Question 1 DFS

Here I implemented a straightforward Depth-First Search (DFS) algorithm, a technique I've applied repeatedly in my CPT and Algorithms courses. util.Stack() and the other provided data structures were self explanatory. The code provided below includes detailed comments to clearly illustrate the implementation and its structure.

```python
# use a stack for LIFO
fringe = util.Stack()
visited = set() # easy visited set to check
# we are pusing (state, actions) so we can see actions as list of moves
fringe.push((problem.getStartState(), []))
# so here while we still have actions in the fringe pop them
while not fringe.isEmpty():
    state, actions = fringe.pop()

    # skip if in set visited
    if state in visited:
        continue

    # then mark as visited so in next while loop iteration we skip
    visited.add(state)

    # just check simple isGoalState(state) with bool output
    if problem.isGoalState(state):
        return actions

    # so for each successor/action we haven't seen just add to fringe to be explored
    # this is the DFS element basically we are looking DEPTH
    for successor, action, stepCost in problem.getSuccessors(state):
        if successor not in visited:
            fringe.push((successor, actions + [action]))

# no solution
return []
```

Question 2 BFS

Here is the BFS implementation. It is almost identical. We just switch our data structure and go from there and the entire implementation changes from depth or one state path to breadth or all the adjacent state paths.

Again image provided and comments do most explaining:

```python
def breadthFirstSearch(problem: SearchProblem) -> List[Directions]:
    """Search the shallowest nodes in the search tree first."""
    # use queue util for FIFO
    fringe = util.Queue()
    # same thing as before set for state management
    visited = set()
    fringe.push((problem.getStartState(), []))

    while not fringe.isEmpty():
        state, actions = fringe.pop()

        # same
        if state in visited:
            continue

        # same
        visited.add(state)

        # same
        if problem.isGoalState(state):
            return actions

        # same just fifo
        for successor, action, stepCost in problem.getSuccessors(state):
            if successor not in visited:
                fringe.push((successor, actions + [action]))

    return []
```

Question 3 Varying Cost Function

(this is just UCS)

This one is slightly different from the previous two. We do the same thing by swapping our data structure, but the priority queue is different because we have a dynamic least cost element we pop

```python
def uniformCostSearch(problem: SearchProblem) -> List[Directions]:
    """Search the node of least total cost first."""
    # use priority queue just so we can get that least cost
    fringe = util.PriorityQueue()
    # same thing here just another set to prevent duplicate states
    visited = set()
    # make a simple base case with parameter: (state, actions, cost)
    fringe.push((problem.getStartState(), [], 0), 0)

    while not fringe.isEmpty():
        state, actions, cost = fringe.pop()

        # again, skip if already in visited set
        if state in visited:
            continue

        # again add the state to visited set, self explanatory by now
        visited.add(state)

        # same
        if problem.isGoalState(state):
            return actions

        # This is really similar but it's our main difference between BFS/DFS implementations
        for successor, action, stepCost in problem.getSuccessors(state):
            if successor not in visited:
                new_cost = cost + stepCost
                # /\/\/\/\/\ above you can see for each unvisited successor we calculate the cumulative cost and add to PQ
                # NOW PQ ensures that the successor with the lowest total cost is expanded next
                fringe.push((successor, actions + [action], new_cost), new_cost)

    return []
```

from the structure. We calculate our cumulative cost as shown in the code for this:

Question 4 A*

This assignment really helped me see how these search algorithms can be implemented so similarly. I was able to reuse a lot of code. Here for A* I just had to make some adjustments, the priority was the hardest part.

I QUERIED AN LLM HERE TO ASK ABOUT PRIORITY IN A*

```python
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic) -> List[Directions]:
    """Search the node that has the lowest combined cost and heuristic first."""
    # we have another PQ but we use a different function than UCS which is : f = g + h
    fringe = util.PriorityQueue()
    start_state = problem.getStartState()

    # set our initial cost as the init state which is always 0, no cost to start!
    best_cost = {start_state: 0}

    # push the start state with priority = 0 because again no cost to start, but look we use heuristic here !
    fringe.push((start_state, [], 0), heuristic(start_state, problem))

    while not fringe.isEmpty():
        # so now we look in our PQ and take that top least cost based on our heuristic
        # starting its just the base case, but as we iterate it becomes the least cost heuristic element
        state, actions, cost = fringe.pop()

        # if or current best_cost is better than the popped cost, we just continue, best_cost is locally optimized for this iteration
        if state in best_cost and cost > best_cost[state]:
            continue

        # again just check for goal
        if problem.isGoalState(state):
            return actions

        # same as UCS for this first part just remember popped element is based on heuristic not FIFO/LIFO
        for successor, action, stepCost in problem.getSuccessors(state):
            new_cost = cost + stepCost

            # now just check if a better path to the successor, update best_cost and push to fringe
            if successor not in best_cost or new_cost < best_cost[successor]:
                best_cost[successor] = new_cost # see we update our best_cost here, when optimized we continue through the loop until we meet goal
                priority = new_cost + heuristic(successor, problem) # then the priority for that fring push becomes cost + heuristic
                fringe.push((successor, actions + [action], new_cost), priority) # simple push to PQ

    return []
```

Question 5 Finding all the Corners

Very easy first two code edits here:

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    # Just a frozenset for Pacman no positions or corners
    return (self.startingPosition, frozenset())

def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
    position, visitedCorners = state
    # goal achieved when the number of visited corners equals the total number of corners
    return len(visitedCorners) == len(self.corners)
```

The next part was again very straightforward. The code basically iterates over all four possible movement directions, checking if each move is legal, and updates the visited corners if a new corner is reached. Then it creates and returns a list of successor tuples containing the new state, action taken, and a step cost of 1. Most if it was already written so the comments here should explain the code implemented

```
def getSuccessors(self, state: Any):
    """
    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 = []
    # use the current state and retreive the current position and the set of visited corners
    position, visitedCorners = state
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        x, y = position
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextPosition = (nextx, nexty)
            # copy the current visited corners and if nextPosition is an unvisited corner, update the set
            newVisitedCorners = visitedCorners
            if nextPosition in self.corners and nextPosition not in visitedCorners:
                newVisitedCorners = visitedCorners.union({nextPosition})
            # finally append the successor state along with the action taken and a cost of 1
            successors.append(((nextPosition, newVisitedCorners), action, 1))

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

def getCostOfActions(self, actions):
```

Question 6 Corners Heuristic

The corners heuristic was simple using problem.corners to append visited and return 0 if we have no visited states. You can then see the heuristic was set to 0 initially past the default return case and from there we use a copy of our unvisited corners and build a heuristic and iterate over remaining to calculate the heuristic using manhattan distance from each corner. You can see the code with comments:

```python
def cornersHeuristic(state: Any, problem: CornersProblem):
    """
    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.
    """
    position, visitedCorners = state
    # make our list of corners that have not been reached yet
    unvisited = []
    for corner in problem.corners:
        if corner not in visitedCorners:
            unvisited.append(corner)
    if not unvisited:
        return 0 # Default trivial solution

    heuristic = 0 # start at 0 and current position
    currentPos = position
    remaining = unvisited[:]  # make a copy of unvisited corners

    #  add the distance to the closest remaining corner
    while remaining:
        distances = [];
        # calculate manhattan distance from currentPos to each corner in remaining
        for corner in remaining: distances.append((abs(currentPos[0] - corner[0]) + abs(currentPos[1] - corner[1]), corner))

        dist, closest = min(distances)   # find the closest corner
        heuristic += dist # add its distance to the heuristic
        currentPos = closest # update currentPos
        remaining.remove(closest) # remove that corner from remaining.

    return heuristic
```

Question 7 Eating All the Dots

This was by far the most complex code. It has complex logic using MST's and I used the mazeDistance for the heuristic. mazeDistance was very simple to work with, however it was difficult to make my MST structure work with key pairs for our nodes.

I QUERIED AN LLM HERE TO ASK ABOUT MEETING BONUS REQUIREMENTS
-> The LLM said to look for a different heuristic and I found mazeDistance. I came up with the MST on my own from algo/CPT.

Here is the code with ample comments:

THIS SECTION IS JUST THE HEADER:

```
class AStarFoodSearchAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)
        self.searchType = FoodSearchProblem

def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
    """
    Your heuristic for the FoodSearchProblem goes here.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your search may have a but our your heuristic is not admissible!  On the
    other hand, inadmissible 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 not foodList:
        return 0
```

CODE I EDITED:

```python
#  find the maze distance from the current position to the closest food or min of distances
#  mazeDistance was how I was able to solve this problem much better
start_to_food = min(mazeDistance(position, food, problem.startingGameState) for food in foodList)

# compute MST cost among food dots with our maze distance
# MST's are great resources for problems like this
# this was my only implementation to get the Bonus correct
mst_cost = 0
if len(foodList) > 1:
    nodes = list(foodList) # first convert foodList to list of our nodes for tree
    mst_set = {nodes[0]}     # then take MST with the first food dot
    remaining = set(nodes[1:])  # use the remaining food dots to connect

    # These next lines contain the most complex logic from the entire project and took me most of my time
    # The loop first finds the closest dot to the current MST set using maze distances
    # then we add the minimal distance to the MST cost, update the MST set by including that dot,
    # and remove it from the remaining set

    while remaining:

        # min_edge stores the smallest distance found, we set to infinite when adding
        # best_node tracks the corresponding node.
        min_edge = float('inf')
        best_node = None
         # loop over each node n1 already in the MST
        for n1 in mst_set:
            # for each node n2 not yet in the MST we then make keys
            for n2 in remaining:
                # we are making keys here for n1, n2 to see if their distance has been computed before
                key = tuple(sorted([n1, n2]))
                # if the distance is cached use it otherwise compute the maze distance and cache it
                if key in problem.heuristicInfo:
                    d = problem.heuristicInfo[key]
                else:
                    d = mazeDistance(n1, n2, problem.startingGameState)
                    problem.heuristicInfo[key] = d
                if d < min_edge:
                    min_edge = d
                    best_node = n2
        # add the smallest found distance to MST total cost
        mst_cost += min_edge
        mst_set.add(best_node)
        #remove added node
        remaining.remove(best_node)

return start_to_food + mst_cost
```

Question 8 Suboptimal Search

Simple one liner method add:

```
class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # just put in this line
            self.actions += nextPathSegment
            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: pacman.GameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        problem = AnyFoodSearchProblem(gameState)
        return search.bfs(problem)
```

Resources used:

https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
https://www.geeksforgeeks.org/heuristic-function-in-ai/
https://stackoverflow.com/questions/18756669/how-to-determine-a-heuristic-for-an-algorithm-say-a-is-a-good-one