Fall 2023 | CS5368 Intelligent Systems | Project 1 Report

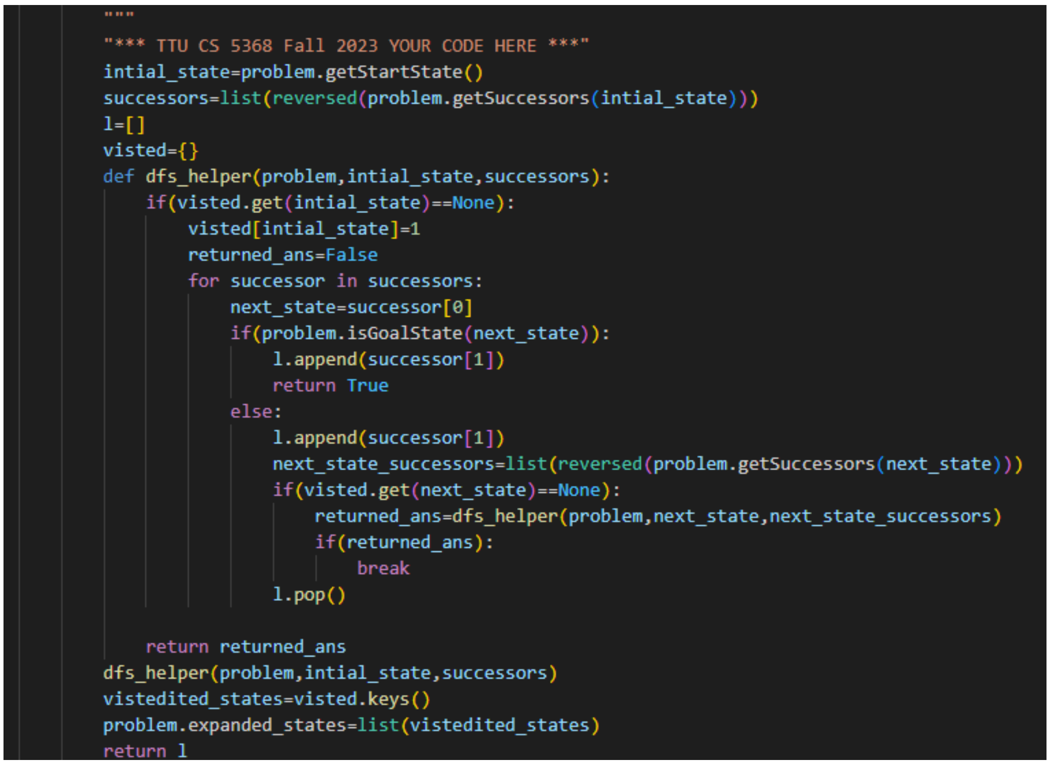
First Name	KanvaKoushik
Last Name	Gopavarapu
Student ID	R11849414

Please answer the following questions and submit them through Blackboard. Be sure to submit it to the Project 0 report submission. DO NOT write the report by hand and submit a scanned version. Just write the answers in a Word document and submit it. Both Word and PDF submissions are accepted.

Basic Questions:

1. **What is your implementation strategy for DFS? Explain**.

   a. With the code attached below, DFS starts with the initial state of the problem. With the help of successors and the function dfs_helper, we implement the recursive function which also checks the visited states or not. The main strategy for the implementation is to recursively explore the search tree while prioritizing deeper exploration before backtracking.

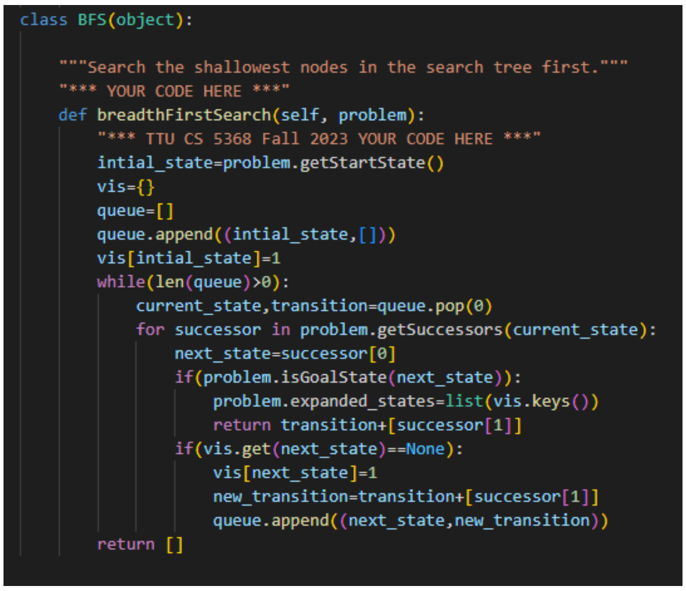


```
"""
"*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
intial_state=problem.getStartState()
successors=list(reversed(problem.getSuccessors(intial_state)))
l=[]
visted={}
def dfs_helper(problem,intial_state,successors):
    if(visted.get(intial_state)==None):
        visted[intial_state]=1
        returned_ans=False
        for successor in successors:
            next_state=successor[0]
            if(problem.isGoalState(next_state)):
                l.append(successor[1])
                return True
            else:
                l.append(successor[1])
                next_state_successors=list(reversed(problem.getSuccessors(next_state)))
                if(visted.get(next_state)==None):
                    returned_ans=dfs_helper(problem,next_state,next_state_successors)
                    if(returned_ans):
                        break
                l.pop()

    return returned_ans
dfs_helper(problem,intial_state,successors)
vistedited_states=visted.keys()
problem.expanded_states=list(vistedited_states)
return l
```

Fig.1. DFS Implementation

2. What is your implementation strategy for BFS? Explain.

a. The BFS implementation is to explore states level by level, starting from the initial place and moving further out. It also ensures that the shallowest nodes are explored first by using a queue to maintain the order of exploration. When the goal is found, the algorithm returns the sequence of actions leading to it.

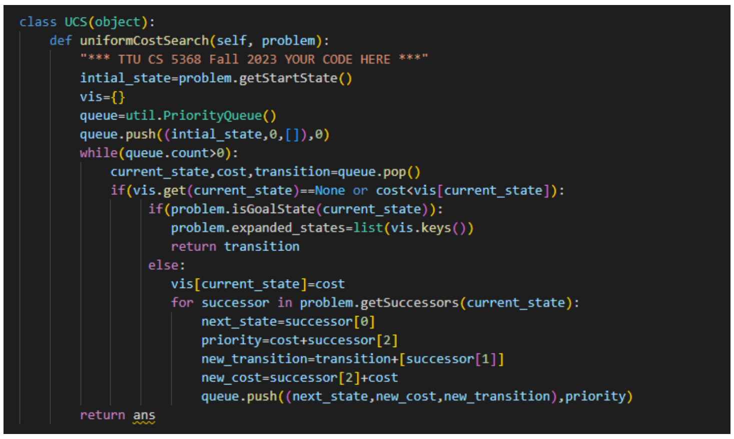


```python
class BFS(object):

    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    def breadthFirstSearch(self, problem):
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        intial_state=problem.getStartState()
        vis={}
        queue=[]
        queue.append((intial_state,[]))
        vis[intial_state]=1
        while(len(queue)>0):
            current_state,transition=queue.pop(0)
            for successor in problem.getSuccessors(current_state):
                next_state=successor[0]
                if(problem.isGoalState(next_state)):
                    problem.expanded_states=list(vis.keys())
                    return transition+[successor[1]]
                if(vis.get(next_state)==None):
                    vis[next_state]=1
                    new_transition=transition+[successor[1]]
                    queue.append((next_state,new_transition))
        return []
```

Fig.2. BFS Implementation

3. What is your implementation strategy for UFS? Explain.

a. The UCS algorithm is for solving a generic search problem. The implementation strategy is to explore the states in order of cost increase. It mainly keeps track of the lower-cost path to each state and prioritizes the states with the lowest cost to be explored first. When the goal is found, the algorithm returns the sequence of actions leading to it.

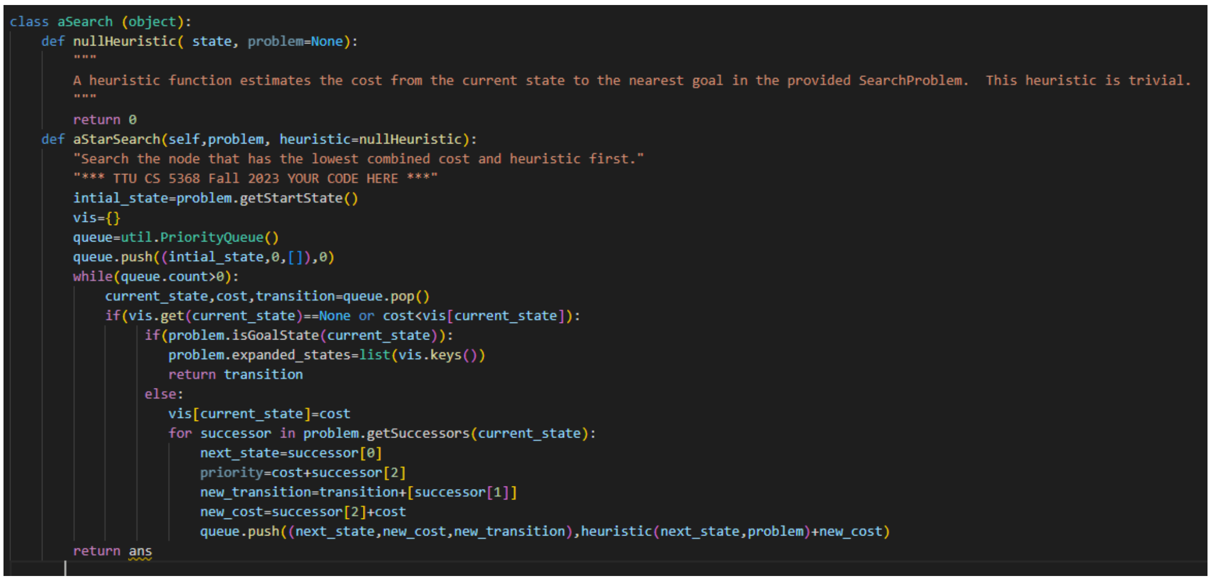


```python
class UCS(object):
    def uniformCostSearch(self, problem):
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        intial_state=problem.getStartState()
        vis={}
        queue=util.PriorityQueue()
        queue.push((intial_state,0,[]),0)
        while(queue.count>0):
            current_state,cost,transition=queue.pop()
            if(vis.get(current_state)==None or cost<vis[current_state]):
                if(problem.isGoalState(current_state)):
                    problem.expanded_states=list(vis.keys())
                    return transition
                else:
                    vis[current_state]=cost
                    for successor in problem.getSuccessors(current_state):
                        next_state=successor[0]
                        priority=cost+successor[2]
                        new_transition=transition+[successor[1]]
                        new_cost=successor[2]+cost
                        queue.push((next_state,new_cost,new_transition),priority)
        return ans
```

Fig.3. UFS Implementation

4. What is your implementation strategy for A*? Explain.

a. A* strategy is implemented to explore the states in order of increasing combined cost and heuristic estimate, prioritizing states that are likely to lead to the goal more efficiently. It makes a queue to maintain the order of priority. When the goal is reached, it returns the sequence of actions.

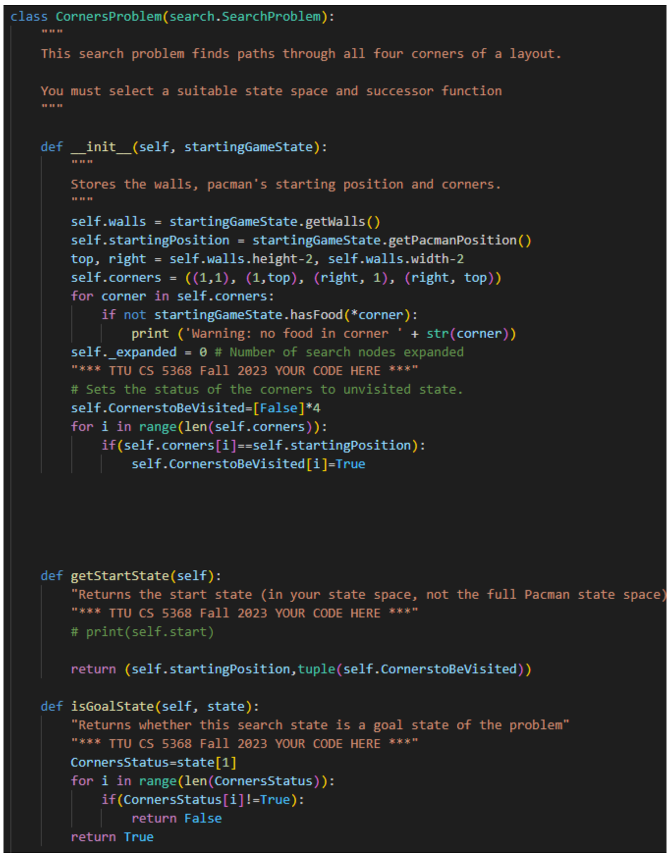


```
class aSearch (object):
    def nullHeuristic( state, problem=None):
        """
        A heuristic function estimates the cost from the current state to the nearest goal in the provided SearchProblem.  This heuristic is trivial.
        """
        return 0
    def aStarSearch(self,problem, heuristic=nullHeuristic):
        "Search the node that has the lowest combined cost and heuristic first."
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        intial_state=problem.getStartState()
        vis={}
        queue=util.PriorityQueue()
        queue.push((intial_state,0,[]),0)
        while(queue.count>0):
            current_state,cost,transition=queue.pop()
            if(vis.get(current_state)==None or cost<vis[current_state]):
                 if(problem.isGoalState(current_state)):
                    problem.expanded_states=list(vis.keys())
                    return transition
                 else:
                    vis[current_state]=cost
                    for successor in problem.getSuccessors(current_state):
                        next_state=successor[0]
                        priority=cost+successor[2]
                        new_transition=transition+[successor[1]]
                        new_cost=successor[2]+cost
                        queue.push((next_state,new_cost,new_transition),heuristic(next_state,problem)+new_cost)
        return ans
```

Fig.4. A search Implementation*

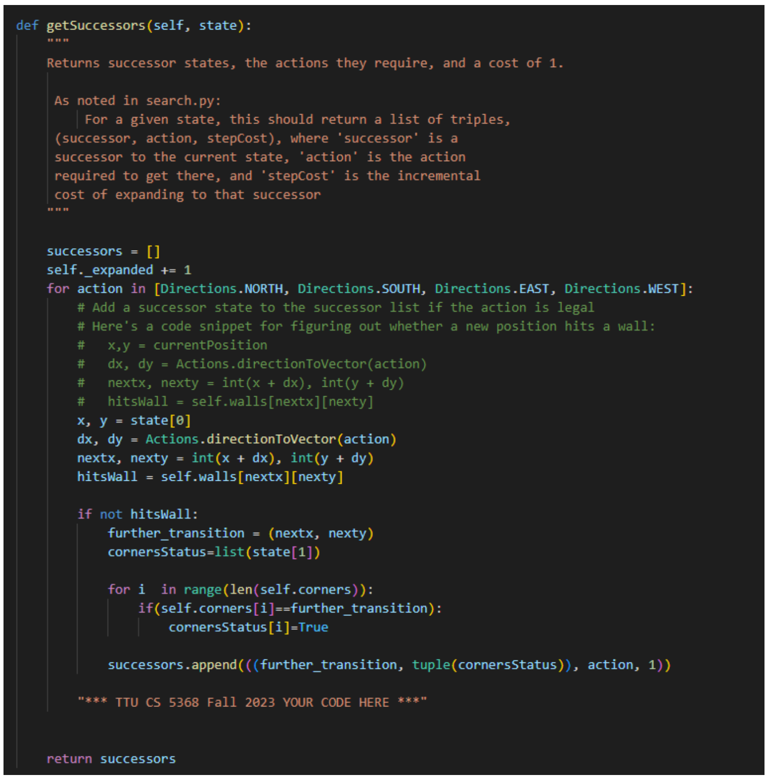
5. Explain your implementation strategy for the Corners Problem. What is your state representation?

a. The below code defines a search problem where Pac-Man needs to visit all four corners of a layout while avoiding walls. The state space includes Pac-Man's position and the status of the corners. The main goal is to visit all the corners, and the function successor generates the states. This uses A* search to find all the optimal paths to visit all the corners. The state representation is a tuple ` (current_position, corners_status) ` where the x-coordinate gives us the position of the Pac-Man and the Y-coordinate gives us the Boolean position.



```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.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 # Number of search nodes expanded
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        # Sets the status of the corners to unvisited state.
        self.CornerstoBeVisited=[False]*4
        for i in range(len(self.corners)):
            if(self.corners[i]==self.startingPosition):
                self.CornerstoBeVisited[i]=True




    def getStartState(self):
        "Returns the start state (in your state space, not the full Pacman state space)
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        # print(self.start)

        return (self.startingPosition,tuple(self.CornerstoBeVisited))

    def isGoalState(self, state):
        "Returns whether this search state is a goal state of the problem"
        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"
        CornersStatus=state[1]
        for i in range(len(CornersStatus)):
            if(CornersStatus[i]!=True):
                return False
        return True
```

Fig.5.1. CornerProblem Implementation

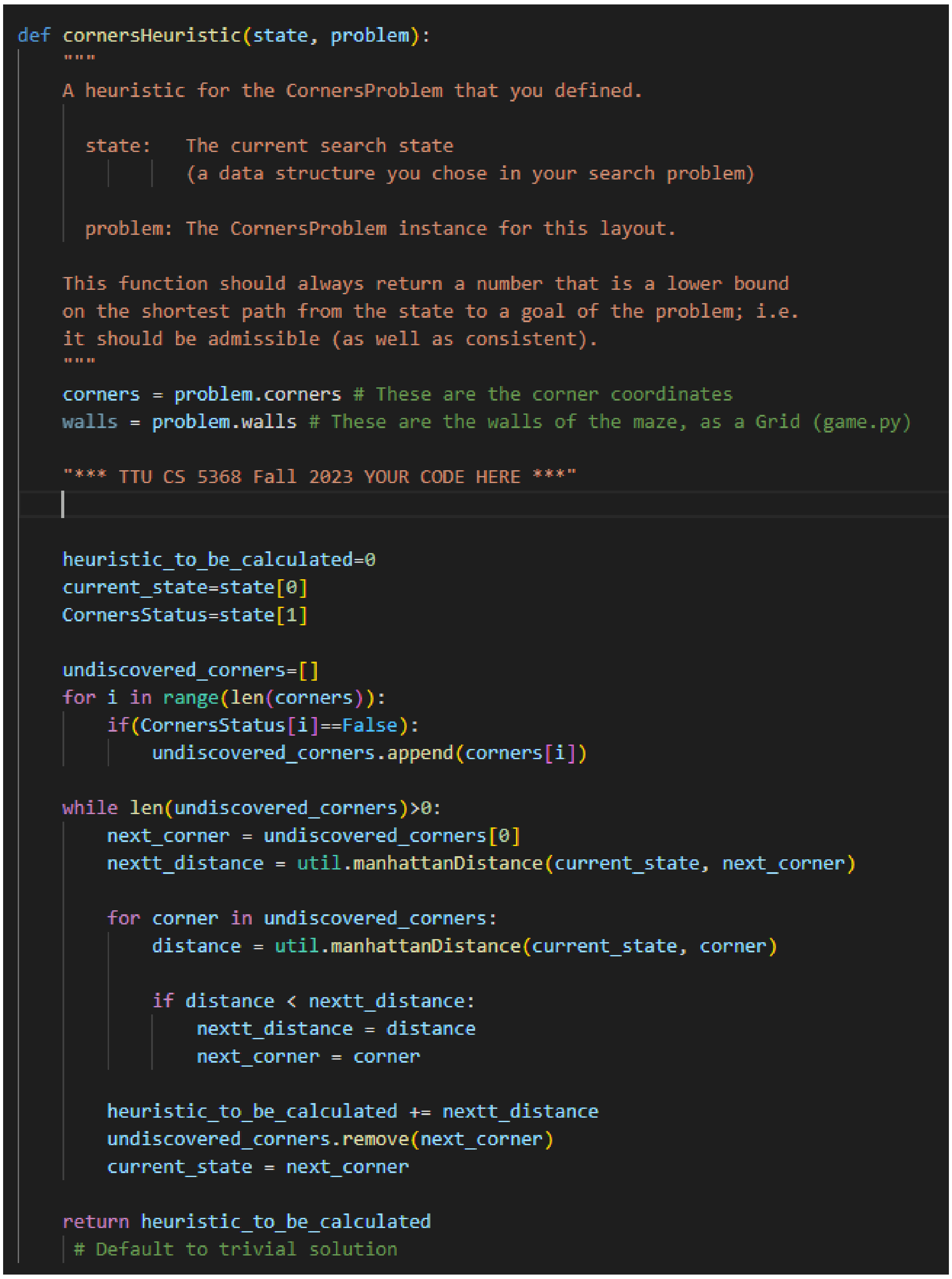


```python
def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     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 = []
    self._expanded += 1
    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)
        #   hitsWall = self.walls[nextx][nexty]
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]

        if not hitsWall:
            further_transition = (nextx, nexty)
            cornersStatus=list(state[1])

            for i  in range(len(self.corners)):
                if(self.corners[i]==further_transition):
                    cornersStatus[i]=True

            successors.append(((further_transition, tuple(cornersStatus)), action, 1))

        "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"


    return successors
```

Fig.5.2. CornerProblem Implementation

6. What are your heuristics for the Corners Problem? How did you implement it?

a. The Heuristics used are Manhattan distance to the nearest unvisited corner along with admissible and consistent. These heuristic guides search algorithms to explore states that are likely to lead to efficient pathfinding in the problem. It prioritizes search algorithms like A* to explore the states that are closer to undiscovered corners.

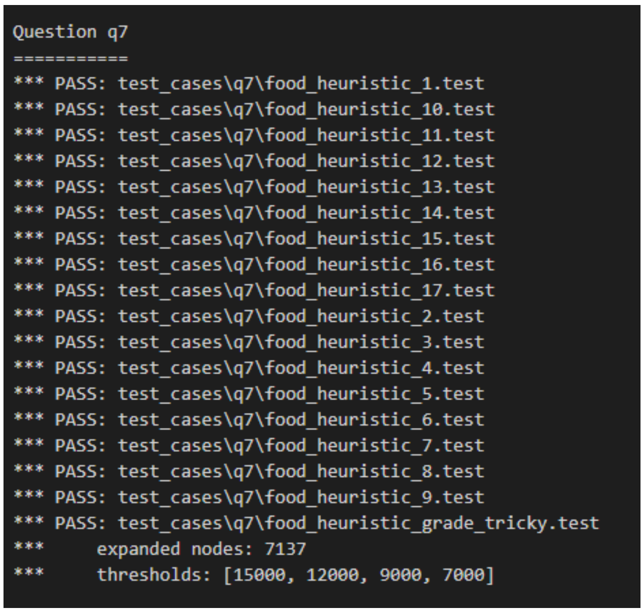


```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      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)

    "*** TTU CS 5368 Fall 2023 YOUR CODE HERE ***"


    heuristic_to_be_calculated=0
    current_state=state[0]
    CornersStatus=state[1]

    undiscovered_corners=[]
    for i in range(len(corners)):
        if(CornersStatus[i]==False):
            undiscovered_corners.append(corners[i])

    while len(undiscovered_corners)>0:
        next_corner = undiscovered_corners[0]
        nextt_distance = util.manhattanDistance(current_state, next_corner)

        for corner in undiscovered_corners:
            distance = util.manhattanDistance(current_state, corner)

            if distance < nextt_distance:
                nextt_distance = distance
                next_corner = corner

        heuristic_to_be_calculated += nextt_distance
        undiscovered_corners.remove(next_corner)
        current_state = next_corner

    return heuristic_to_be_calculated
    # Default to trivial solution
```

Fig.6. Heuristic for CornerProblem Implementation

Advanced Questions:

1. **Based on your implementation of DFS, what happens if you fail to push all successors to the fringe (see Get Successors**
   a. If we fail to push all the successors to the fringe, it can lead to incomplete and inaccurate results which in turn could overflow the stack due to excess memory usage. So, failing to push all the successors violates the DFS and can lead to untrustworthy search results.
2. **Does BFS give fewer actions to reach the goal compared to DFS? Explain.**
   a. Yes, BFS is more likely to find a shorter path to the goal compared to DFS as it explores all paths at each level while giving us the shortest path possible with uniform costs. It also depends on the problem we are facing or trying to solve and within which state of search it is encountered.
3. **Many paths: You have expanded some paths in all search algorithms, which of these algorithms has the least expansion, and why?**
   a. In my opinion, BFS has had optimal search results compared to other algorithms. It explores in a breadth-first manner moving from the current depth before exploring the next deeper levels.
4. **Explain how the implementation of A* differs from UCS.**
   a. A* uses heuristics while prioritizing the nodes whereas UCS relies on the cost of the path and explores the nodes uniformly.
5. **Node expansion: How many nodes were expanded in Question 7?**
   a. In the food heuristic problem, 7137 nodes were expanded from implementation.



```
Question q7
===========
*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 7137
***     thresholds: [15000, 12000, 9000, 7000]
```

Fig.7. Food Heuristic expanded nodes.