# Best First Search

```
def BFS(graph, start, goal):
    q = []
    q.append([start, h_val[start]])
    v = []

    while(q):
        temp = q.pop(0)
        v.append(temp[0])

        if(temp[0] == goal):
            print("Goal Node Found!")
            for node in v:
                print(node, end="--->")

        for node in graph[temp[0]]:
            if(node[0] not in v):
                newNode = [node[0], h_val[node[0]]]
                q.append(newNode)
        q.sort(key = sortQueue)
```

# Uniformed Cost Search

```
def UCS(graph, start, end):
    q = []
    v = []
```

```
q.append([start, 0, []])
while (q):
   print("Status of Queue: " + str(q))
   popedNode = q.pop(0)
   print("Node Poped: " + str(popedNode))
   popedCost = popedNode[1]
   v.append(popedNode[0])
   print("Status of Visited Queue: " + str(v))
   print(end = "\n\n")

   route = popedNode[2]
   routes = copy.deepcopy(route)
   routes.append([popedNode[0]])

   if(popedNode[0] == end):
      print("Goal Node Found With A Cost Of: " + str(popedCost))
      print("Path Taken: " + str(routes))
      return

   for child in graph[popedNode[0]]:
      if(child[0] not in v):
         tempChild = child[0]
         tempCost = child[1] + popedCost
         q.append([tempChild, tempCost, routes])
   q.sort(key = sort)
```

# A* Algorithm

```
def a_star(graph, start, end):
    # insert root node
    # make a distance dictionary
    # make a queue
    # insert root node into queue with a f_cost = g_cost + h_cost
    q =[]
    v = []
    distance = {}
    q.append([start, 0, []])
    distance[start] = 0
    #while queue is not empty
        # pop from queue
    while(q):
        print("Queue Status: " + str(q), end = "\n")
        currNode = q.pop(0)
        print("Node Poped: " + str(currNode), end = "\n\n\n")
        neighbours = graph[currNode[0]]
        v.append(currNode[0])
        #print(neighbours)
        route = currNode[2]
        routes = copy.deepcopy(route)
        routes.append(currNode[0])
        if(currNode[0] == end):
            print("Path Found!!")
            print("Path : " + str(routes))
```

```
        print("Path Cost: " + str(currNode[1]))
        return

      #print(neighbours)
      children = valid_children(graph, currNode, v)
      for child in children:
        g_cost = distance[currNode[0]] + neighbours[child]
        #print(g_cost)
        if(child not in distance or g_cost < distance[child]):
          distance[child] = g_cost
          f_cost = g_cost + heuristic_value[child]
          q.append([child, f_cost, routes])
      q.sort(key = sortfun)
    return
```

# Traveling Salesman Problem

```
def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
      randomCity = cities[random.randint(0, len(cities) - 1)]
      solution.append(randomCity)
      cities.remove(randomCity)

    return solution
```

```python
def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength


def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours


def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength
```

```
def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength
```

# Min Max Algorithm

```
# our min-max function that evaluates the current game board and assigns a score to the board depending on if the user is winning or the computer is winning
def minimax(depth, isMax) :
    # if the computer won we assign in a score of 1
    if (checkForWinner('O') == True) :
        return 1
    # if the player won we assign it a score of -1
    if (checkForWinner('X') == True) :
        return (-1)
    # if there is a tie / no more possible moves we assign it 0
    if (possibleMoves() == []) :
        return 0

    # If this maximizer/computers move
    if (isMax) :
        # we have to maximize negative infinity
        best = -1000
        # traverse the game board
        for i in range(3) :
```

```
            for j in range(3) :
                # if we find an empty cell
                if (board[i][j]=='_') :
                    # we make a move there
                    board[i][j] = 'O'
                    # we call the minimax algorithm but we select the max
of the two miniumum
                    best = max( best, minimax(depth + 1, False))
                    # we undo the move we made
                    board[i][j] = '_'
        return best

    # If this minimizer/players move
    else :
        # we have to minimize infinity
        best = 1000
        # traverse the game board
        for i in range(3) :
            for j in range(3) :
                # if we find an empty cell
                if (board[i][j] == '_') :
                    # we make a move there
                    board[i][j] = 'X'
                    # we call the minimax algorithm but we select the max
of the two miniumum
                    best = min(best, minimax(depth + 1, True))
                    # we undo the move we made
                    board[i][j] = '_'
        return best

# function that will use the min-max algorithm and provide us with the best
possible move for the computer
def findBestMove(board) :
    bestVal = -1000
    bestMove = []

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for i in range(3) :
        for j in range(3) :
            # Check if cell is empty
            if (board[i][j] == '_') :
                # Make the move
                board[i][j] = 'O'
                # compute evaluation function for this
                # move.
                moveVal = minimax(0, False)
                # Undo the move
                board[i][j] = '_'
                # If the value of the current move is
                # more than the best value, then update
                # best/
                if (moveVal > bestVal) :
                    bestMove = [i,j]
                    bestVal = moveVal
    return (bestMove, bestVal)
```

```
# function that implements mini-max algorithm on a game tree
def minimax(currDepth, depth, currNode, score, maximizing):
    # check if we have reached the maximum tree depth
    if(currDepth == depth):
        return score[currNode]
    # if we are maximizing then we need to pick the max of the two minimum 
childs
    if(maximizing == True):
        return max( minimax(currDepth+1, depth, currNode*2, score, False) , 
minimax(currDepth+1, depth, (currNode*2)+1, score, False ) )
    # if we are minimizing then we need to pick two of the maximum childs
    else:
        return min( minimax(currDepth+1, depth, currNode*2, score, True) , 
minimax(currDepth+1, depth, (currNode*2)+1, score, True ) )
```

# Alpha Beta Pruning

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

**Name: Syed Muhammad Ashhar Shah**

**Reg Number: 2020478**

**Intro To Artificial Intelligence Lab**

**TRD**