LAB 1 :- BFS (Water Jug Problem)

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan  6 10:13:45 2025

@author: CSE
"""

# Water Jug Problem using BFS

x_capacity = 4  # Capacity of jug X
y_capacity = 3  # Capacity of jug Y

def bfs(start, goal):
    """Perform BFS to find the shortest path to the goal state."""
    parent = []  # Stores parent-child relationships
    queue = [start]  # BFS queue
    explored = {start}  # Set to track explored states

    while queue:
        current = queue.pop(0)  # Dequeue the first state

        if current == goal:
            print("Solution Found!")
            print("Path:", find_path(parent, start, goal))
            return

        # Generate all possible next states
        next_states = [
            fill_x(current),
            fill_y(current),
            empty_x(current),
            empty_y(current),
            pour_from_x_to_y(current),
            pour_from_y_to_x(current),
        ]

        for state in next_states:
            if state and state not in explored:  # Avoid revisiting states
                queue.append(state)
                explored.add(state)
                parent.append([current, state])  # Store relationship for path tracking
```

```python
    print("No solution found.")

def fill_x(state):
    """Fill jug X to its maximum capacity."""
    x, y = state
    return (x_capacity, y) if x < x_capacity else None

def fill_y(state):
    """Fill jug Y to its maximum capacity."""
    x, y = state
    return (x, y_capacity) if y < y_capacity else None

def empty_x(state):
    """Empty jug X completely."""
    x, y = state
    return (0, y) if x > 0 else None

def empty_y(state):
    """Empty jug Y completely."""
    x, y = state
    return (x, 0) if y > 0 else None

def pour_from_x_to_y(state):
    """Pour water from jug X to jug Y until Y is full or X is empty."""
    x, y = state
    transfer = min(x, y_capacity - y)
    return (x - transfer, y + transfer) if transfer > 0 else None

def pour_from_y_to_x(state):
    """Pour water from jug Y to jug X until X is full or Y is empty."""
    x, y = state
    transfer = min(y, x_capacity - x)
    return (x + transfer, y - transfer) if transfer > 0 else None

def find_path(parent, start, goal):
    """Trace the path from start to goal using the parent list."""
    path = [goal]
    while path[-1] != start:
        for p in parent:
            if p[1] == path[-1]:  # Find the parent of the current state
                path.append(p[0])
                break
    return path[::-1]  # Reverse to get the correct order
```

```
# Initial and goal states
start_state = (0, 0)  # Both jugs start empty
goal_state = (2, 0)  # Goal: 2 liters in jug X and 0 in jug Y

# Run BFS to find the solution
bfs(start_state, goal_state)
```

LAB 2

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 13 04:43:08 2025

@author: CSE
"""
```

PART 1 :- DFS (TREE)

```
g = {1:[2,3],2:[4,5],3:[6,7],4:[],5:[],6:[],7:[]}

def DFS(start, g):
    visited = []
    stack = g[start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.append(node)


    for i in visited:
        print(i)

DFS(1, g)
```

PART 2 :- DFS (GRAPH)

```
graph = {1:[2,3],2:[1,4,5],3:[1,6,7],4:[2],5:[2],6:[3],7:[3]}

def dfs(graph,start, goal):
    visited = []
    stack = [(start,[start])]

    while stack:
        node,path = stack.pop()
        if node not in visited:
            visited.append(node)
            print('node',node,'added')

        if node == goal:
            return visited,path

        for neighbour in graph[node]:
            if neighbour not in visited:
                stack.append([neighbour,path+[neighbour]])

    for i in visited:
        print(i)

visited,path = dfs(graph,1,6)
print(visited)
print(path)
```

PART 3 :- DLS (GRAPH)

```
graph = {1: [2, 3], 2: [1, 4, 5], 3: [1, 6, 7], 4: [2], 5: [2], 6: [3], 7: [3]}

def dls(graph, start, goal, depth_limit):
    """Performs Depth-Limited Search (DLS) from start to goal up to depth_limit."""
    stack = [(start, [start], 0)]  # Stack stores (node, path, depth)

    while stack:
        node, path, depth = stack.pop()

        if node == goal:
            return path  # Return the found path

        if depth < depth_limit:  # Enforce depth limit
            for neighbor in reversed(graph[node]):  # Reverse to maintain order (like DFS)
```

```python
            stack.append((neighbor, path + [neighbor], depth + 1))

    return None  # Return None if no path is found

# Get user input for depth limit
depth_limit = int(input("Input the depth limit for Depth-Limited Search: "))
start_node = 1
goal_node = 6

# Run DLS
result = dls(graph, start_node, goal_node, depth_limit)

# Print result
if result:
    print("Path found:", result)
else:
    print("No path found within depth limit.")
```

LAB 3 :- UCS

PART A:

```python
import heapq
import copy

def uniform_cost_search(graph, start, goal):
    """Finds the least-cost path using Uniform Cost Search (UCS)."""
    path = []  # Stores the path from start to goal
    visited = set()  # Keeps track of visited nodes
    path_cost = 0  # Tracks the total path cost

    if start == goal:
        return path, path_cost, visited

    # Priority queue (min-heap) initialized with (cost, path)
    open_list = [(path_cost, [start])]

    while open_list:
        curr_cost, curr_path = heapq.heappop(open_list)  # Pop the least-cost path
        curr_node = curr_path[-1]
```

```
        if curr_node in visited:
            continue  # Skip already visited nodes

        visited.add(curr_node)  # Mark node as visited

        # If goal is reached, return path details
        if curr_node == goal:
            return curr_path, curr_cost, visited

        # Explore neighbors
        if curr_node in graph:  # Ensure node exists in graph
            for cost, neighbor in graph[curr_node]:
                if neighbor not in visited:
                    new_cost = curr_cost + cost
                    new_path = copy.copy(curr_path) + [neighbor]
                    heapq.heappush(open_list, (new_cost, new_path))

    return [], float('inf'), visited  # Return empty path if no solution found


# Sample Graphs
graph2 = {
    'A': [(3, 'B'), (1, 'C')],
    'B': [(3, 'D')],
    'C': [(2, 'G'), (1, 'D')],
    'D': [(3, 'G')],
    'S': [(12, 'G')]
}

graph3 = {
    0: [(1, 2), (1, 1)],
    1: [(3, 3)],
    2: [(2, 5)],
    3: [(2, 5), (2, 4)],
    4: [(1, 5)],
    5: [(3, 0)]
}

# Running UCS on both graphs
p, c, v = uniform_cost_search(graph3, 0, 5)
print("Graph 3:")
print("Path: ", p)
print("Cost: ", c)
print("Visited: ", v)
```

```
print("\n")

p, c, v = uniform_cost_search(graph2, 'A', 'G')
print("Graph 2:")
print("Path: ", p)
print("Cost: ", c)
print("Visited: ", v)
```

PART B :

```
from heapq import heappush, heappop
from collections import defaultdict

# Graph Representation using defaultdict
graph = defaultdict(list)

graph[1].append((1, 2))
graph[1].append((1, 1))

graph[2].append((3, 4))
graph[2].append((4, 3))

graph[3].append((3, 5))

graph[4].append((3, 5))
graph[4].append((4, 3))

graph[5].append((4, 3))  # Node 5 is connected to Node 3

def ucs(src, dest):
    """Performs Uniform Cost Search (UCS) to find the least-cost path from src to dest."""
    heap = []
    heappush(heap, (0, src, [src]))  # (cost, currentNode, pathTaken)

    while heap:
        dist, currNode, path = heappop(heap)

        if currNode == dest:
```

```
            print(f"Path from {src} to {dest}: {path}")
            print(f"Total Cost: {dist}")
            return

        for wt, nei in graph[currNode]:
            if nei not in path:  # Avoid cycles
                heappush(heap, (dist + wt, nei, path + [nei]))

    print(f"No path found from {src} to {dest}")

# Run Uniform Cost Search
ucs(1, 5)
```

Practical 4:- A* Search

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 27 10:12:44 2025

@author: CSE
"""


#Graph_nodes = {'A': [('B', 2), ('E', 3)], 'B': [('C', 1), ('G', 9)], 'C': None, 'E': [('D', 6)], 'D': [('G', 1)]}
Graph_nodes = {'A': [('B', 4), ('C', 3)], 'B': [('F', 5), ('E', 12)], 'C': [('E', 10), ('D', 7)],'D': [('E', 2), ('C', 7)],
               'E': [('B', 12), ('G', 5)], 'F': [('B', 5), ('G', 16)], 'G': [('F', 16), ('E', 5)]}

def heuristic(n):
    heuristic = {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7,'F':11, 'G': 0}
    return heuristic[n]

def Astar(start, stop):
    open_set = set(start)
    closed_set = set()
    g = {}
    parents = {}
```

```
g[start] = 0
parents[start] = start

while len(open_set) > 0:
    n = None

    for v in open_set:
        if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
            n = v

    if n == stop or Graph_nodes[n] is None:
        pass
    else:
        for (m, weight) in get_neighbours(n):
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
                print('Cost till previous node =', g[n])
                print('Consolidated cost of', m, 'is', g[m])

            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    print('Parent of', m, 'is', parents[m])
                    parents[m] = n
                    print('parents of', m, 'reinitiated')
                    print('new parents of', m, 'is', parents[m])

                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

    if n is None:
        print('Path does not exist!')
        return None

    if n == stop:
        path = []

        while parents[n] != n:
            path.append(n)
            n = parents[n]
        path.append(start)
```

```
            path.reverse()

            print('shortest path found: {}'.format(path))
            print('Path length is', g[stop])
            return path

        open_set.remove(n)
        closed_set.add(n)
        print('visited', closed_set)
        print('Open List', open_set)

    print('Path does not exist')
    return None

def get_neighbours(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None


Astar("A", "G")
```

OR

MODEFIED CODE :-

```
       # Graph representation with adjacency list and weights
Graph_nodes = {
    'A': [('B', 4), ('C', 3)],
    'B': [('F', 5), ('E', 12)],
    'C': [('E', 10), ('D', 7)],
    'D': [('E', 2), ('C', 7)],
    'E': [('B', 12), ('G', 5)],
    'F': [('B', 5), ('G', 16)],
    'G': [('F', 16), ('E', 5)]
}

# Heuristic function (estimated cost from node to goal)
def heuristic(n):
    heuristics = {'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'F': 11, 'G': 0}
    return heuristics[n]
```

```
# Function to get neighboring nodes and their costs
def get_neighbours(node):
    return Graph_nodes.get(node, [])

# A* Algorithm implementation
def Astar(start, goal):
    open_set = set([start])  # Nodes to be evaluated
    closed_set = set()  # Nodes already evaluated
    g = {start: 0}  # Cost from start to current node
    parents = {start: start}  # Parent dictionary for path reconstruction

    while open_set:
        # Select node with lowest f(n) = g(n) + h(n)
        current_node = min(open_set, key=lambda node: g[node] + heuristic(node))

        if current_node == goal:
            path = []
            while parents[current_node] != current_node:
                path.append(current_node)
                current_node = parents[current_node]
            path.append(start)
            path.reverse()

            print(f"Shortest path found: {path}")
            print(f"Path length: {g[goal]}")
            return path

        open_set.remove(current_node)
        closed_set.add(current_node)

        for neighbor, weight in get_neighbours(current_node):
            if neighbor in closed_set:
                continue

            tentative_g = g[current_node] + weight

            if neighbor not in open_set:
                open_set.add(neighbor)
                parents[neighbor] = current_node
                g[neighbor] = tentative_g
            else:
                if tentative_g < g[neighbor]:  # Found a better path
                    g[neighbor] = tentative_g
```

```
                parents[neighbor] = current_node

                if neighbor in closed_set:
                    closed_set.remove(neighbor)
                    open_set.add(neighbor)

        print(f"Visited: {closed_set}")
        print(f"Open List: {open_set}")

    print("Path does not exist!")
    return None

# Running A* search from A to G
Astar("A", "G")
```

Lab 5 :- Sliding Puzzle Problem (Hill Climbing Search)

```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb  3 10:09:37 2025

@author: CSE
"""
import copy

start = [[1,3,5], [4,None,2], [6,7,8]]
goal = [[1,3,None], [4,2,5], [6,7,8]]

def checkgoal(a, b):
    flag = (a == b)
    return flag

def misplaced(a,b):
    count = 0
    for i in range(3):
        for j in range(3) :
            if (a[i][j] == b[i][j]):
                pass
```

```
            else:
                count+=1
    return count

'''def manhatton(a,b):
    if(a != b)
        for i in a:
            for j in '''

def emptytile(a):
    for i in range(3):
        for j in range(3):
            if a[i][j] == None:
                return i,j
                print("Position",i,j, "is Empty")


def generatemoves(state):
    childern = []
    x,y = emptytile(state)
    moveUp = (x-1,y)
    moveDown = (x+1,y)
    moveRight = (x,y+1)
    moveLeft = (x,y-1)

    if(moveUp[0] in [0,1,2]) and (moveUp[1] in [0,1,2]):
        childern.append(moveUp)
    if(moveDown[0] in [0,1,2]) and (moveDown[1] in [0,1,2]):
        childern.append(moveDown)
    if(moveRight[0] in [0,1,2]) and (moveRight[1] in [0,1,2]):
        childern.append(moveRight)
    if(moveLeft[0] in [0,1,2]) and (moveLeft[1] in [0,1,2]):
        childern.append(moveLeft)
    return childern

def generatestates(state):
    list1 = []
    list2 = []

    st = copy.deepcopy(state)
    childmoves = generatemoves(state)
    count = len(childmoves)
    x,y = emptytile(state)
```

```
    while count > 0:
        tpos = childmoves.pop()
        print("Position is",tpos)
        tval = st[tpos[0]][tpos[1]]
        print("tile nuimber is", tval)
        for i in range(0,3):
            for j in range(0,3):
                if st[i][j] == None:
                    tval = st[tpos[0]][tpos[1]]
                    st[tpos[0]][tpos[1]] = None
                    st[i][j] = tval
        tiles = misplaced(st,goal)
        print("mispalced tiles", tiles)

        list2.append(tiles)

        list1.append(st)
        print('generated state is: ',st[:])
        print()
        st = copy.deepcopy(state)
        count = count - 1

    print('The values of misplaced tiles are :', list2)
    v = list2.index(min(list2))
    print("the best chosen stete is: ",list[v])
    print()
    return list1[v]

def hillclimb(s,e):
    f = checkgoal(s,e)
    if f is True:
        print("start state and end state is same")
    else :
        i,maxi=1,7
        while maxi>0:
            print("Iteration Number: ",i,'begins')
            chosen = generatestates(s)
            print("chosen stae returned to main functionn for checking = ",chosen)
            if chosen == e:
                print("Goal reached")
                print('Goal configuration is ',chosen,'in',i,'steps')
                break
            else:
                print('The chosen state is not the goal state\n')
```

```
        s = chosen
        maxi = maxi - 1
        i = i+1

res = misplaced(start,goal)
print(res)
flag = checkgoal(start,goal)
print(flag)
index_i,index_j = emptytile(start)
print(index_i,index_j)
index_i,index_j = emptytile(goal)
print(index_i,index_j)
print(generatemoves(start))
print(generatestates(goal))
```

OR

MODIFIED CODE :-

```
import copy

# Initial and Goal States
start = [[1, 3, 5], [4, None, 2], [6, 7, 8]]
goal = [[1, 3, None], [4, 2, 5], [6, 7, 8]]

# Check if current state is the goal state
def checkgoal(a, b):
    return a == b

# Heuristic: Count misplaced tiles
def misplaced(a, b):
    count = sum(1 for i in range(3) for j in range(3) if a[i][j] != b[i][j])
    return count

# Find the empty tile position
def emptytile(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] is None:
                return i, j

# Generate possible moves for empty tile
def generatemoves(state):
    x, y = emptytile(state)
```

```python
    moves = []

    # Define possible moves (Up, Down, Left, Right)
    possible_moves = {
        "Up": (x - 1, y),
        "Down": (x + 1, y),
        "Left": (x, y - 1),
        "Right": (x, y + 1)
    }

    # Add valid moves
    for move in possible_moves.values():
        if 0 <= move[0] < 3 and 0 <= move[1] < 3:
            moves.append(move)

    return moves

# Generate new states by moving the empty tile
def generatestates(state):
    children = []
    misplaced_counts = []

    x, y = emptytile(state)
    possible_moves = generatemoves(state)

    for new_x, new_y in possible_moves:
        # Copy the current state
        new_state = copy.deepcopy(state)

        # Swap empty tile with new position
        new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]

        # Compute heuristic (misplaced tiles)
        h = misplaced(new_state, goal)

        children.append(new_state)
        misplaced_counts.append(h)

    # Find the best state (with least misplaced tiles)
    if misplaced_counts:
        best_index = misplaced_counts.index(min(misplaced_counts))
        return children[best_index]

    return None  # No better move available
```

```python
# Hill Climbing Algorithm
def hillclimb(start, goal):
    if checkgoal(start, goal):
        print("Start state is already the goal state!")
        return

    max_iterations = 10  # To prevent infinite loops
    current_state = start

    for i in range(1, max_iterations + 1):
        print(f"\nIteration {i}:")
        for row in current_state:
            print(row)

        # Generate the best next state
        next_state = generatestates(current_state)

        # If no better move is found
        if next_state is None or checkgoal(next_state, current_state):
            print("No better state found. Stopping search.")
            return

        # If goal state is reached
        if checkgoal(next_state, goal):
            print("\nGoal state reached!")
            for row in next_state:
                print(row)
            print(f"Total Steps: {i}")
            return

        # Move to the next best state
        current_state = next_state

    print("Max iterations reached. Stopping search.")

# Run the algorithm
hillclimb(start, goal)
```

LAB 6 :- Minimax Algorithm

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth ):
    if (curDepth == targetDepth):
        print('Minimax value is: ', scores[nodeIndex])
        return scores[nodeIndex]

    if(maxTurn):
        Mx = max(minimax(curDepth + 1, nodeIndex*2 ,False, scores,
targetDepth),minimax(curDepth + 1, nodeIndex*2 + 1,False, scores, targetDepth))
        print('Max value selected is =',Mx)

        return Mx
        print("  ")

    else:
        Mn = min(minimax(curDepth + 1, nodeIndex*2 ,True, scores,
targetDepth),minimax(curDepth + 1, nodeIndex*2 + 1,True, scores, targetDepth))
        print('Min value selected is =',Mn)

        return Mn
        print("  ")

scores = [10,24,12,16,2,7,-5,-80,1,12,22,-16,2,7,-15,-8]
treeDepth = math.log(len(scores),2)

print('The optimal value is ', minimax(0 ,0 ,False ,scores ,treeDepth))
# print('The optimal value is ', minimax(0 ,0 ,True ,scores ,treeDepth))
```