**Coursework – AI Methods COB107 – Peter Smith B614743**

**1.** **List and explain the criteria used to evaluate search methods**.

There are four criteria for evaluating a search method:

* **Completeness**

A search algorithm is complete if, when the goal state exists within the data, the algorithm finds it, and otherwise, correctly reports a failure.[1] If the desired data is found, then a success is reported and if the desired data is not present, then we are informed that it is not in the given data. This is important as if an algorithm isn’t complete, then the answer can be uncertain, and the algorithm will likely loop infinitely. A complete algorithm will always return an answer.

* **Time Complexity**

Time Complexity concerns the amount of time it takes for the algorithm to find a solution. Time Complexity often pays attention to the number of iterations the algorithm goes through – for an amount of data *n*, an algorithm may take a relatively short amount of time (n iterations, for example), or a relatively large amount of time (2n iterations, for example, which will become progressively slower as the dataset becomes larger). Quicker algorithms (that have lower time complexity) are considered better and more efficient.

* **Space Complexity**

Space Complexity concerns the memory space used by the search function during the search. This usually includes the amount of space the program itself takes up, and any variables used in the program. A program with less lines of code will take up less space in the memory and is considered to have lower space complexity. Likewise, creating fewer variables will also mean an algorithm is evaluated to have lower space complexity.

* **Optimality**

An algorithm is optimal if it finds the best possible solution, when there exists several different solutions. The highest quality solution usually means the solution that has the lowest ‘depth’, where depth is the number of distinguishable steps from the start state. If a search algorithm finds a non-optimal solution, then the algorithm is not optimal.

**2. Is the following propositional logic sentence valid? Why?**

A Propositional Logic Sentence is valid if and only if it is satisfied by every possible combination of truth values to the variables in it. The sentence is satisfiable if and only if there is an assignment of truth values to the variables in it such that the sentence evaluates to true. Hence, the sentence S is valid if and only if the sentence evaluates to true, for every possible combination of truth values to the variables in S. Truth tables can be used to prove that a given sentence is valid.­[2]

For Truth Table 1, we define S:

**Table 1**, a truth table describing all possible situations and the truth values of S.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| T | T | T | T | T | T | T |
| T | F | F | T | F | F | T |
| F | T | T | F | F | T | T |
| F | F | T | T | T | T | T |

In Truth Table 1, S evaluates to true for every possible interpretation. Therefore, according to the definition above, sentence S is true, as it evaluates to true for every possible assignment of truth values to the variables in S. S is also a tautology.

**3a. Considering the described problem, write and explain a high-level breadth-first search algorithm that takes A, B, and C as input and generates the set of all possible states that can be reached from the start state (0, 0, 0).**I will run through a dynamically changed array using a for loop. For each state I will check through the possible changes you can make. For each new possible state, I will check through the array of existing states so that there are no duplicates. If it is a distinct state, I will append it to the end of the array. The for loop continues to loop through. As I append to the end of the array, the for loop means that every node on a level is checked before going on to the next level – the algorithm checks the nodes in the order they were created. This means that the algorithm is breadth-first. Here is pseudo code on how it may work:

INPUT: INTs A, B and C – Jug capacities  
OUTPUT: S, a set of all possible states from (0,0,0).  
StartState := NEW State(0,0,0,-1);  
StateList := NEW ArrayList.add(StartState);  
FOR LOOP (Int x runs through StateList):  
 TempState := StateList.get(x);  
 CreateStatesChildren(x);  
END LOOP  
//For each element in the array, any possible new states are appended to the end of the array, so the for loop only stops once every state has been checked whether it can create new children.  
PrintArray;  
//The array is only printed after the array has been completed.

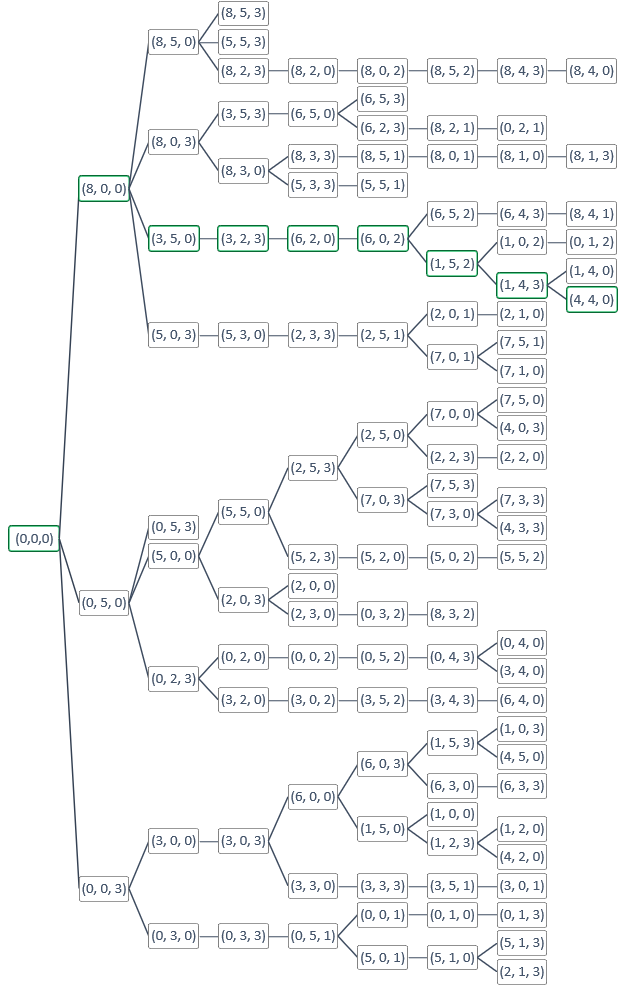
FUNCTION CreateStatesChildren(x);  
INPUT: Int x, the parent index;  
OUTPUT: None;  
FillJugs(x);  
EmptyJugs(x);  
PourJugs(x);  
//This goes through each possible change and checks whether it can be done.

FUNCTION FillJugs(x);  
INPUT: Int x, the parent index;  
OUTPUT: None;  
If(Jug is not at full capacity)  
 AddState(f,g,h,x);  
//f,g,h are the values of water in each jug we want to add to the StateList. Depending on which jug has been filled, that jug will be A, B, or C, the jug’s capacity.

FUNCTION EmptyJugs(x);  
INPUT: Int x, the parent index;  
OUTPUT: None;  
If(Jug is not empty)  
 AddState(f,g,h,x);  
//f,g,h are the values of water in each jug we want to add to the StateList. Depending on which jug has been emptied, that jug will be 0.

FUNCTION PourJugs(x);  
INPUT: Int x, the parent indexl  
OUTPUT: None;  
If(the current sum of the liquid in the two jugs does not exceed the destination jug’s capacity)  
 AddState(f,g,h,x);  
//f,g,h are the values of water in each jug we want to add to the StateList. This will try each combination of jugs. It will completely pour the contents of one jug into the other. The destination jug will then contain the sum of liquid, while the other jug will become 0.  
Else if(the current sum of the liquid in the two jugs does exceeds the destination jug’s capacity)  
 AddState(f,g,h,x);  
//f,g,h are the values of water in each jug we want to add to the StateList. This will try each combination of jugs. It will partially pour the contents of one jug into the other. The destination jug will then contain its maximum capacity, while the other jug will become contain the remainder of liquid.  
  
FUNCTION AddState(x,y,z,parent)  
INPUT: Ints x,y,z (the values of water in each jug that we want to add to the StateList) and Int parent, the parent index.  
OUTPUT: None;  
BOOLEAN included = false;  
FOR LOOP(int m through StateList)  
 IF(StateList.get(m) has the same numbers as x,y and z)  
 included = true;  
IF(Included is false)  
 StateList.add(new State(x,y,z,parent));

//This only adds the new state if it isn’t already in the array.  
//The parent index can be used to trace the node.



**3c.** A tree diagram representing the path the program takes from start state (0,0,0) to (4,4,0).

Depth Increases **NB**: The program moves downwards, from the top.

Depth 8, if the start state is depth 0.

**Bibliography**

1. LAVALLE, S. Planning Algorithms. [online]. [viewed last 11/12/2017). Available from: http://planning.cs.uiuc.edu/node631.html
2. FATIMA, S. Propositional Logic lecture notes. [online]. Slide 29. [viewed last 11/12/2017]. Available from: http://learn.lboro.ac.uk/pluginfile.php/634312/mod\_resource/content/21/proplogic.pdf