|  |  |
| --- | --- |
| Semester | T.E. Semester V |
| Subject | Artificial Intelligence Lab |
| Subject Professor In-charge | Ms. Rasika Ransing |
| Laboratory | L 011 |

|  |  |
| --- | --- |
| Student Name | Harsh Rawte |
| Roll Number | 22101A0047 |

|  |  |
| --- | --- |
| Experiment  Number | 04 |
| Experiment Title | Program on uninformed search methods (DFS) |
| Objectives  (Skill Set /  Knowledge  Tested /  Imparted) | To implement Depth First search  (Find best route from a source to destination)   **Understanding Depth-First Search (DFS)**:   **Pathfinding Algorithms**:   * Knowledge imparted includes which algorithms are optimal for different scenarios, such as unweighted graphs (BFS) and weighted graphs (Dijkstra).    **Problem-Solving and Analytical Thinking**:   * The problem tests students’ ability to apply DFS in various search scenarios and compare it with other pathfinding algorithms. * Analytical thinking about when DFS is more useful vs. other algorithms like Dijkstra for shortest path scenarios is imparted.    **Real-World Applications of Graph Traversal**:   * The project shows how DFS can be applied in real-world scenarios such as searching paths, game solving, or network exploration.    **Coding and Algorithm Implementation**:   * Practical coding skills for DFS, stack-based exploration, and recursive approaches are imparted and tested. |
| Theory  Code | **Depth First search (DFS)**  Depth-First Search (DFS) is a graph traversal algorithm that explores as far along a branch as possible before backtracking. Unlike BFS, which explores level by level, DFS goes deep into one path, visiting nodes in a depth-first manner. It uses a stack (either explicitly or recursively) to track the nodes being explored.  In the context of finding the shortest path between Mumbai and Goa, DFS may not be the best approach for identifying the shortest path because it explores one path to its maximum depth before considering alternatives. While DFS is effective for exploring all possible routes, it does not inherently consider the shortest path unless all paths are evaluated.  If we model the route between Mumbai and Goa as a graph, where cities are nodes and routes between them are edges, DFS can help in exploring all possible routes, but for efficiency in finding the shortest path  import java.util.\*;  public class RouteDFS {  static class Graph {  private Map<String, List<String>> adjList = new HashMap<>();  void addEdge(String from, String to) {  adjList.computeIfAbsent(from, k -> new ArrayList<>()).add(to);  adjList.computeIfAbsent(to, k -> new ArrayList<>()).add(from); // If undirected graph  }  List<String> getAdjacentNodes(String node) {  return adjList.getOrDefault(node, new ArrayList<>());  }  }  static class Route {  String start;  String end;  int distance;  double time;  Graph graph;  Route(String start, String end, int distance, double time) {  this.start = start;  this.end = end;  this.distance = distance;  this.time = time;  this.graph = new Graph();  }  }  public static void dfs(Graph graph, String start, String end, Set<String> visited, List<String> path) {  visited.add(start);  path.add(start);  if (start.equals(end)) {  System.out.println("Route found: " + String.join(" ~> ", path));  return;  }  for (String neighbor : graph.getAdjacentNodes(start)) {  if (!visited.contains(neighbor)) {  dfs(graph, neighbor, end, visited, path);  // Remove node from path after returning to allow other paths  path.remove(path.size() - 1);  }  }  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  // Get source and destination from the user  System.out.print("Enter source city: ");  String source = scanner.nextLine();  System.out.print("Enter destination city: ");  String destination = scanner.nextLine();  // Define the routes  Route route1 = new Route("Mumbai", "Goa", 588, 12);  route1.graph.addEdge("Mumbai", "Panvel");  route1.graph.addEdge("Panvel", "Pali");  route1.graph.addEdge("Pali", "Mangaon");  route1.graph.addEdge("Mangaon", "Mahad");  route1.graph.addEdge("Mahad", "Khed");  route1.graph.addEdge("Khed", "Chiplun");  route1.graph.addEdge("Chiplun", "Sangameshwar");  route1.graph.addEdge("Sangameshwar", "Rajapur");  route1.graph.addEdge("Rajapur", "Kharepatan");  route1.graph.addEdge("Kharepatan", "Kudal");  route1.graph.addEdge("Kudal", "Sawantwadi");  route1.graph.addEdge("Sawantwadi", "Goa");  Route route2 = new Route("Mumbai", "Goa", 600, 12.5);  route2.graph.addEdge("Mumbai", "Khopoli");  route2.graph.addEdge("Khopoli", "Lonavala");  route2.graph.addEdge("Lonavala", "Pune");  route2.graph.addEdge("Pune", "Satara");  route2.graph.addEdge("Satara", "Kolhapur");  route2.graph.addEdge("Kolhapur", "Shiroli");  route2.graph.addEdge("Shiroli", "Goa");  Route route3 = new Route("Mumbai", "Goa", 709, 15);  route3.graph.addEdge("Mumbai", "Lonavala");  route3.graph.addEdge("Lonavala", "Pune");  route3.graph.addEdge("Pune", "Phaltan");  route3.graph.addEdge("Phaltan", "Miraj");  route3.graph.addEdge("Miraj", "Chikodi");  route3.graph.addEdge("Chikodi", "Belagavi");  route3.graph.addEdge("Belagavi", "Goa");  // List of all routes  List<Route> routes = Arrays.asList(route1, route2, route3);  // Display all routes using DFS  System.out.println("\nAll available routes:");  for (Route route : routes) {  System.out.println("\nRoute via " + route.start + " to " + route.end + " (Total Distance: " + route.distance + " km, Total Time: " + route.time + " hours):");  dfs(route.graph, route.start, route.end, new HashSet<>(), new ArrayList<>());  }  // Find the route with the shortest distance  Route shortestRoute = Collections.min(routes, Comparator.comparingInt(route -> route.distance));  // Display shortest route  System.out.println("\nShortest route from " + source + " to " + destination + ":");  System.out.println("Route via " + shortestRoute.start + " to " + shortestRoute.end);  System.out.println("Total Distance: " + shortestRoute.distance + " km");  System.out.println("Total Time: " + shortestRoute.time + " hours");  // Perform DFS on the shortest route  System.out.println("\nDFS for the shortest route:");  dfs(shortestRoute.graph, shortestRoute.start, shortestRoute.end, new HashSet<>(), new ArrayList<>());  }  } |
| Output |  |
| Conclusion | While DFS is a powerful tool for exploring graphs, it is not well-suited for finding the shortest path in a weighted graph (like road distances between cities) without additional logic. For scenarios like finding the shortest path between Mumbai and Goa, using BFS or Dijkstra’s algorithm would be more appropriate, as they consider all potential paths efficiently. DFS, however, remains valuable for exhaustive search tasks and problems that require complete traversal. |