Name:Coffman,Ethan

Email: ejc012@uark.edu

ID: 100468386

Late Days: 1

Problem 1:

a) The neighbors of YD are all other nodes: Ya, Yb, Yc, Ye
b) {30,40,50}
c) D(Ya) = {10,30,40,50}, D(Yb) = {20,30,50}, D(Yc) = {40,50}, D(Yd) = {30,40,50},D(Ye) = {30,40,50}
d) D(Ya) = {30,40,50}, D(Yb) = {10,30,50}, D{Yd) = {30,40,50}
e) Backtrack, the possible domain of Yc is empty

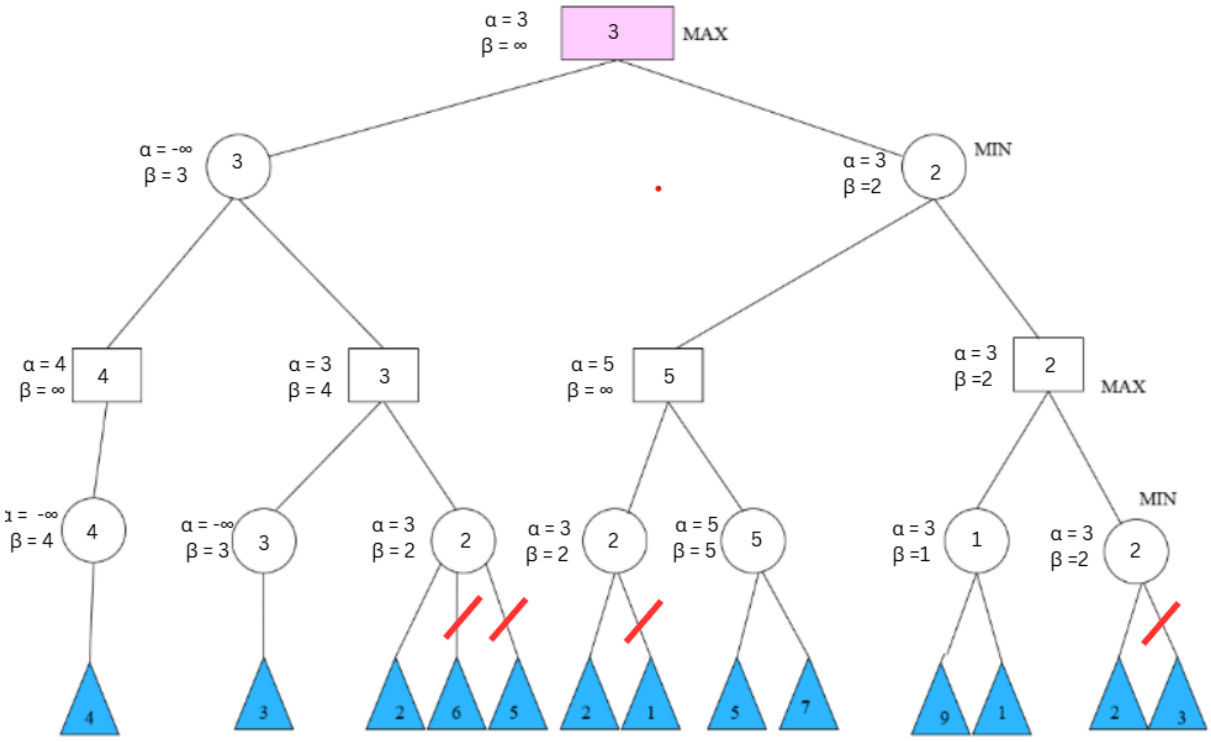
Problem 2:

```
# Function to print the Sudoku board
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 board[i][j] == 0:
                print(".", end=" ")
            else:
                print(board[i][j], end=" ")
        print()
# Find the next empty cell (denoted by 0)
def find_empty(board):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if(board[i][j] == 0):
                return i, j
    return -1, -1
# Check if the current value is valid at the given position
def is_valid(board , num, pos):
    x, y = pos
    if num in board[x]:
```

```
        return False
    for i in range(len(board)):
        if num == board[i][y]:
            return False
    quandrant_x = x // 3 * 3
    quandrant_y = y // 3 * 3
    for i in range(3):
        for j in range(3):
            cur_x = quandrant_x + i
            cur_y = quandrant_y + j
            if num == board[cur_x][cur_y]:
                return False
    return True
# Main backtracking solver
def solve_sudoku(board):
    x,y = find_empty(board)
    if x == -1:
        return True
    for i in range(9):
        num = i + 1
        if(not is_valid(board,num,[x,y])):
            continue
        board[x][y] = num
        if solve_sudoku(board):
            return True
        board[x][y] = 0
    return False
# Sudoku Puzzle (use the image provided to fill in the board)
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],
]
# Solve and print
print("Initial Sudoku Puzzle:")
print_board(sudoku_board)
value = solve_sudoku(sudoku_board)
if value:
```

```
    print("\nSolved Sudoku Puzzle:")
    print_board(sudoku_board)
else:
    print('No solution exists')
```

Problem 3:



Problem 4:

II:

```
    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.initialCorners = [0,0,0,0]

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return (self.startingPosition,self.initialCorners)

    def isWall(self, state):
        "*** YOUR CODE HERE ***"
        x, y = state[0]
        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 ***"
        if(0 in state[1]):
            return False
        else:
            return True



    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
```

```
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        successors = []
        M = self.walls.width
        N = self.walls.height

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:

            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits
a wall:
            #   x, y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   cost = self.costFn((nextx, nexty))
            "*** YOUR CODE HERE ***"
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if 0 <= nextx and nextx < M and 0 <= nexty and nexty < N:
                corners = state[1][:]

                if((nextx,nexty) in self.corners):
                    corners[self.corners.index((nextx,nexty))] = 1

                nextState = ((nextx, nexty), corners)
                cost = self.costFn((nextx, nexty))
                successors.append( ( nextState, action, cost) )

        self._expanded += 1 # DO NOT CHANGE
        return successors
```

```
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> py pacman.py -l tinyCo
rners -p SearchAgent -a fn=bfs,prob=CornersProblem
readCommand argv {argv}
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 18 in 0.2 seconds
Search nodes expanded: 1051
Number of Hitting Walls: 2
Pacman emerges victorious! Score: 522
Average Score: 522.0
Scores:        522.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> py pacman.py -l medium
Corners -p SearchAgent -a fn=bfs,prob=CornersProblem
readCommand argv {argv}
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 77 in 6.0 seconds
Search nodes expanded: 6584
Number of Hitting Walls: 2
Pacman emerges victorious! Score: 463
Average Score: 463.0
Scores:        463.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search>
```

III:

```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 # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"
    from util import manhattanDistance

    # Goal state #
    if problem.isGoalState(state):
        return 0

    else:
```

```
        distancesFromGoals = [] # Calculate all distances from goals(not visited
corners)

        for index,item in enumerate(state[1]):
            if item == 0: # Not visited corner
                # Use manhattan method #

distancesFromGoals.append(manhattanDistance(state[0],corners[index]))

        # Worst case. This guess should be higher than real. Pick higher distance
#
        return max(distancesFromGoals)
```

```
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> py pacman.py -l medium
Corners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
readCommand argv {argv}
[SearchAgent] using function aStarSearch and heuristic cornersHeuristic
[SearchAgent] using problem type CornersProblem
Path found with total cost of 92 in 0.0 seconds
Search nodes expanded: 1080
Number of Hitting Walls: 2
Pacman emerges victorious! Score: 448
Average Score: 448.0
Scores:        448.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search>
```

IV:

```
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()

    "*** YOUR CODE HERE ***"

    from util import manhattanDistance

    problem.heuristicInfo['wallCount'] = problem.walls.count()

    if problem.isGoalState(state):
        return 0

    # Find real distances between position and all of the food #
    distance = []

    for item in foodList:
        distance.append(manhattanDistance(position,item))
    return max(distance)
```

```
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> py pacman.py -l tricky
Search -p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
readCommand argv {argv}
[SearchAgent] using function astar and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 35 in 2.3 seconds
Search nodes expanded: 7619
Number of Hitting Walls: 2
Pacman emerges victorious! Score: 595
Average Score: 595.0
Scores:        595.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search>
```

V:

```python
class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        self.numHits = 0
        while(currentState.getFood().count() > 0):
            nextPathSegment, hits = self.findPathToClosestDot(currentState) # The
missing piece
            self.actions += nextPathSegment
            self.numHits += hits
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal
move: %s!\n%s' % t)
                currentState = currentState.generateSuccessor(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)
        "*** YOUR CODE HERE ***"
        from search import breadthFirstSearch
```

```
        return breadthFirstSearch(problem,self.numHits,True)
```

```
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> python pacman.py -l bi
gSearch -p ClosestDotSearchAgent -z .5
Python was not found; run without arguments to install from the Microsoft Store, or disable this shortcut from Settings
> Apps > Advanced app settings > App execution aliases.
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search> py pacman.py -l bigSea
rch -p ClosestDotSearchAgent -z .5
readCommand argv {argv}
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 348.
Number of Hitting Walls: 2
Pacman emerges victorious! Score: 2362
Average Score: 2362.0
Scores:        2362.0
Win Rate:      1/1 (1.00)
Record:        Win
PS C:\Users\ethan\Documents\School\Artificial_Intelligence\hw2\search (Pacman finds food)\search>
```