UNIVERSITY OF SCIENCE
VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

PROJECT SEARCH

Ares's adventure

Introduction to Artificial Intelligence

Team member

22125023 – Lê Công Quốc Hân

22125094 – Đào Bá Thành

22125118 – Phạm Ngọc Phương Uyên

22125124 – Huỳnh Phan Nhật Vy

Lecturers: Nguyễn Ngọc Thảo

Nguyễn Thanh Tình

Contents

1 Introduction

Artificial Intelligence (AI) is rapidly emerging as a transformative field with vast potential across various sectors, paving the way for new advancements and innovations. The course Introduction to Artificial Intelligence has equipped us with a structured foundation in AI, encompassing both theoretical principles and practical applications. Through exploring essential methods and models, we have gained insights into addressing complex challenges in AI.

Our course project has provided a valuable opportunity to apply our knowledge in a real-world context, marking a significant first step in developing our research skills and advancing our technical expertise. We are deeply grateful to our lecturers and supervisors for their unwavering guidance and support, which has been instrumental in helping us understand and implement key algorithms and theoretical concepts to solve project-specific problems.

While we have tried to deliver a thorough and well-executed project, we recognize that there may be areas for improvement due to our limited experience and knowledge. We look forward to receiving constructive feedback from our professors to further refine our work.

We extend our heartfelt thanks to our professors for their ongoing support and encouragement throughout this journey.

Video URL: https://www.youtube.com/watch?v=dSbHKYCXDgs
Github: https://github.com/nhatvy118/Game

2 Activity Progress

2.1 Task Assignment

Student ID	Name	Assigned Task	Completion Rate
22125023	Le Cong Quoc Han	Implement core architecture, developing algorithms	100%
22125094	Dao Ba Thanh	Optimize algorithms, generate testcases, report	100%
22125118	Pham Ngoc Phuong Uyen	GUI design, developing algorithms, report	100%
22125124	Huynh Phan Nhat Vy	GUI design, optimize algorithms, report	100%

2.2 Timeline

Time	Task
17/10/2024 - 24/10/2024	Learn about algorithms, library to make GUI
25/10/2024 - 6/11/2024	Implementation
7/11/2024 - 11/11/2024	Testing game, optimize algorithm, analysis results

2.3 Self-evaluation

No	Requirement	Complete
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 (12 tests)	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 About Ares's adventure Game

3.1 Interface

The interface is primarily designed using the Pygame library in Python, combined with assets and backgrounds sourced from the internet. The interface is divided into five screens to handle different tasks: the welcome screen, the screen for choosing the algorithm, the screen for choosing the level, the waiting screen while the algorithm is running, and the game screen.

- **Welcome Screen**: This is the first screen displayed when running the program. It serves as the introduction to the game and provides a start button to start the game.
- **Algorithm Selection Screen**: On this screen, the user chooses which algorithm to use (A*, UCS, DFS, BFS). After the user selects the algorithm, the program will proceed to the next screen.
- **Level Selection Screen**: After selecting the algorithm, the user moves to the level selection screen, where different difficulty levels or stages are presented. The user selects a level, and then the program proceeds to the waiting screen.
- **Loading Screen**: This screen is used while waiting for the solution to be computed. To prevent the program from becoming unresponsive during slow algorithm execution, we utilize multiprocessing in Pygame.
- **Game Screen**: This screen illustrates the character animating step-by-step based on the solution path traced by the algorithm. You can revisualize by clicking on the back button.

3.2 Flow of our game

When the user selects the algorithm type and level, the backend executes the algorithm and returns the result to the frontend. The frontend then simulates based on the path provided in the backend's response.

3.3 Screen Images

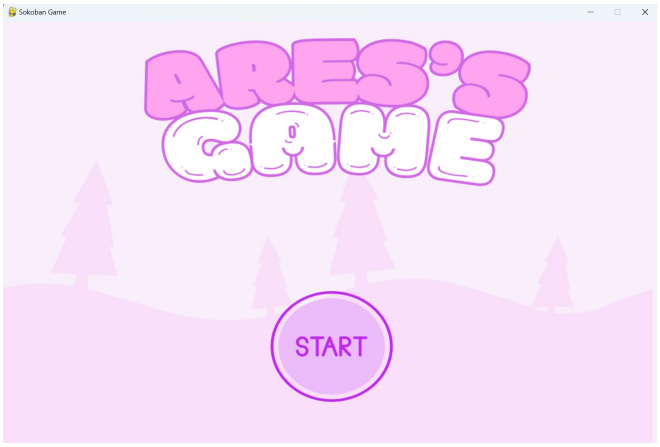


Figure 1: Welcome Screen

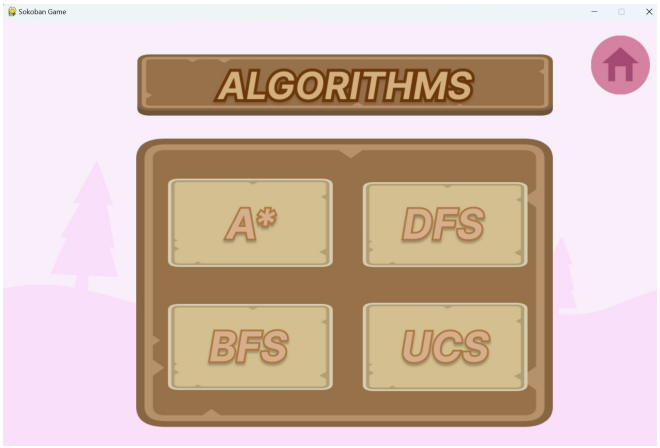


Figure 2: Screen for Algorithm Chosen

Figure 3: Screen for Level Chosen

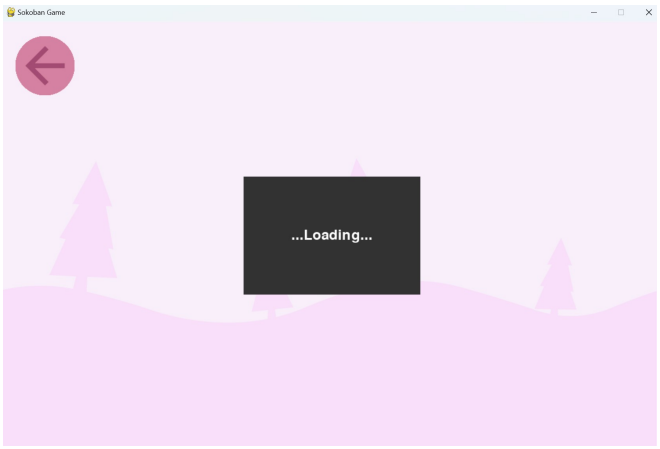


Figure 4: Loading Screen

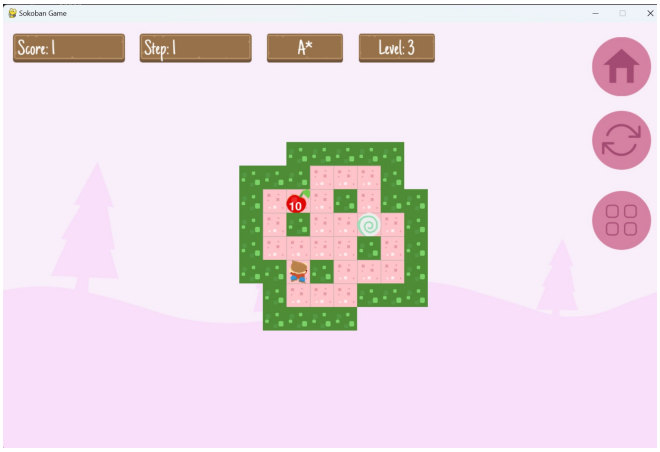


Figure 5: Game Screen

4 Algorithms

4.1 Helper Classes

This section describes the supporting classes in the game logic and operations. These classes handle various aspects of the game state, such as coordinates, cells on the board, or node states, which are used to facilitate the implementation of the algorithms.

4.1.1 Class `Cell`

The `Cell` class represents each individual cell on the board. It stores the type of the cell (e.g., empty, wall, goal) and its weight. These attributes are used to define the state of the board at any given point in time.

```
class Cell:
    def __init__(self, type, weight):
    def __str__(self):
    def __repr__(self):
```

4.1.2 Class `Stone`

The `Stone` class represents a stone object that can be moved around the game board, and display its value. It manages the stone's coordination, its movement logic, and updates its appearance based on its interaction with goals.

```
class Stone:
def __init__(self, x, y, value, on_switch)
def move(self, dx, dy, walls, stones, switches)
def draw(self, screen)
```

4.1.3 Class `Player`

The `Player` class represents the player in the game and handles movement, animation, and interaction with stones and walls.

```
def __init__(self, x, y)
def move(self, dx, dy, stones, walls, switches)
def update_animation(self, frames)
def draw(self, screen)
```

4.1.4 Class `Node`

The `Node` class represents a state of the game, including the board configuration, player position, goals, path taken, and associated cost. It provides methods for checking whether the game has reached the goal state, detecting deadlocks, and validating moves. The class also manages the creation of unique IDs for different game states based on the board's configuration.

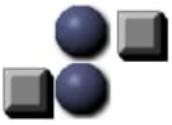
```
class Node:
    def __init__(self, board, player: Pair, goals:list, path:str, cost:int):
    def __lt__(self, other):
    def isGoalState(self):
    def isDeadlocked(self):
    def isPosValid(self, pos: Pair):
    def canMove(self, direction):
    def move(self, direction)
```

Function `isDeadlocked`

The `isDeadlocked` function plays a crucial role in optimizing the state space search by identifying and eliminating deadlock states. When the search space contains many states, it becomes critical to detect these deadlocks early to avoid redundant calculations. By identifying known deadlock patterns, we can prune the search space and reduce the number of generated states.

This function checks if the current configuration of the system matches a pattern where further progress is impossible, typically due to the positions of the stones and blocking obstacles . These patterns can be defined based on the layout of the stones and the surrounding environment.

Below are some examples of deadlock patterns identified and utilized by the `isDeadlocked` function:



These patterns help the algorithm quickly identify configurations where no further valid moves are possible, allowing it to avoid generating additional states that would lead to deadlocks. By incorporating these patterns into the search process, we can significantly reduce the search space and improve the efficiency of the algorithm.

4.2 Breadth-first Search

The Breadth-first Search (BFS) algorithm systematically explores all possible states of the problem by examining nodes level by level. This is one of the four algorithms applied in this game.

Algorithm Overview BFS explores the game state tree level by level. It uses a queue to manage the states being explored and includes an early goal test and find the result without consider the cost

```
1 def bfs(originalBoard, originalPlayer, originalGoals):
```

The `bfs` function accepts:

- `originalBoard`: A 2D list representing the Sokoban game board.
- `originalPlayer`: The initial position of the player.
- `originalGoals`: A list of goal positions on the board.

It returns a tuple containing:

- `path`: The sequence of moves from the start state to the goal.
- `node.cost`: The number of moves required to reach the goal.
- `elapsedTime`: The total time taken to complete the search.
- `memUsed`: Memory used during the search in MB.
- `cntNode`: Number of nodes (states) visited during the search.

Algorithm Steps:

1. **Initialization:** Deep copies of the board, player position, and goal positions are created to preserve original inputs. A `visited` set is initialized, and a `Node` object for the starting state is created. If the start node is not a goal state, it is added to the queue, and its unique ID is added to the `visited` set.

```
    board = copy.deepcopy(originalBoard)
    goals = copy.deepcopy(originalGoals)
    player = originalPlayer
    visited = set()
    startNode = Node(board, player, goals, "", 0)
    q = queue.Queue()
    startTime = time.time()
    if (startNode.isGoalState() == False):
        q.put(startNode)
        visited.add(startNode.ID)

```

2. **Starting the Search:** Nodes are processed from the queue. If a node represents a deadlocked state, it is skipped. If a node is a goal state, the search terminates, and the result is returned. A time limit of 3 minutes (180 seconds) is enforced to terminate excessive searches.

```
    while not q.empty():
        node = q.get()

        if node.isDeadlocked(): continue

        currentTime = time.time()
        if (currentTime - startTime > 180): return "-1", -1, -1, -1, -1

```

3. **Exploring Neighboring States:** The algorithm iterates over all valid directions (up, down, left, right). For each direction, it checks if the player can move in that direction. If a new node results and is not already visited, it is added to the queue and the `visited` set. If the new node is a goal state, the search ends.

```
    for dir in directions:
        if node.canMove(dir):
            newNode = node.move(dir)
            if newNode.ID not in visited:
                if newNode.isGoalState():
                    node = newNode
                    ok = True
                    break
                q.put(newNode)
                cntNode += 1
                visited.add(newNode.ID)

```

4. **Return the Result:** When the goal is reached, or the search completes, the function returns the path, cost, time elapsed, memory usage, and nodes visited.

```
    path = node.path
    endTime = time.time()
    elapsedTime = endTime - startTime
    memUsed = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2
    return path, node.cost, elapsedTime, memUsed, cntNode

```

Key Operations:

- **Node Movement**: `node.move(dir)` generates a new state based on the specified direction.
- **Goal State Detection**: `node.isGoalState()` checks if the current state meets the goal condition.
- **Deadlock Detection**: `node.isDeadlocked()` identifies states where no further progress is possible.
- **State Representation**: Each node is uniquely identified by its state through `node.ID`.
- **Queue Management**: `queue.Queue()` manages level-by-level exploration, ensuring the shortest path to the goal is found first.

Termination Conditions: BFS terminates when:

- The goal state is reached (`node.isGoalState()` returns `True`).
- The 3-minute time limit is exceeded.

Time Complexity: $O(b^d)$, where b is the branching factor, and d is the solution depth.
Space Complexity: $O(b^d)$.

4.3 Depth-First Search (DFS)

Depth-First Search (DFS) is an uninformed search algorithm that explores game states by going as deep as possible along a path before backtracking, with an early goal test and not considering cost.

Algorithm Overview The DFS algorithm starts with three inputs: the initial game board (`originalBoard`), the player's starting position (`originalPlayer`), and the goal states (`originalGoals`). These inputs are deep-copied to preserve the original state for each DFS run.

The algorithm initializes a `Node` object to represent the starting game state, which includes the board, player's position, and goals. A `visited` set is used to avoid redundant state revisits, while a stack `stack` is used to manage the depth-first exploration.

The start time is recorded for time management purposes, and the initial node is added to the stack if it is not already a goal state.

```
board = copy.deepcopy(originalBoard)
goals = copy.deepcopy(originalGoals)
player = originalPlayer
visited = set()
startNode = Node(board, player, goals, "", 0)
stack = deque()
startTime = time.time()

if not startNode.isGoalState():
    stack.append(startNode)
    visited.add(startNode.ID)
```

The DFS algorithm processes nodes from the stack in a depth-first manner, aiming to reach a goal state by exploring each path as deeply as possible before backtracking.

```
def dfs(originalBoard, originalPlayer, originalGoals):
```

The `dfs` function takes three inputs:

- `originalBoard`: A 2D list representing the Sokoban game board.
- `originalPlayer`: The initial position of the player.
- `originalGoals`: A list of goal positions on the board.

It returns a tuple containing:

- `path`: The sequence of moves from the start state to the goal.
- `node.cost`: The number of moves required to reach the goal.
- `elapsedTime`: The total time taken for the search.
- `memUsed`: The memory used during the search in MB.
- `cntNode`: The number of nodes (states) visited during the search.

Algorithm Steps:

1. **Initialization:** Deep copies of the board, player position, and goal positions are created. A `visited` set is initialized to track explored states, and a `Node` object for the start state is created. The initial node is added to the stack if it is not a goal state.

```
    board = copy.deepcopy(originalBoard)
    goals = copy.deepcopy(originalGoals)
    player = originalPlayer
    visited = set()
    startNode = Node(board, player, goals, "", 0)
    stack = deque()
    if not startNode.isGoalState():
        stack.append(startNode)
        visited.add(startNode.ID)

```

2. **Starting the Search:** Nodes are processed from the stack. If a node is deadlocked, it is skipped. If a node is a goal state, the search terminates and the results are returned. A time limit of 180 seconds is enforced to terminate excessive searches.

```
    while stack:
        currentTime = time.time()
        if (currentTime - startTime > 180): return "-1", -1, -1, -1, -1
        node = stack.pop()
        if node.isDeadlocked(): continue

```

3. **Exploring Neighboring States:** For each direction (up, down, left, right), the algorithm checks if the player can move in that direction. If a new node results and is not in `visited`, it is added to the stack and `visited` set. If the new node is a goal state, the search stops.

```
    for dir in directions:
        if node.canMove(dir):
            newNode = node.move(dir)
            if newNode.ID not in visited:
                if newNode.isGoalState():
                    node = newNode
                    ok = True
                    break
                stack.append(newNode)
                visited.add(newNode.ID)

```

4. **Return the Result:** When a goal state is found or the search completes, the function returns the path, cost, time elapsed, memory usage, and nodes visited.

```
    path = node.path
    endTime = time.time()
    elapsedTime = endTime - startTime
    memUsed = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2
    return path, node.cost, elapsedTime, memUsed, cntNode

```

Key Operations:

- **Node Movement**: `node.move(dir)` generates a new state based on the specified direction.
- **Goal State Detection**: `node.isGoalState()` checks if the current state meets the goal condition.
- **Deadlock Detection**: `node.isDeadlocked()` identifies states where no further progress is possible.
- **State Representation**: Each node is uniquely identified by its state through `node.ID`.
- **Stack Management**: `deque()` is used to manage nodes in a depth-first order.

Termination Conditions: DFS terminates when:

- The goal state is reached (`node.isGoalState()` returns `True`).
- The 180-second time limit is exceeded.

Time Complexity: $O(b^d)$, where b is the branching factor and d is the solution depth.
Space Complexity: $O(b \cdot d)$.

4.4 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) is an informed search algorithm that explores the Sokoban game state by expanding the least costly node first, ensuring the shortest path in terms of cost. UCS is similar to Dijkstra's Algorithm and is suitable for search problems where different actions have varying costs.

Algorithm Overview: UCS begins with the initial state and explores neighboring states by always expanding the node with the least accumulated cost. A priority queue (min-heap) manages nodes based on their cost, ensuring the most cost-effective path to a goal state.

```
1 def uniformCostSearch(originalBoard, originalPlayer, originalGoals):
```

The `uniformCostSearch` function accepts three inputs:

- `originalBoard`: The 2D Sokoban game board.
- `originalPlayer`: The starting position of the player.
- `originalGoals`: A list of positions representing the goal locations.

It returns:

- `path`: A string of moves leading to the goal state.
- `node.cost`: The total cost of reaching the goal.
- `elapsedTime`: Time taken for the UCS search.
- `memUsed`: Memory used during the search, in MB.
- `cntNode`: The number of nodes visited in the search process.

Algorithm Steps:

1. **Initialization:** Deep copies of the board, player position, and goals are created. A dictionary `visited` tracks each state and its lowest cost. A `Node` object representing the initial state is created and added to the priority queue (`pq`), with the priority being the cost.

```
    board = copy.deepcopy(originalBoard)
    goals = copy.deepcopy(originalGoals)
    player = originalPlayer
    startTime = time.time()
    visited = {}
    startNode = Node(board, player, goals, "", 0)
    pq = []
    heapq.heappush(pq, startNode)
    cntNode = 1

```

2. **Priority Queue Management:** The priority queue ensures that the node with the least accumulated cost is always processed next. The `heapq.heappop(pq)` function retrieves this node.

```
    heapq.heapify(pq)
    node = heapq.heappop(pq)

```

3. **Goal and Deadlock Detection:** If the current node is a goal state (`node.isGoalState()`), the search ends and results are returned. If the node is deadlocked or does not improve the cost for an already visited state, it is skipped.

```
    if (node.isGoalState()): break
    if (node.isDeadlocked()): continue
    if (node.ID in visited and visited[node.ID] != node.cost): continue

```

4. **Exploring Neighboring States:** For each valid move direction, the algorithm checks if the move can be executed. If so, a new state `newNode` is generated. If this state has not been visited or has a lower cost than previously recorded, it is added to the priority queue and marked as visited.

```
    for dir in directions:
        if (node.canMove(dir)):
            newNode = node.move(dir)
            if (newNode.ID not in visited or
                newNode.cost < visited[newNode.ID]):
                heapq.heappush(pq, newNode)
                visited[newNode.ID] = newNode.cost
                cntNode += 1

```

5. **Time and Memory Management:** The elapsed time is monitored to ensure the search does not exceed 3 minutes (180 seconds). Memory usage is tracked using the `psutil` library.

```
    currentTime = time.time()
    if (currentTime - startTime > 180): return "-1", -1, -1, -1, -1

```

6. **Return the Result:** Upon reaching a goal state, or when the search is complete, the function returns the path, total cost, time taken, memory usage, and nodes visited.

```
    path = node.path
    endTime = time.time()
    elapsedTime = endTime - startTime
    memUsed = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2
    return path, node.cost, elapsedTime, memUsed, cntNode

```

Key Operations:

- **Node Movement:** `node.move(dir)` creates a new state based on the specified direction.
- **Goal State Check:** `node.isGoalState()` determines if the current node meets the goal condition.
- **Deadlock Check:** `node.isDeadlocked()` identifies nodes where no further moves are possible.
- **State Representation:** Each state is uniquely identified by `node.ID`.
- **Priority Queue Management:** `heapq` ensures nodes with the lowest cost are processed first.

Termination Conditions: UCS terminates when:

- A goal state is found (`node.isGoalState()` returns `True`).
- The 3-minute (180 seconds) time limit is exceeded.

Time Complexity: $O(b^d \log b)$, where b is the branching factor and d is the solution depth.
Space Complexity: $O(b^d)$.

4.5 A* Algorithm

The A* algorithm is an informed search algorithm that is widely used in pathfinding and graph traversal problems. It combines the best features of both uniform cost search and greedy best-first search. A* finds the optimal path to the goal by considering both the cost to reach a node (`g(n)`) and the estimated cost to reach the goal from that node (`h(n)`), where the total cost function is defined as:

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

where: $g(n)$ is the cost to reach the node n from the start node, and $h(n)$ is the heuristic estimate of the cost from n to the goal.

The heuristic function used in the implementation is the **Manhattan Distance**. The Manhattan Distance calculates the sum of the absolute differences of the horizontal and vertical distances between the current position and the target. Also, instead of using a greedy approach to directly

find the minimum distance between stones and their respective goals, we implemented the **Hungarian algorithm** to solve the optimal stone-to-goal matching problem. The Hungarian algorithm efficiently finds the best possible pairings of stones and goals by solving the assignment problem. This algorithm ensures that the stones are matched to their corresponding goals in the most effective manner, taking into account all available positions. This method improves the overall solution by avoiding suboptimal choices that might arise from a purely greedy strategy.

```
def AStarAlgorithm(originalBoard, originalPlayer, originalGoals):
```

This function takes the following parameters:

- `originalBoard`: The 2D grid representing the Sokoban game board.
- `originalPlayer`: The initial position of the player on the board.
- `originalGoals`: The goal positions where boxes need to be placed.

It returns:

- `path`: The sequence of moves taken to reach the goal.
- `node.cost`: The total cost incurred in reaching the goal.
- `elapsedTime`: The time taken to find the solution.
- `memUsed`: The memory used by the algorithm during the search.
- `cntNode`: The number of nodes visited during the search.

Algorithm Steps:

1. **Initialization:** The algorithm starts by deep copying the original board, player position, and goals. A dictionary `visited` is initialized to store visited states and their costs to avoid redundant exploration. The start state (node) is created and added to the priority queue.

```
    board = copy.deepcopy(originalBoard)
    goals = copy.deepcopy(originalGoals)
    player = originalPlayer
    visited = {}
    startNode = Node(board, player, goals, "", 0)
    pq = []
    heapq.heappush(pq, startNode)
    cntNode = 1

```

2. **Priority Queue Management:** A priority queue is used to store nodes that are waiting to be expanded. The node with the lowest `f(n)` value is dequeued and expanded.

```
    heapq.heapify(pq)
    node = heapq.heappop(pq)

```

3. **Goal State Detection:** The algorithm checks whether the current node is a goal state using the `node.isGoalState()` method. If a goal state is found, the search stops, and the algorithm returns the path to the goal.

```
    if (node.isGoalState()): break

```

4. **Deadlock Check:** If a node is deadlocked, it is skipped, and the algorithm continues expanding other nodes.

```
    if (node.isDeadlocked()): continue

```

5. **Visited State Check and Update:** For each visited state, the algorithm checks if it has been visited with a lower cost. If the current node's cost is not lower, it skips further expansion.

```
    if (node.ID in visited and visited[node.ID] != node.cost +
        manhattanDistance(node)): continue

```

6. **Exploring Neighboring States:** The algorithm iterates through all possible directions and checks whether the player can move in that direction. For each valid move, a new node is generated, and its heuristic cost is calculated using the `manhattanDistance()` function.

```
    for dir in directions:
        if (node.canMove(dir)):
            newNode = node.move(dir)
            newNode.heuristicCost = manhattanDistance(newNode)
            if (newNode.ID not in visited or newNode.cost
                + newNode.heuristicCost < visited[newNode.ID]):
                heapq.heappush(pq, newNode)
                visited[newNode.ID] = newNode.cost + newNode.heuristicCost
                cntNode += 1

```

7. **Time and Memory Management:** As with other search algorithms, A* monitors the elapsed time to avoid exceeding a time limit (3 minutes). Memory usage is also tracked using the `psutil` library.

```
    currentTime = time.time()
    if (currentTime - startTime > 180): return "-1", -1, -1, -1, -1

```

8. **Return the Result:** The algorithm returns the result once a goal state is found or the search is completed. The result includes the path to the goal, the total cost, the time taken, memory usage, and the number of nodes visited.

```
    path = node.path
    endTime = time.time()
    elapsedTime = endTime - startTime
    memUsed = psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2
    return path, node.cost, elapsedTime, memUsed, cntNode

```

Manhattan Distance: The `manhattanDistance` function calculates the total cost based on the Manhattan Distance heuristic. This is done by first identifying the positions of the boxes and the goal positions on the board. For each box, the distance to each target is calculated and weighted according to the box's weight. The optimal assignment of boxes to targets is then computed using the **Hungarian algorithm** (via the `linear_sum_assignment` function from the `scipy.optimize` module), which minimizes the total cost.

```
def manhattanDistance(node):
    totalCost = 0
    targets = list()
    boxPositions = list()

    for goal in node.goals:
        if (node.board[goal.x][goal.y].type != '$'):
            targets.append(goal)

    for r in range(len(node.board)):
        for c in range(len(node.board[r])):
            if (node.board[r][c].type == '$') and
                (Pair(r, c) not in node.goals):
                boxPositions.append(Pair(r, c))

    distances = []
    for box in boxPositions:
        for target in targets:
            distance = (abs(box.x - target.x) + abs(box.y - target.y)) *
                node.board[box.x][box.y].weight
            distances.append(distance)

    cost_matrix = np.array(distances).reshape(len(targets), len(boxPositions))
    row_ind, col_ind = linear_sum_assignment(cost_matrix)
    totalCost = cost_matrix[row_ind, col_ind].sum()
    return totalCost
```

Termination Conditions: A* terminates when one of the following conditions is met:

- A goal state is found (`node.isGoalState()` returns `True`).
- The time limit (3 minutes) is exceeded.

Time Complexity: The time complexity of A* depends on the heuristic and the branching factor. In the worst case, it is $O(b^d)$, where b is the branching factor and d is the depth of the solution.

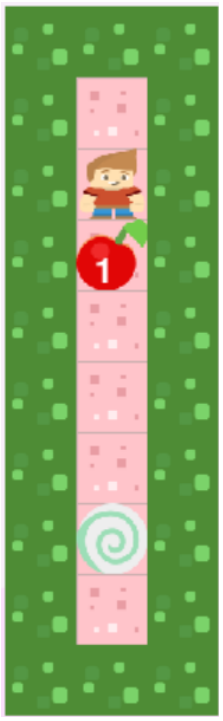
Space Complexity: $O(b^d)$

5 Test Case

To evaluate the performance of the four algorithms, we have created a set of 12 test cases with varying levels of difficulty.We ran our test cases on a MacBook with an M3 Pro chip, 1 TB of storage, and 36 GB of memory. The results are shown below.

5.1 Test Case 1

In this test case, we evaluate the performance of four algorithms on a narrow map with a single solution path.



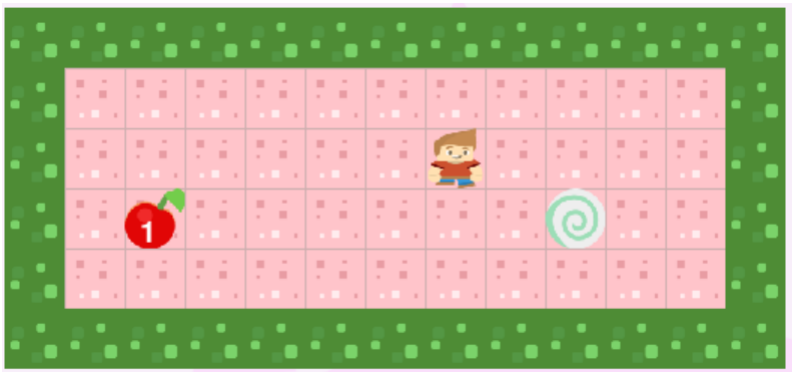
Test Case 1: Narrow, One-Way Solution Map

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	4	8	14	5.52	112.73
DFS	4	8	10	3.52	112.79
BFS	4	8	8	2.00	112.80
UCS	4	8	15	5.03	112.80

The results show that all four algorithms take the same number of both steps (4) and weight (8). The number of nodes visited varies slightly, the time taken for each algorithm is relatively close, and the memory usage is similar across all algorithms. Despite these small differences, the core performance of all algorithms remains quite consistent for this map.

5.2 Test Case 2

In this test case, we evaluate the performance of four algorithms on a large map with no walls.



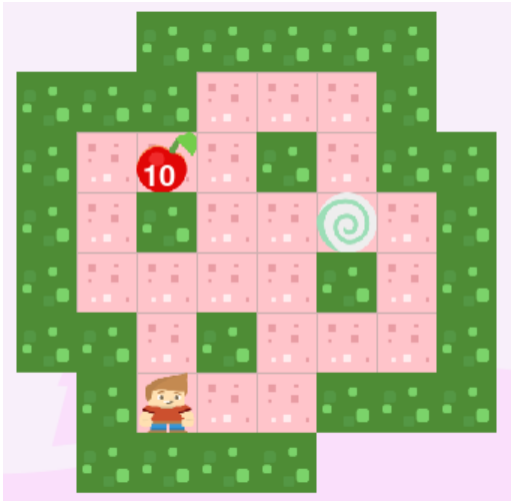
Test Case 2: Large Map, No Walls

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	15	22	322	470.12	112.87
DFS	123	136	1511	2458.42	112.80
BFS	15	22	412	535.44	112.79
UCS	15	22	837	1358.65	112.84

The results indicate that A*, BFS, and UCS all take 15 steps and achieve the same weight of 22. However, A* explores fewer nodes (322) compared to BFS (412) and UCS (837), resulting in significantly lower time usage. DFS, on the other hand, takes considerably more steps (123) and explores many more nodes (1511), leading to much higher computation time. Despite these differences, memory usage remains consistent across all algorithms.

5.3 Test Case 3

In this test case, we evaluate the performance of four algorithms on a map with a single stone and some walls, showcasing the effectiveness of A* in reducing the number of nodes visited.



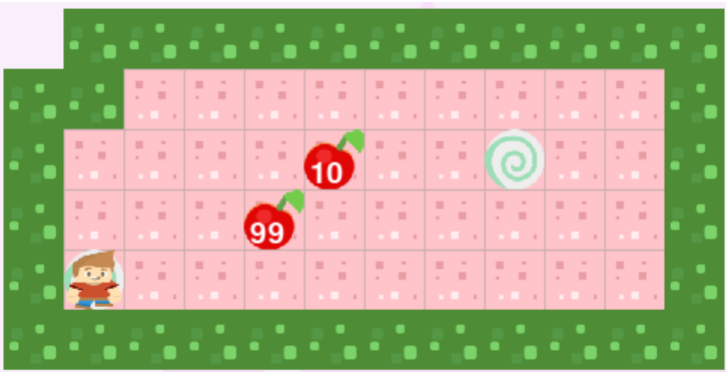
Test Case 3: One Stone and Some Walls

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	37	97	137	129.28	112.86
DFS	67	147	155	120.27	112.90
BFS	37	97	175	152.07	112.85
UCS	37	97	190	161.74	112.89

The results show that A* visits fewer nodes (137) compared to BFS (175) and UCS (190), highlighting its ability to find the optimal path more efficiently. While DFS takes the most steps (67), A* and BFS both take the same number of steps (37), but A* consistently performs better in terms of nodes visited and time taken. The memory usage is similar across all algorithms.

5.4 Test Case 4

In this test case, we assess the performance of four algorithms on a map containing two stones, demonstrating the algorithm's ability to handle scenarios with multiple stones.



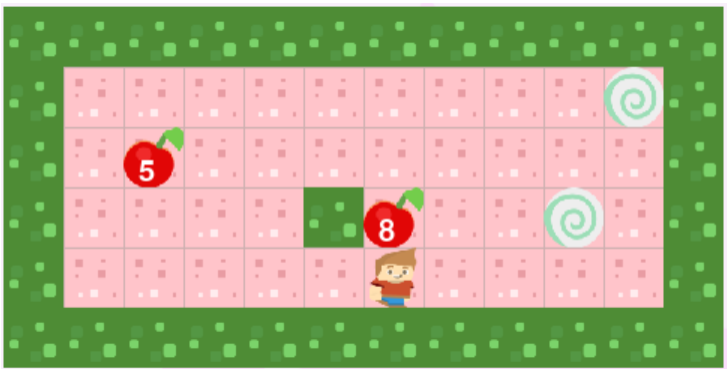
Test Case 4: Two Stones only

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	17	443	1095	572.64	215.80
DFS	148	714	1170	566.59	219.39
BFS	17	443	12603	5231.41	277.19
UCS	17	443	28090	16939.75	276.48

The results show that A* performs significantly better than the other algorithms in terms of the number of nodes visited. Despite having the same number of steps and weight, the BFS and UCS algorithms visit a much larger number of nodes. DFS, on the other hand, takes the longest path in terms of steps but still performs better in terms of memory usage compared to BFS and UCS.

5.5 Test Case 5

In this test case, we evaluate the performance of four algorithms on a map with two stones, where the weights are approximately equal. As a result, the performance of BFS, A*, and UCS algorithms does not differ significantly.



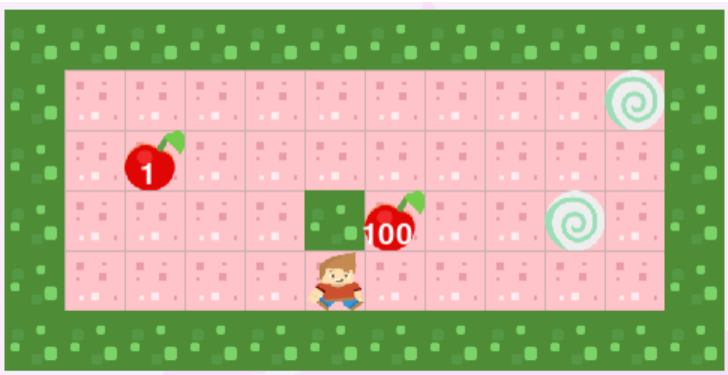
Test Case 5: Two Stones with Approximately Equal Weights

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	28	113	10890	7251.22	275.64
DFS	652	1051	15591	7756.93	278.61
BFS	26	114	27923	13119.88	296.19
UCS	28	113	43787	27058.62	296.47

The results show that three algorithms, except DFS, performed similarly in terms of steps, weight, and memory usage. However, A* takes significantly less time than BFS and UCS. DFS, although it visits the most nodes, still takes less time than both BFS and UCS.

5.6 Test Case 6

In this test case, we evaluate the performance of four algorithms on a map with two stones, where the weight difference between the stones is significant. This allows us to observe how A* and UCS incorporate the weight in their pathfinding, while BFS only aims to find a valid path.



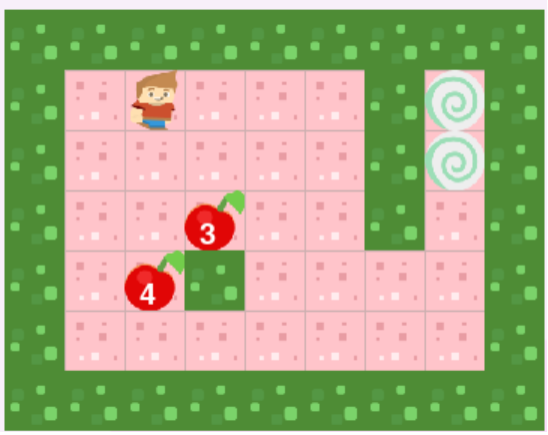
Test Case 6: Two Stones with a Significant Weight Difference

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	28	537	10311	6829.19	295.38
DFS	652	3979	15591	7736.77	298.25
BFS	26	634	27923	13163.75	303.36
UCS	28	537	37362	22596.01	302.39

The results show that A* and UCS effectively incorporate the weight of the stones. In contrast, BFS only focuses on finding a valid path. Although the weight difference is significant, BFS still finds a solution faster in terms of time, but the path is less efficient in terms of node exploration. DFS, although it visits the most nodes, still takes less time than both BFS and UCS.

5.7 Test Case 7

In this test case, we evaluate the performance of four algorithms on a simple maze with two stones and a switch. The focus is to test the stacking phenomenon of switches to prevent one stone from blocking the other if it reaches the switch first.



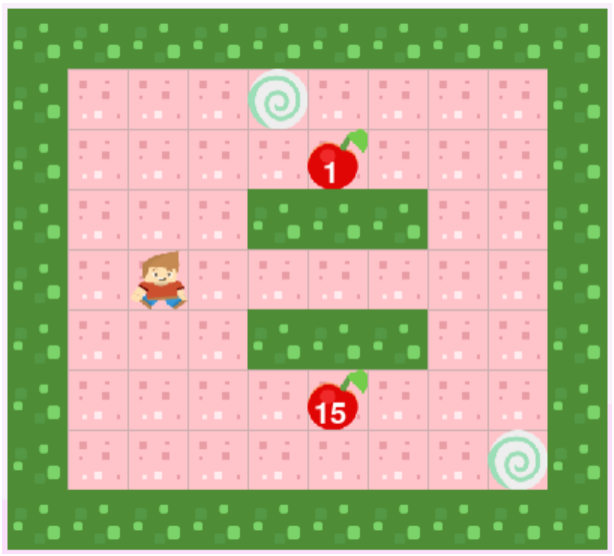
Test Case 7: Simple Maze with Two Stones and Stacking Switches

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	42	102	17855	10157.53	299.11
DFS	782	1204	15528	6196.03	301.70
BFS	42	102	19429	8302.97	301.80
UCS	42	102	19733	9340.11	300.88

The results demonstrate that A*, BFS, and UCS algorithms all performed similarly in terms of steps and weight.The generated node of A* is little smaller because we apply Hungarian algorithm to match the stone and switch.

5.8 Test Case 8

In this test case, we assess the performance of four algorithms on a map with two stones and a wall in the center. The focus is on the limitations of the Manhattan heuristic, which causes A* to perform less optimally than UCS and BFS.



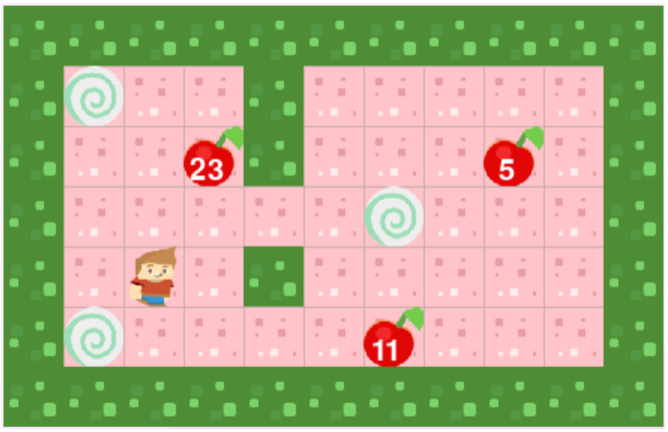
Test Case 8: Two Stones with Weaknesses of Manhattan distance

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	28	92	4689	3091.13	296.44
DFS	1274	1990	24062	14544.25	298.42
BFS	26	92	19730	10340.45	343.05
UCS	26	92	30801	22214.81	325.48

The results demonstrate that while A*, BFS, and UCS performed similarly in terms of steps and weight, DFS still performs with most steps, weight and nodes, and A* does not offer the same optimization as UCS. A* uses the Manhattan distance and Hungarian algorithm, which is not as effectively as UCS and BFS.

5.9 Test Case 9

In this test case, we evaluate the performance of four algorithms on a map with three stones and some walls. The goal is to test if the algorithms work well with more than two stones and account for obstacles.



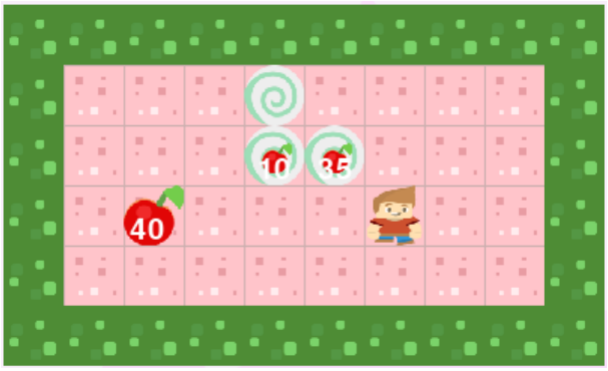
Test Case 9: Three Stones with Walls

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	42	227	38770	35186.43	315.53
DFS	247	638	603	209.63	316.98
BFS	40	271	234081	126470.91	619.25
UCS	-	-	-	*Timeout*	-

The results indicate that the A* algorithm performs reasonably well with three stones, although the number of nodes visited and the execution time are higher compared to simpler scenarios. DFS performs with fewer nodes and takes significantly less time, but it requires way more steps. BFS, while visiting many more nodes, results in much higher time and memory usage. UCS encounters a timeout, likely due to the increased complexity of having three stones and obstacles.

5.10 Test Case 10

In this test case, we check the behavior of BFS and DFS when two out of three stones are already at the goal. This is to ensure that these algorithms still work correctly and do not "work wrong" in this specific scenario.



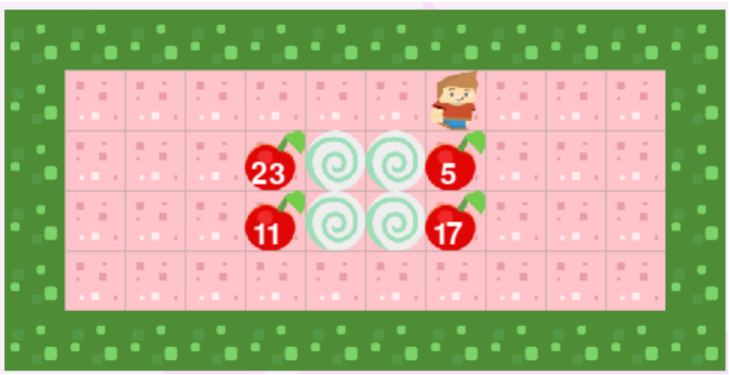
Test Case 10: Three Stones, Two at Goal

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	13	143	135	45.27	484.47
DFS	997	4287	83913	34855.90	436.22
BFS	12	172	3258	903.65	429.22
UCS	13	143	36707	24571.37	425.22

The results indicate that A*, UCS, and BFS, work correctly when two stones are already at the goal. DFS also has solution but it requires way too much steps, which means it works wrong. In addition, A* and UCS show similar performance in terms of steps and weight, with UCS slightly slower in terms of time. BFS takes least steps, and weight in this scenario.

5.11 Test Case 11

This test case upgrades the scenario to 4 stones, with the goal positions placed next to each other. The performance of different algorithms is measured in terms of steps, weight, nodes visited, and execution time.



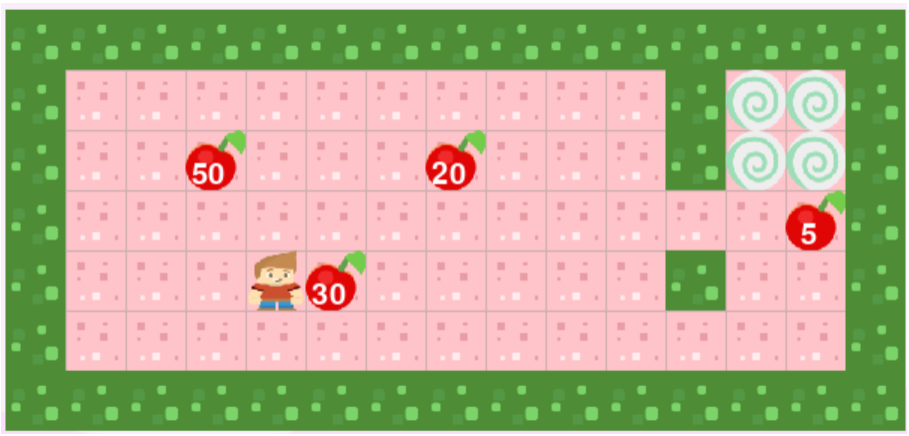
Test Case 11: Four Stones, Goal Next to Each Other

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	17	73	631	286.10	147.70
DFS	-	-	-	*Timeout*	-
BFS	17	73	180140	63592.56	1644.50
UCS	17	73	84078	105102.03	997.14

The results show that A* performs efficiently in terms of steps, nodes visited, time, and memory usage. DFS exceeds its time limit due to the large number of nodes it needs to visit, resulting in a timeout. BFS and UCS perform similarly in terms of steps and weight, and both BFS and UCS consume considerable memory resources, with UCS requiring more time to compute the solution.

5.12 Test Case 12

This test case tests the performance of the algorithms with a large map and 4 stones. Due to the complexity of the scenario, all algorithms failed to provide results within the time limit.



Test Case 12: Four Stones and Large Map

Algorithm	Steps	Weight	Nodes Visited	Time (ms)	Memory (MB)
A*	-	-	-	*Timeout*	-
DFS	-	-	-	*Timeout*	-
BFS	-	-	-	*Timeout*	-
UCS	-	-	-	*Timeout*	-

In this case, all algorithms—A*, DFS, BFS, and UCS—exceeded the time limit due to the large size of the map and the increased number of stones. These factors significantly increase the complexity of the problem, causing each algorithm to fail to compute the solution within the allowed time frame (3 minutes).

6 Compare Algorithms

To understand thoroughly the performance of the four algorithms, we have built some tables and bar charts to have a better illustration

6.1 Steps

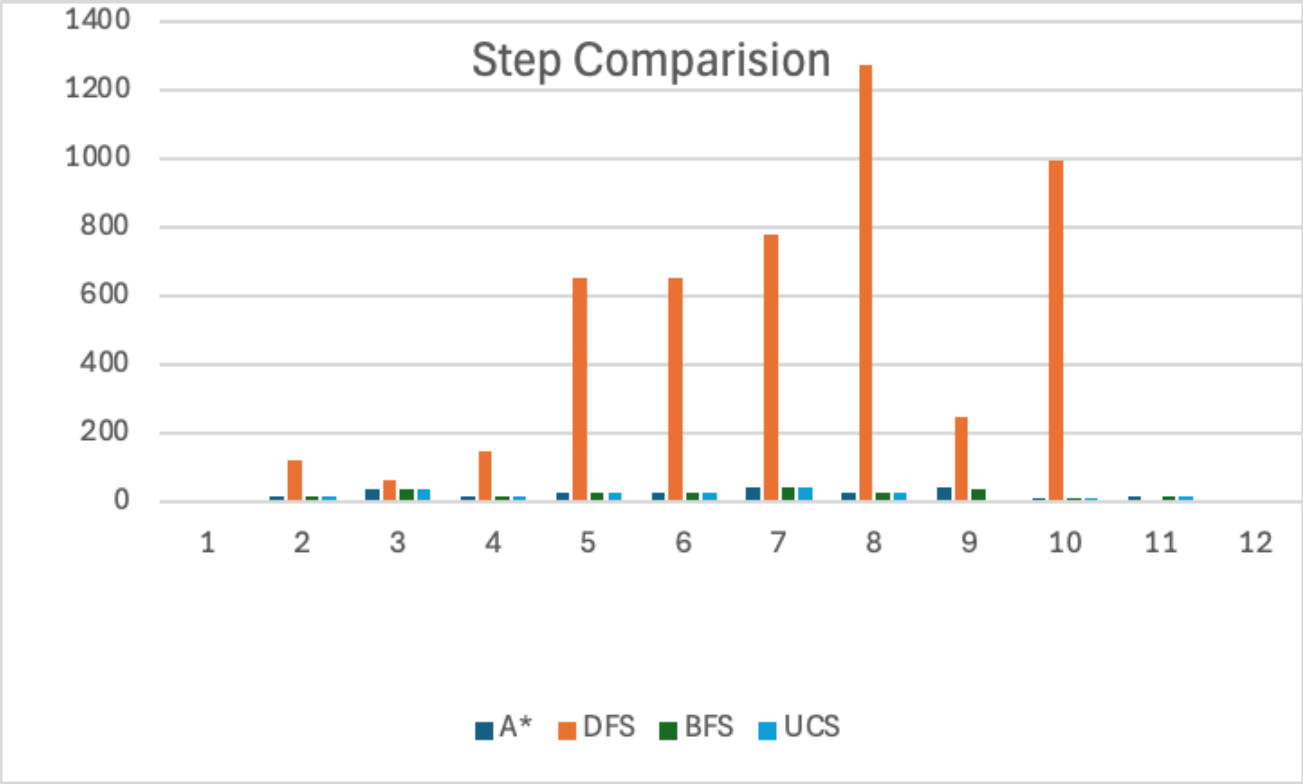


Figure 6: Compare Steps

Algo	1	2	3	4	5	6	7	8	9	10	11	12
A*	4	15	37	17	28	28	42	28	42	13	17	-
DFS	4	123	67	148	652	652	782	1274	247	997	-	-
BFS	4	15	37	17	26	28	42	26	40	12	17	-
UCS	4	15	37	17	28	28	42	26	-	13	17	-

Table 1: Compare Steps

- In the simple Test 1, with a narrow map and only one solution, all four algorithms—DFS, BFS, UCS, and A*—perform equally well, taking the same number of steps. However, in more complex maps with multiple paths and obstacles, DFS performs poorly, as it explores deep paths without efficiently finding optimal solutions, leading to significantly more steps than the other algorithms.
- BFS performs better in terms of steps, as it finds the shortest path by distance without considering path costs. However, it doesn't guarantee a minimum-cost solution. UCS and A* generally perform best in complex cases by finding optimal paths, with A* leveraging heuristics for greater efficiency.

- In Test 12, with a large map and more stones, all four algorithms timeout, indicating the map's complexity exceeds their computational limits. Overall, UCS and A* are most effective for complex Sokoban maps, while DFS is the least efficient.

6.2 Weight

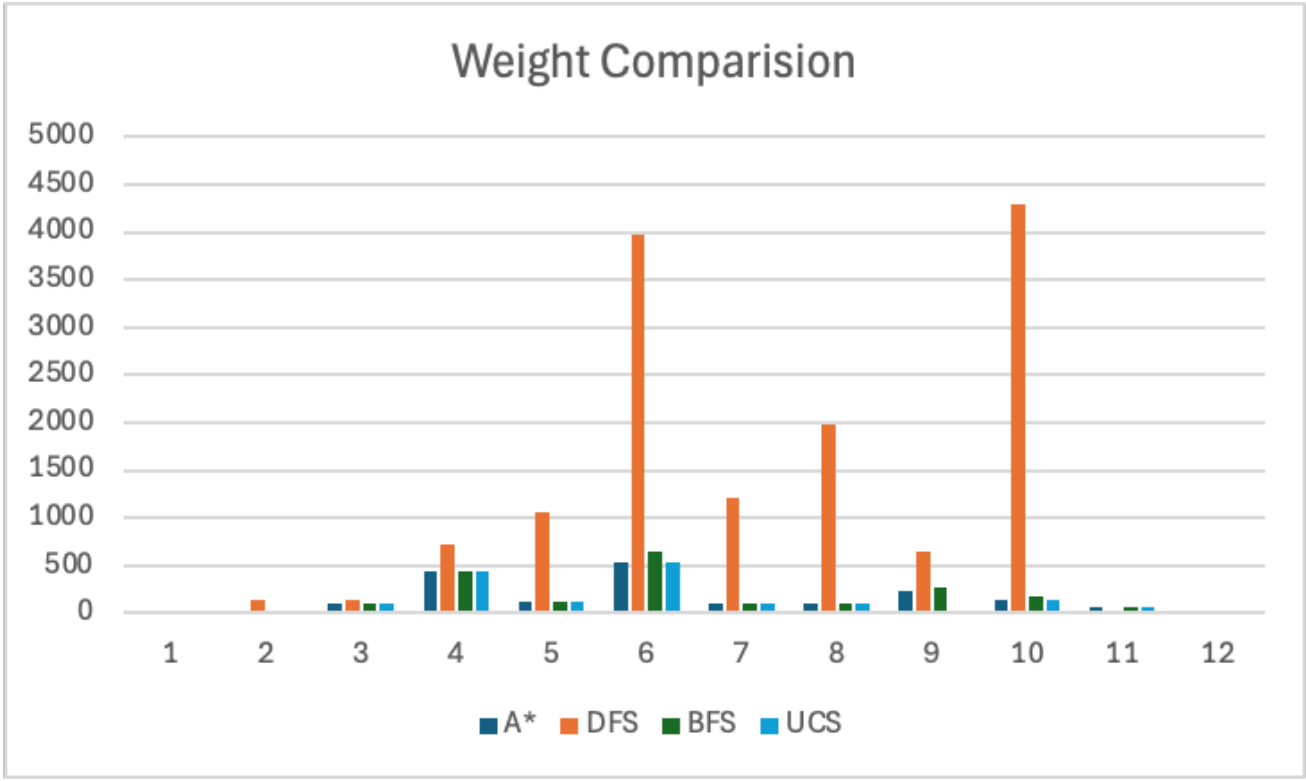


Figure 7: Compare Weight

Algo	1	2	3	4	5	6	7	8	9	10	11	12
A*	8	22	97	443	113	537	102	92	227	143	73	-
DFS	8	136	147	714	1051	3979	1204	1990	638	4287	-	-
BFS	8	22	97	443	114	643	102	92	271	172	73	-
UCS	8	22	97	443	113	537	102	92	-	143	73	-

Table 2: Compare Weight

- DFS has the highest costs in most cases because its depth-first approach often follows longer, non-optimal paths without considering costs. It also times out in Tests 11 and 12, showing inefficiency in complex maps.
- BFS, while finding paths with fewer steps, does not account for weights, resulting in moderate but non-optimal costs. It struggles in complex cases like Test 12.

- UCS finds low-cost paths, making it effective for minimizing path weight, but it times out in Tests 9 and 12 due to high complexity.
- A* generally maintains low weights by using heuristics to find efficient paths, performing better than UCS in avoiding timeouts in complex tests.

6.3 Node

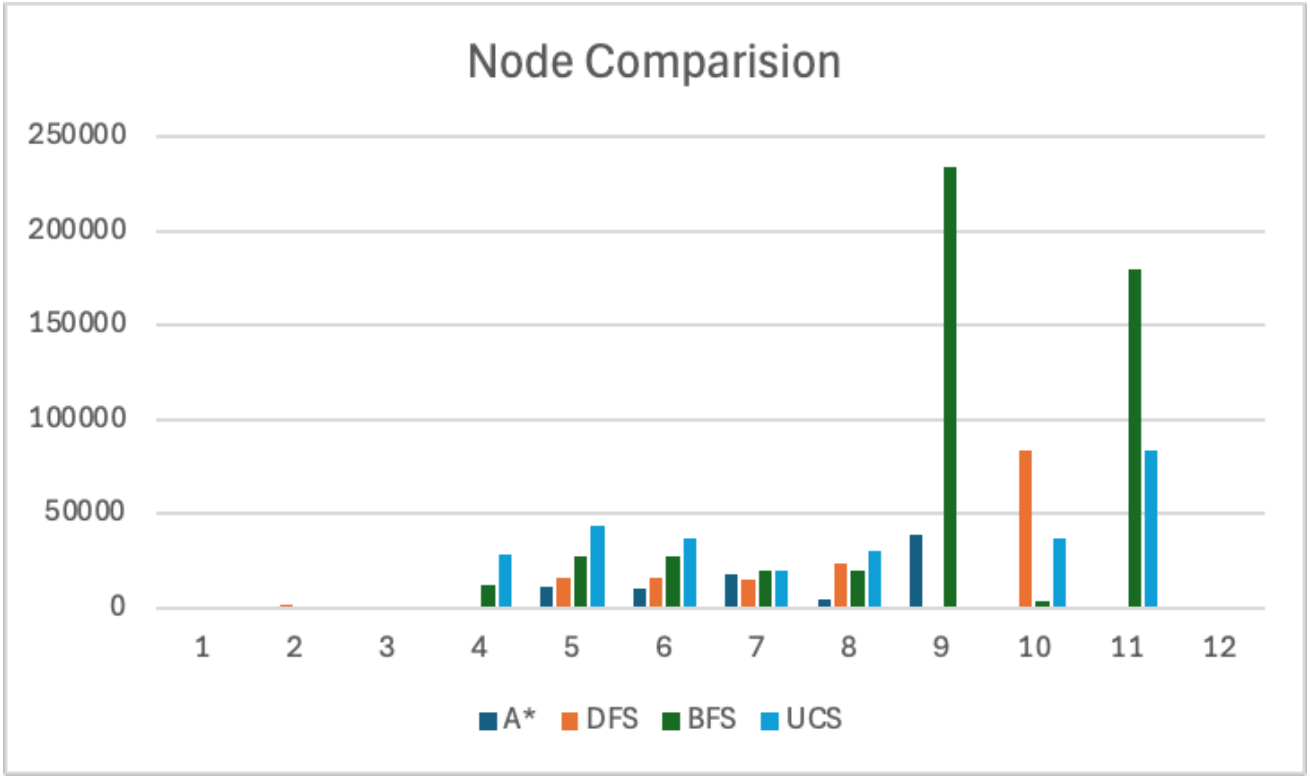


Figure 8: Compare Node

Algo	1	2	3	4	5	6	7	8	9	10	11	12
A*	14	322	137	1095	10890	10311	17855	4689	38770	135	631	-
DFS	10	1511	155	1170	15591	15591	15528	24062	603	83913	-	-
BFS	8	412	175	12603	27923	27923	19429	19730	234081	3258	180140	-
UCS	15	837	190	28090	43787	37362	19733	30801	-	36707	84078	-

Table 3: Compare Node

- A* is the most efficient, exploring fewer nodes in most cases. It performs consistently well, with only slight increases in complex cases.
- DFS and BFS are uninformed searches. By applying an early goal test, they end when the goal state is generated. Compared with UCS (an informed search), DFS and BFS may expand fewer nodes than UCS because of the deadlock detections.

- UCS explores the most nodes, especially in complex cases like Test 9 and 11, highlighting their inefficiency in large search spaces.

6.4 Time complexity

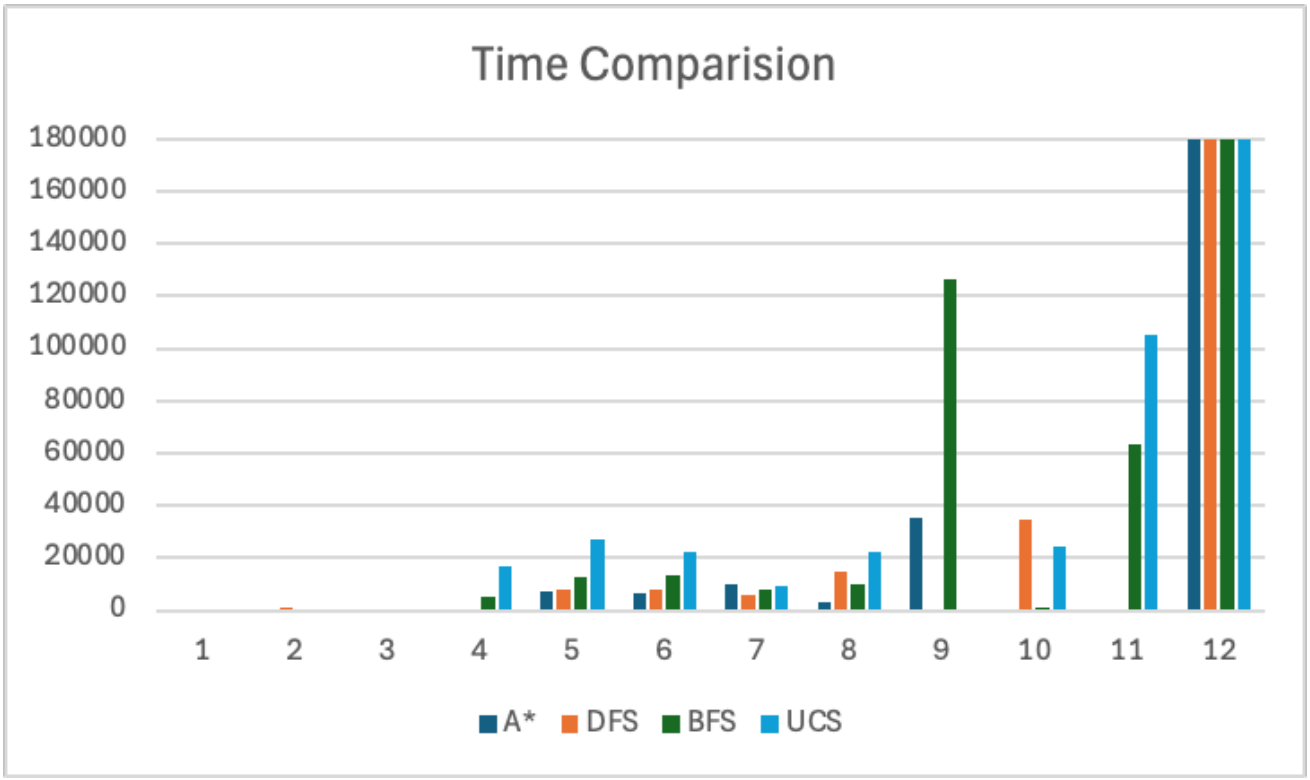


Figure 9: Compare Time Complexity

Algo	1	2	3	4	5	6	7	8	9	10	11	12
A*	14	322	137	1095	10890	10311	17855	4689	38770	135	631	*Timeout*
DFS	10	1511	155	1170	15591	15591	15528	24062	603	83913	*Timeout*	*Timeout*
BFS	8	412	175	12603	27923	27923	19429	19730	234081	3258	180140	*Timeout*
UCS	15	837	190	28090	43787	37362	19733	30801	*Timeout*	36707	84078	*Timeout*

Table 4: Compare Time Complexity

- A* is the fastest in most cases and often completes the task where other algorithms time out. However, A*'s time can increase in complex problems (Test 9 and 12), but it is still more efficient than UCS.
- DFS outperforms in processing time for complex test cases, due to its simple approach, the lack of need to store many states, and deadlock detections.
- BFS and UCS generally take the most time, due to having to traverse many states and nodes before reaching the goal state.

- In summary, A* has the best time efficiency, followed by DFS, while BFS and UCS are slower and prone to timing out in complex cases.

6.5 Memory

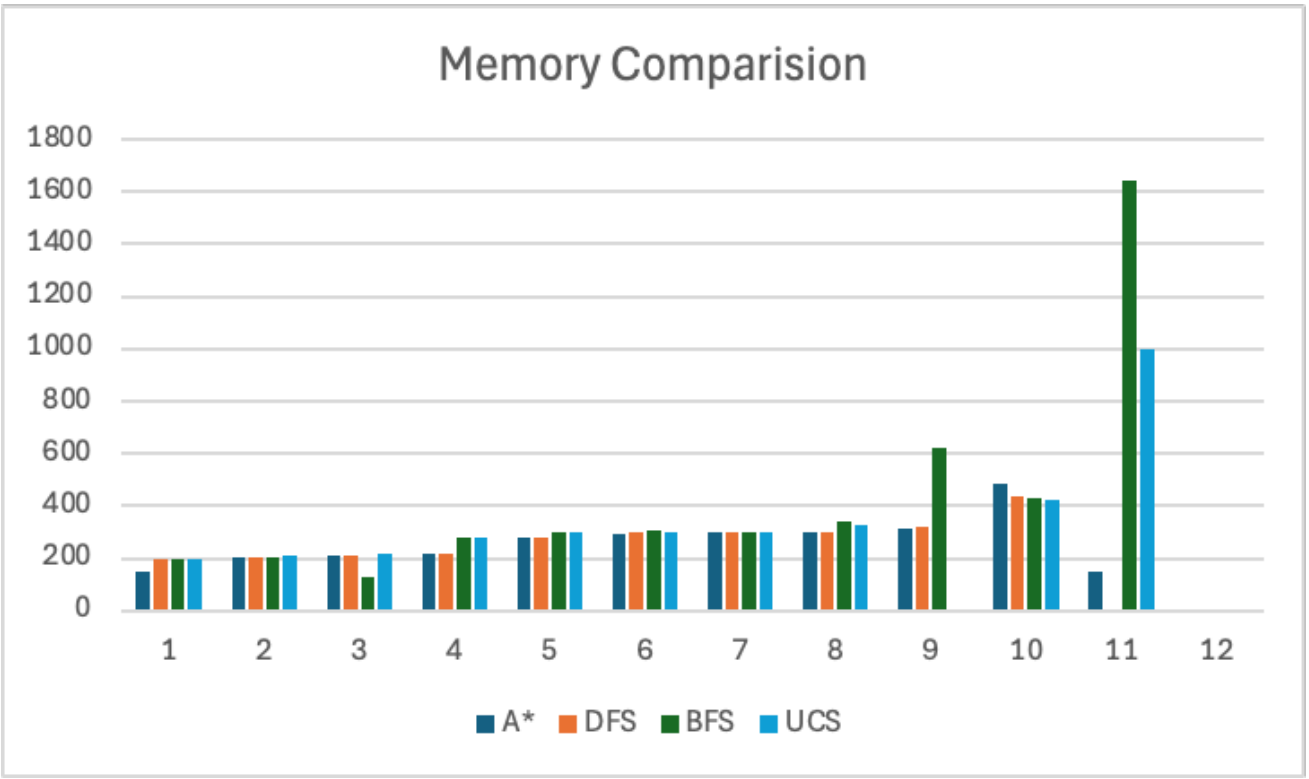


Figure 10: Compare Memory

Algo	1	2	3	4	5	6	7	8	9	10	11	12
A*	145.38	201.69	207.91	215.8	275.64	295.38	299.11	296.44	315.53	484.47	147.7	-
DFS	193.25	203.53	211.19	219.39	278.61	298.25	301.7	298.42	316.98	436.22	-	-
BFS	198.98	205.86	129.67	277.19	296.19	303.36	301.8	343.05	619.25	429.22	1644.5	-
UCS	199.55	207.72	217.59	276.48	296.47	302.39	300.88	325.48	-	425.22	997.14	-

Table 5: Compare Memory

- DFS uses the least memory of the algorithms since it only stores nodes in the current branch.
- BFS and UCS have the largest memory requirements since they store all nodes in an expanding queue. This causes problems when the search space is large.
- A* is worse than DFS but better than BFS and UCS in terms of memory usage, with better memory savings than BFS and UCS due to heuristics.

7 Conclusion

In this project, we investigated different search algorithms to solve the stone puzzle problem. Based on our experiences, we have drawn the following conclusions.

- DFS is efficient in memory usage for simple cases; can quickly find solutions in narrow, straightforward paths. It does not consider path cost, leading to non-optimal solutions.
- BFS finds the shortest path in terms of the number of steps. Ideal for unweighted maps or when the focus is on step count rather than cost. It needs more memory than DFS.
- UCS and A* are more optimal in terms of weights, with A* showing superiority in saving memory and time thanks to heuristics. In cases where high accuracy and resource optimization are required, A* is the best choice. Through the charts, it can be seen that A* is often the optimal solution in terms of performance when it is necessary to balance time, number of steps, memory, and weight optimization, while DFS is the best solution in terms of resources when weights are not required.
- There is an area for improvement is the inclusion of additional features like deadlock detection, where a stone is trapped and cannot reach its destination, and providing a more effective heuristic for estimating the optimal path towards the goal. Through careful project research, understanding related concepts, and efficient task delegation, we ensured smooth progress, uniform algorithm implementation, and minimized challenges. Proper planning and a structured approach were key to the successful completion of the project.

8 References

1. Deadlock Detection: http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks
2. Pygame Documentation: https://www.pygame.org/docs/
3. Solving Sokoban Game: Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf