| Assignment No: 1  Title:Implement depth first search algorithm and Breadth First Search algorithm, use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.  CODE:  from collections import defaultdict, deque  class Graph:  def \_\_init\_\_(self):  self.graph = defaultdict(list)  def add\_edge(self, u, v):  self.graph[u].append(v)  self.graph[v].append(u) # For undirected graph  def dfs(self, vertex, visited):  visited[vertex] = True  print(vertex, end=' ')  for neighbor in self.graph[vertex]:  if not visited[neighbor]:  self.dfs(neighbor, visited)  def recursive\_dfs(self, vertex):  visited = [False] \* (max(self.graph) + 1)  self.\_recursive\_dfs(vertex, visited)  def \_recursive\_dfs(self, vertex, visited):  visited[vertex] = True  print(vertex, end=' ')  for neighbor in self.graph[vertex]:  if not visited[neighbor]:  self.\_recursive\_dfs(neighbor, visited)  def bfs(self, start):  visited = [False] \* (max(self.graph) + 1)  queue = deque([start])  while queue:  current = queue.popleft()  if not visited[current]:  print(current, end=' ')  visited[current] = True  for neighbor in self.graph[current]:  if not visited[neighbor]:  queue.append(neighbor)  # Example usage:  g = Graph()  g.add\_edge(0, 1)  g.add\_edge(0, 2)  g.add\_edge(1, 2)  g.add\_edge(2, 0)  g.add\_edge(2, 3)  g.add\_edge(3, 3)  print("DFS:")  g.dfs(2, [False] \* (max(g.graph) + 1)) # Start DFS from vertex 2  print("\nRecursive DFS:")  g.recursive\_dfs(2) # Start recursive DFS from vertex 2  print("\nBFS:")  g.bfs(2) # Start BFS from vertex 2  from collections import defaultdict, deque  class Graph:  def \_\_init\_\_(self):  self.graph = defaultdict(list)  def add\_edge(self, u, v):  self.graph[u].append(v)  self.graph[v].append(u) # For undirected graph  def dfs(self, vertex, visited):  visited[vertex] = True  print(vertex, end=' ')  for neighbor in self.graph[vertex]:  if not visited[neighbor]:  self.dfs(neighbor, visited)  def recursive\_dfs(self, vertex):  visited = [False] \* (max(self.graph) + 1)  self.\_recursive\_dfs(vertex, visited)  def \_recursive\_dfs(self, vertex, visited):  visited[vertex] = True  print(vertex, end=' ')  for neighbor in self.graph[vertex]:  if not visited[neighbor]:  self.\_recursive\_dfs(neighbor, visited)  def bfs(self, start):  visited = [False] \* (max(self.graph) + 1)  queue = deque([start])  while queue:  current = queue.popleft()  if not visited[current]:  print(current, end=' ')  visited[current] = True  for neighbor in self.graph[current]:  if not visited[neighbor]:  queue.append(neighbor)  # Example usage:  g = Graph()  g.add\_edge(0, 1)  g.add\_edge(0, 2)  g.add\_edge(1, 2)  g.add\_edge(2, 0)  g.add\_edge(2, 3)  g.add\_edge(3, 3)  print("DFS:")  g.dfs(2, [False] \* (max(g.graph) + 1)) # Start DFS from vertex 2  print("\nRecursive DFS:")  g.recursive\_dfs(2) # Start recursive DFS from vertex 2  print("\nBFS:")  g.bfs(2) # Start BFS from vertex 2  OUTPUT:  DFS:  2 0 1 3  Recursive DFS:  2 0 1 3  BFS:  2 0 1 3 |
| --- |

| Assignment No: 2  Title:  Implement A star Algorithm for 8-Puzzle search problem  CODE:  import heapq  import numpy as np  class PuzzleNode:  def \_\_init\_\_(self, state, parent=None, action=None, cost=0, heuristic=0):  self.state = state  self.parent = parent  self.action = action  self.cost = cost  self.heuristic = heuristic  def \_\_lt\_\_(self, other):  return (self.cost + self.heuristic) < (other.cost + other.heuristic)  def print\_solution(node):  if node is not None:  print\_solution(node.parent)  print(node.state)  print("")  def calculate\_manhattan\_distance(state, goal\_state):  distance = 0  for i in range(3):  for j in range(3):  value = state[i][j]  if value != 0:  goal\_position = np.where(goal\_state == value)  distance += abs(i - goal\_position[0][0]) + abs(j - goal\_position[1][0])  return distance  def get\_neighbors(node, goal\_state):  neighbors = []  blank\_position = np.where(node.state == 0)  possible\_moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]  for move in possible\_moves:  new\_blank\_position = (blank\_position[0][0] + move[0], blank\_position[1][0] + move[1])  if 0 <= new\_blank\_position[0] < 3 and 0 <= new\_blank\_position[1] < 3:  new\_state = np.copy(node.state)  new\_state[blank\_position], new\_state[new\_blank\_position] = new\_state[new\_blank\_position], new\_state[blank\_position]  action = f"Move {node.state[new\_blank\_position]} to {blank\_position}"  cost = node.cost + 1  heuristic = calculate\_manhattan\_distance(new\_state, goal\_state)  neighbor\_node = PuzzleNode(new\_state, node, action, cost, heuristic)  neighbors.append(neighbor\_node)  return neighbors  def a\_star(initial\_state, goal\_state):  initial\_node = PuzzleNode(initial\_state)  goal\_node = PuzzleNode(goal\_state)  heap = [initial\_node]  heapq.heapify(heap)  visited = set()  while heap:  current\_node = heapq.heappop(heap)  if np.array\_equal(current\_node.state, goal\_state):  print("Solution found!")  print\_solution(current\_node)  break  if tuple(map(tuple, current\_node.state)) in visited:  continue  visited.add(tuple(map(tuple, current\_node.state)))  neighbors = get\_neighbors(current\_node, goal\_state)  for neighbor in neighbors:  if tuple(map(tuple, neighbor.state)) not in visited:  heapq.heappush(heap, neighbor)  # Example usage:  initial\_state = np.array([[1, 2, 3], [4, 0, 5], [6, 7, 8]])  goal\_state = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])  a\_star(initial\_state, goal\_state)  Output:  Solution found!  [[1 2 3]  [4 0 5]  [6 7 8]]  [[1 2 3]  [4 5 0]  [6 7 8]]  [[1 2 3]  [4 5 8]  [6 7 0]]  [[1 2 3]  [4 5 8]  [6 0 7]]  [[1 2 3]  [4 5 8]  [0 6 7]]  [[1 2 3]  [0 5 8]  [4 6 7]]  [[1 2 3]  [5 0 8]  [4 6 7]]  [[1 2 3]  [5 6 8]  [4 0 7]]  [[1 2 3]  [5 6 8]  [4 7 0]]  [[1 2 3]  [5 6 0]  [4 7 8]]  [[1 2 3]  [5 0 6]  [4 7 8]]  [[1 2 3]  [0 5 6]  [4 7 8]]  [[1 2 3]  [4 5 6]  [0 7 8]]  [[1 2 3]  [4 5 6]  [7 0 8]]  [[1 2 3]  [4 5 6]  [7 8 0]] |
| --- |

| **Assignment No: 3**  **Implement Greedy search algorithm for Job Scheduling Problem**  **CODE:**  def job\_scheduling(jobs):  # Sort jobs in decreasing order of profit  jobs.sort(key=lambda x: x[2], reverse=True)  n = len(jobs)  result = [-1] \* n # Initialize result array to store the scheduled jobs  slots = [False] \* n # Initialize an array to track available slots  for i in range(n):  # Find a suitable slot for the job with the highest profit  for j in range(min(n, jobs[i][1]) - 1, -1, -1):  if not slots[j]:  result[j] = i  slots[j] = True  break  # Filter out the unscheduled jobs  scheduled\_jobs = [(jobs[result[i]][0], jobs[result[i]][1], jobs[result[i]][2]) for i in range(n) if result[i] != -1]  return scheduled\_jobs  # Example usage:  jobs = [('J1', 2, 60), ('J2', 1, 100), ('J3', 3, 20), ('J4', 2, 40)]  scheduled\_jobs = job\_scheduling(jobs)  print("Scheduled Jobs:")  for job in scheduled\_jobs:  print(f"Job {job[0]} with deadline {job[1]} and profit {job[2]}")  **Output:**  Scheduled Jobs:  Job J2 with deadline 1 and profit 100  Job J1 with deadline 2 and profit 60  Job J3 with deadline 3 and profit 20 |
| --- |

| **Assignment No: 4**  **Tile: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph colouring problem**  **CODE:**  class NQueensBacktracking:  def \_\_init\_\_(self, n):  self.n = n  self.board = [[0] \* n for \_ in range(n)]  self.solutions = []  def is\_safe(self, row, col):  # Check if there is a queen in the same column  for i in range(row):  if self.board[i][col] == 1:  return False  # Check upper left diagonal  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  if self.board[i][j] == 1:  return False  # Check upper right diagonal  for i, j in zip(range(row, -1, -1), range(col, self.n)):  if self.board[i][j] == 1:  return False  return True  def solve(self, row=0):  if row == self.n:  # Found a solution  self.solutions.append([row[:] for row in self.board])  return  for col in range(self.n):  if self.is\_safe(row, col):  self.board[row][col] = 1  self.solve(row + 1)  self.board[row][col] = 0 # Backtrack  def get\_solutions(self):  return self.solutions  class NQueensBranchAndBound:  def \_\_init\_\_(self, n):  self.n = n  self.solutions = []  def is\_safe(self, row, col, placement):  # Check if there is a queen in the same column  for i in range(row):  if placement[i] == col or \  placement[i] - i == col - row or \  placement[i] + i == col + row:  return False  return True  def solve(self, row, placement):  if row == self.n:  # Found a solution  self.solutions.append(placement[:])  return  for col in range(self.n):  if self.is\_safe(row, col, placement):  placement[row] = col  self.solve(row + 1, placement)  def branch\_and\_bound\_solve(self):  placement = [-1] \* self.n  self.solve(0, placement)  def get\_solutions(self):  return self.solutions  # Example usage for Backtracking  n\_queens\_backtracking = NQueensBacktracking(4)  n\_queens\_backtracking.solve()  print("Solutions using Backtracking:")  for solution in n\_queens\_backtracking.get\_solutions():  for row in solution:  print(row)  print()  # Example usage for Branch and Bound  n\_queens\_branch\_and\_bound = NQueensBranchAndBound(4)  n\_queens\_branch\_and\_bound.branch\_and\_bound\_solve()  print("\nSolutions using Branch and Bound:")  for solution in n\_queens\_branch\_and\_bound.get\_solutions():  for i in range(len(solution)):  print([1 if j == solution[i] else 0 for j in range(len(solution))])  print()  **Output:**  Solutions using Backtracking:  [0, 1, 0, 0]  [0, 0, 0, 1]  [1, 0, 0, 0]  [0, 0, 1, 0]  [0, 0, 1, 0]  [1, 0, 0, 0]  [0, 0, 0, 1]  [0, 1, 0, 0]  Solutions using Branch and Bound:  [0, 1, 0, 0]  [0, 0, 0, 1]  [1, 0, 0, 0]  [0, 0, 1, 0]  [0, 0, 1, 0]  [1, 0, 0, 0]  [0, 0, 0, 1]  [0, 1, 0, 0] |
| --- |

| **Assignment No: 5**  **Develop an elementary chatbot for any suitable customer interaction application.**  **CODE:**  def elementary\_chatbot():  print("Hello! I'm your elementary chatbot.")  while True:  user\_input = input("You: ").lower()  if "hello" in user\_input or "hi" in user\_input:  print("Chatbot: Hi there! How can I help you?")  elif "how are you" in user\_input:  print("Chatbot: I'm just a computer program, but thanks for asking!")  elif "bye" in user\_input or "exit" in user\_input:  print("Chatbot: Goodbye! Have a great day!")  break  elif "favorite color" in user\_input:  print("Chatbot: I don't have a favorite color. What's yours?")  user\_color = input("You: ")  print(f"Chatbot: {user\_color} is a nice color!")  else:  print("Chatbot: I'm not sure how to respond. Can you rephrase that or ask something else?")  if \_\_name\_\_ == "\_\_main\_\_":  elementary\_chatbot()  **Output:**  **Hello! I'm your elementary chatbot.**  **You:hi** |
| --- |

| Assignment No: 6  Implement Expert System for Hospitals and medical facilities  class SymptomCheckerExpertSystem:  def \_\_init\_\_(self):  self.rules = {  'Fever': ['High body temperature', 'Headache', 'Fatigue'],  'Cough': ['Persistent cough', 'Shortness of breath', 'Chest pain'],  'Sore\_Throat': ['Pain or irritation in the throat', 'Difficulty swallowing'],  'Headache': ['Persistent headache', 'Nausea', 'Sensitivity to light'],  # Add more symptoms and related conditions as needed  }  def get\_conditions(self, symptoms):  conditions = []  for condition, related\_symptoms in self.rules.items():  if all(symptom in symptoms for symptom in related\_symptoms):  conditions.append(condition)  return conditions  def main():  expert\_system = SymptomCheckerExpertSystem()  print("Welcome to the Symptom Checker Expert System!")  print("Enter your symptoms (comma-separated):")  user\_input = input("Symptoms: ").split(',')  conditions = expert\_system.get\_conditions(user\_input)  if conditions:  print("Possible conditions based on your symptoms:")  for condition in conditions:  print(f"- {condition}")  else:  print("No specific conditions identified based on your symptoms. Consult a healthcare professional.")  if \_\_name\_\_ == "\_\_main\_\_":  main()  **Output:**  **Welcome to the Symptom Checker Expert System!**  **Enter your symptoms (comma-separated):**  **Symptoms:Persistent cough, Shortness of breath,Chest pain** |
| --- |

| **Assignment No: 7**  **Title:**  **Implement Tic-Tac -Toe using Min Max Algorithm**  **CODE:**  import random  class TicTacToe:  def \_\_init\_\_(self):  self.board = [' '] \* 9  self.current\_player = 'X'  self.computer\_player = 'O'  def print\_board(self):  for i in range(0, 9, 3):  print(self.board[i], '|', self.board[i + 1], '|', self.board[i + 2])  if i < 6:  print("---------")  def is\_winner(self, player):  # Check rows, columns, and diagonals for a winner  for i in range(3):  if all(self.board[i \* 3 + j] == player for j in range(3)) or \  all(self.board[j \* 3 + i] == player for j in range(3)):  return True  if all(self.board[i] == player for i in [0, 4, 8]) or \  all(self.board[i] == player for i in [2, 4, 6]):  return True  return False  def is\_full(self):  return ' ' not in self.board  def is\_game\_over(self):  return self.is\_winner('X') or self.is\_winner('O') or self.is\_full()  def get\_available\_moves(self):  return [i for i in range(9) if self.board[i] == ' ']  def make\_move(self, position):  if self.board[position] == ' ':  self.board[position] = self.current\_player  self.switch\_player()  def switch\_player(self):  self.current\_player = 'O' if self.current\_player == 'X' else 'X'  def minimax(self, depth, is\_maximizing):  if self.is\_winner('X'):  return -1  if self.is\_winner('O'):  return 1  if self.is\_full():  return 0  if is\_maximizing:  best\_score = float('-inf')  for move in self.get\_available\_moves():  self.make\_move(move)  score = self.minimax(depth + 1, False)  self.board[move] = ' '  best\_score = max(score, best\_score)  return best\_score  else:  best\_score = float('inf')  for move in self.get\_available\_moves():  self.make\_move(move)  score = self.minimax(depth + 1, True)  self.board[move] = ' '  best\_score = min(score, best\_score)  return best\_score  def find\_best\_move(self):  best\_score = float('-inf')  best\_move = None  for move in self.get\_available\_moves():  self.make\_move(move)  score = self.minimax(0, False)  self.board[move] = ' '  if score > best\_score:  best\_score = score  best\_move = move  return best\_move  def main():  game = TicTacToe()  while not game.is\_game\_over():  game.print\_board()  if game.current\_player == 'X':  position = int(input("Enter your move (1-9): ")) - 1  if position not in game.get\_available\_moves():  print("Invalid move. Try again.")  continue  else:  print("Computer's move:")  position = game.find\_best\_move()  print(position + 1)  game.make\_move(position)  game.print\_board()  if game.is\_winner('X'):  print("You win!")  elif game.is\_winner('O'):  print("Computer wins!")  else:  print("It's a draw!")  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |