# COEN 266 ARTIFICIAL INTELLIGENCE

# GROUP ASSIGNMENT 1

NAME: Pauravi Nagarkar Id : W1650209 :50%

Nityanand Pujari Id : W1650422 :50%

## 1. Problem 1: Breadth-First- Search

**FUNCTION :**

```
def breadthFirstSearch(problem):

    # define node, with state as initial starting state and path cost0

    node = {'state': problem.getStartState(), 'action':
problem.getActions(problem.getStartState()), 'cost': 0}

    # check if goalState is current-state then return bode.

    if problem.goalTest(node['state']):
        return []

    # declare the starting cost as 0
    cost = 0

    # Creating the QUEUE to insert the node & storing the node
    frontier = util.Queue()
    frontier.push(node)

    # Creating a set variable to hold the explored states eliminating the re backtracking
problem
    explored = set()

    # Loop until there are elements in the Frontier QUEUE
    while frontier:
        # Popping out the new node
        node = frontier.pop()
        # Marking the state if explored
        explored.add(node['state'])


        # Next_node will be based on actions taken by current node
        nextNode = problem.getActions(node['state'])

   # Iterating through each possible nextNode we have for the current state
```

```
        for nextN in nextNode:
            # Creating the next-node state from the parent node
            child = {'state': problem.getResult(node['state'], nextN),  'action': nextN,
'parent': node,
                     'cost': cost}

# Checking if the next state is present in the explored set, if not then its explored

            if child['state'] not in explored:

 # if current node is goal node then return the path
            if problem.goalTest(child['state']):

                    # Creating ResultPath to store the steps.
                    resultPath = []
                    # Make the current node as the child
                    node = child
                    # Backtrack until we get parent node is root node.
                    while 'parent' in node:
                        # Append the result list with the action taken
                        resultPath.append(node['action'])
                    # Make the node point to the current nodes parent node
                        node = node['parent']
 # Reversing because the first action to be taken will be in the end of the list
                    resultPath.reverse()


                    # Return resultPath that took us to goal node
                    return resultPath
             # if current is not nextNode continue to push node in frontier.
                frontier.push(child)

    util.raiseNotDefined()
```

**COMMENTS:**

- Breadth-first Search is simple strategy in which root node is expanded first, then all the successors of root node are expanded next, then their successor and so on.
- The BFS algorithm uses a FIFO (First in First Out) queue for its frontier set to explore the shallowest unexpanded node first, meaning that nodes that were added first to the frontier are explored before those that were added later.

**frontier = util.Queue()**

- The goal test is executed on each node as soon as it is generated, rather than after it has been expanded. The node's attributes such as state, action, and cost are obtained using the Node class. In the BFS algorithm, all nodes are given equal priority and their cost is the same.

```
                    problem.getActions(problem.getStartState()), 'cost': 0}

                      # check if goalState is current-state then return bode.

                                  if problem.goalTest(node['state']):
                                                return []
```

```
# if current node is goal node then return the path
if problem.goalTest(child['state']):
```

- We use the ‘getActions()’ function to get the succeeding nodes - ‘successors’ variable here for the actions and then get the Child Nodes from those actions.

```
nextNode = problem.getActions(node['state'])
```

- For the Breadth First Search algorithm, we verify if the generated child node is present in the frontier set. If it is not present, it is pushed to the frontier queue. If the child node is the goal node, we retrace the steps taken from the starting state to the goal node and store them. These steps represent the solution path for Pacman to reach its goal, the food. Finally, the reverse of this list of actions is returned as the final output.

**PSEUDO CODE FOR BREATH FIRST SEARCH :**

```
Function Breadth-first-search(problem) returns solution or failure
Node<- node with State = problem.Initial-State , Path-cost =0
If goalTest=node return solution found
If problem.GoalTest(node.State) return Solution
Frontier<- FIFO Queue
Explored <- empty set
Loop Do
    If empty(frontier) then return failure
    Node<- POP(frontier)
    Add node to explored nodes
    For each action in problem.Actions(node.State) do
        Child<-child-node(problem,node,action)
        If child.state is not in explored or frontier then
            If problem.GoalTest(child.State) Then return Solution(child)
            Frontier <- Insert(child,frontier)
```

**EXPLANATION:**

- Define node with state & Path-cost
- In BFS, the goal test is performed on each node as soon as it is created, not after it has been expanded. The node's attributes, including its state, action, and cost, are stored using a Node Class
- Define queue(frontier) to store the nodes, new nodes are stored at back of queue and old nodes are at front

```
frontier = util.Queue()
```

- keep track of explored nodes
- Appy Loop;  if frontier is empty return failure
- Else Insert node into explored List
- The "getActions()" function is used to obtain the successor nodes, stored in the "next-node" variable. The child nodes are then generated from these actions.
- The algorithm checks if a child node is present in the frontier set. If it's not, and it's not the goal node, it is added to the frontier. If it is the goal node, the algorithm traces the steps taken to reach it from the start, stores them as the solution path (goal to start), and returns the reverse of the list of actions as the final output. This provides Pacman with the path to reach the food (goal node).

```
For each action in problem.Actions(node.State) do
        Child<-child-node(problem,node,action)
        If child.state is not in explored or frontier then
            If problem.GoalTest(child.State) Then return Solution(child)
            Frontier <- Insert(child,frontier)
```

- Goal node will be returned by reversing the path, once it will reach the goal node it will reverse the path it followed to reach. That is how solution path is returned.

```
resultPath.reverse()
```

## PROBLEM 2 : A* SEARCH

**FUNCTION :**

```
def aStarSearch(problem, heuristic=nullHeuristic):
    startNode = problem.getStartState()
    # Check if the current state is the goal state. If it is, return.
    if problem.goalTest(startNode):
        return []
    # Use Priority Queue to create Current Path
    currentpath = util.PriorityQueue()
    # Defining a list to store all the visited nodes/places
    explored = []
    # Defining a result list, which store the result path
    result = []
    # Using Priority queue to store all the possible actions at every stage
    priorityQueue = util.PriorityQueue()
    # adding into the queue
    priorityQueue.push(startNode, 0)
    # Positon of current node determined by the queue
    currentNode = priorityQueue.pop()
    # Loop until the current node is the goal node
    while not problem.goalTest(currentNode):
        # Check if the current node visited ?
        if currentNode not in explored:
            # Since its visited now, append current node into explored
            explored.append(currentNode)
            # At this node, get all the possible actions, eg - North, South, East, West
            action = problem.getActions(currentNode)

            # For each action possible, calculate the Pathcost
            for A in action:
                # Since this node is a possible path to the goal, add it to result list
                nextnode = result + [A]
                # Calculate the Pathcost so that the nodes visited will have the
priorities assigned based on Heuristic
                Pathcost = problem.getCostOfActions(nextnode) +
heuristic(problem.getResult(currentNode, A), problem)
                # Explore only if the node is not visited before
                if A not in explored:
                    # Append the possible actions at each stage queue along with the
corresponding Path cost
```

```
                    priorityQueue.push(problem.getResult(currentNode, A), Pathcost)
                    currentpath.push(nextnode, Pathcost)
            # Remove form queue as the current node is already visited.
            currentNode = priorityQueue.pop()
            # Store only the element with lower Pathcost rest pop element out
            result = currentpath.pop()
        return result
util.raiseNotDefined()
```

**Explanation**:

- It evaluates Node By combining g(n), the cost to reach the node and h(n) the cost to get from the node to goal :
    - f(n) = g(n)+h(n)
- since g(n) gives path cost from start node to node n , and h(n) is estimated cost of cheapest path from n to goal
- f(n) = estimated cost of cheapest solution through n,
- Thus, to fine cheapest solution, lets find node with lowest f(n).
- A* is both complete and Optimal.

F(n) = g(n)+h(n)

- The above source code implements A* algorithm to help Pacman reach its goal node, which is the 'Food', in the game.
- The initial step involves obtaining the starting node and evaluating whether it is the goal node or not.
- Next, we initialize the Frontier set as a priority queue, where the nodes are prioritized and stored based on their lowest f(n) value.
- We proceed by exploring the nodes until the goal node is reached. Each time a node is explored, it is added to the list of visited nodes to ensure that it is not revisited. The exploration process continues until the goal node is found.
- The "getActions" function is utilized to generate the child nodes, and the total estimated cost is then calculated based on the cost of reaching the node and the estimated cost to reach the goal from that node.

- The current path is updated by adding the child nodes based on their priority, and the node with the lowest priority (i.e. lowest f(n) or route cost) is removed first.
- The process continues until the goal node is reached and Pacman eats its food.

COMMENTS:

- A* Search is a type of search algorithm that takes into account both the cost to reach the current node and the estimated cost to reach the goal from the current node. The cost to reach the current node is represented by g(n) and the estimated cost to reach the goal from the current node is represented by h(n). The algorithm combines these costs to evaluate the nodes and determine the cheapest solution path.

$$\mathbf{f(n) = g(n) + h(n)}$$

- The A* Search algorithm is applied to the Pacman Game to reach the goal node of finding food. The algorithm starts by obtaining the starting node and determining if it is already the goal node.

```
startNode = problem.getStartState()
# Check if the current state is the goal state. If it is, return.
if problem.goalTest(startNode):
    return []
```

- Next, we initialize the Frontier Set to Priority Queue, where the nodes will be retained based on the lowest f(n) value.

```
# Use Priority Queue to create Current Path
currentpath = util.PriorityQueue()
```

- The exploration of nodes continues until the goal node is found. Whenever a new node is encountered, it is added to the list of visited nodes and checked to ensure that it has not been explored before.
- We use ‘getActions’ function to generate the Child Nodes and then calculate the Total Estimated Cost as below:

```
for A in action:
    # Since this node is a possible path to the goal, add it to result list
    nextnode = result + [A]
    # Calculate the Pathcost so that the nodes visited will have the priorities assigned based on Heuristic
```

```
            Pathcost = problem.getCostOfActions(nextnode) + heuristic(problem.getResult(currentNode, A),
    problem)
```

- The current path is then added with the Child Nodes based on the priority and the one with the lowest priority i.e. Cost f(n) (Pathcost) will be popped first.
- The whole loop goes on till we reach the Goal, and the Pacman eats his food.