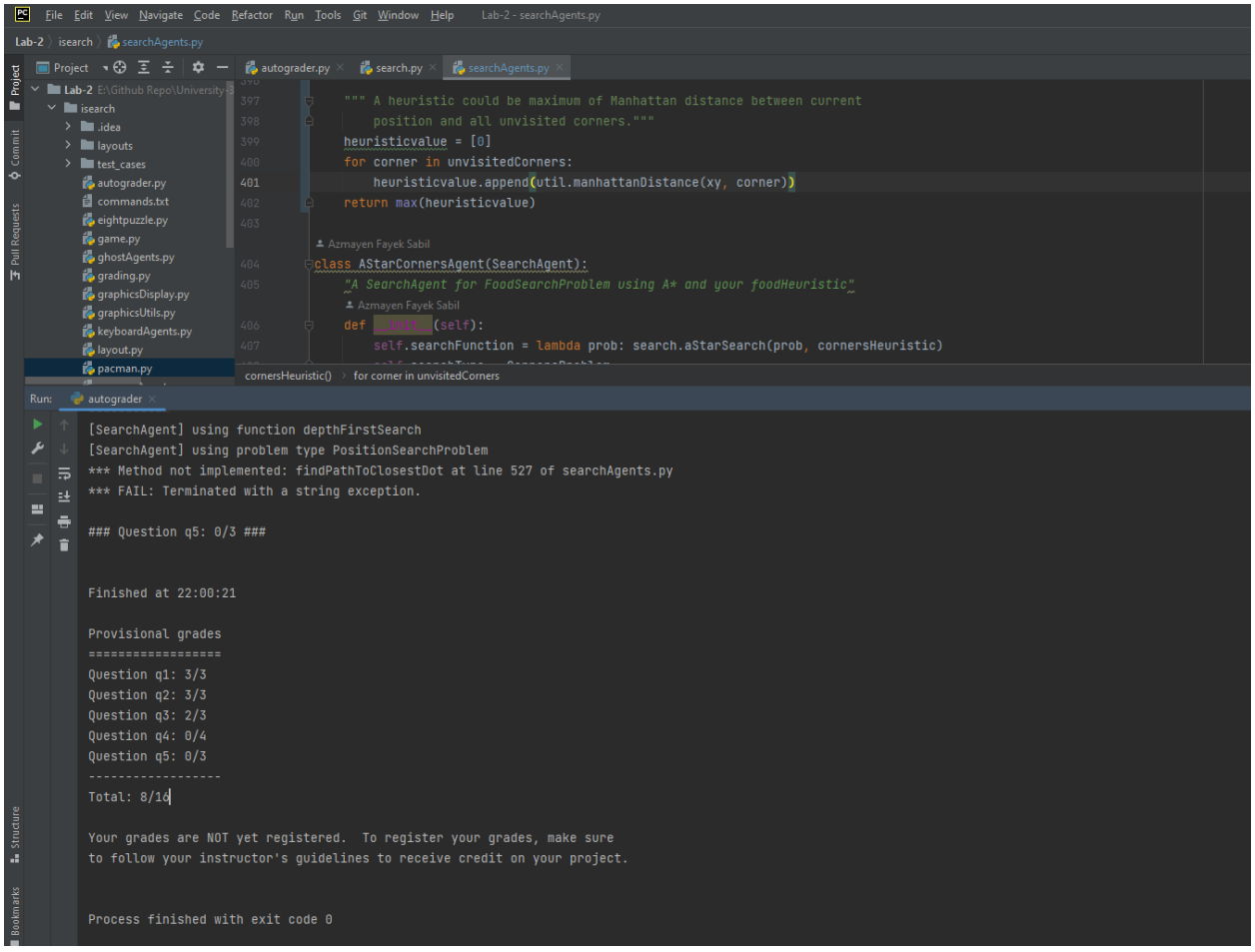
Autograder's Result



After running the autograder my code passed for question 1, 2 and partially for question 3. I couldn't get full grades on my question 3.

QUESTION-1

In this question I had to implement A* search. We have to keep track of both heuristic value and cost of the path to get the best result. We maintained a fringe and a visited array.

```
Azmayen Fayek Sabil
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"

    fringe = util.PriorityQueue()
    h_value = heuristic(problem.getStartState(), problem)
    visited = []

    fringe.push((problem.getStartState(), [], 0), h_value)

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

        if problem.isGoalState(current):
            return actions

        if not current in visited:
            visited.append(current)

            for nextState, action, cost in problem.getSuccessors(current):
                if not nextState in visited:
                    actionList = list(actions)
                    actionList += [action]
                    costActions = problem.getCostOfActions(actionList)
                    get_heuristic = heuristic(nextState, problem)
                    fringe.push((nextState, actionList, 1), costActions + get_heuristic)

    return []

    #util.raiseNotDefined()
```

QUESTION-2

For this question we had to complete a few of the functions. It was easy, just read the function name and returned the desirable value.

```
Azmayen Fayek Sabil
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    return (self.startingPosition, [])
    #util.raiseNotDefined()
```

Here we returned a value that says if we have reached the corner portions or not.

```
Azmayen Fayek Sabil *
def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    pos, visited_corners = state
    return all(corner in visited_corners for corner in self.corners)
    #util.raiseNotDefined()
```

This one was a bit tricky. Here we had to return the successors for a cost and an action. So we had to iterate through every direction and returned from there what actions we were able to perform and associated cost with that.

```
Azmayen Fayek Sabil *
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
    """
    pos, visited_corners = state
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        x, y = pos
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            next_pos = (nextx, nexty)
            next_visited_corners = visited_corners[:]
            if next_pos in self.corners and next_pos not in next_visited_corners:
                next_visited_corners.append(next_pos)
            successors.append(((next_pos, next_visited_corners), action, 1))
    self._expanded += 1
    return successors
```

Here we returned the total cost of a series of actions.

```
Azmayen Fayek Sabil *
def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999.  This is implemented for you.
    """
    if actions == None: return 999999
    x, y = self.startingPosition
    for action in actions:
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]: return 999999
    return len(actions)
    # if actions == None: return 999999
    # x,y= self.startingPosition
    # for action in actions:
    #     dx, dy = Actions.directionToVector(action)
    #     x, y = int(x + dx), int(y + dy)
    #     if self.walls[x][y]: return 999999
    # return len(actions)
```

QUESTION-3

Here we are calculating the heuristic value which is admissible and consistent at the same time.

```
Azmayen Fayek Sabil *
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 ***"
    # return 0 # Default to trivial solution
    xy = state[0]
    visitedCorners = state[1]
    # Finding out the not visited corners
    unvisitedCorners = []
    for corner in corners:
        if not (corner in visitedCorners):
            unvisitedCorners.append(corner)

    """ A heuristic could be maximum of Manhattan distance between current
        position and all unvisited corners."""
    heuristicvalue = [0]
    for corner in unvisitedCorners:
        heuristicvalue.append(util.manhattanDistance(xy, corner))
    return max(heuristicvalue)
```