**Artificial Intelligence Lab 3**

**Exercise 3.1.**

1. Plot the directed and undirected graph for following points



1. Perform a depth-first search of the graph using any internal node
2. Perform a breadth first search of the graph using any internal node

|  |
| --- |
| import networkx as nx  # Define the edges  edges = [(1, 2), (1, 4), (1, 5), (2, 4), (4, 5), (3, 4), (3, 5), (3, 6), (6, 4)]  # Create the directed and undirected graphs  G\_directed = nx.DiGraph(edges)  G\_undirected = nx.Graph(edges)  # Perform depth-first search starting from node 1  dfs\_directed = list(nx.dfs\_preorder\_nodes(G\_directed, source=1))  dfs\_undirected = list(nx.dfs\_preorder\_nodes(G\_undirected, source=1))  # Perform breadth-first search starting from node 1  bfs\_directed = list(nx.bfs\_tree(G\_directed, source=1).nodes)  bfs\_undirected = list(nx.bfs\_tree(G\_undirected, source=1).nodes)  # Print the results  print("Directed graph:")  print(G\_directed.edges())  print("DFS traversal starting from node 1:", dfs\_directed)  print("BFS traversal starting from node 1:", bfs\_directed)  print("\nUndirected graph:")  print(G\_undirected.edges())  print("DFS traversal starting from node 1:", dfs\_undirected)  print("BFS traversal starting from node 1:", bfs\_undirected) |
| Output:  Directed graph:  [(1, 2), (1, 4), (1, 5), (2, 4), (4, 5), (3, 4), (3, 5), (3, 6), (6, 4)]  DFS traversal starting from node 1: [1, 2, 4, 5]  BFS traversal starting from node 1: [1, 2, 4, 5]  Undirected graph:  [(1, 2), (1, 4), (1, 5), (2, 4), (4, 5), (4, 3), (4, 6), (5, 3), (3, 6)]  DFS traversal starting from node 1: [1, 2, 4, 5, 3, 6]  BFS traversal starting from node 1: [1, 2, 4, 5, 3, 6] |

**If plotting is also required**

|  |
| --- |
| import networkx as nx  import matplotlib.pyplot as plt  # Define the edges  s = [1, 1, 1, 2, 3, 3, 3, 4, 6]  t = [2, 4, 5, 5, 6, 7, 4, 1, 4]  edges = [(s[i], t[i]) for i in range(len(s))]  # Create the directed graph  G = nx.DiGraph()  G.add\_edges\_from(edges)  # Plot the directed graph  nx.draw(G, with\_labels=True)  plt.show()  # Create the undirected graph  G = nx.Graph()  G.add\_edges\_from(edges)  # Plot the undirected graph  nx.draw(G, with\_labels=True)  plt.show() |

**Exercise 3.2.**

1. Assign weights to the edges of the graph
2. Find the shortest path of the undirected graph
3. Find the shortest path of the directed graph
4. Compute the shortest path of the graph using weights

|  |
| --- |
| import networkx as nx  import matplotlib.pyplot as plt  import heapq  # Define the graph with weights  edges = [(1, 2, 1), (1, 4, 1), (1, 5, 1), (2, 5, 1), (3, 6, 1), (3, 7, 1), (3, 4, 1), (4, 1, 1), (6, 4, 1)]  graph = {}  for u, v, w in edges:      if u not in graph:          graph[u] = []      graph[u].append((v, w))  # Perform Dijkstra's algorithm  start\_node = 1  distances = {start\_node: 0}  queue = [(0, start\_node)]  while queue:      current\_distance, current\_node = heapq.heappop(queue)      if current\_distance > distances[current\_node]:          continue      for neighbor, weight in graph.get(current\_node, []):          distance = current\_distance + weight          if distance < distances.get(neighbor, float('inf')):              distances[neighbor] = distance              heapq.heappush(queue, (distance, neighbor))  # Print the shortest path to each node  print("\n")  for node, distance in sorted(distances.items()):      print(f"Shortest path to node {node}: {distance}") |

**Exercise 3.3.**

In this task you will consider 8-puzzle problem. The 8-puzzle problem is a puzzle played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in specified order. You are permitted to slide blocks horizontally or vertically into the blank square.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | 3 | 4 | 1 | | 2 | 5 | 6 | | 7 | 8 |  |   Initial State (Randomly Chosen) | |  |  |  | | --- | --- | --- | | 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 |  |   Goal State |

Create a State Space for 8-puzzle problem. Define it in such a way that it may act as a tree.

|  |
| --- |
| #Task 3.3  # define initial and goal states  initial\_state = [[3, 4, 1], [2, 5, 6], [7, 8, 'X']]  goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 'X']]  # function to find possible moves from a given state  def find\_moves(state):      moves = []      x, y = 0, 0      # find the position of the blank square      for i in range(3):          for j in range(3):              if state[i][j] == 'X':                  x, y = i, j                  break      # check if a move to the left is possible      if y > 0:          new\_state = [row[:] for row in state]          new\_state[x][y], new\_state[x][y-1] = new\_state[x][y-1], new\_state[x][y]          moves.append(new\_state)      # check if a move to the right is possible      if y < 2:          new\_state = [row[:] for row in state]          new\_state[x][y], new\_state[x][y+1] = new\_state[x][y+1], new\_state[x][y]          moves.append(new\_state)      # check if a move up is possible      if x > 0:          new\_state = [row[:] for row in state]          new\_state[x][y], new\_state[x-1][y] = new\_state[x-1][y], new\_state[x][y]          moves.append(new\_state)      # check if a move down is possible      if x < 2:          new\_state = [row[:] for row in state]          new\_state[x][y], new\_state[x+1][y] = new\_state[x+1][y], new\_state[x][y]          moves.append(new\_state)      return moves  # create state space tree  state\_space = {str(initial\_state): []}  queue = [initial\_state]  while queue:      state = queue.pop(0)      moves = find\_moves(state)      for move in moves:          move\_str = str(move)          if move\_str not in state\_space:              state\_space[move\_str] = []              queue.append(move)          state\_space[str(state)].append(move\_str)  # find the path to the goal state using breadth-first search  path = []  queue = [(str(initial\_state), path)]  visited = set()  while queue:      state, path = queue.pop(0)      if state == str(goal\_state):          break      if state in visited:          continue      visited.add(state)      for child in state\_space[state]:          queue.append((child, path + [state]))    # print path to goal state  print('Initial state:')  for row in initial\_state:      print(row)  print('\nGoal state:')  for row in goal\_state:      print(row)  print('\nPath to goal state:')  for state in path:      for row in eval(state):          print(row)      print()  for row in goal\_state:      print(row) |

**Exercise 3.4.**

Design a suitable representation and draw the complete search tree for the following problem: A farmer is on one side of a river and wishes to cross the river with a wolf, a chicken, and a bag of grain. He can take only one item at a time in his boat with him. He can’t leave the chicken alone with the grain, or it will eat the grain, and he can’t leave the wolf alone with the chicken, or the wolf will eat the chicken. How does he get all three safely across to the other side?

|  |
| --- |
| #Task 3.4  # Define the initial state  initial\_state = {'farmer': 'left', 'wolf': 'left', 'chicken': 'left', 'grain': 'left', 'boat': 'left'}  # Define the goal state  goal\_state = {'farmer': 'right', 'wolf': 'right', 'chicken': 'right', 'grain': 'right', 'boat': 'right'}  # Define the possible actions  actions = [('farmer',), ('farmer', 'wolf'), ('farmer', 'chicken'), ('farmer', 'grain')]  # Define the state transition function  def transition(state, action):      new\_state = state.copy()      if new\_state['boat'] == 'left':          new\_state['boat'] = 'right'          for item in action:              new\_state[item] = 'right'      else:          new\_state['boat'] = 'left'          for item in action:              new\_state[item] = 'left'      return new\_state  # Define the goal test function  def goal\_test(state):      return state == goal\_state  # Define the valid state function  def is\_valid(state):      if state['wolf'] == state['chicken'] and state['farmer'] != state['wolf']:          return False      elif state['chicken'] == state['grain'] and state['farmer'] != state['chicken']:          return False      else:          return True  # Define the depth-first search function  def dfs(start, goal, actions, transition, goal\_test, is\_valid):      visited = []      stack = [(start, [])]      while stack:          (state, path) = stack.pop()          if state not in visited:              visited.append(state)              if goal\_test(state):                  return path + [state]              for action in actions:                  new\_state = transition(state, action)                  if is\_valid(new\_state) and new\_state not in visited:                      stack.append((new\_state, path + [state]))      return None  # Solve the problem using DFS  solution = dfs(initial\_state, goal\_state, actions, transition, goal\_test, is\_valid)  # Print the solution  if solution is None:      print("No solution found.")  else:      print("Solution:")      for i, state in enumerate(solution):          print(f"{i}. {state['farmer']} {state['wolf']} {state['chicken']} {state['grain']} {state['boat']}") |