P18-0126 AI lab task 07

```
#modified bfs for greedy first search also

import queue as que
def bfs(self, matrix, root, goal):
        cost = -1
        visited, queue,opened = set(), collections.deque([root]),que
        opened.append(root)

        while queue:

            explore=True  #set it to by-default True
            print("BFS QUEUE ")

            for q in queue:
                if opened!=None:
                    print(q.key, q.location)
                if opened==None:
                    return queue

            #Dequeue a Node from queue\n"
            node = queue.popleft()
            visited.enqueue(node)
            print(\"\\nExpanding Node: \"+ str(node.key) + \" \", end=\"\")

            #check if current node is a goal
            if self.goalTest(node.key, goal):
                print(\"\\n\\n===================\\nHurrah! Found Goal!\
\n==================\\n\\n\")
                #call to a function: findGoalPath\
                #calculating total cost and path from goal-node to the initial state

                opened.dequeue(node)
                visited.enqueue(node)
                self.findGoalPath(node)
                break
                 return

            #call to function: possibleActions\n",
            ans = self.possibleActions(matrix, node.key, node.location)
            print("Locations of all possible actions =")
            cost += 1

            #this function returns an iterable list of 2D-Matrix-indices (i,j) where agent can move
next!\n",
            for nextActionDirection, nextActionLoc in ans.items():
                i, j = nextActionLoc[0], nextActionLoc[1]

                newNodeVal = matrix[i][j]
                newNodeLoc = tuple((i,j))
                newNodeLabel = nextActionDirection

                #for first iteration, don't check parent of Root Node; No need\n",
                if cost<=0:
```

```
                newNode = root.insert(node.key,  node.location, newNodeVal, newNodeLoc,
newNodeLabel)
                #print(\"New node = \", newNode.key, newNode.location)\n",
                #for the first level\n",
                visited.add(newNode)
                queue.append(newNode)
                opened.dequeue(newNode)

            #check not to add parent of a node under its child\n",
            if cost > 0 and node.parent.location is not newNodeLoc:
                newNode = root.insert(node.key, node.location, newNodeVal, newNodeLoc,
newNodeLabel)

                #check the neighbours of a node if they're already visited or not\n",
                #node's can have same value but Unique (row,col) location on matrix\n",
                for eachNode in visited:
                    if eachNode.location == newNodeLoc:
                        explore=False
                # If not visited, mark it as visited, and enqueue it\n",
                if explore:
                    visited.add(newNode)
                    queue.append(newNode)
                    opened.dequeue(newnNode)
        return None

    #==========================================\n",
    #End of Class\n",
    #==========================================\n",
```

A* search general algorithm still working on its final form will submit it you allow me later couldn't complete due to some technical issues

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach
goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be
(re-)expanded.
    // Initially, only the start node is known.
```

```
    // This is usually implemented as a min-heap or priority queue
rather than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding
it on the cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from
start to n currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n]
represents our current best guess as to
    // how short a path from start to finish can be if it goes
through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a
min-heap or a priority queue
        current := the node in openSet having the lowest fScore[]
value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from
current to neighbor
            // tentative_gScore is the distance from start to the
neighbor through current
            tentative_gScore := gScore[current] + d(current,
neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any
previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```