9 - Experiment results

5.2.10 Test case 10

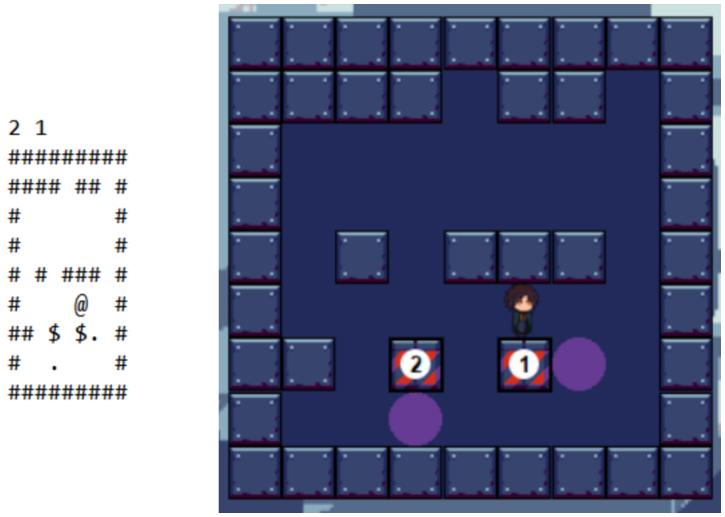

Figure 13: Test case 10

Algorithm	Steps	Weight	Nodes	Time (ms)	Memory (MB)
BFS	12	21	5813	1818.0623054504395	86.28
DFS	101	141	21483	90207.24844932556	86.27
UCS	5	8	437	390.3498649597168	86.19
A*	5	8	50	385.6086730957031	86.22

Table 12: Test cases 10 - Experiment results

5.3 Highlight challenges and compare overall behavior of algorithms

5.3.1 Challenges

- BFS suffers in terms of space usage, particularly in high branching factor problems, because it must maintain a large queue. For deep search spaces, it may also be prohibitively slow.
- DFS may get trapped in deep, irrelevant paths, especially if the search space is vast or infinite. Additionally, the lack of optimality and potential for non-termination are significant drawbacks in many applications.
- UCS faces challenges with high memory usage, particularly if there are numerous nodes with similar costs. It also struggles with large search spaces where keeping track of explored paths is essential to avoid redundant work.

- The primary challenge with A* is balancing heuristic quality and computation time. A poorly chosen heuristic can result in excessive memory usage and high computation time, particularly in large or complex search spaces. For the heuristic approach implemented in our project, its complexity may occasionally be "overkill" for simpler cases.

5.3.2 Comparing overall behavior

- **Completeness:** All algorithms in this game after checking deadlocks, visited nodes are completed for depth-first search(DFS) and admissible heuristic function for A* algorithm.
- **Optimality:** BFS, UCS and A* all return optimal solution. Even though the paths returned by BFS, UCS, and A* are different from each other, their minimum weight is the same as well as their shortest path, meaning that they all found an optimal solution. However, DFS is very inefficient since the path returned by DFS has a much higher weight than the minimum weight achieved by the other algorithms. This is because of the nature of DFS to penetrate deep in one direction without any kind of regard to the path cost itself and often settle for much longer and less efficient paths.
- **Time and Space complexity:** While DFS takes the most time and memory as it explores a huge number of nodes, UCS and A* take less time and space. In particular, A* is the most efficient concerning nodes. It also explores the same but takes a bit more time because of the heuristic calculations.

6 Project Demo

Google Drive Link to Demo Video:

https://drive.google.com/file/d/1pjQUfOUBStyQKjuapHeExQHlWUuY6O2m/view?usp=sharing

References

[1] TakeTake. (n.d.). *The Difference between Manhattan Distance and Euclidean Distance*. Retrieved from https://taketake2.com/N102_en.html