Assessment Cover Sheet

This Assessment Cover Sheet is only to be attached to hard copy submission of assessments.

ASSESSMENT DETAILS

Unit title	INTRODUCTION TO ARTIFICIAL INTELLIGENCE	Tutorial /Lab Group	FRI, 10-12AM	Office use only
Unit code	COS30019	Due date	18 OCT 2021	
Name of lecturer/tutor	DR. JOEL THAN MING CHIA			
Assignment title	ASSIGNMENT 01			Faculty or school date stamp

STUDENT(S) DETAILS

	Student Name(s)	Student ID Number(s)
(1)	KATHY WONG HUI YING	101212259
(2)		
(3)		
(4)		
(5)		
(6)		

DECLARATION AND STATEMENT OF AUTHORSHIP

1. I/we have not impersonated, or allowed myself/ourselves to be impersonated by any person for the purposes of this assessment.
2. This assessment is my/our original work and no part of it has been copied from any other source except where due acknowledgement is made.
3. No part of this assessment has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/tutor concerned.
4. I/we have not previously submitted this work for this or any other course/unit.
5. I/we give permission for my/our assessment response to be reproduced, communicated, compared and archived for plagiarism detection, benchmarking or educational purposes.

I/we understand that:

6. Plagiarism is the presentation of the work, idea or creation of another person as though it is your own. It is a form of cheating and is a very serious academic offence that may lead to exclusion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.

Student signature/s

I/we declare that I/we have read and understood the declaration and statement of authorship.

(1)	KATHY W. HY	(4)	
(2)		(5)	
(3)		(6)	

Further information relating to the penalties for plagiarism, which range from a formal caution to expulsion from the University is contained on the Current Students website at **https://www.swinburne.edu.my/current-students/manage-course/exams-results-assessment**

Copies of this form can be downloaded from the Student Forms web page at **https://www.swinburne.edu.my/current-students/manage-course/exams-results-assessment/how-to-submit-work.php**

Table of Contents

Introduction

In this assignment, two puzzles are given. They are to be solved through implementing a python program that will apply two search strategies (uninformed and informed search strategies), in order to obtain the solution.

The main objective of this assignment is to showcase understanding in regards how the applied search strategies work, and their process of obtaining the solution.

Puzzle-1

Task Described

The given task presents three bottles, b1, b2 and b3 of differing capacities. There are no markings on the bottles to measure the water volume inside when not at maximum capacity, so the bottles will not be filled partially from the supply. The objective for this task is to achieve the desired amount of water in the bottles (end goal), as specified by the question.

The bottles can be either filled completely, emptied, or "pour" their contents into another bottle. Any excess will remain in the original bottle (the one being poured).

Assuming an unlimited water supply, the following scenarios are provided for this problem:

<table>
<tr><th colspan="2">Situation-1</th><th colspan="2">Situation-2</th></tr>
<tr><td colspan="2">Maximum bottle capacity:
b1 = 10
b2 = 6
b3 = 5</td><td colspan="2">Maximum bottle capacity:
b1 = 11
b2 = 7
b3 = 4</td></tr>
<tr><td>Start:
b1 = 10
b2 = 0
b3 = 0</td><td>End goal:
b1 = 8
b2 = 3
b3 = 0</td><td>Start:
b1 = 10
b2 = 0
b3 = 0</td><td>End goal:
b1 = 8
b2 = 3
b3 = 0</td></tr>
</table>

Problem Representation

For both situations, each state is to be represented in (b1, b2, b3) form. So, if b1 contains 10 liters, b2 contains 0 liters, and b3 0 liters too, therefore the state is represented by (10, 0, 0).

The maximum capacities of each bottle will be denoted by b1max, b2max and b3max.

Puzzle-1 State Representation			
Note: $0 \leq b1 \leq b1max$ $0 \leq b2 \leq b2max$ $0 \leq b3 \leq b3max$			
Step	Operator	State Condition/ Precondition	Result of Operation/ Effect
1	Empty b1	b1 > 0	(0, b2, b3)
2	Empty b2	b2 > 0	(b1, 0, b3)
3	Empty b3	b3 > 0	(b1, b2, 0)
4	Fill the b1	b1 < b1max	(b1max, b2, b3)
5	Fill the b2	b2 < b2max	(b1, b1max, b3)
6	Fill the b3	b3 < b3max	(b1, b2, b3max)
7	Pour water from b1 into b2 (with excess kept in b1)	0 < (b1 + b2) > b2max & b1 > 0	((b1+b2-b2max), b2max, b3)
8	Pour water from b1 into b3 (with excess kept in b1)	0 < (b1 + b3) > b3max & b1 > 0	((b1+b3-b3max), b2, b3max)
9	Pour water from b2 into b1 bottle (with excess kept in b2)	0 < (b2 + b1) > b1max & b2 > 0	(b1max, (b2+b1-b1max), b3)
10	Pour water from b2 into b3 (with excess kept in b2)	0 < (b2 + b3) > b3max & b2 > 0	(b1, (b2+b3-b3max), b3max)
11	Pour water from b3 into b1 (with excess kept in b3)	0 < (b3 + b1) > b1max & b3 > 0	(b1max, b2, (b3+b1-b1max))
12	Pour water from b3 into b2 (with excess kept in b3)	0 < (b3 + b2) > b2max & b3 > 0	(b1, b2max, (b3+b2-b2max))
13	Pour water from b1 into b2 (no excess in b1)	0 < (b1 + b2) ≤ b2max &	(0, (b1+b2), b3)

		$b1 \geq 0$	
14	Pour water from b1 into b3 (no excess in b1)	$0 < (b1 + b3) \leq b3max$ & $b1 \geq 0$	(b1, 0, (b1+b3))
15	Pour water from b2 into b1 (no excess in b2)	$0 < (b2 + b1) \leq b1max$ & $b2 \geq 0$	((b2+b1), 0, b3)
16	Pour water from b2 into b3 (no excess in b2)	$0 < (b2 + b3) \leq b3max$ & $b3 \geq 0$	(b1, 0, (b2+b3))
17	Pour water from b3 into b1 (no excess in b3)	$0 < (b3 + b1) \leq b1max$ & $b1 \geq 0$	((b3+b1), b2, 0)
18	Pour water from b3 into b2 (no excess in b3 bottle)	$0 < (b3 + b2) \leq b2max$ & $b2 \geq 0$	(b1, (b3+b2), 0)

Puzzle-2

Task Described

The given task presents four entities (man, wolf, goat, cabbages) and a rowboat. The man has to transport the other entities across the river. However, the rowboat only has room for the man and one other entity.

If the man takes the cabbage across the river, the wolf will eat the goat. If the man takes the wolf, the goat will eat the cabbage. If the man is present, the goat and cabbage will be safe from their 'enemy'.

The end goal in this problem is to ensure the safe transport of all entities across the river.

Problem Representation

The following assumptions are made for this solution:

1. The river separates two banks, left and right.
    a. All entities are assumed to initially be situated on the left river bank.

b. The man is expected to transport all entities across the river to the right river bank.

2. The rowboat will not be explicitly shown in the state representation, but is assumed to be the agent for change for the entities being on the left to right bank, or vice versa.
3. The solution fails if the wolf is left alone with the goat, or the goat with the cabbage. Thus, such states will not be further explored.

Each state is represented in (m(X), w(X), g(X), c(X)) form, where m = man, w = wolf, g = goat and c = cabbage. X can either be 'l' or 'r', to indicate the left or right bank position.

One restriction imposed is that not more than two entities can change their bank position for each state change, and these entities must consist of **one man** and **one other entity**.

Problem 2 State Representation			
Initial state defined: (m(l), w(l), g(l), c(l)) Goal state defined: (m(r), w(r), g(r), c(r)) X indicates don't care (can be either l or r)			
Step	Operator	State Condition/ Precondition	Result of Operation/ Effect
1	Only man goes to left	m(r)	(m(l), w(X), g(X), c(X))
2	Only man goes to right	m(l)	(m(r), w(X), g(X), c(X))
3	Man and wolf goes to left	m(r) & w(r)	(m(l), w(l), g(X), c(X))
4	Man and wolf goes to right	m(l) & w(l)	(m(r), w(r), g(X), c(X))
5	Man and goat goes to left	m(r) & g(r)	(m(l), w(X), g(l), c(X))
6	Man and goat goes to right	m(l) & g(l)	(m(r), w(X), g(r), c(X))
7	Man and cabbage goes to left	m(r) & c(r)	(m(l), w(X), g(X), c(l))
8	Man and cabbage goes to right	m(l) & c(l)	(m(r), w(X), g(X), c(r))

**Note: Due to limited knowledge in python, in the source code, the states will be represented as (m, w, g, c) form, where m = man, w = wolf, g = goat and c = cabbage.*

Search Strategies Considered

**Notation for space/time complexity:*

m = maximum branching factor of search tree

d = depth of the solution

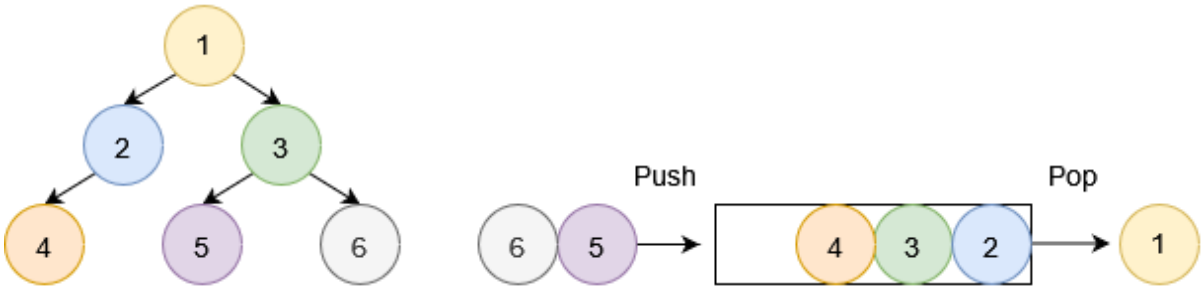
m = maximum depth of state space

Uninformed Search

Breadth-First-Search

In Breadth-First-Search, the root node of the tree is first expanded. Then, the child nodes will be expanded one by one, going across each layer (either left to right or vice versa). In other words, the shallowest unexpanded node is first expanded before proceeding with the next level.

Breadth-First-Search utilizes the Queue for storing visited nodes, operating on a First-In-First-Out (FIFO) basis. Unexplored nodes are pushed into the queue, and will be 'popped' out of it in the same sequence as being pushed in, to be explored (expanded).



Its space and time complexities are both $O(b^d)$. The more possible actions that can be taken, the higher the number of branching factors. Also, the more moves it takes to obtain the solution, there will be a greater depth to the solution. If both these variables are high, the storage required to allocate for the nodes will be high and the search time will be very lengthy.

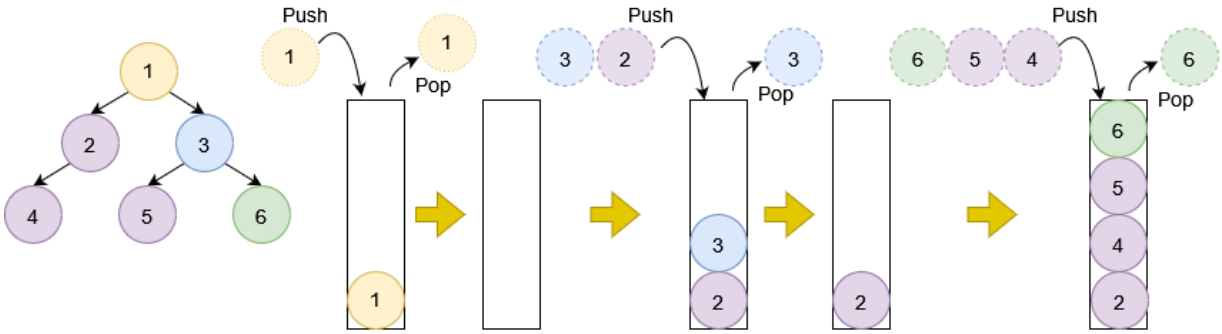
Breath-First Search will work well in cases with low branching factors (not so many possible actions) or with lower solution depth (less moves to obtain solution).

Depth-First-Search

In Depth-First-Search, the root node is first expanded all the way down, or as far as it can go down a given path (branch), until there are no successive nodes. In other words, the deepest

unexpanded node is first expanded. When that branch is fully explored, the search backtracks, then finds another unexplored path to traverse.

Depth-First-Search utilizes the Stack for storing visited nodes, operating on a First-In-Last-Out (FILO) basis. Unexplored nodes are pushed into the stack from one end to another, and the last node to be pushed in will be the first to be popped out (while the very first node to be pushed in will become the last to be popped out).



Its space and time complexities are O(bm) and O(b^m) respectively.

Depth-First-Search only requires enough memory space to save the number of nodes that make up the maximum path (maximum height/depth) of the tree. In cases where the search space is infinite (where there is an infinite set of possible valid moves), Depth-First-Search will be incomplete— meaning, this strategy will fail, as the maximum depth could potentially be infinite.

Informed Search

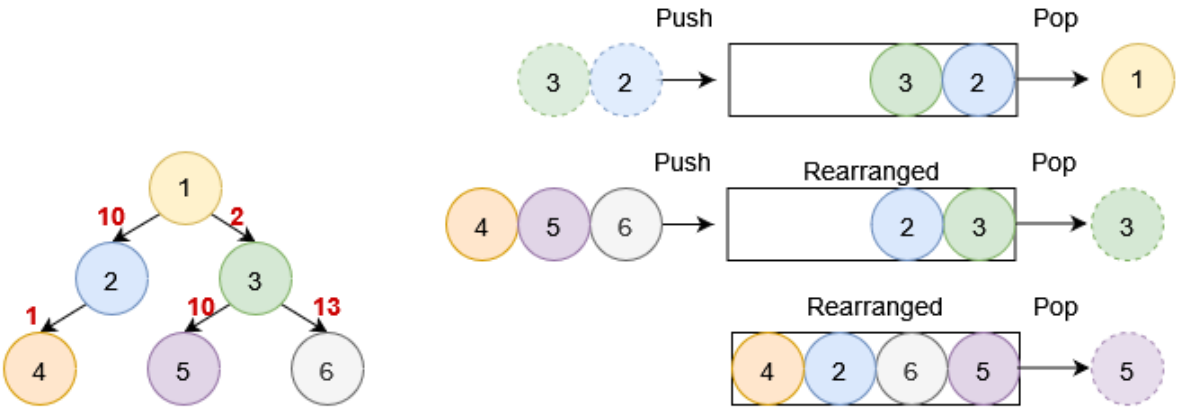
Greedy Best-First search

Greedy-Best-First-Search will always expand the node that appears closest to the goal: it does this by evaluating how far each node is from the goal. The further the node is, the more the path would 'cost'— so Greedy-Best-First-Search will go for the one with the least 'cost'.

It does not revise or retrace its steps, but instead just goes for the next immediate cheap cost. As a result, it is possible for this search to get stuck in loops.

Heuristic used: **f(n) = h(n)**, where h(n) is the estimated cost of path from the state at a certain node to the goal state. The heuristic is thus to choose whichever node has the least cost.

Greedy-Best-First-Search utilizes the Priority Queue for storing visited nodes: Nodes will be pushed into the queue, and then rearranged according to heuristic value. Nodes with the highest priority (in this case, lowest path cost) will be first 'popped' out of the queue to be expanded.



Its space and time complexities are both $O(b^m)$. All nodes are kept in memory as the search has to scour each level of the tree for the node with least path cost before expanding it.

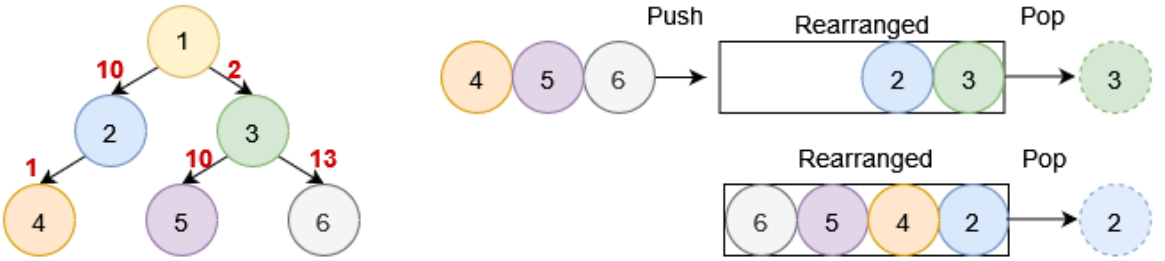
A*-Search

A*-Search takes into account the 'bigger picture': how far the current node is from the start state (root node), and how far it is from the goal state. The further the paths are, the more cost they incur, and A*-Search avoids expanding paths that are costly.

This will be very useful in a big search tree, as the solution can always be found (provided that there is a solution in a first place) as long as it is optimal. A*-Searches are optimal only when its heuristics are admissible: meaning, they do not overestimate a certain value, and can be accepted.

Heuristic used: **f(n) = h(n) + g(n)**, where h(n) is the estimated cost of path from the state at a certain node to the goal state, and g(n) is the estimated cost from the start state to reach the current node. The heuristic is thus to choose whichever node has the least total cost down the tree nodes. It can also evaluate if the current path really has the cheapest cost to the goal, or if other nodes will give better options.

Similar to Greedy Best-First Search, A*-Search also utilizes the Priority Queue for storing visited nodes: Nodes are pushed into the queue, and then rearranged according to heuristic value and popped out accordingly. In this case, the nodes with the highest priority are those that incur the lowest total path cost.



Its space complexity is $O(b^m)$. The time complexity increases exponentially with the size of the search tree.

Search Strategies Chosen (Comparison and Why)

For Uninformed Search, Depth-First-Search was chosen:

In both Puzzles 1 and 2, the search space is finite: in Puzzle-1, the bottles have limited capacity, thus there is a fixed amount of volume each can hold. In Puzzle-2, there are only two positions: left, or right.

Both Puzzles require some form of solving, to obtain the goal state. There is a result that needs to be evaluated upon reaching each node in the tree. The further down the nodes of the search tree go, the more 'moves' are made to solve the puzzle, the higher the chance a state would resemble the goal state.

Breadth-First-Search is not optimal in this situation as it goes horizontally across the nodes, exploring each possible state caused by only one move. Based on both space and time complexity, the memory required to store the nodes and time duration to reach the goal state would be very large.

Therefore Depth-First-Search is chosen, as it explores the nodes path by path (by the sequence of actions) and is more likely to find the goal state in a shorter time.

For Informed Search, A*-Search was chosen:

Using the analogy of drivers on long road trips, Greedy-Best-First-Search is the driver who prefers driving short distances between cities to reach the final destination, even though the cumulative distance is not actually the shortest route.

This is very notable in Puzzle-1, where, given the goal state (8, 3, 0), a node with the state (8, 0, 0) will be considered to have a cheap path to reach the goal (as it's very close to the goal state). However, due to the puzzle rules, going from (8, 0, 0) to (8, 3, 0) is not straightforward—there will be other actions involved to actually achieve (8, 3, 0).

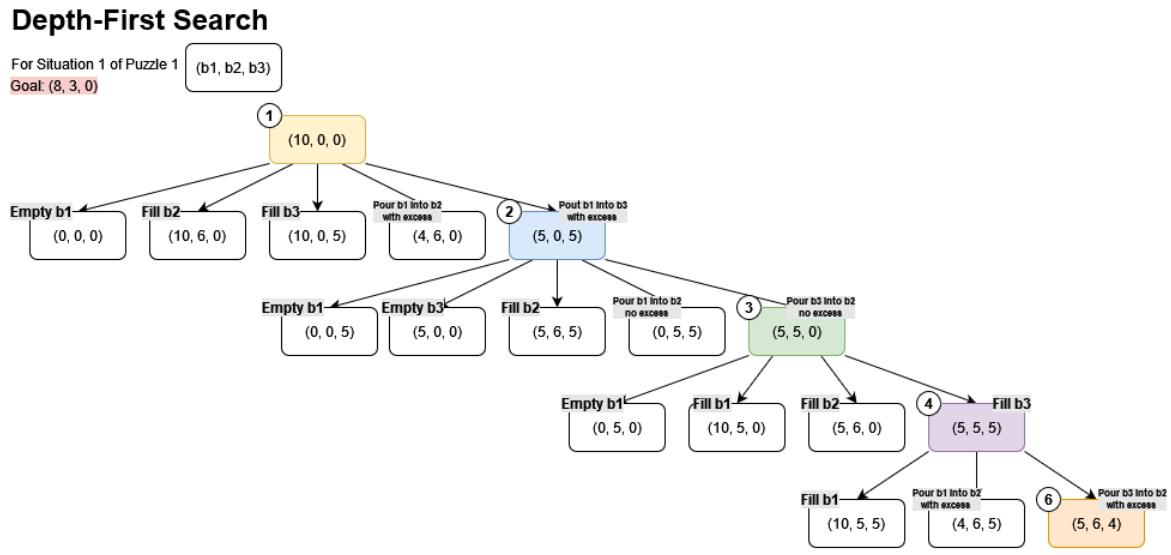
On the other hand, A*-Search is the driver who would consider how far to drive in total to reach the final destination. This driver would not mind driving the occasional long distance between cities if it meant the overall distance actually bears the shortest route.

A*-Search will ensure the path found is optimal, and always finds the solution (provided there is one). It also overcomes Greedy Best-First Search's weakness of getting stuck in loops. Therefore, A*-Search is chosen, as it meets the requirement of implementing a search strategy to find the solution with the fewest steps

Search Trees

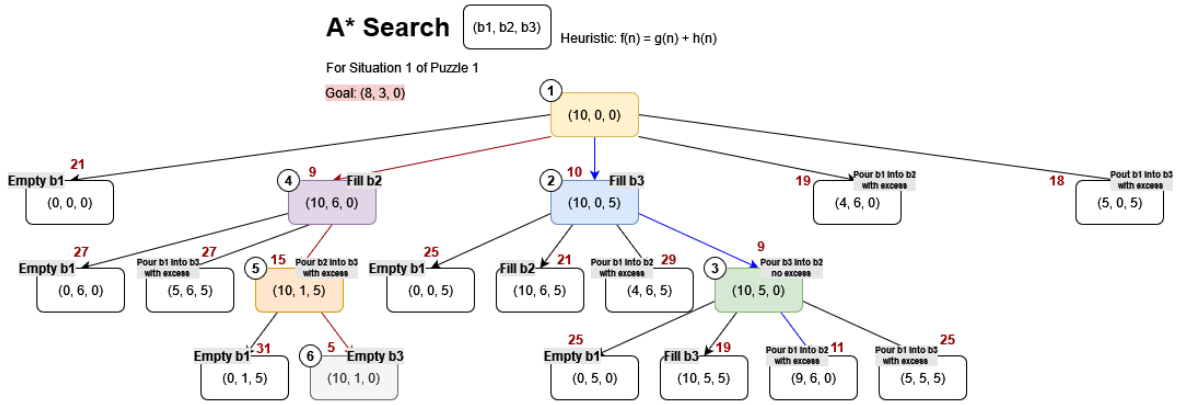
Puzzle-1

Depth-First-Search (Uninformed)



Here, it is observed the Depth-First-Search strategy takes the right-most unexplored nodes, and keeps expanding them.

A*-Search (Informed)



Heuristic applied: f(n) = g(n) + h(n)

How far we are from the start, g(n) + how far we are from the goal, h(n).

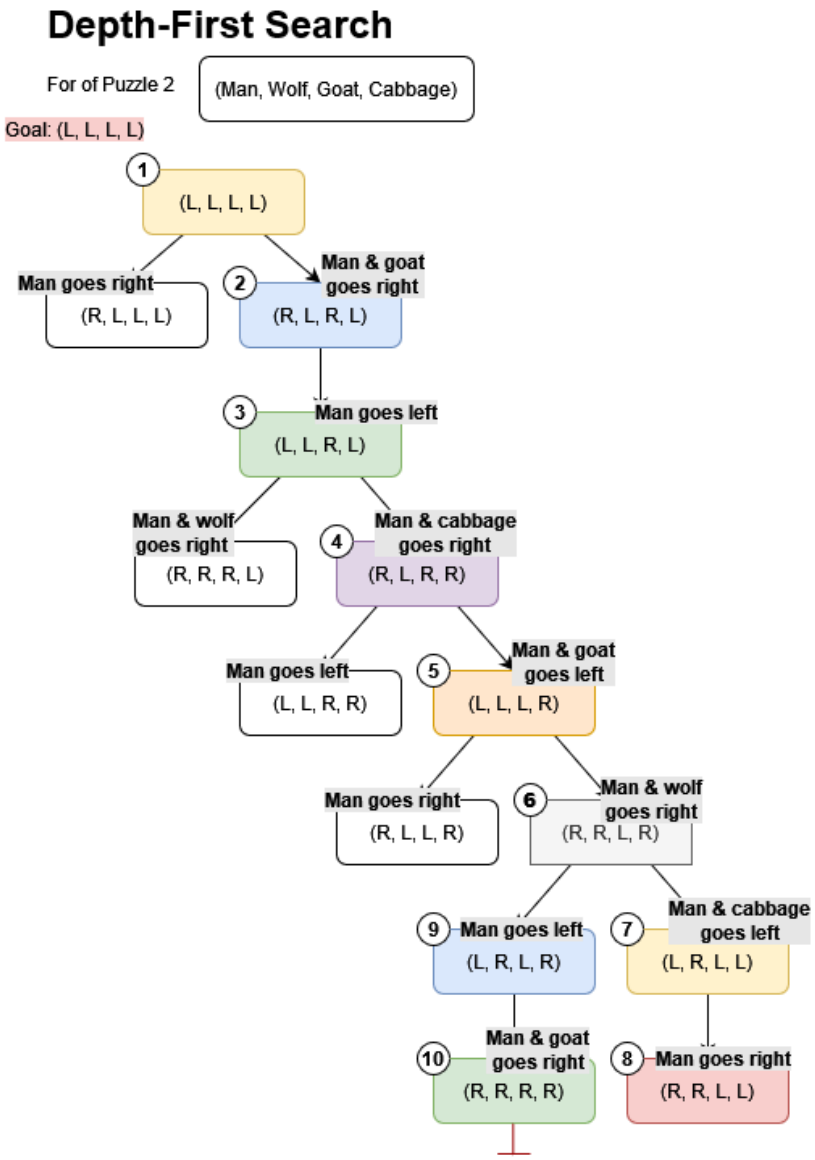
A*-Search chooses which nodes to expand based on the costs calculated by the heuristic. The path cost of each node is numbered in red.

The search initially opts for path 1->2->3, choosing the cheapest total path (10+9=19). However, upon expanding to the next layer, the search again evaluates all the nodes. It finds there is a better path with a less total cost: path 4->5->6 (9+15+5=29), and thus chooses that.

Puzzle-2

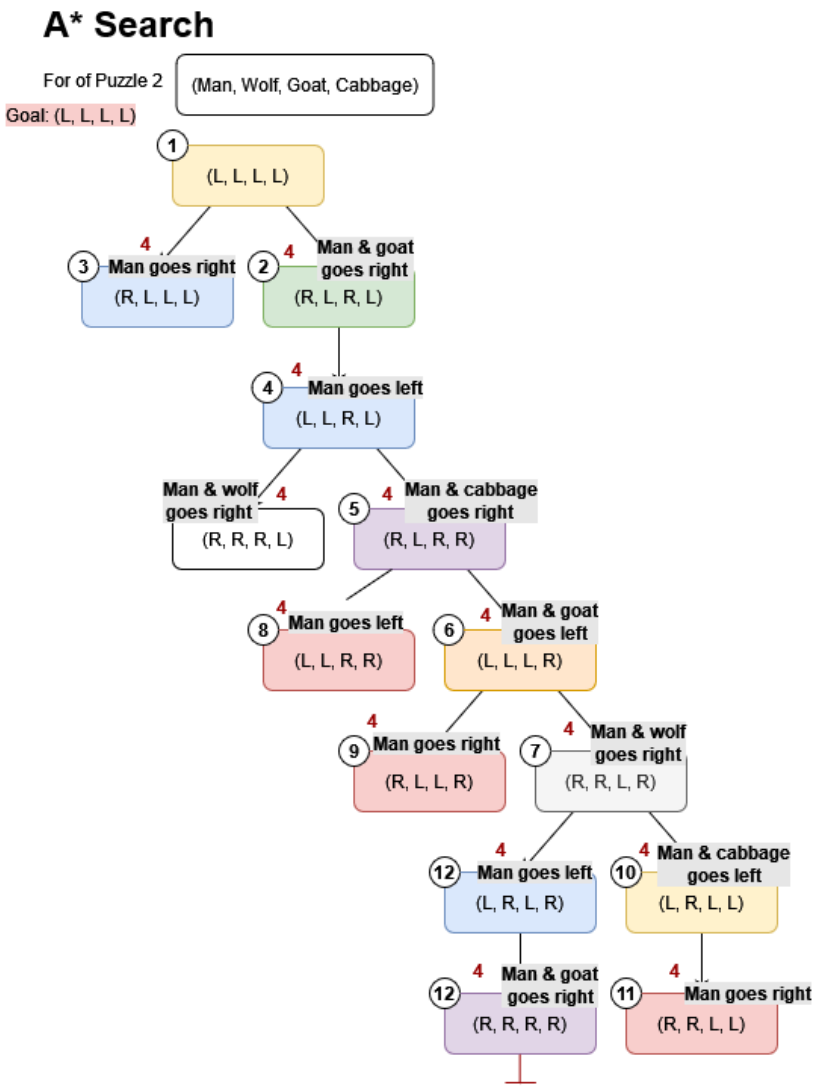
(The nodes colored in red indicate states that 'break' the rules of the puzzle. They will not be further expanded).

Depth-First-Search (Uninformed)



Here, it is observed the Depth-First-Search strategy takes the right-most unexplored nodes, and keeps expanding them. When the end of one path is reached, but the solution is yet to be found, Depth-First-Search will then proceed to backtrack upwards to the next adjacent undiscovered branch and starts expanding downwards from there (as seen reaching node 8, then backtracking to node 9).

A*-Search (Informed)



Heuristic applied: f(n) = g(n) + h(n)

How far we are from the start, g(n) + how far we are from the goal, h(n).

However, as observed from the diagram above, all the nodes cost the same. (This is likely due to them method used for calculating the cost for the heuristic: since there are only two possible states for each entity, left or right, and eventually at all times, the total cost for how far the node is from the start and how far it is from the Goal will always add up to 4).

The node at 2 is not expanded due to repeated states. At node 6, the search strategy backtracks in an attempt to expand a solution from other nodes in order to reduce the total path cost. However, because the chosen nodes break the rules of the puzzle, the search is forced to again go downwards in order to find the goal state.

Observation and Discussion

Based on observation, the number of nodes explored for each puzzle before reaching goal state are as shown below:

Puzzle		**Depth-First-Search Nodes Expanded**	**A*-Search Nodes Expanded**
Puzzle-1	Situation-1	19	39
	Situation-2	39	13
Puzzle-2		8 + 1 = 0 (1 node is not further expanded due to breaking puzzle rules).	8 + 3 = 12 (1 node not further expanded due to repeated state, the other 2 nodes due to breaking puzzle rules).

Assuming that as the more nodes are explored, the more time is taken to carry out the search: it is deduced that the time taken for Depth-First-Search is less than that for A*-Search, since there is less backtracking and considering which has the cheapest path cost. So, based on the results for Puzzle-1 Situation-1, Depth-First-Search is quicker to obtain the goal state than A*-Search. This correlates with the time complexity for each of the search strategies, which are $O(b^m)$ and exponential depending on heuristic, respectively.

However, in Puzzle-1 Situation-2, where the puzzle becomes more complex (different maximum capacities), Depth-First-Search takes longer than A*-Search. This process that A*-Search is more useful in a bigger search tree, for a puzzle with more moves.

By mapping out the search tree based on the source code output, it will be possible to see whether the number of steps taken to reach goal step is more than in Depth-First-Search than A*-Search (or equivalent). By theory, this should be true.

In both puzzles, for uninformed search, the choice of state representation will affect the efficiency of the problem solving, as each expanded node generated is following the arrangement of operators being defined in the state representation table.

To every solvable puzzle, there is a specific sequence of steps (moves) to be carried out in order to obtain the solution. It is possible that the possible actions are arranged in such a way to allow the search to locate the solution much quicker, as indicated how the goal state was easily reached after expanding just 19 nodes for Situation-1 of Puzzle-1, but 39 nodes for Situation-2. This can help with uninformed searches, but likely bear no effect in informed searches.

Further Discussion

If Puzzle-1 becomes a four-bottle problem

For a three-bottle problem, the maximum number of child nodes possibly produced from a single parent node is 18. If another bottle is added, the maximum would increase to 32. This results in an increased branching factor, as each node can expand into a maximum of 32 more nodes. Also, the maximum depth of the search tree would increase as the goal state is likely to be found much deeper down the search tree.

For a Depth-First-Search of O(bm) and O(b^m) space and time complexity respectively, the memory space required to store the nodes would be increased, and so would the time taken to reach the goal state. The same also applies for A*-Search, where both the space and time complexity would increase. However, A* might show a better performance compared to Depth-First-Search, in this increased puzzle complexity, due to its nature.

If Puzzle-2 has another item added to the list

For a Puzzle-2, the maximum number of child nodes possibly produced from a single parent node is 7 (as there is always one entity on one side of the river). If another item is added to the list (for example, a dog, assuming it is an entity that can only be brought across the river by the farmer), the maximum number of expanded nodes would increase to 9. It is likely there would not be much increase in regards to the branching factor and maximum depth of the search tree.

Thus, for both the uninformed and informed search strategies applied, there should likely not be a significant increase in the memory storage or time taken to locate the goal state.

Conclusion

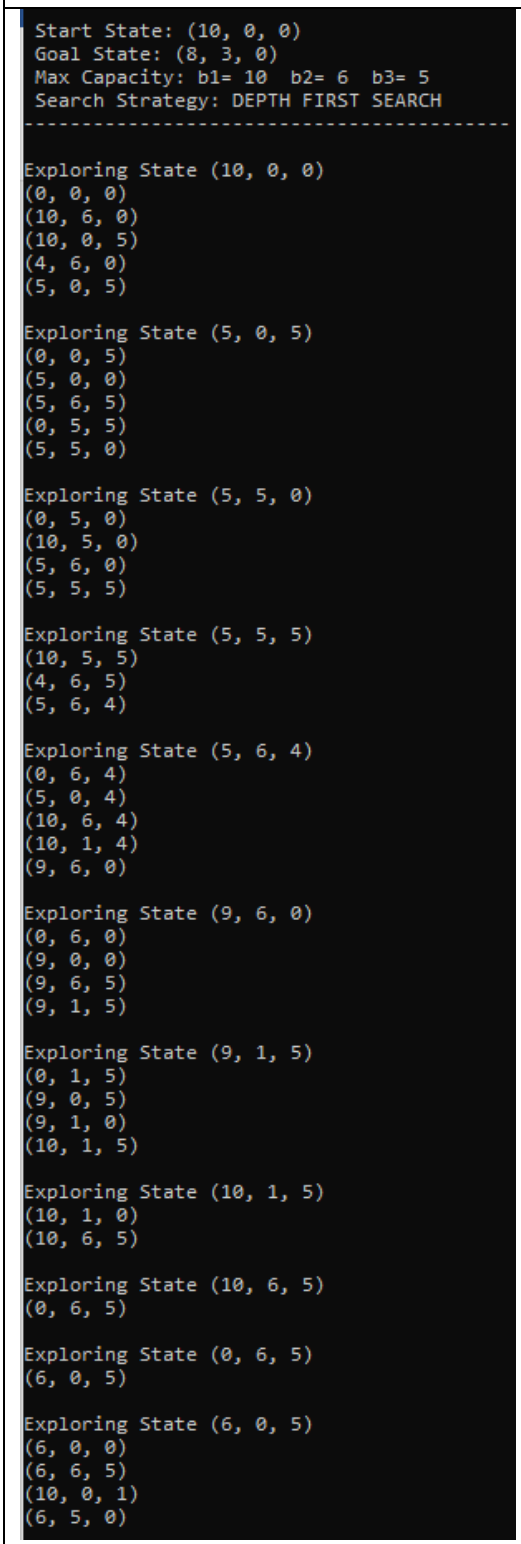
In conclusion, from this report it is observed that: Depth-First-Search is useful in small but dense search trees (or for relatively simple puzzles), but A*-Search is more efficient in complex puzzles compared to simpler ones. The state representation also can affect the performance of Depth-First-Search in reaching the goal state efficiently.

References

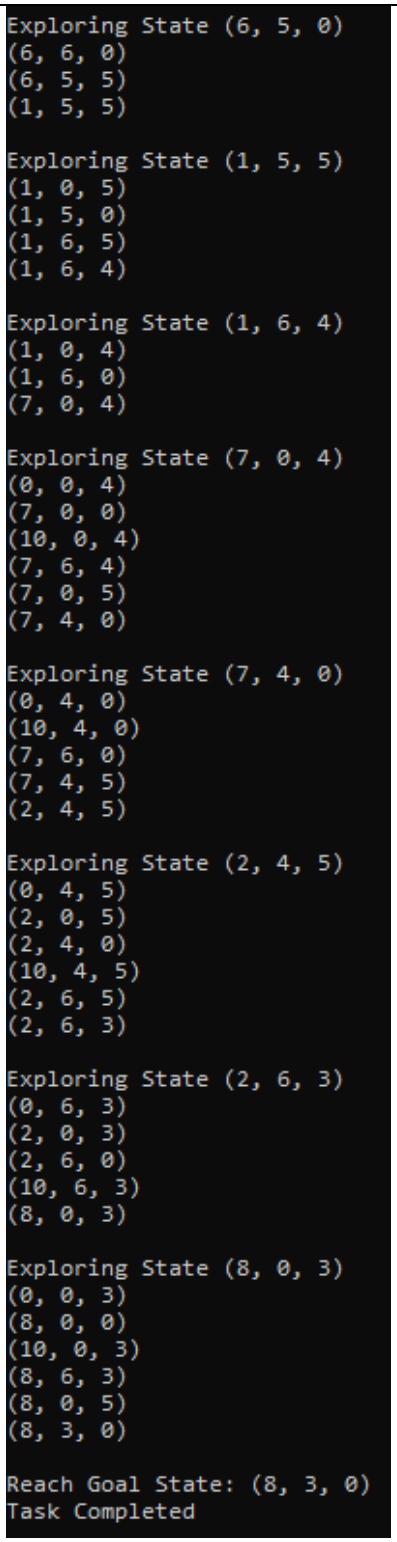
1. BRILLIANT, 2021, Depth-First Search (DFS), viewed 17 October 2021, <https://brilliant.org/wiki/depth-first-search-dfs/>.
2. TechVidvan, 2021, Heuristic Search Techniques in Artificial Intelligence, viewed 17 October 2021, <https://techvidvan.com/tutorials/ai-heuristic-search/>.
3. Gurasis, S 2019, Algorithms: Breadth-First Search vs. Depth-First Search, viewed 17 October 2021, <https://betterprogramming.pub/algorithms-searching-through-a-tree-33610e4577bd>.

Program Output Screenshots

Puzzle 1 Situation 1 [Depth First Search]



```
 Start State: (10, 0, 0)
 Goal State: (8, 3, 0)
 Max Capacity: b1= 10  b2= 6  b3= 5
 Search Strategy: DEPTH FIRST SEARCH
----------------------------------------

Exploring State (10, 0, 0)
(0, 0, 0)
(10, 6, 0)
(10, 0, 5)
(4, 6, 0)
(5, 0, 5)

Exploring State (5, 0, 5)
(0, 0, 5)
(5, 0, 0)
(5, 6, 5)
(0, 5, 5)
(5, 5, 0)

Exploring State (5, 5, 0)
(0, 5, 0)
(10, 5, 0)
(5, 6, 0)
(5, 5, 5)

Exploring State (5, 5, 5)
(10, 5, 5)
(4, 6, 5)
(5, 6, 4)

Exploring State (5, 6, 4)
(0, 6, 4)
(5, 0, 4)
(10, 6, 4)
(10, 1, 4)
(9, 6, 0)

Exploring State (9, 6, 0)
(0, 6, 0)
(9, 0, 0)
(9, 6, 5)
(9, 1, 5)

Exploring State (9, 1, 5)
(0, 1, 5)
(9, 0, 5)
(9, 1, 0)
(10, 1, 5)

Exploring State (10, 1, 5)
(10, 1, 0)
(10, 6, 5)

Exploring State (10, 6, 5)
(0, 6, 5)

Exploring State (0, 6, 5)
(6, 0, 5)

Exploring State (6, 0, 5)
(6, 0, 0)
(6, 6, 5)
(10, 0, 1)
(6, 5, 0)
```



```
Exploring State (6, 5, 0)
(6, 6, 0)
(6, 5, 5)
(1, 5, 5)

Exploring State (1, 5, 5)
(1, 0, 5)
(1, 5, 0)
(1, 6, 5)
(1, 6, 4)

Exploring State (1, 6, 4)
(1, 0, 4)
(1, 6, 0)
(7, 0, 4)

Exploring State (7, 0, 4)
(0, 0, 4)
(7, 0, 0)
(10, 0, 4)
(7, 6, 4)
(7, 0, 5)
(7, 4, 0)

Exploring State (7, 4, 0)
(0, 4, 0)
(10, 4, 0)
(7, 6, 0)
(7, 4, 5)
(2, 4, 5)

Exploring State (2, 4, 5)
(0, 4, 5)
(2, 0, 5)
(2, 4, 0)
(10, 4, 5)
(2, 6, 5)
(2, 6, 3)

Exploring State (2, 6, 3)
(0, 6, 3)
(2, 0, 3)
(2, 6, 0)
(10, 6, 3)
(8, 0, 3)

Exploring State (8, 0, 3)
(0, 0, 3)
(8, 0, 0)
(10, 0, 3)
(8, 6, 3)
(8, 0, 5)
(8, 3, 0)

Reach Goal State: (8, 3, 0)
Task Completed
```

Puzzle 1 Situation 2 [Depth First Search]

```
 Start State: (10, 0, 0)
 Goal State: (8, 3, 0)
 Max Capacity: b1= 11  b2= 7  b3= 4
 Search Strategy: DEPTH FIRST SEARCH
----------------------------------------

Exploring State (10, 0, 0)
(0, 0, 0)
(11, 0, 0)
(10, 7, 0)
(10, 0, 4)
(3, 7, 0)
(6, 0, 4)

Exploring State (6, 0, 4)
(0, 0, 4)
(6, 0, 0)
(11, 0, 4)
(6, 7, 4)
(0, 6, 4)
(6, 4, 0)

Exploring State (6, 4, 0)
(0, 4, 0)
(11, 4, 0)
(6, 7, 0)
(6, 4, 4)
(2, 4, 4)

Exploring State (2, 4, 4)
(0, 4, 4)
(2, 0, 4)
(2, 4, 0)
(11, 4, 4)
(2, 7, 4)
(2, 7, 1)

Exploring State (2, 7, 1)
(0, 7, 1)
(2, 0, 1)
(2, 7, 0)
(11, 7, 1)
(2, 0, 3)
(9, 0, 1)

Exploring State (9, 0, 1)
(0, 0, 1)
(9, 0, 0)
(11, 0, 1)
(9, 7, 1)
(9, 0, 4)
(9, 1, 0)

Exploring State (9, 1, 0)
(0, 1, 0)
(11, 1, 0)
(9, 7, 0)
(9, 1, 4)
(5, 1, 4)

Exploring State (5, 1, 4)
(0, 1, 4)
(5, 0, 4)
(5, 1, 0)
(11, 1, 4)
(5, 7, 4)
(5, 5, 0)
```

```
Exploring State (5, 5, 0)
(0, 5, 0)
(5, 0, 0)
(11, 5, 0)
(5, 7, 0)
(5, 5, 4)
(1, 5, 4)

Exploring State (1, 5, 4)
(0, 5, 4)
(1, 0, 4)
(1, 5, 0)
(11, 5, 4)
(1, 7, 4)
(1, 7, 2)

Exploring State (1, 7, 2)
(0, 7, 2)
(1, 0, 2)
(1, 7, 0)
(11, 7, 2)
(1, 0, 3)
(8, 0, 2)

Exploring State (8, 0, 2)
(0, 0, 2)
(8, 0, 0)
(11, 0, 2)
(8, 7, 2)
(8, 0, 4)
(8, 2, 0)

Exploring State (8, 2, 0)
(0, 2, 0)
(11, 2, 0)
(8, 7, 0)
(8, 2, 4)
(4, 2, 4)

Exploring State (4, 2, 4)
(0, 2, 4)
(4, 0, 4)
(4, 2, 0)
(11, 2, 4)
(4, 7, 4)
(4, 6, 0)

Exploring State (4, 6, 0)
(0, 6, 0)
(4, 0, 0)
(11, 6, 0)
(4, 7, 0)
(4, 6, 4)

Exploring State (4, 6, 4)
(11, 6, 4)
(3, 7, 4)
(4, 7, 3)
(8, 6, 0)

Exploring State (8, 6, 0)
(8, 6, 4)
(7, 7, 0)
(11, 3, 0)
```

```
Exploring State (11, 3, 0)
(0, 3, 0)
(11, 7, 0)
(11, 3, 4)
(7, 3, 4)
(11, 0, 3)

Exploring State (11, 0, 3)
(0, 0, 3)
(11, 7, 3)

Exploring State (11, 7, 3)
(0, 7, 3)
(11, 7, 4)
(10, 7, 4)

Exploring State (10, 7, 4)
(0, 7, 4)

Exploring State (0, 7, 4)
(0, 7, 0)
(7, 0, 4)

Exploring State (7, 0, 4)
(7, 0, 0)
(7, 7, 4)
(7, 4, 0)

Exploring State (7, 4, 0)
(7, 4, 4)
(3, 4, 4)

Exploring State (3, 4, 4)
(3, 0, 4)
(3, 4, 0)
(3, 7, 1)

Exploring State (3, 7, 1)
(3, 0, 1)
(10, 0, 1)

Exploring State (10, 0, 1)
(10, 7, 1)
(10, 1, 0)

Exploring State (10, 1, 0)
(10, 1, 4)
(6, 1, 4)

Exploring State (6, 1, 4)
(6, 1, 0)
(6, 5, 0)

Exploring State (6, 5, 0)
(6, 5, 4)
(2, 5, 4)

Exploring State (2, 5, 4)
(2, 5, 0)
(2, 7, 2)

Exploring State (2, 7, 2)
(2, 0, 2)
(9, 0, 2)

Exploring State (9, 0, 2)
(9, 7, 2)
(9, 2, 0)
```

```
Exploring State (9, 2, 0)
(9, 2, 4)
(5, 2, 4)

Exploring State (5, 2, 4)
(5, 2, 0)
(5, 6, 0)

Exploring State (5, 6, 0)
(5, 6, 4)
(1, 6, 4)

Exploring State (1, 6, 4)
(1, 6, 0)
(1, 7, 3)

Exploring State (1, 7, 3)
(8, 0, 3)

Exploring State (8, 0, 3)
(8, 7, 3)
(8, 3, 0)

Reach Goal State: (8, 3, 0)
Task Completed
```

Puzzle 1 Situation 1 [A* Search]

```
Start State: (10, 0, 0)
Goal State: (8, 3, 0)
Max Capacity: b1= 10  b2= 6  b3= 5
Search Strategy: A* SEARCH
----------------------------------------

Exploring State (10, 0, 0)
(0, 0, 0)
(10, 6, 0)
(10, 0, 5)
(4, 6, 0)
(5, 0, 5)

Exploring State (10, 6, 0)
(0, 6, 0)
(10, 6, 5)
(5, 6, 5)
(10, 1, 5)

Exploring State (10, 0, 5)
(0, 0, 5)
(4, 6, 5)
(10, 5, 0)

Exploring State (10, 5, 0)
(0, 5, 0)
(10, 5, 5)
(9, 6, 0)
(5, 5, 5)

Exploring State (9, 6, 0)
(9, 0, 0)
(9, 6, 5)
(9, 1, 5)

Exploring State (9, 0, 0)
(9, 0, 5)
(3, 6, 0)
(4, 0, 5)

Exploring State (9, 0, 5)
(3, 6, 5)
(10, 0, 4)
(9, 5, 0)

Exploring State (9, 5, 0)
(9, 5, 5)
(8, 6, 0)
(4, 5, 5)
(10, 4, 0)

Exploring State (10, 4, 0)
(0, 4, 0)
(10, 4, 5)
(5, 4, 5)

Exploring State (8, 6, 0)
(8, 0, 0)
(8, 6, 5)
(8, 1, 5)
```

```
Exploring State (8, 6, 0)
(8, 0, 0)
(8, 6, 5)
(8, 1, 5)

Exploring State (8, 0, 0)
(8, 0, 5)
(2, 6, 0)
(3, 0, 5)

Exploring State (10, 0, 4)
(0, 0, 4)
(10, 6, 4)
(4, 6, 4)

Exploring State (8, 0, 5)
(2, 6, 5)
(10, 0, 3)
(8, 5, 0)

Exploring State (8, 5, 0)
(8, 5, 5)
(7, 6, 0)
(3, 5, 5)
(10, 3, 0)

Exploring State (10, 3, 0)
(0, 3, 0)
(10, 3, 5)
(5, 3, 5)

Exploring State (10, 0, 3)
(0, 0, 3)
(10, 6, 3)
(4, 6, 3)

Exploring State (7, 6, 0)
(7, 0, 0)
(7, 6, 5)
(7, 1, 5)

Exploring State (7, 0, 0)
(7, 0, 5)
(1, 6, 0)
(2, 0, 5)

Exploring State (8, 1, 5)
(0, 1, 5)
(8, 1, 0)
(10, 1, 3)

Exploring State (8, 1, 0)
(0, 1, 0)
(10, 1, 0)
(3, 1, 5)
(8, 0, 1)

Exploring State (10, 1, 0)
(5, 6, 0)
(5, 1, 5)
(10, 0, 1)

Exploring State (8, 0, 1)
(0, 0, 1)
(8, 6, 1)
(2, 6, 1)
```

```
Exploring State (10, 0, 1)
(10, 6, 1)
(4, 6, 1)
(6, 0, 5)

Exploring State (10, 1, 3)
(0, 1, 3)
(5, 6, 3)

Exploring State (8, 6, 1)
(0, 6, 1)
(10, 4, 1)
(8, 2, 5)

Exploring State (10, 4, 1)
(0, 4, 1)
(6, 4, 5)

Exploring State (10, 6, 1)
(6, 6, 5)
(10, 2, 5)

Exploring State (8, 2, 5)
(0, 2, 5)
(8, 2, 0)
(10, 2, 3)

Exploring State (8, 2, 0)
(0, 2, 0)
(10, 2, 0)
(3, 2, 5)
(8, 0, 2)

Exploring State (10, 2, 0)
(6, 6, 0)
(5, 2, 5)
(10, 0, 2)

Exploring State (8, 0, 2)
(0, 0, 2)
(8, 6, 2)
(2, 6, 2)

Exploring State (10, 0, 2)
(10, 6, 2)
(4, 6, 2)

Exploring State (10, 2, 3)
(0, 2, 3)
(6, 6, 3)

Exploring State (6, 6, 0)
(6, 0, 0)
(1, 6, 5)
(6, 1, 5)

Exploring State (6, 0, 0)
(1, 0, 5)

Exploring State (8, 6, 2)
(0, 6, 2)
(10, 4, 2)
(8, 3, 5)
```

```
Exploring State (10, 4, 2)
(0, 4, 2)
(7, 4, 5)

Exploring State (8, 3, 5)
(0, 3, 5)
(8, 3, 0)

Reach Goal State: (8, 3, 0)
Task Completed
```

Puzzle 1 Situation 2 [A* Search]

```
 Start State: (10, 0, 0)
 Goal State: (8, 3, 0)
 Max Capacity: b1= 11  b2= 7  b3= 4
 Search Strategy: A* SEARCH
-----------------------------------------

Exploring State (10, 0, 0)
(0, 0, 0)
(11, 0, 0)
(10, 7, 0)
(10, 0, 4)
(3, 7, 0)
(6, 0, 4)

Exploring State (11, 0, 0)
(11, 7, 0)
(11, 0, 4)
(4, 7, 0)
(7, 0, 4)

Exploring State (10, 0, 4)
(0, 0, 4)
(10, 7, 4)
(3, 7, 4)
(11, 0, 3)
(10, 4, 0)

Exploring State (10, 4, 0)
(0, 4, 0)
(11, 4, 0)
(10, 4, 4)
(7, 7, 0)
(6, 4, 4)
(11, 3, 0)

Exploring State (11, 3, 0)
(0, 3, 0)
(11, 3, 4)
(7, 3, 4)

Exploring State (11, 4, 0)
(11, 4, 4)
(8, 7, 0)
(7, 4, 4)

Exploring State (8, 7, 0)
(0, 7, 0)
(8, 0, 0)
(8, 7, 4)
(4, 7, 4)
(8, 3, 4)

Exploring State (8, 0, 0)
(8, 0, 4)
(1, 7, 0)
(4, 0, 4)

Exploring State (8, 0, 4)
(1, 7, 4)
(11, 0, 1)
(8, 4, 0)

Exploring State (8, 4, 0)
(8, 4, 4)
(5, 7, 0)
(4, 4, 4)
(11, 1, 0)
```

```
Exploring State (8, 4, 0)
(8, 4, 4)
(5, 7, 0)
(4, 4, 4)
(11, 1, 0)

Exploring State (11, 1, 0)
(0, 1, 0)
(11, 1, 4)
(7, 1, 4)

Exploring State (11, 0, 1)
(0, 0, 1)
(11, 7, 1)
(4, 7, 1)

Exploring State (8, 3, 4)
(0, 3, 4)
(8, 3, 0)

Reach Goal State: (8, 3, 0)
Task Completed
```

Puzzle 2 [Depth First Search]

```
 Start State: ('L', 'L', 'L', 'L')
 Goal State: ('R', 'R', 'R', 'R')
 Search Strategy: DEPTH FIRST SEARCH
----------------------------------------

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'L', 'L')
('R', 'L', 'L', 'L')
('R', 'L', 'R', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'R', 'L')
('L', 'L', 'R', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'R', 'L')
('R', 'R', 'R', 'L')
('R', 'L', 'R', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'R', 'R')
('L', 'L', 'R', 'R')
('L', 'L', 'L', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'L', 'R')
('R', 'L', 'L', 'R')
('R', 'R', 'L', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'R', 'L', 'R')
('L', 'R', 'L', 'R')
('L', 'R', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'R', 'L', 'L')
('R', 'R', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'R', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'R', 'L', 'R')
('R', 'R', 'R', 'R')

Reach Goal State: ('R', 'R', 'R', 'R')
Task Completed
```

Puzzle 2 [A* Search]

```
 Start State: ('L', 'L', 'L', 'L')
 Goal State: ('R', 'R', 'R', 'R')
 Search Strategy: A* SEARCH
----------------------------------------

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'L', 'L')
('R', 'L', 'L', 'L')
('R', 'L', 'R', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'R', 'L')
('L', 'L', 'R', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'R', 'L')
('R', 'R', 'R', 'L')
('R', 'L', 'R', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'R', 'R')
('L', 'L', 'R', 'R')
('L', 'L', 'L', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'L', 'R')
('R', 'L', 'L', 'R')
('R', 'R', 'L', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'L', 'R', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'L', 'L', 'R')

Exploring State (Man, Wolf, Goat, Cabbage): ('R', 'R', 'L', 'R')
('L', 'R', 'L', 'R')
('L', 'R', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'R', 'L', 'L')
('R', 'R', 'L', 'L')

Exploring State (Man, Wolf, Goat, Cabbage): ('L', 'R', 'L', 'R')
('R', 'R', 'R', 'R')

Reach Goal State: ('R', 'R', 'R', 'R')
Task Completed
```