Project 1. Search

Ares's adventure

CS420

Group's member:

Le Van Cuong - 22125013

Nguyen Minh Quan - 22125079

Bui Danh Nghe - 22125063

Phung Khanh Vinh - 22125122

Contents

1 Task Distribution

Table 1: Task Distribution & Completion Rate

	(22125013)	(22125063)	(22125079)	(22125122)
Console game (5% Total)			100%	100%
UI game (5% Total)	100%	100%		
DFS (10% Total)			100%	100%
BFS (10% Total)				100%
UCS (10% Total)		100%		
A* (10% Total)	100%			
Test cases (10% Total)	100%	100%	100%	100%
Output file (5% Total)			100%	
Fix bug + help each others (15% Total)	100%	100%	100%	100%
Demo (5% Total)		100%	100%	
Write report (15% Total)	100%	100%	100%	100%
Total	100%	100%	100%	100%

2 Youtube demo

Part 1: https://www.youtube.com/watch?v=JwVBm4KaOnw

Part 2: https://www.youtube.com/watch?v=XJQL6F59yjo

3 Introduction

This report summarizes our development and implementation of the solution for the "Ares's Adventure" problem. The goal here is to help a young adventurer, Ares, who has to navigate his way through a really complex maze with walls, stones, and switches to move the stones onto designated switches that unlock a gate toward a treasure. This calls for strategic planning and precision in moving the stones to particular locations.

These include the implementation of Breadth-First Search, Depth-First Search, Uniform Cost Search, and A* Search with heuristic guidance in guiding Ares efficiently. Each of these algorithms explores different aspects of search optimization and alternative problem-solving techniques, thus allowing us to evaluate their performances within a constrained environment.

4 Self-evaluation

We successfully implemented and tested each algorithm using custom-designed test cases. Additionally, we developed a robust GUI to visualize the paths returned by the algorithms, which allowed for effective analysis and demonstration of the results. Overall, we are satisfied with our achievements and outcomes.

Accomplishments

- Met all project objectives
- Developed a GUI for algorithm demonstration
- Created unique test cases to highlight the strengths and weaknesses of each search algorithm

Possible Improvement

- Enhance the GUI aesthetics

5 Algorithm Explaination

5.1 Definition

5.1.1 State

A state of our game is define by $k+1$ pair of coordinates in the form of:

$$(Ares_x, Ares_y), (Stone_{1_x}, Stone_{1_y}), (Stone_{2_x}, Stone_{2_y}), ..., (Stone_{k_x}, Stone_{k_y})$$

Which represented the position of Ares, and the position of each of the k stone.

5.2 Utils Function

Here is the list of function shared by our search Algorithm:

5.2.1 heuristic(x)

This function return the heuristic value of x (x is a state)
The heuristic value is calculate as follow: The sum of manhattan distance between each stone to the closest button multiply by the stone weight (Even if a button is closest to more than one stone). We can easily prove that this heuristic is admissible as for any solution, each stone must at least move to the closest button.
In addition, this heuristic is also consistent. As after each move, if it push the box and reduce or heuristics, the change is always at most the cost of that move.
There are likely better heuristic. However, this heuristics is chosen for the balance between time and memory consumption.

5.2.2 canMove(dr, dc):

Check if whether the agent (Ares) can move in the given direction or not.

5.2.3 move(direction)

If the Ares can move in the given direction, change the current state into the next state. Otherwise, do nothing.

5.2.4 isGoalState(x)

This function return whether x is the goal state, i.e whether all button is pressed in state x.

5.2.5 getSuccessors(x)

Get all the successors of state x. The function check in 4 directions (U, D, L, R) which one Ares is capable of moving, then add the after-move state to the list. Each successor represents a new game state after Ares moves in one of the four possible directions (up, down, left, right). The function returns a list of tuples, where each tuple contains a new game state, the direction of the move, and the cost of that move.

5.3 Search Algorithm

Here is the explanation of each of our search algorithm. The algorithm here is implemented closely to the one from the theory session.

5.3.1 bfs(x)

The bfs function is an implementation of the Breadth-First Search (BFS) algorithm for the MazeGame class. The algorithm do as follow: Start from the initial state. By maintaining and processing through a FIFO queue, it push into the queue all the neighbor state of the current that have not been explored and not in the queue. It repeat this process until the goal state is processed. It explores all possible paths from the starting state to a goal state and returns the solution with information such as the total cost, the number of steps, the time taken, memory usage, and the path taken to reach the goal.

5.3.2 dfs(x)

The dfs function is an implementation of the Depth-First Search (DFS) algorithm for the MazeGame class. The algortithm itself is not quite different from the BFS algorithm. Instead of using a FIFO queue, DFS use a stack that follows the Last In, First Out (LIFO) principle. The function checks each state to determine if it's a goal state; if found, it returns with details on the solution path, including total cost, steps, nodes generated, time taken, memory usage, and the path taken to reach the goal.

5.3.3 ucs(x)

The ucs function implements the Uniform Cost Search (UCS) algorithm for the MazeGame class. Starting from the initial state, UCS uses a priority queue to explore paths with the lowest cumulative cost, expanding nodes in increasing order of path cost. Each successor state is added to the queue if it hasn't been explored or if a cheaper path is found. The search continues until the goal state is reached, at which point it returns the solution, including the total cost, number of steps, nodes generated, time taken, memory usage, and the path taken to reach the goal.

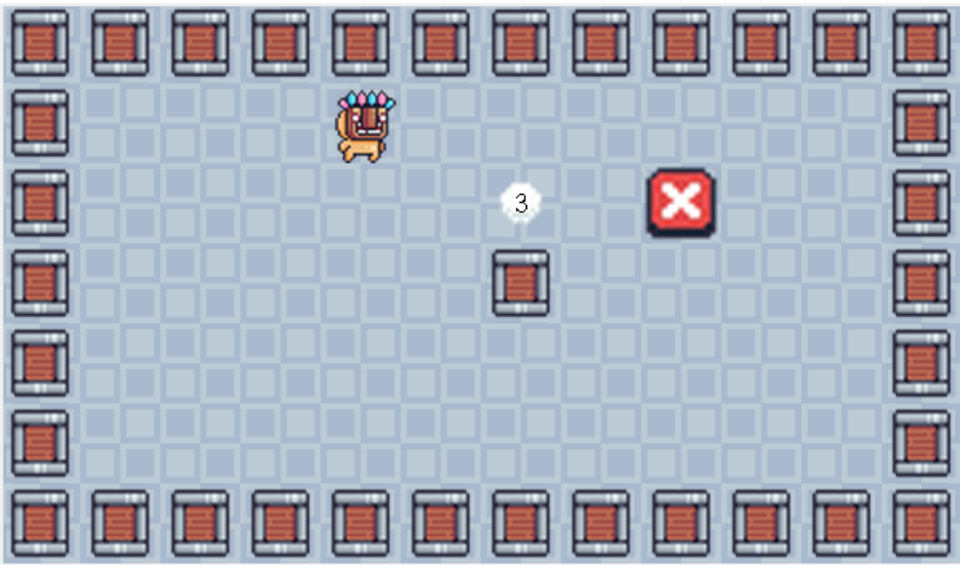
5.3.4 A*(x)

The A* function implements the A* Search algorithm for the MazeGame class. This algorithm starts from the initial state and uses a priority queue to explore paths with the lowest estimated total cost. Each node is prioritized based on the sum of its path cost and a heuristic that estimates the remaining cost to the goal. The function checks if the goal is reached at each step; if not, it generates and enqueues all unexplored successors with their updated path costs and heuristic values. The process continues until the goal is reached, and it returns the solution details, including the path cost, number of steps, nodes generated, time taken, memory usage, and the solution path.

6 Testing and Experiment

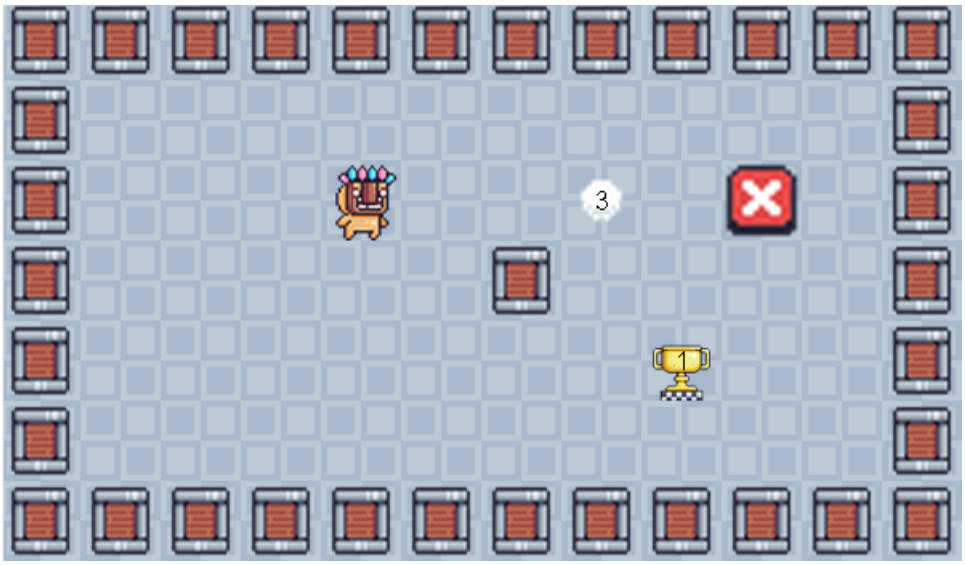
The fun part. We have prepare 10 input, each with their specific purpose.

Input 1: Simple test case for demonstration



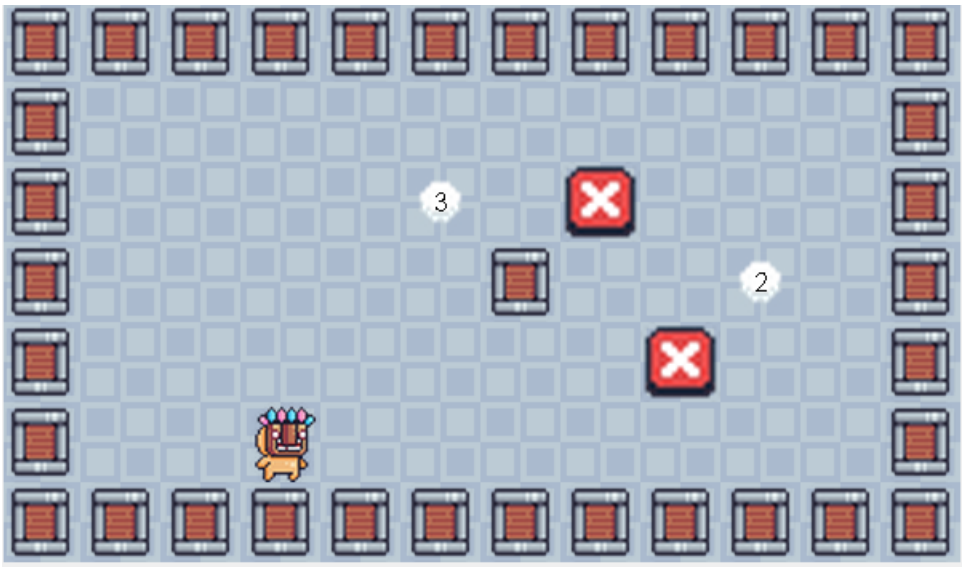
Simple test case, mostly for demonstration, use to show the type of path these algorithm take.

Input 2: Simple test case that all algorithm perform well



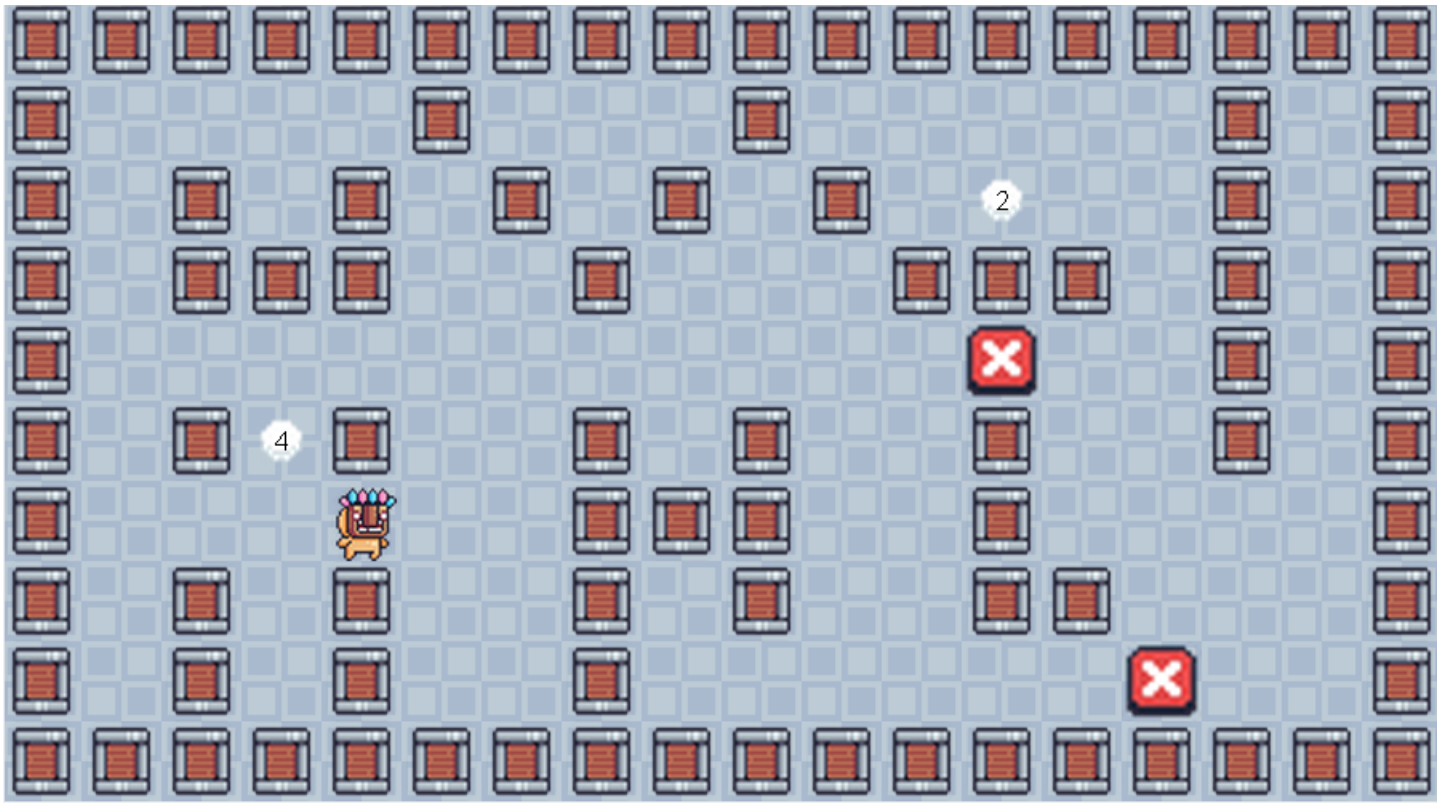
Very simple test case, in which the solution is favourable to all search algorithm

Input 3: Random test case with few walls and many states (DFS performs poorly)



Random test case with minimal walls, resulting in a large search space. DFS performs poorly here due to its tendency to explore deeply without optimizing for path cost.

Input 4: Random test case with many walls, resulting in fewer states (DFS performs better)



A random configuration with many walls, reducing the number of available paths. DFS performs relatively better in this setup since fewer paths need to be explored.

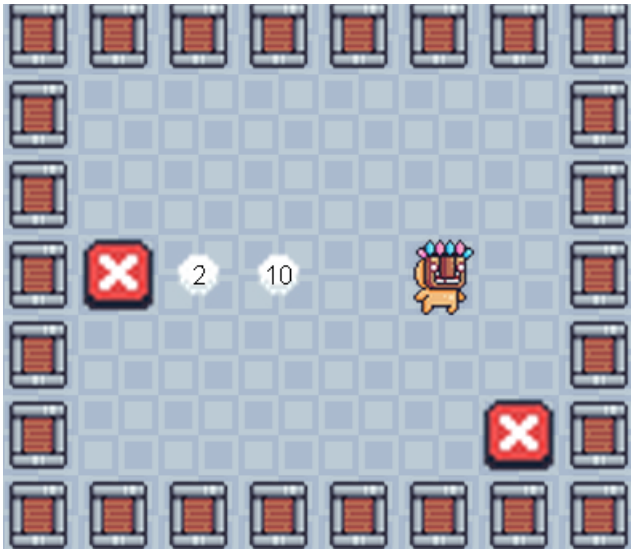
Input 5: Simple test case with very few states

A straightforward test case with limited states, allowing all algorithms to find solutions quickly. This demonstrates their efficiency in a small search space.

Input 6: Test case with no solution

A setup where the goal is unreachable, designed to test how quickly each algorithm can exhaustively search all possible solution.

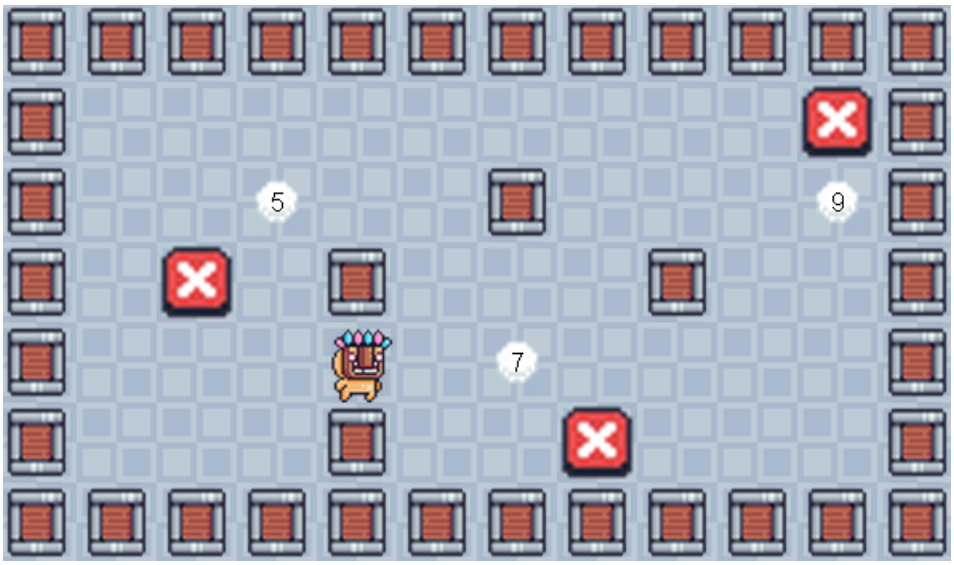
Input 7: Test case for UCS and A* optimality



This layout is tailored to highlight the optimality of UCS and A* in finding the lowest-cost path, demonstrating their effectiveness in a weighted environment.

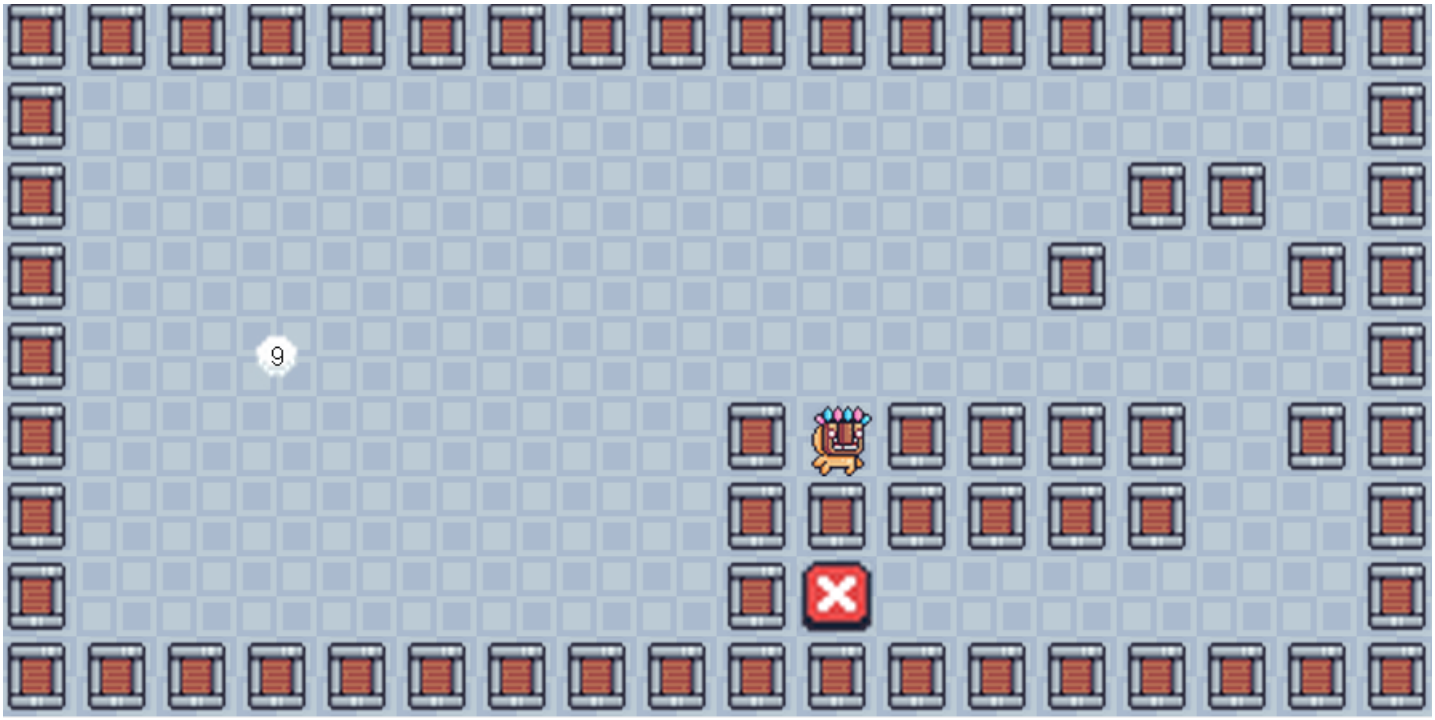
	Steps	Cost	Nodes Generated	Time (ms)	Memory (MB)
BFS	12	64	5905	896.63	2.52
UCS	17	49	36250	6160.53	1.89
A Star	17	49	5702	5386.91	1.58
DFS	225	445	4794	694.23	0.88

Input 8: Random complex maze



A more intricate maze that challenges the optimality and efficiency of the algorithms in complex search spaces.

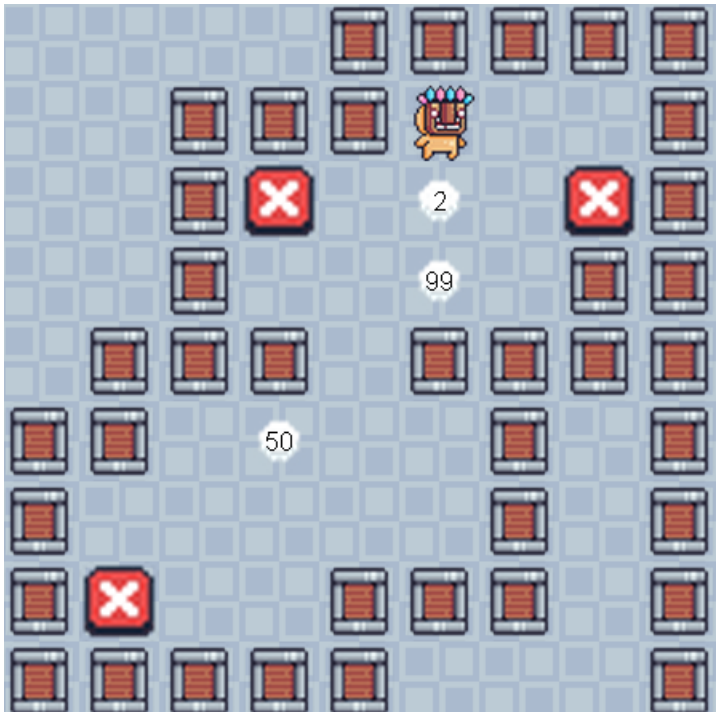
Input 9: Special hand-crafted maze where DFS performs best



A custom-designed maze where DFS finds the solution more effectively than other algorithms due to a depth-oriented path that aligns well with DFS's search pattern as the end path is narrow.

	Steps	Cost	Nodes Generated	Time (ms)	Memory (MB)
BFS	35	215	23332	5369.1	2.49
UCS	35	215	26048	6160.53	1.89
A Star	35	215	7253	5386.91	1.58
DFS	43	223	1918	416.91	0.3

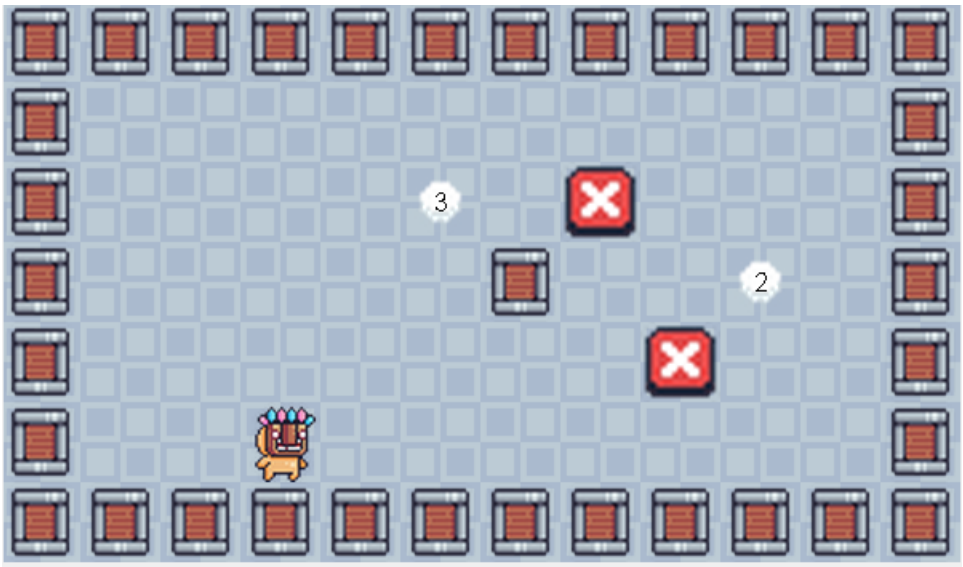
Input 10: Challenging puzzle for human



An intricate and challenging maze intended to test both human problem-solving and algorithm robustness in highly complex scenarios. I can't passed this level, but can the algorithm?

6.1 General Case:

The representative for this case is the input #3. This is a simple maze with very few walls.



The detailed statistics return as follow:

	Steps	Cost	Nodes Generated	Time (ms)	Memory (MB)
BFS	15	25	6469	343.31	2.52
UCS	15	25	9126	487.74	2.62
A Star	15	25	408	52	0.28
DFS	1140	1417	290705	15966.83	48.37

This give us some insight into the general performance of these algorithm:

- A*, BFS, UCS seem to give us the optimal path.
- A* is the fastest and most memory efficient of the 4.
- DFS is horrible.

However, are any of these assumption correct. There is only one way to know, do more testing.

6.2 Optimality

Here, we will try to see whether any of our algorithm A*, BFS, UCS, DFS give us the optimal path. As we know, **A*** give optimal solution only when the heuristics are consistent. In which our heuristics does (see proof above), so it should give the optimal path.

UCS also guarantee optimal solution as the cost of moving one state to another is 1 or larger (which is non-negative).

BFS is not always optimal, however in some case, it give us the best solution with lower time and memory compare to UCS (like in the input #7). This is because it return shortest path (by step) to any goal state, i.e all move is consider to be of weight 1. In some case, this may also the optimal path, but in general, BFS do not return the optimal path.

DFS is worst-optimal in testing. In most of the cases, it not only give us the very costly path, but also took a really long time processing.

Overall, A* and UCS guarantee optimality, while BFS and DFS do not.

6.3 Efficiency

Here, we will try to find out whether our algorithm A*, BFS, UCS, DFS perform efficiently in real cases.

First, we compare 2 optimal algorithms: *A** and *UCS* A* generally have the better performance in both time and memory. The main reason for this is the use of **heuristics** in A* search, which helps guide the search more efficiently towards the goal compared to UCS. In our project, we optimize the heuristics function to make that difference even further.

Next, we consider the last 2 algorithms: *DFS* and *BFS*

From the real results after testing, BFS tends to perform better in both time and space. One of the reason is because we do not have a particular direction for DFS. It always search the most-left of the search tree first (in which the agent go right) and it expand the tree to the very end. So if our solution is on the right side, and the left side of the search tree have many nodes, this algorithm will take a long time wasting generate unnecessary nodes. While in BFS, time and space complexity only depend on how deep the solution is (which side of the search tree also matters, but the difference is not considerable in the same level).