Northeastern University

CS 5100 Foundations of Artificial Intelligence

Homework and PA 1

NAME: **PARSHVA TIMBADIA**

EMAIL: timbadia.p@northeastern.edu

NUID: **001091783**

1.

a. Playing Soccer

PEAS:

P- Performance	**E- Environment**
Victory / Defeat**,**	Football,
Number of Goals Scored (GF),	Football pitch,
Number of Goal Scored by Opponent (GA)	Team Players,
	Opponents
A- Actuators	**S- Sensors**
Head,	Eyes,
Legs,	Ears
Chest	

Properties of Task Environment:

Observable	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Known**
Partially	Multi-agent	Stochastic	Sequential	Dynamic	Continuous	Unknown

b. Shopping for used AI books on the Internet

PEAS:

P- Performance	E- Environment
Low Price,	Website's that sell Used books
Edition,	
Quality	
A- Actuators	S- Sensors
Mouse,	Web Interface
Keyboard	

Properties of Task Environment:

Observable	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Unknown**
Partially	Multi-agent	Stochastic	Sequential	Dynamic	Discrete	Unknown

c. Playing a tennis match.

P- Performance Scoring more for Victory	E- Environment Tennis Ball, Opponent, Tennis Court
A- Actuators Racket, Legs	S- Sensors Eyes, Ears

Properties of Task Environment:

Observable	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Known**
Partially	Multi-agent	Stochastic	Sequential	Dynamic	Continuous	Unknown

d. Performing a high jump.

P- Performance Jumped over the horizontal bar without dislodging it or not	E- Environment Jumping Horizontal Bar, Track
A- Actuators Legs	S- Sensors Eyes

Properties of Task Environment:

Observable	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Known**
Observable	Single Agent	Stochastic	Sequential	Dynamic	Continuous	Known

e. Knitting a sweater.

P- Performance Comfort, Size, Looks (Style)	E- Environment A spot to perform the task
A- Actuators Hands	S- Sensors Eyes

Properties of Task Environment:

Observable	**Agents**	**Deterministic**	**Episodic**	**Static**	**Discrete**	**Known**
Observable	Single-Agent	Stochastic	Sequential	Dynamic	Continuous	Unknown

2>

Agents: An entity which can sense the surroundings to learn or understand the situation and later perform some actions to make changes in the environment accordingly, showcasing intelligence-based behavior.

Agent Function: A function that provides an ability to the agent to perform certain actions as output, by taking in certain percept, but in the mathematical abstract.

Agent Program: Makes use of Agent Function for the actual implementation within some physical system.

Rationality: The actions for optimizing or maximizing its own performance. In other words, maximizing the anticipated utility.

Autonomy: The ability of the agent to learn and act on its own over the period of time rather than always relying on initial knowledge.

Reflex Agent: An agent that reacts only on the basis of its current states and does not take into consideration its previous actions.

Model-Based Agent: This is an agent that makes decision-based on its past knowledge and internal memory. Basically, the internal state helps in making the agent aware related to the working of the environment and aspects of the world which the agent might not be able to perceive currently.

Goal-Based Agent: An agent that is aware of the goal state and align its actions in that direction to achieve the goal.

Utility-Based Agent: An agent whose performance measure is given by the optimal way to reach the goal state based on the preference, ie. it makes use of the utility function.

Learning Agent: An agent that is able to adapt from the experience by learning and later enhance its performance.

3>

GOAL-BASED AGENT

So basically Goal-based agent is the extension of the Model-Based Agent. Here the agents will percept the environment that is what is happening in the present and what my actions do. In the goal-based Agent, the agent possesses the knowledge of where the Goal state is, and hence it aligns its actions based on the current state it is in and the Goal State.

For instance, let's consider a GPS system where the user inputs its final Destination and the current location can be tracked by the GPS. Here the agent will act in a certain way that will help to reach the user to the reach desired location.

So the below pseudo-code is trying to explain that the **State** gets updated after every action and based on the **Sate** and the **goals** the next **ACTION based on searching and planning** will be taken by the agent.

GOAL BASED AGENT:

```
function GOAL-BASED-AGENT(percept) returns an action
    persistent: state, the agent's current idea of the world
                action, the most recent action, initially none.
                goal, the final destination or the desire that needs to be achieved.
    State ← UPDATE_STATE(State, percept, action, goal)
    If State== goal:
        STOP
    action ← ACTION(State, goals ).ACTION
    return action
```

UTILITY-BASED AGENT

The Utility-based agent is again the extension of the Goal-based Agent, here reaching the final destination is not the goal but at the same time it should also take the Utility into the consideration.

Considering the same GPS example like the above, now the agent will also take into account the utility, meaning that it will provide the best possible route which might be either comfortable or short and so on. It selected the best possible action based on the utility to provide the maximum happiness.

So in the below pseudo-code is trying to explain that the **State** gets updated based on the previous actions and the current percept from the environment and the **BEST_ACTION represents the action that maximize the Utility** based on the **State**(Current Updated State) and **possible_states.**

```
UTILITY BASED AGENT:

function UTILITY_BASED-AGENT(percept) returns an action
    persistent: state, the agent's current idea of the world
        action, the most recent action, initially none.
        possible_states, every possible states from the current state.
    State ← UPDATE- STATE(State, action, percept, possible_states)
    action ← BEST_ACTION( State, possible_states)
    return action
```

4>

a>

Initial State: All the regions in the provided map will be uncolored

Actions: Assigning a color to a region.

Transition Model: The uncolored region will be assigned with some color.

Goal State: Every region colored and no two adjacent regions should contain the same color.

Path Cost: Total number of regions provided on the map.

b>

Initial State: Initial position of monkey in a room and bananas suspended from the ceiling.

Actions: Climb on the crate, jump off the crate, stacking both the crates one on another, move the crate around, walking around, and grabbing bananas while on the crate.

Transition Model: The boxes have been stacked, or moved.

Goal State: Monkey get the bananas

Path Cost: Total number of actions required.

5>

State: Whenever the agent performs a certain action, it is represented in terms of state.

State Space: A state space is the set of states connected to each other if there is an action to perform, and that action can lead to a change in the state from one state to another.

Search Tree: A tree where the root node is considered as the initial position and one can traverse from node to another by performing certain action.

Search Node: A node in the search tree.

Goal: The final state where the agent wants to reach.

Action: When an agent is in a particular state, it can perform certain **actions** that lead to the change from the previous state or sometimes into the same state.

Transition Model: It provides the general idea of what the particular action leads to.

Branching Factor: The branching factor indicates the total number of possible moves, the agent can make.

6>

a>

State Representation of the Sudoku can be given as (3x3) x9. As there are 9 sub-boxes of the dimension 3x3. Each of the single squares should be filled with number 1-9 such that there is no duplicate found in any of the rows, columns or any sub-boxes (3x3 grid) square.

Therefore, 9x9= 81

b>

So basically what I am trying to do is first the algorithm will start implementing by searching for the empty node in the grid. If it finds the empty node it will try to assign the value from 1 to 9 for it by checking the possible function. The possible function will return True if the element is unique in the entire same row, column, and 3x3 gird. Next, it will assign the value and if the value and add the current state that is the gird in my case to the successor list. Further, it will continue assigning the value and if some other value fails to satisfy the Sudoku constraint it will backtrack to the previous stages and try to re-assign different values for the location and continue further.

```
Successor =[] #Declaring the Successor list

    for y = 1 to 9:
          for x= 1 to 9:
                if grid[y][x]==0: #This means that the Grid is empty
                      for n= 1 to 9: # Number to assign
                         IF possible(y, x, n):
                            grid[y][x]=n #This is the current state.

                            ADD the element into the Successor List
                        ENDIF
                      ENDFOR
                ENDIF
          ENDFOR
    ENDFOR

FUNCTION possible(y,x, n):
    “””
          This function makes sure that none of the element in the horizontal, vertical
and in the 3x3 gird matches.
    “””
    FOR i= 1 to 9:
          If gird[y][i]==n:
                RETURN FALSE
    FOR i= 1 to 9:
            If gird[i][y]==n:
                RETURN FALSE

    // Now checking for the 3x3 GRID
    // Hence reducing the size of 9x9 to 3x3 GRID

    X_reduced= (x//3)*3
    Y_reduced = (y//3)*3
    FOR i= 0 to 2:
            FOR j = 0 to 2:
                IF grid[X_reduced+i][Y_reduced+j]==n:
                        return FALSE
                ENDIF
            ENDFOR
    ENDFOR

    return TURE
```

In the above function, the X_reduced and Y_reduced play a role in keeping the grid within the specific 3x3 gird. For example, if I want to select the last 3x3 gird of the from the top. Then if my current state is at 1, 8 for instance. Then the Y_reduced would be 1, but X_reduced would be 7 and thereby support the for loop function in the further steps to be in the 3x3 grid.

c>

Now the above current state will add the number to the gird if that particular number is not present in the particular rows, columns and the 3x3 gird but if it does not find the number, that is it will run out of options to input any digit in the square it will remain as empty. So our successor function should check if the any state is **Empty or Not**. If **ANY** state is Empty it will return False.

ALL ← The entire 9x9 grid should be filled.

ANY ← If any of the square is kept empty it should return False.

Now let us consider C= Current State,

```
If ALL(ANY(C==0)):
    Return False
```

```
ENDIF
Return True.
```

d>

As we have 9x9 grid which makes total of 81 squares out of which 28 digits have already been filled, so to reach the goal state we need to fill in only the remaining blocks that is 53.

$$L = 81 - 28 = 53$$

7>

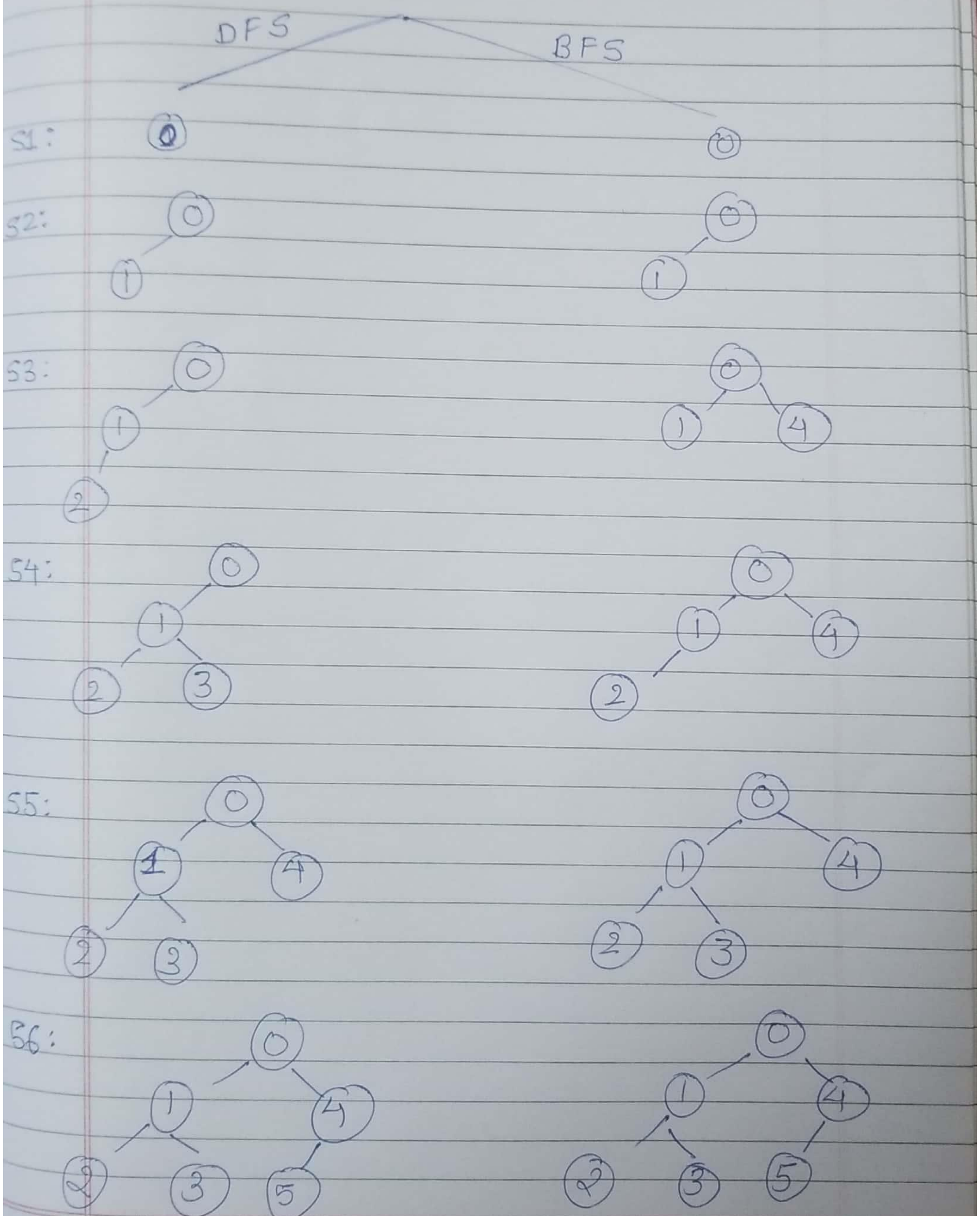
State Representation: Every parent node has two child node, so the total number of state in the above tree can be given by $\mathbf{2^{(h+1)}-1}$**.** Where h stands for the height of the tree and it is calculated from the root node. The height of the root node is considered as zero.

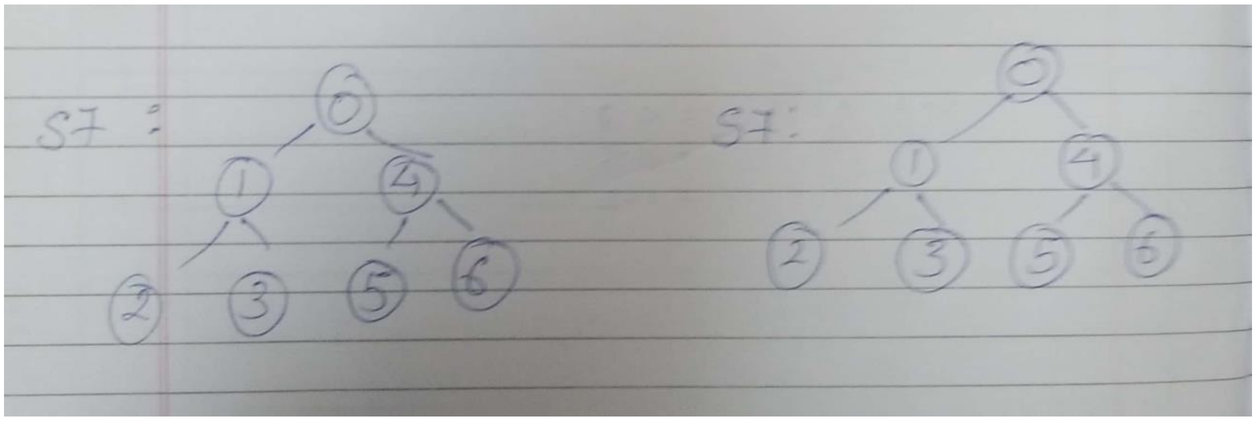
For the diagram provided the state representation will be $\mathbf{2^{(2+1)}-1}$ that is **7.**

The DFS will travel the deepest node first, while the BFS will traverse the shallowest one first. The traversal for the given diagram is attached below along with the code.

Formulation:

Initial: Root Node
Final: All nodes should be visited
Action: Based on DFS, we will do it till the deepest node first. For BFS the shallowest one
Path Cost: The number of nodes visited
Transition Model: To make the unvisited node as visited.


DFS
BFS
S1:
0
0
S2:
0
1
0
1
S3:
0
1
2
0
1
4
S4:
0
1
2
3
0
1
4
2
S5:
0
1
4
2
3
0
1
4
2
3
S6:
0
1
4
2
3
5
0
1
4
2
3
5



DFS:

```python
# Using a Python dictionary for adjacency list
graph = {
    0 : [1,4],
    1 : [2, 3],
    4 : [5,6],
    2 : [],
    3 : [],
    5 : [],
    6:[]
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node, end=' ')
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 0)
```

Output:

```
0 1 2 3 4 5 6
Executed in: 0.033 sec(s)
Memory: 4176 kilobyte(s)
```

BFS:

Code:

```
# Using a Python dictionary for adjacency list
graph = {
  0 : [1,4],
  1 : [2, 3],
  2 : [],
  3 : [],
  4 : [5,6],
  5 : [],
  6:[]
}

visited = [] # List to keep track of visited nodes.
queue = []   #Initializing a queue

def BFS(visited, graph, node):
  queue.append(node)
  while queue:
    element = queue.pop(0) #pop the left most element and mark it visited
    visited.append(element)
    print(element, end=' ')
    for ele in graph[element]: #Include the neighbours
      if ele not in visited:
        queue.append(ele)


# Calling the function with the starting node
BFS(visited, graph, 0)
```

Output:

```
0 1 4 2 3 5 6
Executed in: 0.038 sec(s)
Memory: 4476 kilobyte(s)
```