ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

Lab Report 2

CSE 4712

Md. Bakhtiar Hasan

24th September, 2022

## Question 01

The first problem requires us to implement the A\* graph search algorithm. This is simple to implement, since it builds upon the Uniform Cost Search algorithm we had implemented in the previous lab. The only new part is the use of a heuristic, but this problem does not require us to create our own heuristic. Instead, we use a provided heuristic function.

To recap the way in which the Uniform Cost Search algorithm was implemented, we used a priority queue to hold state values that we wished to explore. Initially, this queue only had the starting state. Each state was a tuple of our position in that state and the list of actions which brough us to the state. The initial list of actions was an empty list. Along with the state, the priority queue also stored a cost associated with moving to the state. The cost for the initial state was 0.

After this, we expanded the states in the queue one by one, popping them in order of least cost. For each popped state, we first checked if this was the goal state. If it was not, we retrieved the successors and added each successor in turn to the queue. We appended the action associated with reaching the successor to the list of actions when pushing it to the queue. The cost was the sum of the cumulative cost of reaching the current state and the cost associated with moving to the successor. This process was repeated until we either ran out of states to expand or reached the goal state.

The only difference to the above mechanism when implementing the A\* graph search algorithm is the cost. The cost as described above is called the backward cost in this algorithm. In addition to this, we also have the forward cost, which is the cost provided by the heuristic for this state. When pushing a state to the queue, the cost associated with the state is the sum of the forward and backward cost for the state.

def aStarSearch(problem, heuristic=nullHeuristic):  
 action\_list = []  
 visited = []  
 start\_state = (problem.getStartState(), action\_list)  
 forward\_cost = heuristic(problem.getStartState(), problem)  
   
 fringe = util.PriorityQueue()  
 fringe.push(start\_state, 0 + forward\_cost)  
  
 while not fringe.isEmpty():  
 current\_state, action\_list = fringe.pop()  
   
 if problem.isGoalState(current\_state):  
 return action\_list  
   
 if current\_state not in visited:  
 visited.append(current\_state)  
 successors = problem.getSuccessors(current\_state)  
 for successor, action, cost in successors:  
 next\_state = (successor, action\_list + [action])  
 backward\_cost = problem.getCostOfActions(action\_list) + cost  
 forward\_cost = heuristic(successor, problem)  
 fringe.push(next\_state, backward\_cost + forward\_cost)  
 return None

PYTHON

## Question 02

The second problem requires us to implement the CornersProblem class, which is a special implementation of the SearchProblem class. This was implemented for us, but the code is being examined, nonetheless.

The problem involves visiting each of the corners of a provided map, regardless of whether the corner has food in it or not. To start off, we need to keep track of unvisited corners. To do this, a dictionary is added to the \_\_init\_\_ method of the class, which initially is set to False for all of the corners. Note that a portion of the method that initialized other variables is not shown since that section was provided in the question.

def \_\_init\_\_(*self*, startingGameState):  
 *self*.visitedCorners = {}  
 for corner in *self*.corners:  
 *self*.visitedCorners[corner] = False

PYTHON

Four methods of the SearchProblem class have to be overridden and modified to fit the new problem. The first of these is the getStartState method. For our problem, we require this method to return a tuple of the coordinates of the starting position and the list of visited corners.

def getStartState(*self*):  
 startState = (*self*.startingPosition, *self*.visitedCorners)  
 return startState

PYTHON

The second overridden method is isGoalState, which checks if the current state is one of the goal states. This method loops over the list of visited corners and checks if any of them are False, meaning they are unvisited. If there is even one such corner, the method returns False. Otherwise, it returns True.

def isGoalState(*self*, state):  
 \_, visitedCorners = state  
 for corner in *self*.corners:  
 if visitedCorners[corner] == False:  
 return False  
 return True

PYTHON

We also need to override the getSuccessors method, which returns a list of successors. In this method, for each possible action, we get the new position that will be reached for that action from our current position. If this new position is not a wall, it is added to the list of successors. This list is returned by the method.

def getSuccessors(*self*, state):  
 successors = []  
 for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:  
 currentPosition, visitedCorners = state  
 x, y = currentPosition  
 dx, dy = Actions().directionToVector(action)  
 nextX, nextY = int(x + dx), int(y + dy)  
 nextVisitedCorners = deepcopy(visitedCorners)  
 if not *self*.walls[nextX][nextY]:  
 nextPosition = (nextX, nextY)  
 if nextPosition in *self*.corners:  
 nextVisitedCorners[nextPosition] = True  
 nextState = (nextPosition, nextVisitedCorners)  
 successors.append((nextState, action, 1))  
  
 *self*.\_expanded += 1 # DO NOT CHANGE  
 return successors

PYTHON

The final method that must be overridden is getCostOfActions, which returns the total cost of a list of actions. This method was provided along with the question. In our problem, we are assuming that each action has a cost of 1, so the total cost is the length of the list. There is one small caveat which requires us to return the value 999999 if the list happens to include an illegal action, i.e., one which places us at the position of a wall.

def getCostOfActions(*self*, 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)

PYTHON

## Question 03

The third problem requires us to implement a heuristic for the Corners Problem described above. The heuristic is required to be admissible as well as consistent. The straightforward answer to such a requirement is to use the Manhattan Distance metric, which leads to a successful result.

To implement this metric, we must loop over the unvisited corners and return the distance between our current position and the corner which is furthest from us. This part is a little counterintuitive. Initially, the distance to the corner which was nearest was chosen. Although this meets the criteria of being admissible and consistent, we fail to achieve a satisfactory result due to poor performance. The performance improves if we manage to provide a heuristic which gives us values that are closer to the true cost. The true cost in this case is the total cost of visiting all the corners. As such, the distance to the furthest corner will give us a value closer to the true cost than the distance to the closest corner.

def cornersHeuristic(state, problem):  
 position, corners = state  
 corners = [corner for corner in corners if corners[corner] == False]  
 max\_cost = None  
   
 for corner in corners:  
 cost = util.manhattanDistance(position, corner)  
 if max\_cost == None or cost > max\_cost:  
 max\_cost = cost  
 return 0 if max\_cost == None else max\_cost

PYTHON

## Question 04

The fourth problem requires us to implement an admissible and consistent heuristic for the problem of eating all the dots on the board. The initial idea behind this was to use the exact same algorithm as the previous problem. We loop over all the positions on the board with remaining dots and return the Manhattan distance to the dot which is furthest way from us. This does work but fails to succeed for all the test cases due to poor performance. Too many nodes are being expanded.

A hint in the comments provided above the code leads to using a new method defined at the bottom of the file, mazeDistance. Using this distance as the metric instead of the Manhattan distance leads to a successful result.

def foodHeuristic(state, problem):  
 position, foodGrid = state

max\_cost = None  
 for food in foodGrid.asList():  
 cost = mazeDistance(position, food, problem.startingGameState)  
 if max\_cost == None or cost > max\_cost:  
 max\_cost = cost  
 return 0 if max\_cost == None else max\_cost

PYTHON

A quick examination of the mazeDistance method reveals that it takes the walls of the problem into account, which is how it can determine a distance metric closer to the actual cost. However, it does this by utilizing the Breadth First Search algorithm. This metric is essentially solving a part of the total problem to achieve a better heuristic. It can be argued that this tricks the grader, which does not consider the nodes being expanded when using Breadth First Search in this section. Using a search algorithm when calculating a heuristic will also have a large computation cost associated with it. As such, the benefits of using this metric are debatable.

## Question 05

The last problem is relatively simpler since it does not put any requirements on us for optimality. We are only required to find a path to the closest dot. This can be achieved using Breadth First Search.

def findPathToClosestDot(*self*, gameState):  
 problem = AnyFoodSearchProblem(gameState)  
 return search.bfs(problem)

PYTHON

To be able to do this, a new problem type was defined for us in the class AnyFoodSearchProblem. This class, however, has an unimplemented method to check for the goal state. The goal state in this case is our current position being the position of the nearest dot, which is the same as saying our current position has a dot at all. As such, the implementation of this method simply requires us to check if our current position is in the list of dot positions.

def isGoalState(*self*, state):  
 return state in *self*.food.asList()

PYTHON