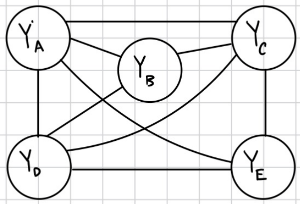
AI HW 2

Jackson Baker
jab132@uark.edu
011029933

September 2025

1 Question I

Q 1



$D(Y_A) = \{10, 20, 30, 40, 50\}$
$D(Y_B) = \{10, 20, 30, 50\}$
$D(Y_C) = \{10, 20, 40, 50\}$
$D(Y_D) = \{10, 30, 40, 50\}$
$D(Y_E) = \{10, 30, 40, 50\}$

(a) The neighbors of Y_D are Y_A, Y_B, Y_C, Y_E

(b) if $Y_A = 10$ and $Y_B = 20$

Y_D could equal $\cancel{10}, 30, 40, 50$

(c) $Y_A = 10$
$Y_B = 20$
$Y_C : \{40, 50\}$
$Y_D : \{30, 40, 50\}$
$Y_E : \{30, 40, 50\}$

(d) If $D(Y_C) = \{20\}$ and $D(Y_E) = \{10\}$

Then
$Y_A : \{30, 40, 50\}$
$Y_B : \{10, 30, 50\}$
$Y_C = 20$
$Y_D : \{30, 40, 50\}$
$Y_E = 10$

(e) If
$Y_A = 10$
$Y_B = 20$
$Y_D = 40$
$Y_E = 50$
Then options are
$Y_C : \{\emptyset\}$

We should backtrack because there are no options for Y_C available that fit the constraints.

2 Question II

2.1 Python Implementation of Sudoku Solver

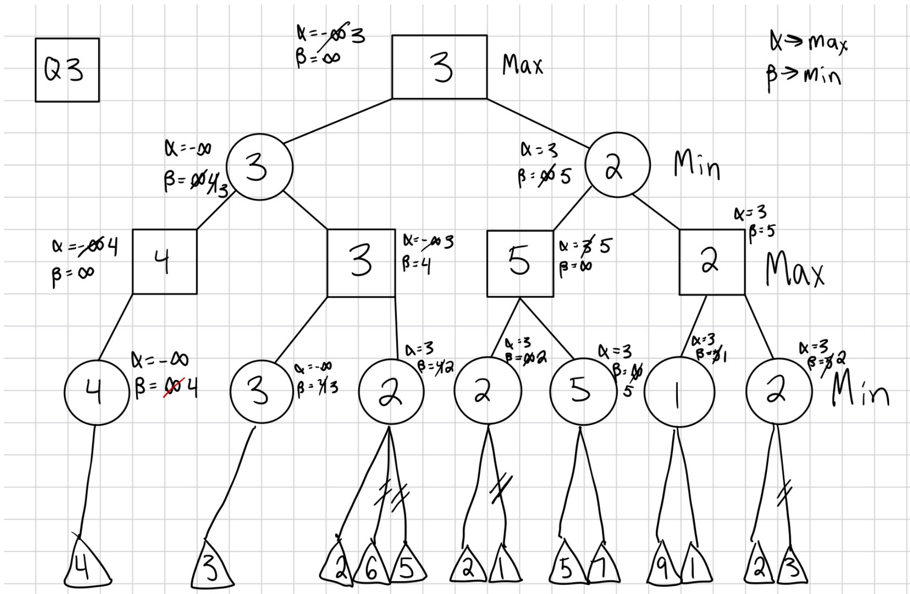
```python
def print_board(board):
    for i in range(len(board)):
        if i % 3 == 0 and i != 0:
            print("-" * 21)
        for j in range(len(board[0])):
            if j % 3 == 0 and j != 0:
                print("|", end=" ")
            if j == 8:
                print(board[i][j])
            else:
                print(str(board[i][j]) + " ", end="")
    print()

def find_empty(board) -> tuple[int, int]:
    for i in range(len(board)):
        try:
            return i, board[i].index(0),
        except:
            continue
    return -1, -1


def is_valid(board, guess, pos) -> bool:
    row, col = pos

    # Check row
    if guess in board[row]:
        return False

    # Check column
    for r in range(9):
        if board[r][col] == guess:
            return False

    # Check grid
    grid_row, grid_col = (row // 3) * 3, (col // 3) * 3

    for r in range(grid_row, grid_row + 3):
        for c in range(grid_col, grid_col + 3):
            if board[r][c] == guess:
                return False

    return True

def solve_sudoku(board) -> bool:
    row, col = find_empty(board)

    if row == -1 and col == -1:
        return True

    for guess in range(1, 10):
        if is_valid(board, guess, (row, col)):
            board[row][col] = guess
```

```

            if solve_sudoku(board):
                return True

            # backtrack
            board[row][col] = 0

    return False


sudoku_board = [
    [0,1,3,0,0,0,7,0,0],
    [0,0,0,5,2,0,4,0,0],
    [0,8,0,0,0,0,0,0,0],
    [0,0,0,0,1,0,0,8,0],
    [9,0,0,0,0,0,6,0,0],
    [2,0,0,0,0,0,0,0,0],
    [0,5,0,4,0,0,0,0,0],
    [7,0,0,6,0,0,0,0,0],
    [0,0,0,0,0,0,0,1,0]
]

another_sudoku_board = [
    [2,6,0,0,0,8,0,0,0],
    [0,8,0,0,0,9,6,0,0],
    [0,0,0,0,5,0,0,0,0],
    [0,9,4,3,0,0,5,0,0],
    [0,0,2,0,7,0,0,0,0],
    [0,5,0,0,0,0,8,0,4],
    [0,3,5,8,0,0,0,0,0],
    [0,0,0,0,0,0,3,0,1],
    [7,0,0,0,6,0,0,0,0]
]

print("Initial Sudoku Board:")
print_board(sudoku_board)

if solve_sudoku(sudoku_board):
    print("Solved Sudoku Board:")
    print_board(sudoku_board)
else:
    print("No solution exists.")
```

2.2 Output

```
Initial Sudoku Board:
0 1 3 | 0 0 0 | 7 0 0
0 0 0 | 5 2 0 | 4 0 0
0 8 0 | 0 0 0 | 0 0 0
---------------------
0 0 0 | 0 1 0 | 0 8 0
9 0 0 | 0 0 0 | 6 0 0
2 0 0 | 0 0 0 | 0 0 0
---------------------
0 5 0 | 4 0 0 | 0 0 0
7 0 0 | 6 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 1 0

Solved Sudoku Board:
5 1 3 | 9 6 4 | 7 2 8
6 9 7 | 5 2 8 | 4 3 1
4 8 2 | 1 3 7 | 5 6 9
---------------------
3 6 5 | 7 1 9 | 2 8 4
9 7 1 | 8 4 2 | 6 5 3
2 4 8 | 3 5 6 | 1 9 7
---------------------
1 5 6 | 4 9 3 | 8 7 2
7 2 9 | 6 8 1 | 3 4 5
8 3 4 | 2 7 5 | 9 1 6
```

3 Question III



4 Question IV

4.1 CornersProblem Implementation

```python
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.startingGameState = startingGameState
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        self.costFn = lambda x: 1
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.cornersEaten = (False, False, False, False)

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        # Check if starting position is already at a corner
        visitedList = [False, False, False, False]
        if self.startingPosition in self.corners:
            index = self.corners.index(self.startingPosition)
            visitedList[index] = True

        # State is (position, tuple of visited corners)
        return (self.startingPosition, tuple(visitedList))

    def isWall(self, state):
        """
        Check if a state position is a wall.
        """
        # Handle both state formats
        if isinstance(state, tuple) and len(state) == 2:
            if isinstance(state[0], tuple):
                # state is ((x, y), visited)
                position, visited = state
```

```python
                x, y = position
            else:
                # state is just (x, y)
                x, y = state

        return True if self.walls[x][y] else False

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        _, visited = state
        return visited == (True, True, True, True)


    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.
        """
        successors = []
        M = self.walls.width
        N = self.walls.height

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            (x, y), visited = state
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)

            # Include ALL positions within bounds (including walls)
            if 0 <= nextx and nextx < M and 0 <= nexty and nexty < N:
                nextPosition = (nextx, nexty)
                visitedList = list(visited)

                # Check if we've reached a corner (only count if it's not a wall)
                if nextPosition in self.corners and not self.walls[nextx][nexty]:
                    index = self.corners.index(nextPosition)
                    visitedList[index] = True

                nextVisited = tuple(visitedList)
                cost = self.costFn(nextPosition)
                successors.append(((nextPosition, nextVisited), action, cost))

        self._expanded += 1
        return successors

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions. This is implemented for you.
```

```
        """
        if actions == None: return 999999 # Pacman doest not move
        return len(actions)
```

4.2 cornerHeuristic Implementation

```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e. it should be
    admissible (as well as consistent).
    """
    corners = problem.corners
    walls = problem.walls

    position, visited = state

    # Get list of unvisited corners
    unvisitedCorners = [corners[i] for i in range(4) if not visited[i]]

    # If all corners visited, heuristic is 0
    if len(unvisitedCorners) == 0:
        return 0

    minDistance = float('inf')
    for corner in unvisitedCorners:
        manhattanDist = abs(position[0] - corner[0]) + abs(position[1] - corner[1])
        if manhattanDist < minDistance:
            minDistance = manhattanDist

    if len(unvisitedCorners) > 1:

        maxCornerDist = 0
        for i in range(len(unvisitedCorners)):
            for j in range(i + 1, len(unvisitedCorners)):
                dist = abs(unvisitedCorners[i][0] - unvisitedCorners[j][0]) + \
                       abs(unvisitedCorners[i][1] - unvisitedCorners[j][1])
                if dist > maxCornerDist:
                    maxCornerDist = dist

        return minDistance + maxCornerDist

    return minDistance
```

4.3 foodHeuristic Implementation

```python
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness. First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    foodList = foodGrid.asList()

    # If no food left, heuristic is 0
    if len(foodList) == 0:
        return 0

    maxDistance = 0
    for food in foodList:
        manhattanDist = abs(position[0] - food[0]) + abs(position[1] - food[1])
        if manhattanDist > maxDistance:
            maxDistance = manhattanDist

```

```
    if len(foodList) > 1:
        maxFoodSpan = 0
        for i in range(len(foodList)):
            for j in range(i + 1, len(foodList)):
                dist = abs(foodList[i][0] - foodList[j][0]) + abs(foodList[i][1] - foodList[j][1])
                if dist > maxFoodSpan:
                    maxFoodSpan = dist

        return max(maxDistance, maxFoodSpan // 2)

    return maxDistance
```

4.4 ClosestDotSearchAgent's findPathToClosestDot Implementation

```python
def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        # Use regular BFS with wall tracking
        # Pass current wall hit count and get updated count back
        if not hasattr(self, 'totalHits'):
            self.totalHits = 0

        actions, hitWalls = search.bfs(problem, initialHit=self.totalHits, returnHit=True)
        self.totalHits += hitWalls
        return actions
```