1. **Write a Program to Implement Breadth First Search using Python**

**Procedure**

**Algorithm Steps**

1. Create a queue and enqueue the starting node
2. Mark the starting node as visited
3. While the queue is not empty:  
   a. Dequeue a node from the front  
   b. Process the node (print it, check for goal, etc.)  
   c. Enqueue all adjacent unvisited nodes  
   d. Mark them as visited
4. Repeat until queue is empty

**Program**

from collections import deque

def bfs(graph, start):

"""

Perform Breadth-First Search on a graph

Args:

graph (dict): Adjacency list representation of the graph

start (str/int): Starting node for BFS

Returns:

list: The order in which nodes were visited

"""

# Create a queue for BFS

queue = deque()

# Mark the start node as visited and enqueue it

visited = set()

visited.add(start)

queue.append(start)

# List to store the traversal order

traversal\_order = []

while queue:

# Dequeue a vertex from queue

current\_node = queue.popleft()

traversal\_order.append(current\_node)

# Get all adjacent vertices of the dequeued vertex.

# If an adjacent hasn't been visited, mark it visited and enqueue it

for neighbor in graph[current\_node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

return traversal\_order

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Sample graph represented as adjacency list

graph = {

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

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

print("BFS Traversal:")

print(bfs(graph, 'A')) # Starting from node 'A'

**Output**

A,B,C,D,E,F

1. **Write a Program to Implement Depth First Search using Python**

**Procedure**

**Algorithm Steps**

1. Start at the initial node and mark it as visited
2. Explore an adjacent unvisited node (choose one if multiple exist)
3. Repeat the process with the new node
4. When you reach a node with no unvisited neighbors, backtrack
5. Continue until all reachable nodes are visited

**Program**

# Recursive DFS

def dfs\_recursive(graph, node, visited=None):

if visited is None:

visited = set()

visited.add(node)

print(node, end=" ")

for neighbor in graph[node]:

if neighbor not in visited:

dfs\_recursive(graph, neighbor, visited)

# Iterative DFS

def dfs\_iterative(graph, start):

visited = set()

stack = [start]

while stack:

node = stack.pop()

if node not in visited:

visited.add(node)

print(node, end=" ")

# Push neighbors in reverse order to visit them left-to-right

for neighbor in reversed(graph[node]):

if neighbor not in visited:

stack.append(neighbor)

# Example graph

graph = {

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

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

print("Recursive DFS:")

dfs\_recursive(graph, 'A')

print("\nIterative DFS:")

dfs\_iterative(graph, 'A')

**Output**

A B D E F C

A depth-first search (DFS) starting from node 'A' in the given graph would explore the graph by going as deep as possible along each branch before backtracking. A possible traversal order is A, B, D, E, F, C.

Here's a step-by-step explanation:

1. **Start at A:** The search begins at node A.
2. **Explore B:** 'B' is a neighbor of A.
3. **Explore D:** 'D' is a neighbor of B.
4. **Backtrack:** There are no unvisited neighbors of D, so the search backtracks to B.
5. **Explore E:** 'E' is a neighbor of B (and not yet visited).
6. **Explore F:** 'F' is a neighbor of E.
7. **Backtrack:** There are no unvisited neighbors of F, so the search backtracks to E, then B, then back to A.
8. **Explore C:** 'C' is a neighbor of A (and not yet visited).
9. **Backtrack:** There are no unvisited neighbors of C, so the search backtracks to A and terminates as all reachable nodes have been visited.
10. **Write a Program to Implement** **Tic Tac Toe using Python**

Program

import random

def print\_board(board):

print("\n")

print(f" {board[0]} | {board[1]} | {board[2]} ")

print("-----------")

print(f" {board[3]} | {board[4]} | {board[5]} ")

print("-----------")

print(f" {board[6]} | {board[7]} | {board[8]} ")

print("\n")

def check\_winner(board):

# Check all possible winning combinations

winning\_combinations = [

[0, 1, 2], [3, 4, 5], [6, 7, 8], # rows

[0, 3, 6], [1, 4, 7], [2, 5, 8], # columns

[0, 4, 8], [2, 4, 6] # diagonals

]

for combo in winning\_combinations:

if board[combo[0]] == board[combo[1]] == board[combo[2]] != " ":

return board[combo[0]] # returns the winning player (X or O)

if " " not in board:

return "Tie"

return None

def player\_move(board):

while True:

try:

move = int(input("Enter your move (1-9): ")) - 1

if 0 <= move <= 8 and board[move] == " ":

return move

else:

print("Invalid move. Try again.")

except ValueError:

print("Please enter a number between 1 and 9.")

def computer\_move(board):

# Simple AI: first checks for winning move, then blocks player, then random

empty\_spots = [i for i, spot in enumerate(board) if spot == " "]

# Check for winning move

for spot in empty\_spots:

board\_copy = board.copy()

board\_copy[spot] = "O"

if check\_winner(board\_copy) == "O":

return spot

# Block player's winning move

for spot in empty\_spots:

board\_copy = board.copy()

board\_copy[spot] = "X"

if check\_winner(board\_copy) == "X":

return spot

# Choose center if available

if 4 in empty\_spots:

return 4

# Choose a corner if available

corners = [0, 2, 6, 8]

available\_corners = [c for c in corners if c in empty\_spots]

if available\_corners:

return random.choice(available\_corners)

# Choose a random spot

return random.choice(empty\_spots)

def play\_game():

board = [" "] \* 9

current\_player = "X" # Player is X, computer is O

print("Welcome to Tic-Tac-Toe!")

print("Enter numbers 1-9 to make your move:")

print\_board(["1", "2", "3", "4", "5", "6", "7", "8", "9"])

while True:

if current\_player == "X":

move = player\_move(board)

board[move] = "X"

else:

print("Computer's turn...")

move = computer\_move(board)

board[move] = "O"

print(f"Computer chooses position {move + 1}")

print\_board(board)

result = check\_winner(board)

if result:

if result == "Tie":

print("It's a tie!")

elif result == "X":

print("Congratulations! You win!")

else:

print("Computer wins!")

break

current\_player = "O" if current\_player == "X" else "X"

if \_\_name\_\_ == "\_\_main\_\_":

while True:

play\_game()

play\_again = input("Play again? (y/n): ").lower()

if play\_again != 'y':

print("Thanks for playing!")

break

Algorithm

1. **Initialize the Game**
   * Create a 3x3 board (represented as a list with 9 empty spaces)
   * Set starting player (typically human as "X" and computer as "O")
2. **Display the Board**
   * Print the current state of the board with positions 1-9 for reference.
3. **Game Loop**
   * Repeat until the game ends (win, lose, or tie):
     + **Human Player's Turn ("X")**
       1. Ask the player to input a move (1-9).
       2. Validate the input:
          - Check if the position is within range (1-9).
          - Ensure the selected cell is empty.
       3. Update the board with "X" at the chosen position.
     + **Check for Win/Tie**
       1. If the human wins → End game, declare winner.
       2. If the board is full → Declare a tie.
     + **Computer's Turn ("O")**
       1. Determine the best move using AI logic:
          - First, check if the computer can win immediately.
          - If not, block the human's winning move.
          - Prefer the center (position 5) if available.
          - Choose a corner if available.
          - Otherwise, pick a random valid move.
       2. Update the board with "O" at the chosen position.
     + **Check for Win/Tie Again**
       1. If the computer wins → End game, declare winner.
       2. If the board is full → Declare a tie.
4. **Game Over**
   * Print the final result (win/lose/tie).
   * Ask if the player wants to play again.
5. Program to Implement 8-Puzzle problem using Python.

**Procedure**

1. **Flatten the board** into a tuple (immutable, so it can be stored in a dictionary).

2. Store this tuple in a dictionary (dict) where the value is the number of moves taken to reach it.

3. Perform **BFS** level by level:

* Expand all nodes at depth cnt.
* Generate all valid next moves by sliding the 0 (blank space).
* If the goal state (0,1,2,3,4,5,6,7,8) is found → return the number of moves.

4. If no solution exists, return -1.

**Program**

class Solution:

def solve(self, board):

state\_dict = {}

flatten = []

# Flatten the board

for i in range(len(board)):

flatten += board[i]

flatten = tuple(flatten)

# Store initial state

state\_dict[flatten] = 0

# Goal state

if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):

return 0

return self.get\_paths(state\_dict)

def get\_paths(self, state\_dict):

cnt = 0

while True:

# Nodes at current depth

current\_nodes = [x for x in state\_dict if state\_dict[x] == cnt]

if len(current\_nodes) == 0:

return -1 # no solution

for node in current\_nodes:

next\_moves = self.find\_next(node)

for move in next\_moves:

if move not in state\_dict:

state\_dict[move] = cnt + 1

if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):

return cnt + 1

cnt += 1

def find\_next(self, node):

# Possible moves for each index of 0

moves = {

0: [1, 3],

1: [0, 2, 4],

2: [1, 5],

3: [0, 4, 6],

4: [1, 3, 5, 7],

5: [2, 4, 8],

6: [3, 7],

7: [4, 6, 8],

8: [5, 7],

}

results = []

pos\_0 = node.index(0) # position of the blank

for move in moves[pos\_0]:

new\_node = list(node)

# Swap 0 with the chosen move

new\_node[move], new\_node[pos\_0] = new\_node[pos\_0], new\_node[move]

results.append(tuple(new\_node))

return results

# Test

ob = Solution()

matrix = [

[3, 1, 2],

[4, 7, 5],

[6, 8, 0]

]

print(ob.solve(matrix)) # prints minimum number of moves

5. Program to Implement Water-Jug problem using Python.

**Procedure**

1. **Start** from the initial state (0,0) → both jugs empty.

2. Use a **queue** (FIFO) to store states and their paths.

3. Maintain a **visited set** to avoid repeating states.

4. At each step, generate all possible states by applying the operations:

* Fill Jug1
* Fill Jug2
* Empty Jug1
* Empty Jug2
* Pour Jug1 → Jug2
* Pour Jug2 → Jug1

5. If at any state, the amount of water in Jug1 or Jug2 equals the target d, **stop** and return the path.

6. If the queue becomes empty, it means no solution exists.

**Program**

from collections import deque

def water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target):

# Initial state (both jugs empty)

start = (0, 0)

# Queue for BFS

q = deque([(start, [])]) # (state, path of states)

# Visited states to avoid repetition

visited = set([start])

while q:

(x, y), path = q.popleft()

# If we reach target

if x == target or y == target:

return path + [(x, y)]

# All possible operations

possible\_states = [

(jug1\_capacity, y), # Fill Jug1

(x, jug2\_capacity), # Fill Jug2

(0, y), # Empty Jug1

(x, 0), # Empty Jug2

# Pour Jug1 -> Jug2

(max(0, x - (jug2\_capacity - y)), min(jug2\_capacity, y + x)),

# Pour Jug2 -> Jug1

(min(jug1\_capacity, x + y), max(0, y - (jug1\_capacity - x))),

]

for state in possible\_states:

if state not in visited:

visited.add(state)

q.append((state, path + [(x, y)]))

return None # No solution

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

jug1\_capacity = 4

jug2\_capacity = 3

target = 2

result = water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target)

if result:

print("Solution found:")

for step in result:

print(step)

else:

print("No solution exists.")