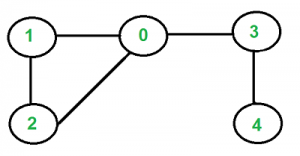
# Question 1

Depth-first search is good for computers with low memory, as it doesn’t store nodes previously visited – like breadth-first search. However, for more complex graphs which contain cycles as seen in the following graph:

this proves detrimental as the search gets stuck in an infinite loop. For this reason, the search algorithm is considered **incomplete**. The search can’t escape this loop without checking the next node hasn’t already been visited – which involves storing all visited nodes.

Greedy search is another search algorithm which naïvely bases the choice for the next child node to visit on a heuristic function. The child with the highest heuristic value is chosen. This algorithm is very simple which can also often make it very efficient. However, the solution given will obviously not always be the optimal solution, as the search can fall into what is analogous to a local-maximum, due to following what *seemed* to be the best path.

# Question 2

## Part 1

By simplifying the propositional-logic sentence: in the following steps:

1. – Implication converted

1. – Double negation rule applied

1. – Absorption equivalence rule applied to simplify term

We end up at the sentence which is obviously satisfiable by setting either A or B to true.

It’s not valid as this sentence can also be made to result in false, by setting either A or B to false.

## Part 2

By simplifying the propositional-logic sentence: in the following steps:

1. – Outer implication converted
2. – Inner implication converted

1. – De Morgan’s law used to bring negation into brackets

1. – De Morgan’s law used to bring negation into brackets

1. – Double negation rule applied

1. – Eliminated brackets
2. – Absorption equivalence rule applied to simplify term

In this final – disjunctive normal – form, , we can see it’s obvious that this sentence is satisfiable by setting at least one of A, B or C to false. The negation will result in one of the terms becoming true, and because it’s in DNF, when one term is true, the entire term becomes true. It’s not valid as this sentence can also be made to result in false, by setting at least one of A, B or C to true.

# Question 3

## Part A

To search this jug problem, we should first establish the type of graph we’re searching.

* Each node in the graph will represent a **State**.
* This state will consist of 3 values, corresponding to the current fill level of each jug
* The fill levels in the state should never be higher than the capacities specified
* Each branch in the graph will represent an **Operation** (such as “**fill Jug A**”on the state, resulting in a new (child) state
* The child state returned by the operation can be one that is already visited

For example, from the state (0, 0, 0), the operations “**empty jug B**” returns to the same visited state.

* There are 12 different operations
  + 3 Fill operations (1 per jug)
  + 3 Empty operations (1 per jug)
  + 6 Transfer operations (2 other jugs to transfer to, per jug). The number of transfer operations is represented by the equation ) where n is the number of jugs beings used per state – 3 for this example.

From the start state, (0,0,0), we will head traverse a branch (perform an operation) to reach a child state. For example; if our jug capacities for A, B and C were 3, 4 and 5 respectively; we could say the first operation was “**fill Jug A**” which would result in the new state (3, 0, 0). We then continue traversal from the new child state, recursively, until we stop generating new states. Unlike traversing a graph which has already been fully mapped out, we cannot determine which states are leaf nodes, without traversing them; and in this case the verdict still isn’t discerned traditionally. Usually, a leaf will have no children, however, obviously every state can get to another state – even if the new state is identical to itself.

Instead, we stop when the child state produced has already been visited. This provides **drastic** performance benefits as we will never need to re-query entire sections of the graph already been visited. This also solves our issue of the depth-first-search not being complete. Because there are no nodes in the tree without children, we obviously have infinite branches to traverse, so a traditional depth-first search will never end. However, this also means we’re no longer getting the same memory-efficiency of DFS, because like BFS, we’re storing all visited nodes.

Once establishing we’ve reached a child node, we will then backtrack through the graph to the last visited node with unvisited children. A recursive search is then conducted on these nodes, sequentially – but in an order that doesn’t matter.

The outline for a search procedure is the optimal, complete, algorithm for searching our problem-space using depth-first search.

### Pseudocode

The outline can be written in high level pseudocode as so:

Stack statesToExplore = {startState};

While (statesToExplore isn't empty) {

currentState = statesToExplore.pop();

for (op: 1 to 12) {

childState = state.traverseOperation(op);

if (childState hasn’t been visited) {

// DISCOVERED NEW CHILD STATE: childState

statesToExplore.push(childState);

alertNewState(childState);

}

}

}

## Part C

Using the program built from Part B, I’m able to conclude that there are 160 unique jug states attainable with the jug capacities 8, 5 and 3.

## Part D

To demonstrate how my program arrives at the goal state, (3, 2, 3) from the start state of (0, 0, 0), I’ve outlined all the operations going on within the program, including the pushing / popping of states from the stack.

I’ve consciously omitted including lines of the form “skipping child state, as already visited,” where the child state generated by an operation has already been searched. This was due to the inclusion of such lines increasing the document size to over 25 pages.

### Finding the goal state

#### Start state (0, 0, 0) is explored

- Fill Jug A: child State [8, 0, 0] is pushed to stack to be visited.

- Fill Jug B: child State [0, 5, 0] is pushed to stack to be visited.

- Fill Jug C: child State [0, 0, 3] is pushed to stack to be visited.

Popped: State [0, 0, 3] from stack to explore

- Transfer from Jug C to Jug A: child State [3, 0, 0] is pushed to stack to be visited.

- Transfer from Jug C to Jug B: child State [0, 3, 0] is pushed to stack to be visited.

- Fill Jug A: child State [8, 0, 3] is pushed to stack to be visited.

- Fill Jug B: child State [0, 5, 3] is pushed to stack to be visited.

Popped: State [0, 5, 3] from stack to explore

- Transfer from Jug B to Jug A: child State [5, 0, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [3, 5, 0] is pushed to stack to be visited.

- Fill Jug A: child State [8, 5, 3] is pushed to stack to be visited.

Popped: State [8, 5, 3] from stack to explore

- Empty Jug C: child State [8, 5, 0] is pushed to stack to be visited.

Popped: State [8, 5, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [5, 5, 3] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 2, 3] is pushed to stack to be visited.

Popped: State [8, 2, 3] from stack to explore

- Empty Jug A: child State [0, 2, 3] is pushed to stack to be visited.

- Empty Jug C: child State [8, 2, 0] is pushed to stack to be visited.

Popped: State [8, 2, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [5, 2, 3] is pushed to stack to be visited.

- Transfer from Jug A to Jug B: child State [5, 5, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 0, 2] is pushed to stack to be visited.

- Empty Jug A: child State [0, 2, 0] is pushed to stack to be visited.

Popped: State [0, 2, 0] from stack to explore

- Transfer from Jug B to Jug A: child State [2, 0, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [0, 0, 2] is pushed to stack to be visited.

Popped: State [0, 0, 2] from stack to explore

- Fill Jug B: child State [0, 5, 2] is pushed to stack to be visited.

Popped: State [0, 5, 2] from stack to explore

- Transfer from Jug B to Jug A: child State [5, 0, 2] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [0, 4, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [2, 5, 0] is pushed to stack to be visited.

- Fill Jug A: child State [8, 5, 2] is pushed to stack to be visited.

Popped: State [8, 5, 2] from stack to explore

- Transfer from Jug A to Jug C: child State [7, 5, 3] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 4, 3] is pushed to stack to be visited.

Popped: State [8, 4, 3] from stack to explore

- Empty Jug C: child State [8, 4, 0] is pushed to stack to be visited.

Popped: State [8, 4, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [5, 4, 3] is pushed to stack to be visited.

- Transfer from Jug A to Jug B: child State [7, 5, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 1, 3] is pushed to stack to be visited.

- Empty Jug A: child State [0, 4, 0] is pushed to stack to be visited.

Popped: State [0, 4, 0] from stack to explore

- Transfer from Jug B to Jug A: child State [4, 0, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [0, 1, 3] is pushed to stack to be visited.

Popped: State [0, 1, 3] from stack to explore

- Transfer from Jug B to Jug A: child State [1, 0, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [3, 1, 0] is pushed to stack to be visited.

- Empty Jug C: child State [0, 1, 0] is pushed to stack to be visited.

Popped: State [0, 1, 0] from stack to explore

- Transfer from Jug B to Jug A: child State [1, 0, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [0, 0, 1] is pushed to stack to be visited.

- Fill Jug A: child State [8, 1, 0] is pushed to stack to be visited.

Popped: State [8, 1, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [5, 1, 3] is pushed to stack to be visited.

- Transfer from Jug A to Jug B: child State [4, 5, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 0, 1] is pushed to stack to be visited.

Popped: State [8, 0, 1] from stack to explore

- Transfer from Jug A to Jug C: child State [6, 0, 3] is pushed to stack to be visited.

- Transfer from Jug A to Jug B: child State [3, 5, 1] is pushed to stack to be visited.

- Fill Jug B: child State [8, 5, 1] is pushed to stack to be visited.

Popped: State [8, 5, 1] from stack to explore

- Transfer from Jug A to Jug C: child State [6, 5, 3] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [8, 3, 3] is pushed to stack to be visited.

- Empty Jug A: child State [0, 5, 1] is pushed to stack to be visited.

Popped: State [0, 5, 1] from stack to explore

- Transfer from Jug B to Jug A: child State [5, 0, 1] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [0, 3, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [1, 5, 0] is pushed to stack to be visited.

Popped: State [1, 5, 0] from stack to explore

- Transfer from Jug B to Jug A: child State [6, 0, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 2, 3] is pushed to stack to be visited.

- Fill Jug C: child State [1, 5, 3] is pushed to stack to be visited.

Popped: State [1, 5, 3] from stack to explore

Popped: State [1, 2, 3] from stack to explore

- Transfer from Jug B to Jug A: child State [3, 0, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [4, 2, 0] is pushed to stack to be visited.

- Empty Jug C: child State [1, 2, 0] is pushed to stack to be visited.

Popped: State [1, 2, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [0, 2, 1] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 0, 2] is pushed to stack to be visited.

Popped: State [1, 0, 2] from stack to explore

- Transfer from Jug A to Jug B: child State [0, 1, 2] is pushed to stack to be visited.

- Fill Jug B: child State [1, 5, 2] is pushed to stack to be visited.

Popped: State [1, 5, 2] from stack to explore

- Transfer from Jug B to Jug A: child State [6, 0, 2] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 4, 3] is pushed to stack to be visited.

Popped: State [1, 4, 3] from stack to explore

- Transfer from Jug C to Jug A: child State [4, 4, 0] is pushed to stack to be visited.

- Empty Jug C: child State [1, 4, 0] is pushed to stack to be visited.

Popped: State [1, 4, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [0, 4, 1] is pushed to stack to be visited.

- Transfer from Jug B to Jug A: child State [5, 0, 0] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 1, 3] is pushed to stack to be visited.

Popped: State [1, 1, 3] from stack to explore

- Transfer from Jug B to Jug A: child State [2, 0, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [4, 1, 0] is pushed to stack to be visited.

- Empty Jug C: child State [1, 1, 0] is pushed to stack to be visited.

Popped: State [1, 1, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [0, 1, 1] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 0, 1] is pushed to stack to be visited.

Popped: State [1, 0, 1] from stack to explore

- Fill Jug B: child State [1, 5, 1] is pushed to stack to be visited.

Popped: State [1, 5, 1] from stack to explore

- Transfer from Jug B to Jug A: child State [6, 0, 1] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [1, 3, 3] is pushed to stack to be visited.

Popped: State [1, 3, 3] from stack to explore

- Transfer from Jug B to Jug A: child State [4, 0, 3] is pushed to stack to be visited.

- Transfer from Jug C to Jug A: child State [4, 3, 0] is pushed to stack to be visited.

- Empty Jug C: child State [1, 3, 0] is pushed to stack to be visited.

Popped: State [1, 3, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [0, 3, 1] is pushed to stack to be visited.

- Fill Jug A: child State [8, 3, 0] is pushed to stack to be visited.

Popped: State [8, 3, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [5, 3, 3] is pushed to stack to be visited.

- Transfer from Jug A to Jug B: child State [6, 5, 0] is pushed to stack to be visited.

Popped: State [6, 5, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [3, 5, 3] is pushed to stack to be visited.

- Transfer from Jug B to Jug C: child State [6, 2, 3] is pushed to stack to be visited.

Popped: State [6, 2, 3] from stack to explore

- Transfer from Jug C to Jug A: child State [8, 2, 1] is pushed to stack to be visited.

- Empty Jug C: child State [6, 2, 0] is pushed to stack to be visited.

Popped: State [6, 2, 0] from stack to explore

- Transfer from Jug A to Jug C: child State [3, 2, 3] is pushed to stack to be visited.

Popped: State [3, 2, 3] from stack

#### Goal state (3, 2, 3) is reached.

### Final Path

The final path taken from my program from (0,0,0) to (3,2,3) is the following 37 steps:

#### Start State (0, 0, 0)

1. Fill Jug C: State [0, 0, 3]
2. Fill Jug B: State [0, 5, 3]
3. Fill Jug A: State [8, 5, 3]
4. Empty Jug C: State [8, 5, 0]
5. Transfer from Jug B to Jug C: State [8, 2, 3]
6. Empty Jug C: State [8, 2, 0]
7. Empty Jug A: State [0, 2, 0]
8. Transfer from Jug B to Jug C: State [0, 0, 2]
9. Fill Jug B: State [0, 5, 2]
10. Fill Jug A: State [8, 5, 2]
11. Transfer from Jug B to Jug C: State [8, 4, 3]
12. Empty Jug C: State [8, 4, 0]
13. Empty Jug A: State [0, 4, 0]
14. Transfer from Jug B to Jug C: State [0, 1, 3]
15. Empty Jug C: State [0, 1, 0]
16. Fill Jug A: State [8, 1, 0]
17. Transfer from Jug B to Jug C: State [8, 0, 1]
18. Fill Jug B: State [8, 5, 1]
19. Empty Jug A: State [0, 5, 1]
20. Transfer from Jug C to Jug A: State [1, 5, 0]
21. Transfer from Jug B to Jug C: State [1, 2, 3]
22. Empty Jug C: State [1, 2, 0]
23. Transfer from Jug B to Jug C: State [1, 0, 2]
24. Fill Jug B: State [1, 5, 2]
25. Transfer from Jug B to Jug C: State [1, 4, 3]
26. Empty Jug C: State [1, 4, 0]
27. Transfer from Jug B to Jug C: State [1, 1, 3]
28. Empty Jug C: State [1, 1, 0]
29. Transfer from Jug B to Jug C: State [1, 0, 1]
30. Fill Jug B: State [1, 5, 1]
31. Transfer from Jug B to Jug C: State [1, 3, 3]
32. Empty Jug C: State [1, 3, 0]
33. Fill Jug A: State [8, 3, 0]
34. Transfer from Jug A to Jug B: State [6, 5, 0]
35. Transfer from Jug B to Jug C: State [6, 2, 3]
36. Empty Jug C: State [6, 2, 0]
37. Transfer from Jug A to Jug C: State [3, 2, 3]

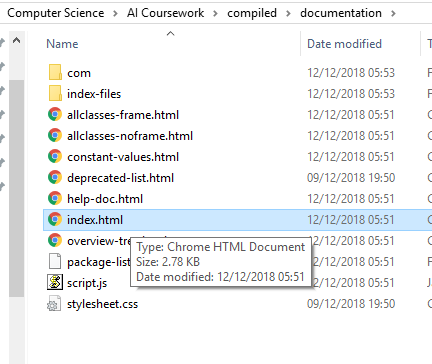
#### Goal Reached (3, 2, 3)

# Appendix

CONTENTS OF INSTRUCTIONS.txt

Documentation is a standard JavaDoc generated from extensive JavaDoc comments.

To view the documentation, please open the "index.html" web page inside the "documentation" folder



documentation/index.html

Alternatively, you can open the "documentation" folder in a browser and it will automatically load the page.

----

To run the java program, you can either pass in command line arguments, or use the console to enter

in your jug capacities, at run-time.

You will need to open a console in this directory and run the following:

if 0 capacity arguments are passed, you will be prompted to enter a number once the program starts.

Otherwise, the capacities passed will be validated (only to make sure they're integers).

Finally, the program will log all discovered states generated from the start state.

It will then finish execution, after logging some statistics about the search.