COMP 30030: Introduction to AI
Assignment 1

Student: Daniel Gallagher
Student ID: 18401492

Question 1(a)

The Turing Test is an operational test for intelligent behaviour in machines, originally published in 1950 by Alan Turing.

It involves three elements: a machine (what we think of now as a computer), a human interrogator and a human evaluator. The machine and the human evaluator and machine will interact through a system which is moderated by the interrogator.

The goal of the evaluator is to decide whether he/she is speaking to a real person or a machine by the end of their conversation.

Alan Turing first proposed this in a paper entitled “Computing Machinery and Intelligence”. Many regard this paper as one of the most important contributions to the development of the field of AI.

Two Aspects Of Intelligence That The Turing Test Evaluates:

- Linguistic Intelligence - Ability to understand and use written/spoken language.
- Logic Intelligence - Ability to reason and make connections.

Two Aspects Of Intelligence That The Turing Test Does Not Evaluate:

- Spatial Intelligence - Ability to process our surroundings.
- Kinaesthetic Intelligence - Ability to move around within a space.

Question 1(b)

Operators For State Transformation:
Operator: ***pour_A_into_B(A, B)***
Parameters A, B := Jug A, Jug B
*Pour the water in Jug A into Jug B, until capacity is reached**

Operator: ***fill_A_with_pump(A)***
Parameters: A := Jug A
Fills a jug with water from the pump until capacity is reached.

Operator: ***empty_A_into_drain(A)***
Parameters: A := Jug A
Empties all the water from Jug A into the drain.

Solution

Summary
State represented as tuple (x, y), where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug.
0 <= x <= 4
0 <= y <= 3

Details
Initial Start State: (0,0)
Operators Actions: Defined Above
Goal: Reach State (2,0)

Steps
Initial State: **(0, 0)**

Operator: fill_A_with_pump(y)
State 2: **(0, 3)**

Operator: pour_A_into_B(y, x)
State 3: **(3, 0)**

Operator: fill_A_with_pump(y)
State 4: **(3, 3)**

Operator: pour_A_into_B(y, x)
State 5: **(4, 2)**

Operator: empty_A_into_drain(x)
State 6: **(0, 2)**

Operator: pour_A_into_B(y, x)
State 7: **(2, 0)**

Goal Of State (2,0) Reached

Question 2

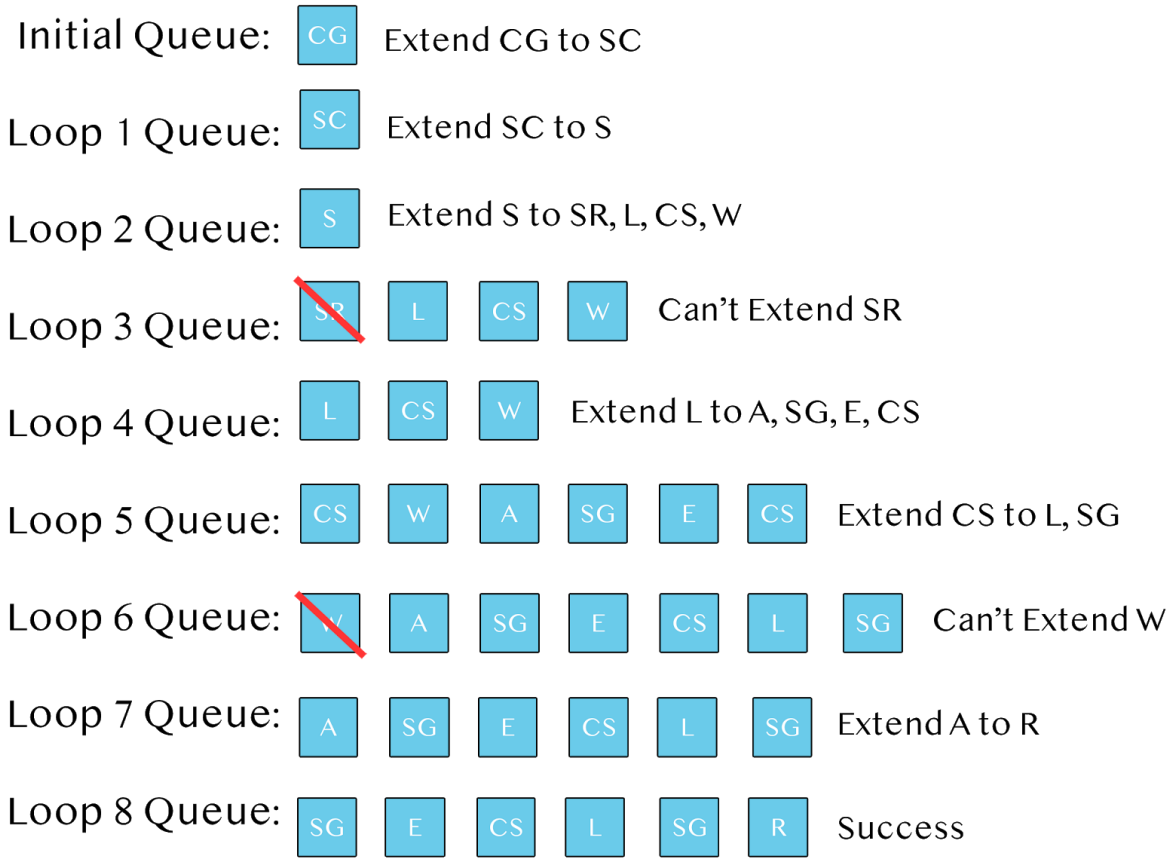
Description

We are asked to perform a complete trace of the BFS procedure on the UCD tree. This tree was originally presented in the lecture slides, however BFS was not used.

Goal

Our goal for this problem is to find the state SG using BFS.

Solution

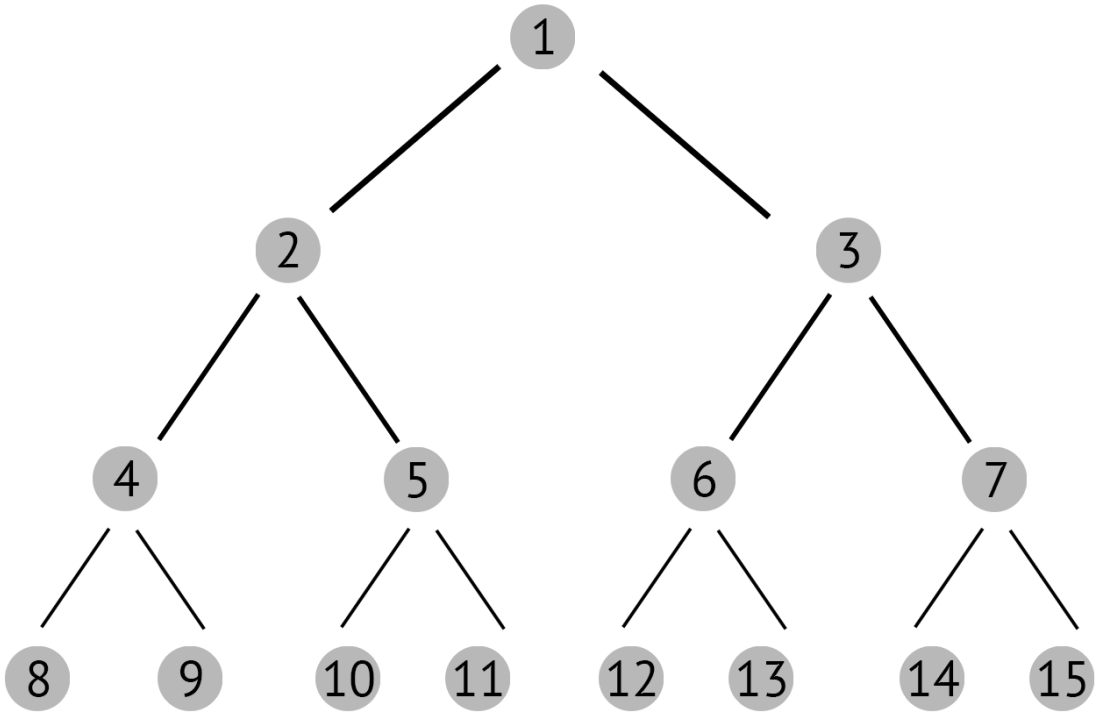


Order Of Nodes Visited

CG → SC → S → L → CS → W → A → SG

Question 3(a)

Search Tree Corresponding To State Space For States 1 - 15



Question 3(b)

Breadth-First Search

First, let's see our queue as we iterate through the tree:

1
2 | 3
3 | 4 | 5
4 | 5 | 6 | 7
5 | 6 | 7 | 8 | 9
6 | 7 | 8 | 9 | 10 | 11
7 | 8 | 9 | 10 | 11 | 12 | 13
8 | 9 | 10 | 11 | 12 | 13 | 14 | 15
9 | 10 | 11 | 12 | 13 | 14 | 15
10 | 11 | 12 | 13 | 14 | 15
11 | 12 | 13 | 14 | 15 *[Success]*

From this, we can list the nodes that were visited:
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7
Node 8
Node 9
Node 10
Node 11 *[Success]*

Depth-Limited Search

We'll start with the queue again:
1
2 | 3
4 | 5 | 3
8 | 9 | 5 | 3
9 | 5 | 3
5 | 3
10 | 11 | 3
11 | 3

Thus, we gather the order of nodes visited:
Node 1
Node 2
Node 4
Node 8
Node 9
Node 5
Node 10
Node 11 *[Success]*

Notice the shorter search time of the DFS approach in this scenario. Lastly, we will try the iterative deepening approach.

Iterative Deepening Search

Our queue:
1
[end]

1
2 | 3
3
[end]

1
2 | 3
4 | 5 | 3
5 | 3
3
6 | 7
7
[end]

1
2 | 3
4 | 5 | 3
8 | 9 | 5 | 3
9 | 5 | 3
5 | 3
10 | 11 | 3
11 | 3
[success]

From this we can gather the order of nodes visited:
Node 1
Node 1
Node 2
Node 3
Node 1
Node 2
Node 4
Node 5
Node 3
Node 6
Node 7
Node 1
Node 2
Node 4
Node 8
Node 9
Node 5
Node 5
Node 10
Node 11

This approach takes many more steps than BFS and DLS in this example.

Question 4

The Problem
To understand the problem presented by this tree, let us first follow our DFS algorithm through the tree and see the queue that forms:

i
s | f
h | f
n | c | f
c | f
f
h
n | c
c

Notice the areas which I have highlighted in bold, they appear to be the same. This points towards a fundamental problem with using DFS on this tree, **we must iterate through the**

same nodes more the once. This is inefficient, and in a more complex problem could lead to huge time costs.

Why This Problem Occurs
There are two reasons that this problem presents itself.

Firstly, **Node s and Node f point to the exact same nodes.** Both of their neighbours are an empty node and the node h. In a sense, this means that node s and f are the same node, and having them on the tree causes unnecessary problems. This is the first element of the problem, the tree itself.

Secondly, the DFS algorithm plays its own role in the issue. Depth-First Search (on its own) **cannot distinguish between nodes that we have previously visited and nodes that we have not**. If it had this feature, we would be able to say a big "NO!" to the second iteration of the node h, however as far as DFS knows this is a completely new and unvisited node.

This second reason will play a crucial role in solving this problem.

The Solution
We can find the solution to this problem by, in a sense, hiring a bouncer. We need something to ensure that the nodes we will be iterating through next haven't been visited already.

To solve this problem we'll use an array called "*visited"* which will tell us if the next node has been visited before. This array will be initialized with all values set to *false*.
Each time we are expanding a new node, we will check if it has been visited, and deny its addition to the queue if we have. Otherwise, we will set its value to true, as we have now visited it.

Below, you can find the pseudocode that will allow us to fix this problem with DFS:

```
Array: visited [entries initialized to false]

DFS(Node A):
    visited[A] = true

    foreach node n in A:
        if not visited[n]:
            DFS(n)
```