Artificial Intelligence Project Final Report

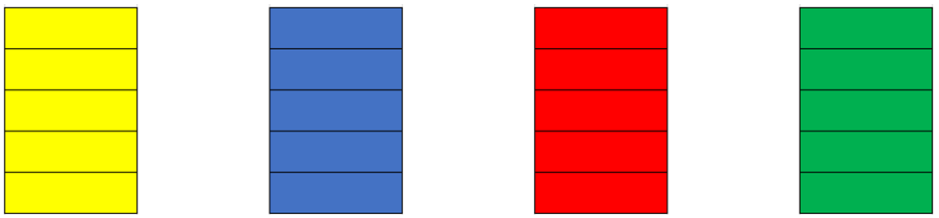
Group Members

Name	Student ID	Email
Dang Kieu Trinh	20214933	trinh.dk214933@sis.hust.edu.vn
Dao Ha Xuan Mai	20210562	mai.dhx210562@sis.hust.edu.vn
Nguyen Ba Duong	20214886	duong.nb214886@sis.hust.edu.vn

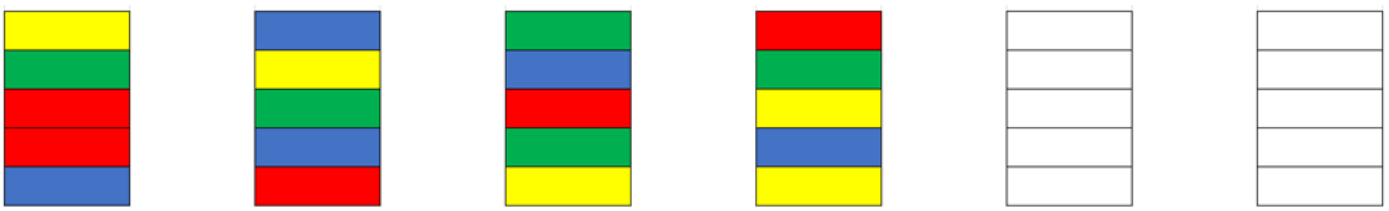
Problem Description

We have B bottles of equal size, each bottle can contain only N equal objects (blocks of water) with the same color and different bottles also differ in the color of their objects. There are exactly N objects have the same color and total of B*N objects.

Example (A) with B = 4 and N = 5:



Then, B*N objects are mixed, and two empty bottles are added.



In this game, there is a simple set of rules:

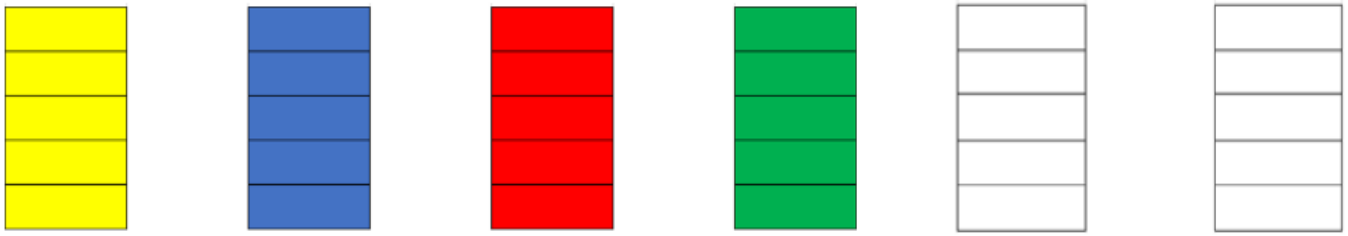
- You can only move the topmost water from one bottle into another.

- Unless the destination bottle is empty, you can only move the water into the bottle that has the topmost water with the same color.

- You can only move an object into another bottle if the destination bottle is not already at its maximum capacity.

Our goal is to have all objects with the same color in one bottle by using AI knowledge.

Goal of example (A):



Problem Solution

Problem Formulation

Initial Stage: B bottles, each containing N objects. Bottles are not uniformly colored.

Goal Stage: All B bottles are uniformly colored.

Actions: Pour up to N objects from one source bottle to one destination bottle with a condition that after the action, the destination bottle contains no more than N objects.

Path Cost: Each step cost 1, hence path cost equals the number of steps in path.

Algorithms

First, we will set up the game. We will input the number of colors and the height of the column with the condition that: The number of colors must be greater than or equal to 2, and to keep the number from being too large so the program doesn't become overloaded, the number of colors must be less than 52. The height of the column must be greater than 2.

Then we will use letters (string) to denote colors. For example, if there are 4 colors in the game, we will denote them with the 4 characters 'a', 'b', 'c', 'd'.

The columns will be stored as an array. In this game, the number of empty bottles is fixed at 2 and is represented by an empty array. Here is an example of a game that has been created with 4 colors and the height of each column is 5:

```
[['b', 'a', 'a', 'b', 'b'],
 ['d', 'a', 'c', 'b', 'd'],
 ['b', 'd', 'c', 'c', 'd'],
 ['c', 'a', 'c', 'a', 'd'],
 [],
 []]
```

BFS (Breadth-First Search)

Pseudo Code:

```
function solvePuzzle(puzzle, bottleHeight=None, visitedPositions=set(), answer=[]):
    if the bottleHeight is not given then
        bottleHeight <- the maximum length of the puzzle
    add the puzzle (in canonical string form) to visitedPositions
    create an empty queue
    for each bottle in the puzzle do
        for each other bottle in the puzzle do
            if the bottles are the same then
                skip this iteration
            if moving water from the current bottle to the candidate bottle is a valid move then
                make a deep copy of the puzzle
                move water from the current bottle to the candidate bottle in the copy
                if the copy puzzle is solved then
                    add the copy puzzle to the answer list
                    return True
                if the copy puzzle (in canonical string form) has not been visited then
                    add the copy puzzle to the queue
                    add the copy puzzle (in canonical string form) to visitedPositions
    while the queue is not empty do
        the current puzzle = the first puzzle in the queue
        remove the first puzzle from the queue
        if the current puzzle is solved then
            add the puzzle to the answer list
            return True
        else do
            try solvePuzzle(with the current puzzle as input)
    return False
```

By using the Breadth-First Search (BFS), we will always find the shortest path to a solution (if it exists, the solution is optimal), sacrificing the time.

The starting pattern is evaluated to find all possible moves, and this forms a queue of next patterns. Then each pattern in the queue is evaluated one-by-one, finding all possible moves that have not already been visited and placing them into the queue. This is repeated until the queue is empty or a solution is found.

DFS (Depth-First Search)

Pseudo Code:

```
function solvePuzzle(puzzle, bottleHeight=None, visitedPositions=set(), answer=[]):
    if bottleHeight is not given then
        bottleHeight <- the maximum length of the puzzle
    add the puzzle (in canonical string form) to visitedPositions
    for each bottle in the puzzle do
        for each other bottle in the puzzle do
            if the bottles are the same then
                skip this iteration
            if moving water from the current bottle to the candidate bottle is a valid move then
                make a deep copy of the puzzle
                move water from the current bottle to the candidate bottle in the copy
                if the copy puzzle is solved then
                    add the copy puzzle to the answer list
                    return True
                if the copy puzzle (in canonical string form) has not been visited then
                    try solvePuzzle(the copy puzzle, bottleHeight, visitedPositions, and answer as input)
                    if the copy puzzle is solved then
                        add the copy puzzle to the answer list
                        return True
    return False
```

Depth-First Search is a search algorithm that explores a tree or graph by traversing it as deeply as possible along each branch before backtracking.

When using the DFS algorithm, we will try to solve the puzzle by making valid moves on the puzzle and then recursively calling itself to solve the resulting puzzle..

In the context of this problem, the main function starts at the root node (the initial stage of the puzzle) and explores as many nodes as possible along each branch before backtracking and exploring other branches.

The function continues this process until it finds a node that represents a solved puzzle and then appends the node (the solved puzzle) to a list of solutions. If the function reaches a dead end (a node with no valid moves), it backtracks to the previous node and continues the search from there.

A*

Pseudo code:

```
function getHeuristic(puzzle, bottleHeight): return (totalWaters - numCorrectWaters)
    totalWaters <- the number of waters expected in the puzzle
    numCorrectWaters <- 0
    for each bottle in the puzzle do
        if the bottle is not empty then
            if the bottle contains only one color then
                numCorrectWaters <- numCorrectWaters + the number of waters in the bottle

function solvePuzzle(puzzle, bottleHeight=None, visitedPositions=set(), answer=[]):
    if bottleHeight is not given then
        bottleHeight <- the maximum length of the puzzle
    add the puzzle (in canonical string form) to visitedPositions
    create a priority queue ordered by getHeuristic()
    for each bottle in puzzle do
        for each other bottle in the puzzle do
            if the bottles are the same then
                skip this iteration
            if moving water from the current bottle to the candidate bottle is a valid move then
                make a deep copy of the puzzle
                move water from the current bottle to the candidate bottle in the copy puzzle
                if the copy puzzle is solved then
                    add the copy puzzle to the answer list
                    return True
                if the copy puzzle (in canonical string form) has not been visited then
                    add the copy puzzle to the priority queue
                    add the copy puzzle (in canonical string form) to visitedPositions

    while the priority queue is not empty:
        get the current puzzle from the priority queue
        try solvePuzzle(with the current puzzle, bottleHeight, visitedPositions, and answer as input)
        if the puzzle is solved then
            add the puzzle to the answer list
            return True
    return False
```

A* search is an algorithm that combines the strengths of BFS with the use of heuristics (estimates of the cost of reaching the goal) to guide the search and achieve faster results.

In the context of this game, the main function (the one that we use to solve the puzzle) starts at the root node (the initial state of the puzzle) and then generates a list of all the valid moves that can be made from this node. For each valid move, it creates copies of the current puzzle and adds them to a priority queue. This priority queue is sorted based on the value of a heuristic function. The heuristic function estimates the number of balls that are not in their correct tubes, with the goal of minimizing this value.

The key advantage of A* search is that it uses heuristics to guide the search towards the most promising areas of the search space, rather than exploring the entire space randomly as in BFS. This allows it to find a solution more efficiently in many cases.

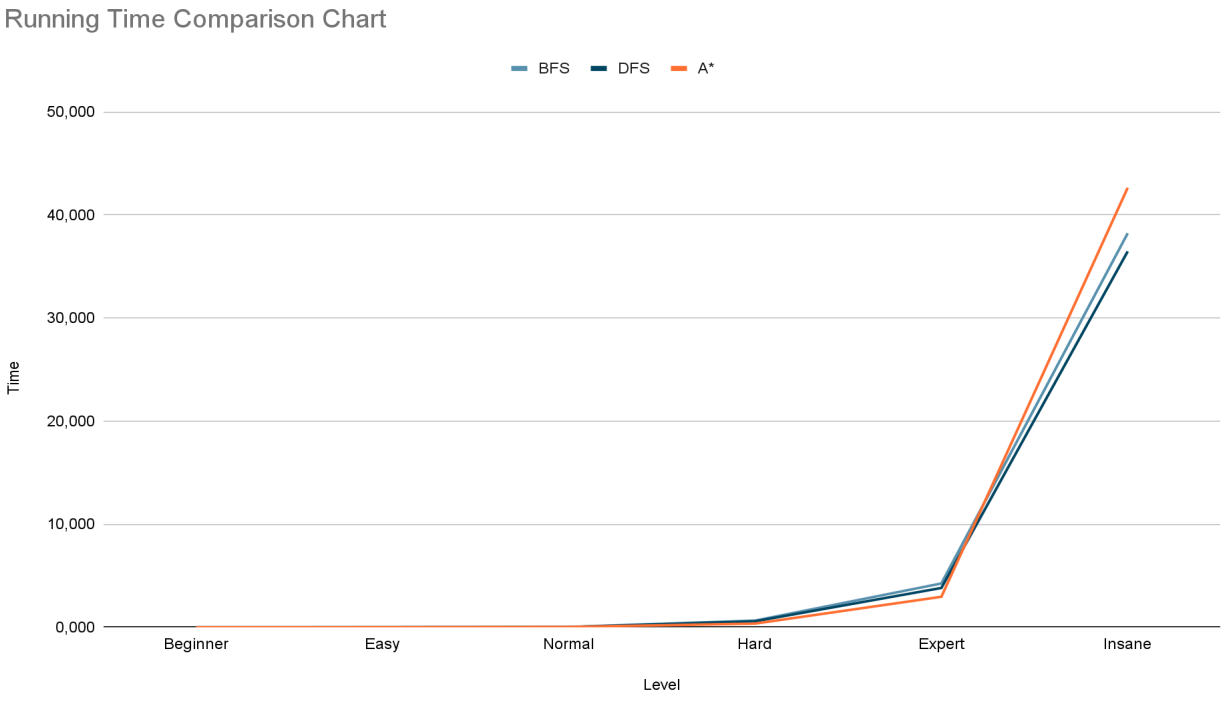
Analyzing Results and Conclusion

To compare the three algorithms, we have designed some game examples with 6 increasing difficulty levels: Beginner, Easy, Normal, Hard, Expert and Insane.

Level	Number of colors	Height of each bottle
Beginner	4	4
Easy	6	6
Normal	8	8
Hard	10	10
Expert	12	12
Insane	14	14

Then we use these examples to measure the running time, memory usage and number of steps to solve the problem obtained from each algorithm, and we have the following 3 comparison charts:

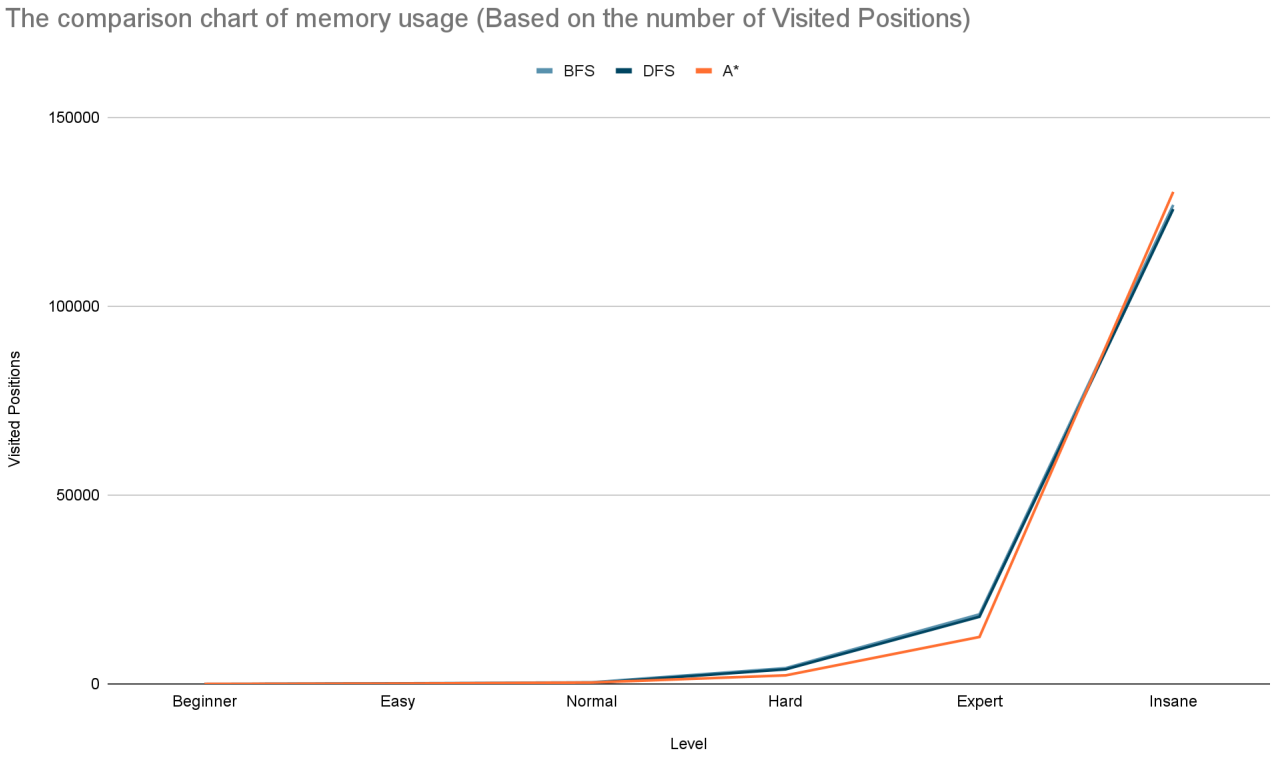
1. **The comparison chart of running time**



- In most cases, DFS is the algorithm with the shortest running time, while BFS is the algorithm with the longest running time. This is because DFS typically has a lower time complexity than BFS, meaning that it is able to solve problems more quickly in the average cases.

- When the problem becomes increasingly complex (such as at the “Insane” level), the A* algorithm takes the longest time. This is because A* is a more complex algorithm than DFS or BFS. It is designed to find optimal solutions by incorporating additional information about the search space. This can make it more efficient than DFS or BFS in most cases, but it can also make it more computationally intensive and slower to execute.

2. The comparison chart of memory usage (Based on Visited Positions)

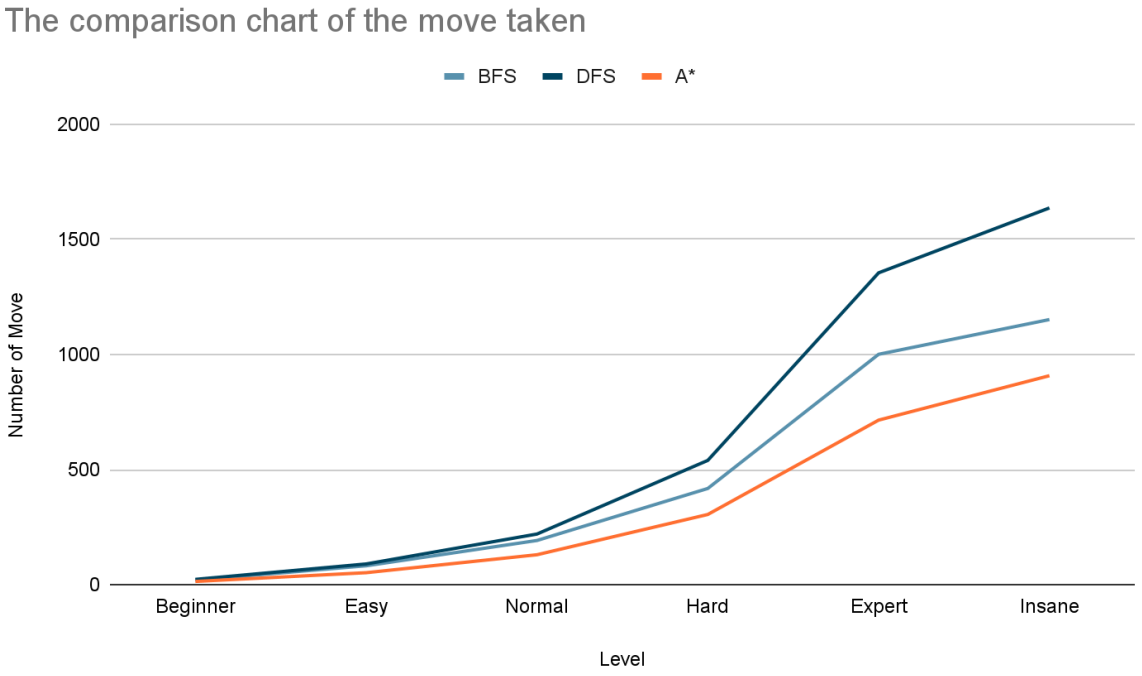


Since visitedPositions is a set that stores information about the puzzle positions that have been visited, it can be used as a factor to measure the memory usage of the function. This can be used to compare the memory usage of each algorithm.

- In most cases, DFS is the algorithm that uses the least memory.

- In almost every case, BFS is the algorithm that uses the most memory out of the three algorithms. However, when the problem becomes more difficult (such as at the "Insane" levels), A* becomes the algorithm that uses the most memory. This is because A* is a more complex algorithm than DFS or BFS, and it requires more memory to store additional information about the search space.

3. The comparison chart of the number of steps



- A* is an algorithm designed to find optimal solutions. As a result, it always finds solutions with the fewest number of steps while DFS results in solutions with the largest number.

Conclusion:

Overall, it depends on which specific characteristics of the problem are being solved. Each algorithm has different strengths and weaknesses.

In general, A* is the ideal algorithm if finding the shortest solution is the most important consideration, especially in the context of this game - all the moves cost the same. When speed and memory usage are higher priorities, DFS may be a more suitable one.

List of references

- Consulting the Solver for Ball Sort Puzzle on GitHub.