AI Search Algorithms Python Programs:

**Problem Statement-1**

Write a Python program to implement **Breadth-First Search (BFS)** on the following tree structure. The BFS should start from the root node and explore all nodes level by level.

A

/ \

B C

/ \ / \

D E F G

**Code:**

**--------------------------------------------------------**

# Breadth-First Search (BFS) without OOP, just a simple tree representation

# Function to perform BFS

def bfs(tree, root):

# Create an empty list to keep track of visited nodes

visited = []

# Create a queue with the root node to begin the search

queue = [root]

# While the queue is not empty, keep exploring nodes

while queue:

# Pop (remove) the first node from the queue and visit it

node = queue.pop(0)

# Add the visited node to the visited list

visited.append(node)

# Add all the children (neighbors) of the current node to the queue

for neighbor in tree[node]:

queue.append(neighbor)

# Return the list of visited nodes in the order they were visited

return visited

# Define a simple tree structure as a dictionary

# Each key is a node, and the value is a list of its children

tree = {

'A': ['B', 'C'], # A is the root, B and C are its children

'B': ['D', 'E'], # B has two children: D and E

'C': ['F', 'G'], # C has two children: F and G

'D': [], # D is a leaf node, no children

'E': [], # E is a leaf node, no children

'F': [], # F is a leaf node, no children

'G': [] # G is a leaf node, no children

}

# Call the BFS function starting from the root 'A'

bfs\_result = bfs(tree, 'A')

# Display the order in which nodes were visited

print("Order of nodes visited in BFS:", bfs\_result)

--------------------------------------------------------------------------------------------------------------------------------------

Sample Output:

Order of nodes visited in BFS: ['A', 'B', 'C', 'D', 'E', 'F', 'G']

**Problem Statement -2:**

Write a Python program to implement **Depth-First Search (DFS)** on the following tree structure. The DFS should start from the root node and explore as deeply as possible before backtracking.

A

/ \

B C

/ \ / \

D E F G

----------------------------------------------------------------------------------------------------------------

**Code:**

# Depth-First Search (DFS) without OOP, simple tree representation

# Function to perform DFS

def dfs(tree, node, visited):

# Add the current node to the visited list

visited.append(node)

# Recursively visit each neighbor (child) of the current node

for neighbor in tree[node]:

dfs(tree, neighbor, visited)

return visited

# Define a simple tree structure as a dictionary

# Each key is a node, and the value is a list of its children

tree = {

'A': ['B', 'C'], # A is the root, B and C are its children

'B': ['D', 'E'], # B has two children: D and E

'C': ['F', 'G'], # C has two children: F and G

'D': [], # D is a leaf node, no children

'E': [], # E is a leaf node, no children

'F': [], # F is a leaf node, no children

'G': [] # G is a leaf node, no children

}

# Initialize an empty list to keep track of visited nodes

visited\_nodes = []

# Call the DFS function starting from the root 'A'

dfs\_result = dfs(tree, 'A', visited\_nodes)

# Display the order in which nodes were visited

print("Order of nodes visited in DFS:", dfs\_result)

Expected Output:

Order of nodes visited in DFS: ['A', 'B', 'D', 'E', 'C', 'F', 'G']

**Problem Statement-3:**

Write a Python program to implement **Depth-Limited Search (DLS)** on the following tree structure. The search should stop when a specified depth limit is reached.

A

/ \

B C

/ \ / \

D E F G

Code:

# Depth-Limited Search (DLS) without OOP, simple tree representation

# Function to perform DLS

def dls(tree, node, limit, visited, depth):

# Add the current node to the visited list

visited.append(node)

# Stop searching deeper if the depth limit is reached

if depth == limit:

return visited

# Recursively visit each neighbor (child) of the current node, incrementing the depth

for neighbor in tree[node]:

dls(tree, neighbor, limit, visited, depth + 1)

return visited

# Define a simple tree structure as a dictionary

# Each key is a node, and the value is a list of its children

tree = {

'A': ['B', 'C'], # A is the root, B and C are its children

'B': ['D', 'E'], # B has two children: D and E

'C': ['F', 'G'], # C has two children: F and G

'D': [], # D is a leaf node, no children

'E': [], # E is a leaf node, no children

'F': [], # F is a leaf node, no children

'G': [] # G is a leaf node, no children

}

# Initialize an empty list to keep track of visited nodes

visited\_nodes = []

# Set the depth limit for the search

depth\_limit = 2

# Call the DLS function starting from the root 'A' and an initial depth of 0

dls\_result = dls(tree, 'A', depth\_limit, visited\_nodes, 0)

# Display the order in which nodes were visited

print("Order of nodes visited in DLS (limit =", depth\_limit, "):", dls\_result)

Expected Output for Depth Limit 2:

Order of nodes visited in DLS (limit = 2): ['A', 'B', 'D', 'E', 'C', 'F', 'G']

**Problem Statement-4:**

Write a Python program to implement **Iterative Deepening Depth-First Search (IDDFS)** on the following tree structure. The search should progressively deepen until the maximum depth limit is reached.

A

/ \

B C

/ \ / \

D E F G

# Depth-Limited Search (DLS) helper function

def dls(tree, node, limit, visited, depth):

# Add the current node to the visited list

visited.append(node)

# Stop searching deeper if the depth limit is reached

if depth == limit:

return visited

# Recursively visit each neighbor (child) of the current node, incrementing the depth

for neighbor in tree[node]:

dls(tree, neighbor, limit, visited, depth + 1)

return visited

# Iterative Deepening Depth-First Search (IDDFS) function

def iddfs(tree, root, max\_depth):

# Iterate over all possible depth limits from 0 to max\_depth

for limit in range(max\_depth + 1):

# Initialize a new visited list for each depth-limited search

visited = []

print(f"Depth limit: {limit}")

# Perform DLS for the current depth limit

dls(tree, root, limit, visited, 0)

# Display the visited nodes for the current depth limit

print("Visited nodes:", visited)

# Define a simple tree structure as a dictionary

# Each key is a node, and the value is a list of its children

tree = {

'A': ['B', 'C'], # A is the root, B and C are its children

'B': ['D', 'E'], # B has two children: D and E

'C': ['F', 'G'], # C has two children: F and G

'D': [], # D is a leaf node, no children

'E': [], # E is a leaf node, no children

'F': [], # F is a leaf node, no children

'G': [] # G is a leaf node, no children

}

# Set the maximum depth limit for the iterative deepening search

max\_depth = 3

# Call the IDDFS function starting from the root 'A'

iddfs(tree, 'A', max\_depth)

----------------------------------------------------------------------------------------------------------------

Expected Output for Maximum Depth Limit 3:

Depth limit: 0

Visited nodes: ['A']

Depth limit: 1

Visited nodes: ['A', 'B', 'C']

Depth limit: 2

Visited nodes: ['A', 'B', 'D', 'E', 'C', 'F', 'G']

Depth limit: 3

Visited nodes: ['A', 'B', 'D', 'E', 'C', 'F', 'G']

**Problem Statement-5:**

Write a Python program to solve the **8-puzzle problem** using **Breadth-First Search (BFS)**. The 8-puzzle problem consists of a 3x3 grid where a tile is missing (represented by 0). The goal is to arrange the tiles to match the goal state by sliding tiles into the blank space.

Initial state:

1 2 3

4 0 5

7 8 6

Goal state:

1 2 3

4 5 6

7 8 0

Code:

from collections import deque

# Define the goal state for the 8-puzzle

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]]

# Function to find the position of the blank tile (0)

def find\_blank(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

# Function to check if two states are equal

def is\_goal(state):

return state == goal\_state

# Function to create a new state by swapping the blank with a neighboring tile

def move\_tile(state, x1, y1, x2, y2):

new\_state = [row[:] for row in state] # Deep copy of the state

new\_state[x1][y1], new\_state[x2][y2] = new\_state[x2][y2], new\_state[x1][y1]

return new\_state

# Function to get all possible moves from the current state

def get\_neighbors(state):

neighbors = []

x, y = find\_blank(state) # Find the blank tile

# Define possible moves: left, right, up, down

possible\_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Try each possible move

for dx, dy in possible\_moves:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < 3 and 0 <= new\_y < 3: # Make sure the move is within bounds

new\_state = move\_tile(state, x, y, new\_x, new\_y)

neighbors.append(new\_state)

return neighbors

# Function to solve the 8-puzzle using Breadth-First Search (BFS)

def bfs(initial\_state):

# Create a queue to store the states to explore

queue = deque([(initial\_state, [])]) # Each item in the queue is (state, path to the state)

# Set to store visited states to avoid revisiting

visited = set()

visited.add(tuple(map(tuple, initial\_state))) # Convert the 2D list to a hashable tuple

while queue:

# Get the current state and the path to reach it

current\_state, path = queue.popleft()

# If we reach the goal state, return the path

if is\_goal(current\_state):

return path + [current\_state]

# Get the neighboring states (possible moves)

for neighbor in get\_neighbors(current\_state):

neighbor\_tuple = tuple(map(tuple, neighbor)) # Convert to a tuple for hashing

# If the neighbor hasn't been visited, add it to the queue

if neighbor\_tuple not in visited:

visited.add(neighbor\_tuple)

queue.append((neighbor, path + [current\_state])) # Add the neighbor with the updated path

return None # Return None if no solution is found

# Define the initial state for the 8-puzzle

initial\_state = [[1, 2, 3],

[4, 0, 5],

[7, 8, 6]]

# Call the BFS function and store the result path

solution\_path = bfs(initial\_state)

# Display the solution path, if found

if solution\_path:

print("Solution found:")

for step in solution\_path:

for row in step:

print(row)

print() # Newline between steps

else:

print("No solution found.")

----------------------------------------------------------------------------------------------------------------

Sample Output:

Solution found:

[1, 2, 3]

[4, 0, 5]

[7, 8, 6]

[1, 2, 3]

[4, 5, 0]

[7, 8, 6]

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]