|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Pratik Pujari** | | |
| **UID no.** | **2020300054** | **Class:** | **Comps C Batch** |
| **Experiment No.** | 6 | | |

|  |  |
| --- | --- |
| **AIM:** | To implement Dijkstra algorithm |
| **THEORY:** | **What is BackTracking?**  Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed.  It uses recursive calling to find a solution set by building a solution step by step, increasing levels with time. In order to find these solutions, a search tree named state-space tree is used. In a state-space tree, each branch is a variable, and each level represents a solution.  A backtracking algorithm uses the depth-first search method. When it starts exploring the solutions, a bounding function is applied so that the algorithm can check if the so-far built solution satisfies the constraints. If it does, it continues searching. If it doesn’t, the branch would be eliminated, and the algorithm goes back to the level before.  **When to Use a Backtracking Algorithm**  The backtracking algorithm is applied to some specific types of problems. For instance, we can use it to find a feasible solution to a decision problem. It was also found to be very effective for optimization problems.  For some cases, a backtracking algorithm is used for the enumeration problem in order to find the set of all feasible solutions for the problem.  On the other hand, backtracking is not considered an optimized technique to solve a problem. It finds its application when the solution needed for a problem is not time-bounded.  **Dijkstra's Algorithm**  Dijkstra’s algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.  Graph  The graph has the following:   * vertices, or nodes, denoted in the algorithm by v*v* or u*u*; * weighted edges that connect two nodes: (u,v*u*,*v*) denotes an edge, and w(u,v)*w*(*u*,*v*) denotes its weight. In the diagram on the right, the weight for each edge is written in gray.   This is done by initializing three values:   * dist*dist*, an array of distances from the source node s*s* to each node in the graph, initialized the following way: dist*dist*(s*s*) = 0; and for all other nodes v*v*, dist*dist*(v*v*) = \infty∞. This is done at the beginning because as the algorithm proceeds, the dist*dist* from the source to each node v*v* in the graph will be recalculated and finalized when the shortest distance to v*v* is found * Q*Q*, a queue of all nodes in the graph. At the end of the algorithm's progress, Q*Q* will be empty. * S*S*, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S*S* will contain all the nodes of the graph.   It is easier to start with an example and then think about the algorithm.  Start with a weighted graphStart with a weighted graphChoose a starting vertex and assign infinity path values to all other devicesChoose a starting vertex and assign infinity path values to all other devicesGo to each vertex and update its path length  Go to each vertex and update its path lengthIf the path length of the adjacent vertex is lesser than new path length, don't update it  If the path length of the adjacent vertex is lesser than new path length, don't update itAvoid updating path lengths of already visited vertices  Avoid updating path lengths of already visited verticesAfter each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7  After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7Notice how the rightmost vertex has its path length updated twice  Notice how the rightmost vertex has its path length updated twice  Repeat until all the vertices have been visited  Repeat until all the vertices have been visited |
| **PSEUDOCODE:** | 1: function Dijkstra(Graph, source):  2: for each vertex v in Graph: // Initialization  3: dist[v] := infinity // initial distance from source to vertex v is set to infinite  4: previous[v] := undefined // Previous node in optimal path from source  5: dist[source] := 0 // Distance from source to source  6: Q := the set of all nodes in Graph // all nodes in the graph are unoptimized - thus are in Q  7: while Q is not empty: // main loop  8: u := node in Q with smallest dist[ ]  9: remove u from Q  10: for each neighbor v of u: // where v has not yet been removed from Q.  11: alt := dist[u] + dist\_between(u, v)  12: if alt < dist[v] // Relax (u,v)  13: dist[v] := alt  14: previous[v] := u  15: return previous[ ] |
| **EXPERIMENT 1** | |
| **CODE:** | import java.util.\*;  public class dijkstra {      int source;  *// The main method is where the program starts.*      void dijkstra\_solve(int[][] graph) {          int count = graph.length;          boolean[] visited = new boolean[count];          int[] distance = new int[count];          for (int i = 0; i < distance.length; i++) {              distance[i] = Integer.MAX\_VALUE;          }          distance[source] = 0;          for (int k = 0; k < distance.length - 1; k++) {              int minVertex = findMin(distance, visited);              visited[minVertex] = true;  *// explore the neighbours*              for (int i = 0; i < distance.length; i++) {                  if (graph[minVertex][i] != 0 && distance[minVertex] != Integer.MAX\_VALUE) {  *// checking if the there exists an edge between the two vertices, the neighbour*  *// should not be visited*  *// adding the weight of the edge to the distance of the min vertex*                      int newDistance = distance[minVertex] + graph[minVertex][i]; *// Relaxation*                      if (newDistance < distance[i]) {  *// updating the distance of the vertex if the value is lesser than the*  *// previous value of the same vertex*                          distance[i] = newDistance;                      }                  }              }          }          System.out.println("\nOutput\n(Vertex-> Distance):");          for (int i = 0; i < distance.length; i++) {              if (distance[i] == Integer.MAX\_VALUE || distance[i] == 0) {                  continue;              }              System.out.println(i + "\t" + distance[i]);          }      }  *// finding the minimum distance vertex*      static int findMin(int[] distance, boolean[] visited) {          int minVertex = -1; *// initializing the minimum vertex to -1*          for (int i = 1; i < distance.length; i++) {  *// if the vertex is not visited and the distance is lesser than the min vertex*              if ((minVertex == -1 || distance[i] < distance[minVertex]) && !visited[i]) {                  minVertex = i;              }          }          return minVertex; *// returning the minimum vertex*      }      public static void main(String[] args) throws Exception {          try (*// Driver code*                  Scanner sc = new Scanner(System.in)) {              System.out.println("----------------Dijkstra's Algorithm----------------");              System.out.println("\nInput(TestCases-> Vertices-> Edges-> Each edge with weights)\n");              int testCases = sc.nextInt();              dijkstra T = new dijkstra();              int negativeChecker = 0;              for (int i = 0; i < testCases; i++) {                  int v = sc.nextInt(); *// vertices*                  int e = sc.nextInt(); *// edges*                  int[][] graph = new int[1024][1024]; *// adjacency matrix*                  int src = sc.nextInt();                  T.source = src;                  int dest = sc.nextInt();                  int cost = sc.nextInt(); *// cost of the edge // set to store the vertices*  *// if cost is negative, then the edge is not added*                  graph[src][dest] = cost;                  if (cost < 0) {                      negativeChecker = 1;                      System.out.print("\nNegative edge not Added");                      continue;                  }                  for (int j = 0; j < e - 1; j++) {                      int p = sc.nextInt(); *// source*                      int q = sc.nextInt(); *// destination*                      cost = sc.nextInt(); *// cost of the edge*                      if (cost < 0) {                          negativeChecker = 1;                          System.out.print("\nNegative edge not allowed");                          break;                      }                      graph[p][q] = cost;                  }                  if (negativeChecker != 1)                      T.dijkstra\_solve(graph);              }              sc.close();          }      }  } |
| **OUTPUT:** | Written Sum for the Dijkstra |
| **TIME COMPLEXITY:** | Both the time and space complexity obtained for the Dijkstra’s algorithm is O(V\*V ) where V is the number of vertices. While if an adjacency list is used the time complexity will come out to be O(E\*logV) where E is the number of edges. Since the Dijkstra’s algorithm works on Greedy approach, it has a limitation that it doesn’t work for the negative values of the distances because negative value and since Dijkstra works on the principle that once a vertex is discovered, it can’t go back. Thus, it can’t be used for the negative values |
| **CONCLUSION:** Learnt during the procedural programming of solving the problem   * Learnt about BackTracking and Dijsktra Algorithm * Learnt how to use adjacency matrix in order to store the graph * Learnt about time and space complexity of the shortest path algorithm * Learnt about the advantages and disadvantages of dijsktra algorithm * Learnt about the applications of a Dijsktra/ shortest path finder algorithm | |