Q1. Implement Dijkstra's Algorithm to find the shortest path from a single source for the following graph G.

**SAVITRIBAI PHULE PUNE UNIVERSITY**

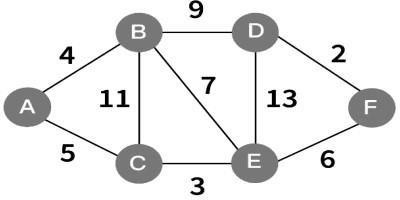
**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

[10 Marks]

[20 Marks]



import heapq

# Graph represented using an adjacency list

graph\_dijkstra = {

'A': [('B', 4), ('C', 5)],

'B': [('A', 4), ('D', 9), ('E', 7)],

'C': [('A', 5), ('E', 3)],

'D': [('B', 9), ('F', 2)],

'E': [('B', 7), ('C', 3), ('F', 13)],

'F': [('D', 2), ('E', 13)]

}

def dijkstra(graph, start):

# Distance dictionary with initial distances set to infinity

distances = {node: float('inf') for node in graph}

distances[start] = 0 # Start node has a distance of 0

priority\_queue = [(0, start)] # Priority queue to get the node with minimum distance

while priority\_queue:

current\_distance, current\_node = heapq.heappop(priority\_queue)

# Only continue if this is the shortest recorded distance to the node

if current\_distance > distances[current\_node]:

continue

# Check neighbors

for neighbor, weight in graph[current\_node]:

distance = current\_distance + weight

# If a shorter path to neighbor is found

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

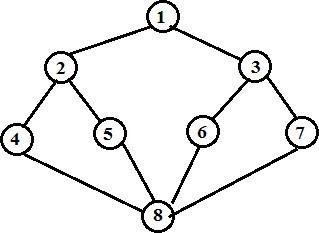
return distances

# Example usage of Dijkstra's Algorithm starting from node 'A'

dijkstra\_result = dijkstra(graph\_dijkstra, 'A')

print("Shortest paths from node A:", dijkstra\_result)

Q2. Write a Python program to implement Depth First Search algorithm. Refer the following graph as an input for the program. [Initial node=1,Goal node=8]



# Graph represented as an adjacency list

graph\_dfs = {

1: [2, 3],

2: [1, 4, 5],

3: [1, 6, 7],

4: [2, 8],

5: [2, 8],

6: [3, 8],

7: [3, 8],

8: [4, 5, 6, 7]

}

def dfs(graph, start, goal, path=None, visited=None):

if path is None:

path = []

if visited is None:

visited = set()

# Add the current node to the path and mark it as visited

path.append(start)

visited.add(start)

# If we have reached the goal node, return the path

if start == goal:

return path

# Recursive DFS on all unvisited neighbors

for neighbor in graph[start]:

if neighbor not in visited:

result = dfs(graph, neighbor, goal, path, visited)

if result: # If a path is found, return it

return result

# Backtrack if no path is found

path.pop()

return None

# Example usage of DFS from node 1 to node 8

dfs\_path\_result = dfs(graph\_dfs, 1, 8)

print("Path from node 1 to node 8 using DFS:", dfs\_path\_result)

OR

Q2. Write a Program to Implement Monkey Banana Problem using Python.

[20 Marks]

class MonkeyBananaProblem:

def \_\_init\_\_(self):

# Initial state

self.monkey\_position = "at\_door" # Options: at\_door, at\_middle, at\_box

self.box\_position = "middle\_of\_room" # Options: middle\_of\_room, under\_banana

self.monkey\_on\_box = False

self.has\_banana = False

def move(self, position):

""" Move the monkey to a specified position """

print(f"Monkey moves to {position}.")

self.monkey\_position = position

def push\_box(self):

""" Push the box under the banana if monkey is at the box position """

if self.monkey\_position == "at\_box" and self.box\_position != "under\_banana":

print("Monkey pushes the box under the banana.")

self.box\_position = "under\_banana"

else:

print("Monkey cannot push the box from here.")

def climb\_box(self):

""" Climb on the box if it is under the banana """

if self.monkey\_position == "at\_box" and self.box\_position == "under\_banana":

print("Monkey climbs on the box.")

self.monkey\_on\_box = True

else:

print("Monkey cannot climb the box from here.")

def grab\_banana(self):

""" Grab the banana if the monkey is on the box under the banana """

if self.monkey\_on\_box and self.box\_position == "under\_banana":

print("Monkey grabs the banana!")

self.has\_banana = True

else:

print("Monkey cannot reach the banana from here.")

def solve(self):

""" Solve the Monkey Banana Problem """

# Step 1: Move to the box

self.move("at\_box")

# Step 2: Push the box under the banana

self.push\_box()

# Step 3: Climb on the box

self.climb\_box()

# Step 4: Grab the banana

self.grab\_banana()

# Check if the goal is achieved

if self.has\_banana:

print("Monkey successfully got the banana!")

else:

print("Monkey failed to get the banana.")

# Run the Monkey Banana Problem solver

problem = MonkeyBananaProblem()

problem.solve()

Q1. Write a Program to Implement Depth First Search using Python.

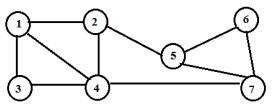
**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

[10 Marks]



# Graph represented as an adjacency list

graph = {

1: [2, 3, 4],

2: [1, 5, 6],

3: [1, 4],

4: [1, 3, 5],

5: [2, 4, 6, 7],

6: [2, 5],

7: [5]

}

# DFS function

def dfs(graph, start, visited=None):

if visited is None:

visited = set() # Use a set to keep track of visited nodes

visited.add(start)

print(start, end=" ") # Print the node as we visit it

# Recur for all the vertices adjacent to this vertex

for neighbor in graph[start]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

# Run DFS starting from node 1

dfs(graph, 1)

Q2. Write a program to implement A\* algorithm.

[20 Marks]

[20 Marks]

import heapq

# Manhattan distance heuristic function

def manhattan\_distance(start, goal):

return abs(start[0] - goal[0]) + abs(start[1] - goal[1])

# A\* algorithm

def a\_star(start, goal, grid):

# Open list (priority queue)

open\_list = []

heapq.heappush(open\_list, (0 + manhattan\_distance(start, goal), 0, start)) # (f, g, node)

# Came from map to reconstruct the path

came\_from = {}

# Cost from start to the current node

g\_cost = {start: 0}

while open\_list:

# Get the node with the lowest f value

\_, current\_g, current\_node = heapq.heappop(open\_list)

if current\_node == goal:

# Reconstruct the path

path = []

while current\_node in came\_from:

path.append(current\_node)

current\_node = came\_from[current\_node]

path.append(start)

return path[::-1] # Return the path from start to goal

# Get neighbors

neighbors = [(current\_node[0] + 1, current\_node[1]), (current\_node[0] - 1, current\_node[1]),

(current\_node[0], current\_node[1] + 1), (current\_node[0], current\_node[1] - 1)]

for neighbor in neighbors:

# Check if neighbor is within grid bounds and is walkable (assuming grid is a 2D list)

if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]) and grid[neighbor[0]][neighbor[1]] != 1:

tentative\_g = current\_g + 1

if neighbor not in g\_cost or tentative\_g < g\_cost[neighbor]:

g\_cost[neighbor] = tentative\_g

f = tentative\_g + manhattan\_distance(neighbor, goal)

heapq.heappush(open\_list, (f, tentative\_g, neighbor))

came\_from[neighbor] = current\_node

return None # If there's no path

# Example usage

grid = [

[0, 0, 0, 0],

[0, 1, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]

]

start = (0, 0)

goal = (3, 3)

path = a\_star(start, goal, grid)

print("Path from start to goal:", path)

OR

Q2. Write a Program to Implement Tic-Tac-Toe game using Python.

SLIP 2

# Function to print the Tic-Tac-Toe board

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

# Function to check if there's a winner

def check\_winner(board, player):

# Check rows, columns, and diagonals for a win

for i in range(3):

if all([cell == player for cell in board[i]]) or all([board[j][i] == player for j in range(3)]):

return True

if board[0][0] == player and board[1][1] == player and board[2][2] == player:

return True

if board[0][2] == player and board[1][1] == player and board[2][0] == player:

return True

return False

# Function to check if the board is full (no more moves)

def is\_board\_full(board):

for row in board:

if ' ' in row:

return False

return True

# Main function to play Tic-Tac-Toe

def tic\_tac\_toe():

# Initialize the board

board = [[' ' for \_ in range(3)] for \_ in range(3)]

current\_player = 'X'

while True:

print\_board(board)

print(f"Player {current\_player}'s turn")

# Get the player's move

row, col = map(int, input(f"Enter row and column (0-2) for {current\_player}: ").split())

if board[row][col] != ' ':

print("Cell already taken! Try again.")

continue

# Place the player's symbol on the board

board[row][col] = current\_player

# Check if the current player has won

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

# Check if the board is full

if is\_board\_full(board):

print\_board(board)

print("It's a tie!")

break

# Switch player

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

# Run the Tic-Tac-Toe game

tic\_tac\_toe()

Q1. Given an array of integrs, write a Python Program to sort the array in ascending order using Selection Sort.

[10 Marks]

[20 Marks]

[20 Marks]

def selection\_sort(arr):

n = len(arr)

for i in range(n):

# Assume the current index as the minimum

min\_index = i

# Find the minimum element in the unsorted part

for j in range(i + 1, n):

if arr[j] < arr[min\_index]:

min\_index = j

# Swap the found minimum element with the first element

arr[i], arr[min\_index] = arr[min\_index], arr[i]

# Example usage

array = [64, 25, 12, 22, 11]

print("Original array:", array)

selection\_sort(array)

print("Sorted array:", array)

Q2. Implement Minimum Spanning Tree using Kruskal’s Algorithm.

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.edges = []

def add\_edge(self, u, v, w):

self.edges.append((w, u, v))

def find(self, parent, i):

if parent[i] == i:

return i

return self.find(parent, parent[i])

def union(self, parent, rank, x, y):

xroot = self.find(parent, x)

yroot = self.find(parent, y)

if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]:

parent[yroot] = xroot

else:

parent[yroot] = xroot

rank[xroot] += 1

def kruskal\_mst(self):

result = [] # Store the resultant MST

i, e = 0, 0 # Initialize indices for sorted edges

self.edges = sorted(self.edges) # Sort edges by weight

parent = []

rank = []

for node in range(self.V):

parent.append(node)

rank.append(0)

while e < self.V - 1:

w, u, v = self.edges[i]

i += 1

x = self.find(parent, u)

y = self.find(parent, v)

if x != y:

e += 1

result.append((u, v, w))

self.union(parent, rank, x, y)

# Print the resultant MST

print("Edges in the MST:")

for u, v, w in result:

print(f"{u} -- {v} == {w}")

# Example usage

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

g.kruskal\_mst()

OR

Q2. Write a Program to implement 8-Puzzle problem using Python.

SLIP 3

import heapq

# Function to calculate Manhattan distance

def manhattan\_distance(state, goal):

distance = 0

for i in range(3):

for j in range(3):

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

x, y = divmod(goal.index(state[i][j]), 3)

distance += abs(x - i) + abs(y - j)

return distance

# A\* algorithm for 8-puzzle

def a\_star\_8\_puzzle(start, goal):

goal\_flat = [tile for row in goal for tile in row]

start\_flat = [tile for row in start for tile in row]

# Priority queue

open\_list = []

heapq.heappush(open\_list, (0, start\_flat, [])) # (f, state, path)

# Set of visited states

visited = set()

while open\_list:

\_, current, path = heapq.heappop(open\_list)

# Check if we have reached the goal

if current == goal\_flat:

return path + [goal]

visited.add(tuple(current))

# Find the position of the empty tile (0)

zero\_pos = current.index(0)

x, y = divmod(zero\_pos, 3)

# Possible moves (up, down, left, right)

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

for dx, dy in moves:

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

if 0 <= new\_x < 3 and 0 <= new\_y < 3:

# Calculate new position of 0

new\_pos = new\_x \* 3 + new\_y

new\_state = current[:]

new\_state[zero\_pos], new\_state[new\_pos] = new\_state[new\_pos], new\_state[zero\_pos]

if tuple(new\_state) not in visited:

g = len(path) + 1

h = manhattan\_distance([new\_state[i:i+3] for i in range(0, 9, 3)], goal)

f = g + h

heapq.heappush(open\_list, (f, new\_state, path + [[new\_state[i:i+3] for i in range(0, 9, 3)]]))

return None # No solution

# Example usage

start = [

[1, 2, 3],

[4, 0, 5],

[7, 8, 6]

]

goal = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

solution = a\_star\_8\_puzzle(start, goal)

if solution:

for step in solution:

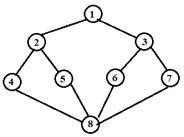
for row in step:

print(row)

print()

else:

print("No solution found.")

Q1. Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1, Goal node=8].

[10 Marks]

[20 Marks]

[20 Marks]

# Graph represented as an adjacency list

graph = {

1: [2, 3],

2: [4, 5],

3: [6, 7],

4: [],

5: [8],

6: [8],

7: [8],

8: []

}

# Depth First Search implementation

def dfs(graph, start, goal, visited=None):

if visited is None:

visited = set()

# Mark the current node as visited

visited.add(start)

print(start, end=" ")

# If the goal node is found, stop the search

if start == goal:

return True

# Visit each neighbor

for neighbor in graph[start]:

if neighbor not in visited:

if dfs(graph, neighbor, goal, visited):

return True

return False

# Example usage

start\_node = 1

goal\_node = 8

print("DFS Path from node", start\_node, "to node", goal\_node, ":")

dfs(graph, start\_node, goal\_node)

Q2. Write a program to implement AO\* algorithm.

class Node:

def \_\_init\_\_(self, name, is\_goal=False):

self.name = name

self.is\_goal = is\_goal

self.children = [] # AND-OR children

self.cost = float('inf') # Initial cost is set to infinity

self.best\_child = None # Best child (if any)

def add\_child(self, node, cost):

self.children.append((node, cost))

def ao\_star(node):

# Base case: return if node is a goal node

if node.is\_goal:

node.cost = 0

return node.cost

# If node has children, recursively calculate cost

min\_cost = float('inf')

for child, cost in node.children:

child\_cost = ao\_star(child)

total\_cost = cost + child\_cost

if total\_cost < min\_cost:

min\_cost = total\_cost

node.best\_child = child

# Update the node's cost with the minimum cost found

node.cost = min\_cost

return node.cost

# Example usage

n1 = Node('Start')

n2 = Node('A')

n3 = Node('B', is\_goal=True)

n4 = Node('C')

n5 = Node('D', is\_goal=True)

# Add connections (AND-OR structure)

n1.add\_child(n2, 1)

n1.add\_child(n3, 2)

n2.add\_child(n4, 1)

n2.add\_child(n5, 2)

# Calculate AO\* solution

print("AO\* Algorithm:")

cost = ao\_star(n1)

print(f"Minimum cost to goal: {cost}")

print(f"Best child from start node: {n1.best\_child.name}")

OR

Q2. Write a Program to Implement Water-Jug problem using Python.

SLIP 4

from collections import deque

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

# Initialize a queue and set of visited states

queue = deque([(0, 0)]) # Starting state with both jugs empty

visited = set((0, 0))

while queue:

jug1, jug2 = queue.popleft()

# Print the state

print(f"Jug1: {jug1}, Jug2: {jug2}")

# Check if we reached the target

if jug1 == target or jug2 == target:

return True

# Generate all possible moves

moves = [

(jug1\_capacity, jug2), # Fill jug1

(jug1, jug2\_capacity), # Fill jug2

(0, jug2), # Empty jug1

(jug1, 0), # Empty jug2

(min(jug1 + jug2, jug1\_capacity), max(0, jug2 - (jug1\_capacity - jug1))), # Pour jug2 into jug1

(max(0, jug1 - (jug2\_capacity - jug2)), min(jug1 + jug2, jug2\_capacity)) # Pour jug1 into jug2

]

# Enqueue all valid moves

for new\_jug1, new\_jug2 in moves:

if (new\_jug1, new\_jug2) not in visited:

visited.add((new\_jug1, new\_jug2))

queue.append((new\_jug1, new\_jug2))

return False

# Example usage

jug1\_capacity = 4

jug2\_capacity = 3

target = 2

print("Water Jug Problem solution steps:")

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

Q1. Implement Dijkstra’s algorithm to find the shortest path from a source node to all other nodes in a weighted graph.

[10 Marks]

[20 Marks]

[20 Marks]

import heapq

def dijkstra(graph, start):

# Initialize distances with infinity for all nodes except the start node

distances = {node: float('infinity') for node in graph}

distances[start] = 0

# Priority queue to store (distance, node)

priority\_queue = [(0, start)]

while priority\_queue:

current\_distance, current\_node = heapq.heappop(priority\_queue)

# If we encounter a greater distance in the queue, skip it

if current\_distance > distances[current\_node]:

continue

# Explore neighbors

for neighbor, weight in graph[current\_node].items():

distance = current\_distance + weight

# Only consider this new path if it's better

if distance < distances[neighbor]:

distances[neighbor] = distance

heapq.heappush(priority\_queue, (distance, neighbor))

return distances

# Example usage

graph = {

'A': {'B': 1, 'C': 4},

'B': {'A': 1, 'D': 2, 'E': 5},

'C': {'A': 4, 'F': 3},

'D': {'B': 2, 'H': 7, 'I': 6},

'E': {'B': 5, 'I': 3},

'F': {'C': 3, 'G': 8, 'K': 2},

'G': {'F': 8},

'H': {'D': 7},

'I': {'D': 6, 'E': 3},

'K': {'F': 2}

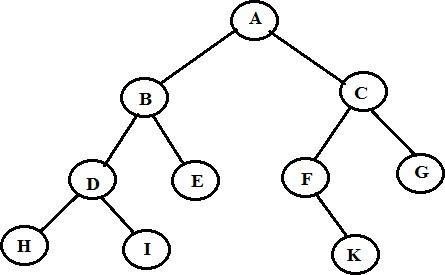
}

# Start from node 'A'

start\_node = 'A'

shortest\_paths = dijkstra(graph, start\_node)

print("Shortest distances from node", start\_node, ":", shortest\_paths)

Q2. Write a program to implement Iterative Deepening DFS algorithm. [ Goal Node =G]

def depth\_limited\_search(graph, start, goal, limit):

if start == goal:

return [start]

if limit <= 0:

return None

for neighbor in graph[start]:

path = depth\_limited\_search(graph, neighbor, goal, limit - 1)

if path is not None:

return [start] + path

return None

def iterative\_deepening\_dfs(graph, start, goal):

depth = 0

while True:

path = depth\_limited\_search(graph, start, goal, depth)

if path is not None:

return path

depth += 1

# Graph structure from the given diagram

graph = {

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

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

'C': ['F', 'G'],

'D': ['H', 'I'],

'E': [],

'F': [],

'G': ['K'],

'H': [],

'I': [],

'K': []

}

# Starting from node 'A' with the goal node 'G'

start\_node = 'A'

goal\_node = 'G'

path\_to\_goal = iterative\_deepening\_dfs(graph, start\_node, goal\_node)

print("Path to goal node", goal\_node, ":", path\_to\_goal)

OR

Q2. Write a Program to Implement Travelling Salesman Problem using Python.

from itertools import permutations

def traveling\_salesman(graph, start):

# List of all nodes except the start node

nodes = list(graph.keys())

nodes.remove(start)

# Initialize minimum path variables

min\_path\_cost = float('inf')

best\_path = []

# Generate all permutations of nodes

for perm in permutations(nodes):

# Calculate the cost of the path starting and ending at the start node

current\_cost = 0

current\_path = [start] + list(perm) + [start]

for i in range(len(current\_path) - 1):

current\_cost += graph[current\_path[i]][current\_path[i + 1]]

# Update minimum path cost if this path is cheaper

if current\_cost < min\_path\_cost:

min\_path\_cost = current\_cost

best\_path = current\_path

return min\_path\_cost, best\_path

# Example usage

graph = {

'A': {'B': 10, 'C': 15, 'D': 20},

'B': {'A': 10, 'C': 35, 'D': 25},

'C': {'A': 15, 'B': 35, 'D': 30},

'D': {'A': 20, 'B': 25, 'C': 30}

}

# Start from node 'A'

start\_node = 'A'

min\_cost, best\_route = traveling\_salesman(graph, start\_node)

print("Minimum cost:", min\_cost)

print("Best route:", best\_route)

Q1. Given an array of integers, write a Python Program to sort the array in descending order using Selection Sort.

[10 Marks]

[20 Marks]

[20 Marks]

def selection\_sort\_descending(arr):

# Traverse through all elements in the array

n = len(arr)

for i in range(n):

# Find the index of the maximum element in the unsorted part of the array

max\_index = i

for j in range(i + 1, n):

if arr[j] > arr[max\_index]:

max\_index = j

# Swap the found maximum element with the first element of the unsorted part

arr[i], arr[max\_index] = arr[max\_index], arr[i]

return arr

# Example usage

array = [64, 25, 12, 22, 11]

sorted\_array = selection\_sort\_descending(array)

print("Sorted array in descending order:", sorted\_array)

Q2. Develop an elementary Chabot for any suitable customer interaction application.

def chatbot():

print("Hello! I am your customer support assistant. How can I help you today?")

print("Type 'bye' to exit the chat.")

# Dictionary of responses

responses = {

"hello": "Hi there! How can I assist you today?",

"hi": "Hello! How can I help you?",

"help": "I'm here to help! You can ask me about our products, your order status, and more.",

"order": "Can you please provide your order number for more details?",

"status": "Checking order status... Please provide your order ID.",

"product": "We have a variety of products. Can you specify which one you're interested in?",

"return": "To initiate a return, please visit our Returns page or provide your order ID here.",

"refund": "Refunds are processed within 5-7 business days after the return is received.",

"bye": "Thank you for visiting! Have a great day!"

}

while True:

# Get user input and convert it to lowercase

user\_input = input("You: ").lower()

if user\_input == "bye":

print("Chatbot: " + responses["bye"])

break

# Check for keywords in the user's input

response\_found = False

for keyword, response in responses.items():

if keyword in user\_input:

print("Chatbot:", response)

response\_found = True

break

# If no keyword matched, provide a generic response

if not response\_found:

print("Chatbot: I'm sorry, I didn't understand that. Could you please rephrase?")

# Start the chatbot

chatbot()

OR

Q2. Write a Program to Implement Water-Jug problem using Python.

SLIP 6

from collections import deque

# Function to solve the Water Jug problem using BFS

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

# Queue for BFS

queue = deque()

# Set to store visited states

visited = set()

# Initialize queue with the starting state (0, 0)

queue.append((0, 0))

visited.add((0, 0))

while queue:

# Get the current state

jug1, jug2 = queue.popleft()

# If either jug has the target amount, return the solution

if jug1 == target or jug2 == target:

return True

# Possible next states

possible\_states = set()

# Fill Jug 1

possible\_states.add((jug1\_capacity, jug2))

# Fill Jug 2

possible\_states.add((jug1, jug2\_capacity))

# Empty Jug 1

possible\_states.add((0, jug2))

# Empty Jug 2

possible\_states.add((jug1, 0))

# Pour Jug 1 -> Jug 2

pour\_to\_jug2 = min(jug1, jug2\_capacity - jug2)

possible\_states.add((jug1 - pour\_to\_jug2, jug2 + pour\_to\_jug2))

# Pour Jug 2 -> Jug 1

pour\_to\_jug1 = min(jug2, jug1\_capacity - jug1)

possible\_states.add((jug1 + pour\_to\_jug1, jug2 - pour\_to\_jug1))

# Explore each possible state

for state in possible\_states:

if state not in visited:

visited.add(state)

queue.append(state)

# If no solution was found

return False

# Example usage

jug1\_capacity = 4

jug2\_capacity = 3

target\_amount = 2

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

print("Solution found to measure", target\_amount, "liters.")

else:

print("No solution found.")

Q1. Write a Program to Implement Breadth First Search using Python.

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

[10 Marks]

[20 Marks]

[20 Marks]

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

while queue:

# Dequeue a node from the front

node = queue.popleft()

print(node, end=" ")

# Go through all the neighbors of this node

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

# Graph representation of the given tree

graph = {

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

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

'C': ['F', 'G'],

'D': ['H', 'I'],

'E': [],

'F': [],

'G': ['K'],

'H': [],

'I': [],

'K': []

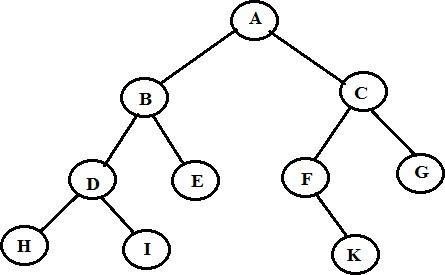
}

# Perform BFS from the starting node 'A'

print("BFS traversal starting from node A:")

bfs(graph, 'A')

Q2. Write a program to implement Iterative Deepening DFS algorithm. [ Goal Node =G]



def dls(graph, node, goal, depth):

# Perform Depth-Limited Search up to a certain depth

if depth == 0 and node == goal:

return True

if depth > 0:

for neighbor in graph.get(node, []):

if dls(graph, neighbor, goal, depth - 1):

return True

return False

def iddfs(graph, start, goal):

# Iteratively increase depth limit until we find the goal node

depth = 0

while True:

print(f"Searching at depth {depth}")

if dls(graph, start, goal, depth):

print(f"Goal node '{goal}' found at depth {depth}")

return True

depth += 1

# Graph representation of the given tree

graph = {

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

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

'C': ['F', 'G'],

'D': ['H', 'I'],

'E': [],

'F': [],

'G': ['K'],

'H': [],

'I': [],

'K': []

}

# Perform IDDFS to find the goal node 'G' starting from 'A'

print("\nIDDFS search for node 'G':")

iddfs(graph, 'A', 'G')

OR

Q2. Write a program to conduct min - max algorithm

**\_**

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Q1. Write a program to implement Best First Search.

[10 Marks]

[20 Marks]

[20 Marks]

Q2. Write a Program to implement 8-Puzzle problem using Python.

OR

Q2. Solve traveling salesman problem using artificial intelligence technique.

Q3. Viva [5 Marks]

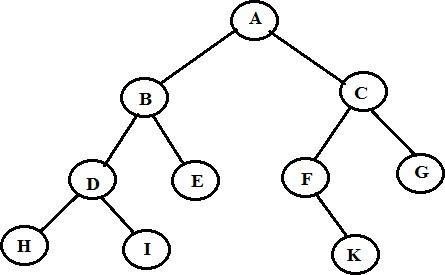
SLIP 8

Q1. Write a Program to Implement Depth First Search using Python.

[10 Marks]

Q2. Write a program to implement Iterative Deepening DFS algorithm. [ Goal Node =G] [

20 Marks]



OR

Q2. Develop Healthcare Appointment Bot (simple chatbot that helps patients book appointments or find clinic information.)

Q3. Viva [5 Marks]

SLIP 9

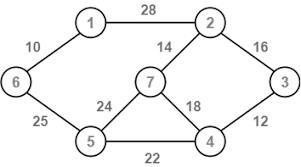
[20 Marks]

Q1. Given a number of cities and the cost of connecting them, find the minimum cost to connect all cities. Use Prim's algorithm to solve.

[10 Marks]

[20 Marks]

[20 Marks]



Q2. Write a program to implement A\* algorithm.

OR

Q2. Write a Python program to implement Mini-Max Algorithm.

Q3. Viva [5 Marks]

SLIP 10

**\_**

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

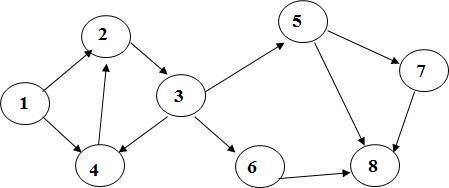
**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Q1. Given an array of integers, write a Python Program to sort the array in ascending order using Selection Sort.

[10 Marks]

Q2. Write a Python program to implement Breadth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1,Goal node=8]

[ 20 Marks ]

[20 Marks]

OR

Q2. Write a Program to Implement Monkey Banana Problem using Python.

Q3. Viva [5 Marks]

SLIP 11

Q1. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph.

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

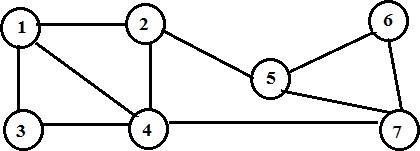
**Time: 3 Hours Max. Marks:35**

[10 Marks]

Q2. Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=2,Goal node=7]

[20 Marks]

[20 Marks]



OR

Q2. Write a Program to Implement Tic-Tac-Toe game using Python.

Q3. Viva [5 Marks]

SLIP 12

Q3. Viva [5 Marks]

SLIP 13

Q1. Write a Program to Implement Breadth First Search using Python. Q2. Implement AO star Algorithm.

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

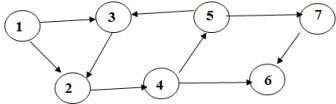
[10 Marks]

[20 Marks]

[20 Marks]

OR

Q2. Write a Program to Implement Depth First Search using Python.

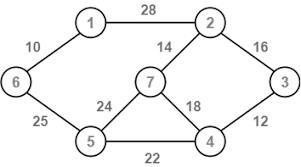


Q1. Given a number of cities and the cost of connecting them, find the minimum cost to connect all cities. Use Prim's algorithm to solve.

[10 Marks]

[20 Marks]

[20 Marks]



Q2. Write a python program to implement A star Algorithm.

OR

Q2. Write a program to solve Missionaries and Cannibals problem.

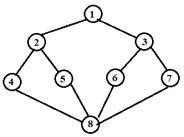
Q3. Viva [5 Marks]

SLIP 14

Q1. Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1,Goal node=8]

[10 Marks]

[20 Marks]



Q2. Write a Program to Implement Alpha-Beta Pruning using Python.

OR

Q2. Develop a Restaurant Reservation Assistant (A simple chatbot that helps customers make reservations or ask about the menu.)

Q3. Viva [5 Marks]

SLIP 15

[20 Marks]

Q3. Viva [5 Marks]

SLIP 16

Q1. Write a Python program to implement Depth First Search algorithm. Refer the following graph as an Input for the program. [Initial node=1,Goal node=8].

**SAVITRIBAI PHULE PUNE UNIVERSITY**

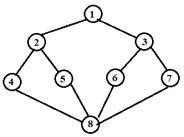
**M.Sc. (Computer Application) Sem-III Practical Examination (2023 Pattern) SUBJECT: CA-604-MJP**

**Lab Course on CA-601-MJ (Artificial Intelligence)**

**Time: 3 Hours Max. Marks:35**

[10 Marks]

[20 Marks]

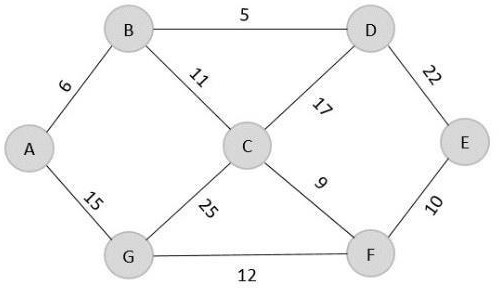


Q2. Write a Program to Implement Tic-Tac-Toe game using Python.

OR

Q2. Given a number of cities and the cost of connecting them, find the minimum cost to connect all cities. Use Kruskal's algorithm to solve.

[20 Marks]



Q1. Write a Program to Implement Depth First Search using Python. Q2. Write a Python program to solve water jug problem.

[10 Marks]

[20 Marks]

[20 Marks]

OR

Q2. Write a program to implement Tic-Tac-Toe\_take.

Q3. Viva [5 Marks]

SLIP 17

Q1. Write a Program to Implement Breadth First Search using Python.

[10 Marks]

[20 Marks]

Q2. Write a Program to Implement Tic-Tac-Toe game using Python.

OR

Q2. Develop a FAQ Bot for a University Website (This chatbot can help students with common questions about admissions, course registration, or contact details.)

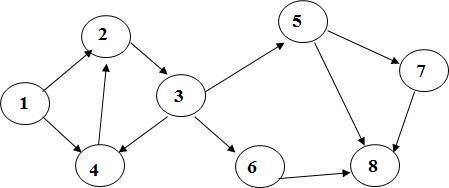
Q3. Viva [5 Marks]

SLIP 18

[20 Marks]

Q1. Write a Program to Implement Depth First Search using Python.

[10 Marks]

Q2. Write a Python program to implement Breadth First Search algorithm. Refer the following graph as an Input for the program.[Initial node=1,Goal node=8]

[20 Marks]

[20 Marks]

OR

Q2. Write a program to implement A\* algorithm.

Q3. Viva [5 Marks]

SLIP 19

Q3. Viva [5 Marks]

SLIP 20

Q1. Write a Program to Implement Depth First Search using Python. Q2. Write a program to implement AO\* algorithm.

[10 Marks]

[20 Marks]

OR

Q2. Given a number of cities and the cost of connecting them, find the minimum cost to connect all cities. Use Kruskal's algorithm to solve.

[20 Marks]

