Homework 1

Part A: Theory and Algorithms

Julian Torres

1) 2.4

For each of the following activites, give a PEAS description of the task environment and characterize it in terms of the properties listed in section 2.3.2

- playing soccer
- Exploring the subsurface oceans of Titan.
- Shopping for used AI books on the Internet.
- Playing a tennis match.
- Practicing tennis against a wall.
- Performing a high jump.
- Knitting a sweater.
- Bidding on an item at an auction.

Playing soccer

Performance Measure: maximize goals scored, minimize goals allowed, minimize penalties, minimize injuries

Environment: Soccer field - grass or turf, various weather conditions

Actuators: those available to a human athlete - running, kicking, jumping, sliding, diving

Sensors: Camera, radar, accelerometer, microphones, force-torque sensors, GPS, encoders

---

Exploring the subsurface oceans of Titan.

Performance Measure: maximize area covered by agent, maximize samples taken per unit area,

Environment: subsurface oceans of titan- varying levels of pressure depending on level of submersion. Low visibility

Actuators: Common submarine propulsion system

Sensors: Barameter, GPS, lidar, camera

Shopping for used AI books on the Internet.

Performance Measure: minimize damage on each book found, maximize specificity of content to AI (relevance of the book's subject matter to AI topics), minimize cost, minimize shipping time

Environment: ecommerce platforms (e.g. Amazon), online forums/marketplaces

Actuators: Search actions, selection actions, sorting actions, purchasing actions, negotiation actions

Sensors: Book listing details, pricing information, reviews/ratings, search feedback

---

Playing a tennis match.

Performance Measure: maximize points scored, minimize points allowed, minimize penalties, minimize errors, maximize stamina (performance at beginning of match compared to end)

Environment: Physical court (clay, grass, or hard), weather conditions,

Actuators: Running, backpedaling, sidestepping, serving, forehand hits, backhand hits

Sensors: Visual input: Ball position, speed, and trajectory. Opponent's position, movement, and racket motion. Court boundaries and net position. Tactile input: Grip on the racket. Feedback from hitting the ball (e.g., power, spin). Auditory input: Sounds of the ball hitting the racket or court. Calls from the umpire or line judges (e.g., "Out" or "Fault"). Proprioceptive input: Body position and balance during movement and strokes.

Practicing Tennis Against a Wall

Performance Measure:

- Maximize successful ball returns
- Minimize missed shots
- Maximize control over shot placement
- Minimize fatigue

Environment:

- A flat, vertical wall
- A smooth, hard playing surface
- Variable weather conditions if outdoors

- Possible external distractions

Actuators:

- Running, sidestepping, backpedaling
- Forehand and backhand strokes
- Adjusting shot power and spin

Sensors:

- **Visual input:** Ball trajectory, rebound angle, speed
- **Tactile input:** Feedback from racket grip and ball contact
- **Auditory input:** Sound of the ball bouncing off the wall and racket
- **Proprioceptive input:** Body positioning and balance

Performing a High Jump

Performance Measure:

- Maximize jump height
- Minimize contact with the bar
- Maximize proper form and technique
- Minimize injury risk

Environment:

- A high jump track with a bar set at a specific height
- Landing area with a cushioned mat
- Possible environmental factors: wind, temperature, surface traction

Actuators:

- Running and accelerating during approach
- Jumping with explosive force
- Arching body over the bar
- Landing safely on the mat

Sensors:

- **Visual input:** Bar position, approach distance, and takeoff spot
- **Proprioceptive input:** Body orientation, jump timing, and muscle feedback
- **Tactile input:** Ground contact during takeoff and landing

---

Knitting a Sweater

Performance Measure:

- Maximize accuracy of stitch patterns
- Minimize errors and unraveling
- Maximize speed without compromising quality
- Minimize material waste

Environment:

- A controlled indoor workspace
- Yarn and knitting needles of varying materials and sizes
- Patterns and instructions to follow

Actuators:

- Hand and finger movements to loop, stitch, and adjust tension
- Adjusting needle positioning and yarn control

Sensors:

- **Tactile input:** Yarn tension, needle grip, and stitch tightness
- **Visual input:** Stitch patterns, errors, and progress tracking
- **Proprioceptive input:** Hand positioning and muscle control

---

Bidding on an Item at an Auction

Performance Measure:

- Minimize cost while winning the item
- Maximize bidding efficiency (timing and amount)
- Minimize overbidding
- Maximize awareness of competing bidders

Environment:

- Live auction or online auction platform
- Competing bidders with unknown budgets and strategies
- Auctioneer moderating the process (in a live auction)

Actuators:

- Placing bids via hand gestures, paddles, or digital inputs
- Monitoring and adjusting bid amounts
- Timing bids strategically

Sensors:

- **Visual input:** Competing bids, auctioneer signals, item details
- **Auditory input:** Auctioneer's calls, competitor reactions

- **Digital feedback:** Online bid status updates, bid increments, and auction countdown

In [ ]:

2) 3.6a

Give a complete problem formulation for following. Choose a formulation that is precise enough to be implemented. Using only four colors, you have to color a planar map in such a way that no two adjacent regions have the same color.

State: defined by the assignment of colors to regions in the map. Call them R1, R2, R3,...,Rn. Each region can take one of {red, blue, green, yellow, None (region yet to be colored)}. Ex) R1=red

Initial State: No regions colored initially.

Actions: assign a color {red, blue, green, yellow} to a region R such that it does not violate the adjacency constraint. Ex) assign blue to R2 such as Color(R2, blue)

Transition Model: Coloring a region changes it's state from `None` to some color

Ex)

```
print(R2) -> None
color(R2, blue)
print(R2) -> blue
```

Goal Test: goal is reached if every region is colored with no two adjacent regions being the same color. Adjacency can be defined by an input graph G, where nodes represent regions and edges represent adjancencies between regions. Test can be implemented as: `for every edge (Ri, Rj) in G; Color(Ri) != Color(Rj)`

Path Cost: total number of coloring steps taken, each coloring action has an assumed cost of 1

3.6b

A 3-foot-tall monkey is in a room where some bananas are suspended from the 8-foot ceiling. He would like to get the bananas. The room contains two stackable, movable, climbable 3-foot-high crates.

State: represents the position of the monkey in the room M = (x,y), position of crates in the room Ci = (x,y) [i = 1,2], whether a crate is stackable (`bool`), position of the bananas (fixed at 8 ft)

Initial State: Monkey at some start position on floor (xi, yi), The two crates placed someone separately on the floor. Ex) C1 = (1,2), C2 = (4, 3)

Actions:

- monkey can move from position (x,y) to (x', y').
- It can push a crate from one position to another. Ex) `push(C1, (x1,y1), (x2,y2))` moves crate C1 from x1y1 to x2y2
- stack(C1, C2) stacks C1 on top of C2
- climb(C1) makes the monkey climb on top of C1
- grab banana: grab(B) (if the monkey is at the bananas position)

Transition Model:

- Move: Changes the monkey's position without affecting other elements.
- Push Crate: Changes both the monkey's and the crate's position.
- Stack Crates: Combines two crates into a single stacked entity.
- Unstack Crates: Separates stacked crates into two separate crates.
- Climb: Changes the monkey's height to the height of the crate or stack.
- Grab Bananas: Ends the problem successfully if the monkey is at the correct position and height.

Goal Test: goal is reached if monkey reaches position of banana and grabs it

Path Cost: Nominally assume each action has a cost of 1

3) 3.9a

The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).

a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.

Problem Definition

missionaries := M

cannibals != C

B := boat's position (R or L)

r, l := right and left side of river, respectively

State Representation = (M_l, C_l, B, M_r, C_r)

Initial State: All three missionaries (M) and three cannibals (C) are on the left bank of the river, along with a boat that can hold one or two people.

Goal State: All missionaries and cannibals are on the right bank of the river, with no violations of the constraints.

Constraints and Valid State Conditions

- Missionary Safety Constraint: On either side of the river, the number of missionaries must never be outnumbered by the cannibals (M > 0 \Rightarrow M \geq C).
- Boat Capacity: The boat can carry one or two people at a time.
- Valid Moves: The boat can transport: • One missionary • One cannibal • Two missionaries • Two cannibals • One missionary and one cannibal

Goal Condition:

When (0, 0, R, 3, 3) is reached, the problem is solved.

State Space Representation:

state space can be visualized as a graph with nodes representing valid states and edges representing legal moves.

4) Posing the N-Queens problem as a search problem (Deepak AI, pg 50)

State: a "snapshot" of the board, denoting what is in every square on the board at a given instance. Can be represented in python as a `list[list[int]]`, where the `int` could be 1 if a queen is in square i,j else 0.

State Space: All combintations of states. There would be 2^M states in the state space, where M = total number of squares on the board (NxN).

Initial State: Prior to placing any Queens on the board, the initial state is simply a `list[list[0]]`, i.e. all cells are marked as 0.

Action: the act of placing a queen at some location on the board, say `action(state) = place1At(state, i,j)`, where `place1At(i,j)` simply adds a 1 to position i,j on state.

Transition Model: `result(state, action) = newState`, where `newState` is simply `state` plus an additional 1 somewhere on the board

Goal States: states that satisfy the problem description of N queens placed on an NxN chessboard, such that no queen attacks another, as per chess rules.

Action Cost: Examples in this section of the textbook merely say "assume a cost of 1". But for a more elaborate model of action cost, we can introduce an additional `int` of -1 to `state = list[list[int]]`, where the -1 denotes a cell that is both empty and cannot contain a queen. For example, if we place a queen at cell i,j then all cells horizontal, vertical, and diagonal to i,j would be populated with a -1, as they are now in that queen's attacking range (as per chess rules)

5) Rabbit Leap Problem (Deepak AI, pg 51)

6) Open and Closed lists for DFS and BFS

7) Sudoku

a: State Representation

A state in sudoku search is a partially filled, valid grid where no row or column contains duplicates of nonzero numbers, no 3x3 sub-grid contains duplicates.(1-9) **State Representation**: `set(set[int])` - the grid is defined by nested sets, and `int` shows the value in a given row, col. Use 0 to denote an empty cell.

b: Pseudocode for Successor Function

```
def generate_successors(state):
    """
    Generate a list of valid successor states from the
current Sudoku grid.
    """
    # Find the first empty cell (0)
    for row in range(9):
        for col in range(9):
            if state[row][col] == 0:  # Found an empty cell
                successors = []
                # Try all numbers from 1 to 9
                for num in range(1, 10):
                    if is_valid_move(state, row, col, num):
                        # Create a new state with the number
placed
                        new_state = deepcopy(state)
                        new_state[row][col] = num
                        successors.append(new_state)

                return successors  # Return all valid states
from this first empty cell

    return []  # No empty cell found, meaning the puzzle is
solved

def is_valid_move(state, row, col, num):
    """
    Check if placing 'num' at (row, col) is valid (no
duplicates in row, column, or 3x3 box).
    """
    # Check row
    if duplicated([state[row][c] for c in range(9) if
state[row][c] != 0] + [num]):
        return False

    # Check column
    if duplicated([state[r][col] for r in range(9) if
state[r][col] != 0] + [num]):
        return False

    # Check 3x3 sub-grid
    box_row, box_col = (row // 3) * 3, (col // 3) * 3  # Top-
left corner of the 3x3 box
    box_values = [
        state[r][c]
        for r in range(box_row, box_row + 3)
        for c in range(box_col, box_col + 3)
        if state[r][c] != 0
    ]

    if duplicated(box_values + [num]):
        return False

    return True

def duplicated(values):
    """
    Returns True if there are duplicate non-zero values in
the list.
    """
    return len(values) != len(set(values))
```

c: Pseudocode for goal function

```
def is_goal(state):
    """
    Returns True if the given Sudoku grid is a goal state.
    """
    # Check if all cells are filled (no zeros)
    for row in range(9):
        for col in range(9):
            if state[row][col] == 0:
                return False  # Incomplete board

    # Check rows for uniqueness
    for row in range(9):
        if not is_unique(state[row]):  # Row contains
duplicates
            return False

    # Check columns for uniqueness
    for col in range(9):
        column_values = [state[row][col] for row in range(9)]
        if not is_unique(column_values):  # Column contains
duplicates
            return False

    # Check 3x3 sub-grids for uniqueness
    for box_row in range(0, 9, 3):
        for box_col in range(0, 9, 3):
            box_values = [
                state[r][c]
                for r in range(box_row, box_row + 3)
                for c in range(box_col, box_col + 3)
            ]
            if not is_unique(box_values):  # 3x3 grid
contains duplicates
                return False

    return True  # All constraints satisfied

def is_unique(collection):
    """
    Returns True if the collection contains all digits 1-9
exactly once.
    """
    return sorted(collection) == list(range(1, 10))  # Should
contain exactly {1,2,3,4,5,6,7,8,9}
```

d: find shortest path

81 total cells -28 filled in initially = 53 moves in shortest path

8) DFS and BFS

State Representation: A state here is considered to be the current node. The data in a node consists of it's value and optionally its left and right child. For example, starting at the root of the tree, the state would be a value of 0, with left child being the 1 node and right child being the 4 node. In the code below, the state is represented as the `node` variable.

DFS: 0->1->2->3->4->5->6

BFS: 0->1->4->2->3->5->6

In [ ]:
```
from collections import deque

class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Tree
two = Node(2)
three = Node(3)
five = Node(5)
six = Node(6)
one = Node(1, two, three)
four = Node(4, five, six)
root = Node(0, one, four)

def dfs(root: Node):
    stack = deque([root])

    while stack:
        node = stack.pop()
        print(node.val)
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

def bfs(root: Node):
    que = deque([root])
    while que:
        node = que.popleft()
        print(node.val)
        if node.left:
            que.append(node.left)
        if node.right:
            que.append(node.right)

print("DFS")
dfs(root)
```

```
print("BFS")
bfs(root)
```