**SIMULATION OF ROUTING PROTOCOLS**

**MINOR PROJECT REPORT**

Submitted by

**Aashna Garg (2K11/SE/001)**

**Charmy Panigrahi (2K11/SE/022)**

**Jasmin Joy (2K11/SE/035)**

BACHELOR OF ENGINEERING

IN

SOFTWARE ENGINEERING

Under the Guidance

of

**Mr. R.K, Yadav**

**(Asst. Professor, CSE)**



November 2013

**DELHI TECHNOLOGICAL UNIVERSITY**

**DELHI TECHNOLOGICAL**

**UNIVERSITY**

****

**CERTIFICATE**

This is to Certified that this project report “**SIMULATION OF ROUTING PROTOCOLS**” is submitted by **Aashna Garg (2K11/SE/001)** ,**Charmy Panigrahi (2K11/SE/022) and Jasmin Joy (2K11/SE/035)** andwho carried out the project work under my supervision.

I approve this project for submission of the Bachelor of Engineering in the Department of Computer Science & Engineering, Delhi Technological University.

**Dr. Daya Gupta Prof. R. K. Yadav Prof. Ruchika malhotra**

**(HOD, CSE) (Project Guide) (Project Coordinator)**

**ACKNOWLEDGEMENT**

It gives us immense pleasure to express our deepest sense of gratitude and sincere thanks to our highly respected and esteemed guide **Prof. Rajesh Kumar Yadav, Asst. Professor** , for his valuable guidance, encouragement and help for completing this work. His useful suggestions for this whole work and co-operative behavior are sincerely acknowledged.

We would like to express our sincere thank to, our Project Coordinator **Dr. Ruchika Malhotra, Asst. Professor** for giving us this opportunity to undertake this project. We would also like to thank **Dr. Daya Gupta, Head of Computer Engineering Department** for whole hearted support.

We are also grateful to our teacher **Ms. Richa Mishra, Asst. Professor** for her constant support and guidance.

At the end we would like to express our sincere thanks to all our friends and others who helped us directly or indirectly during this project work.

**Aashna Garg (2K11/SE/001)**

**Charmy Panigrahi (2K11/SE/022)**

**Jasmin Joy (2K11/SE/035)**

1. **Introduction**

On the network layer, achieving routing convergence, the process in which routing tables are updated, is a crucial and complex process. At every topology change, including a link failure or recovery, the routing tables need to be updated at which time the convergence process takes place.

The task of updating these tables is accomplished by routers that communicate according to a set of rules set by routing protocols. The main goals of any routing protocol are to achieve fast convergence, while remaining simple, flexible, accurate and robust.

In this project, we analyze and compare the convergence times of three protocols: Routing Information Protocol (RIP), Open Shortest Path First (OSPF), and Enhanced Interior Gateway Routing Protocol (EIGRP).

We will consider network of different sizes, each of which will be simulated on C++ and Matlab. We will simulate each Network with Dijkstra, Prims and Bellman-Ford collect statistics such as distance and shortest path.

We will also analyze the routing tables of network in order to study the metrics of each protocol and gain a better understanding of how routes are chosen. By examining the results, we will identify the routing protocol with the best performance for a large, realistic network.

Finally, we will discuss the limitations that exist within our project and network implementations of the routing protocols. Furthermore, we will provide possible modifications that could be explored for future work.

**1.2 Literature survey**

Routing links together small networks to form huge internetworks that span vast regions. This cumbersome task makes the network layer the most complex in the TCP/IP reference model. The network layer provides the transfer of packets across the network.

Routing protocols define the path of each packet from source to destination. To complete this task, routers use routing tables, which contain information about possible destinations in the network and the metrics (distance, cost, bandwidth, etc.) to these destinations.

Routers have information regarding the neighbor routers around them. The degree of a router’s network knowledge and awareness depends on the routing protocol it uses. At every change in the network, including link failure and link recovery, routing tables must be updated. The efficiency of these updates determines the efficiency of the routing protocols.

There are two main types of routing protocols: static routing and dynamic routing.

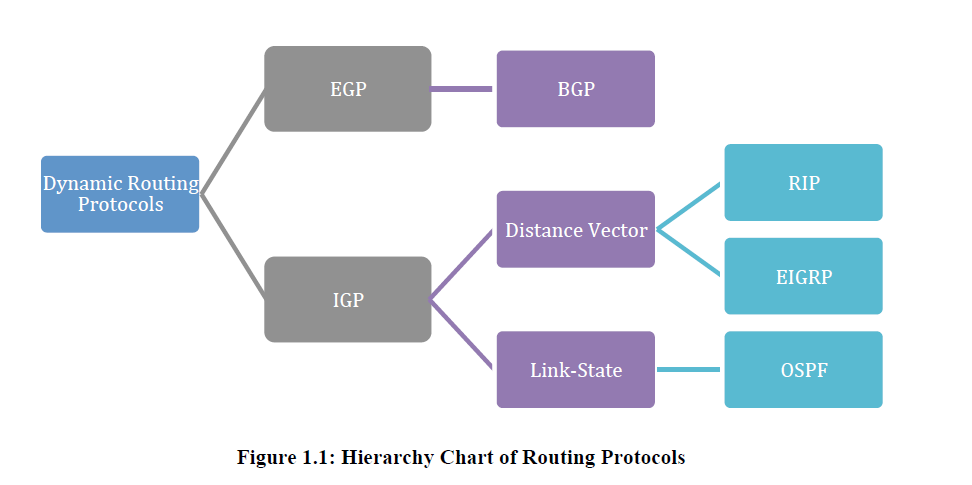
Static routing assumes that the network is fixed, meaning no nodes are added or removed and routing tables are therefore only manually updated.

Dynamic or adaptive routing, more commonly used for internetworking, allows changes in the network topology by using routing tables that update with each network change.

In this report we will only consider dynamic routing protocols. Within the class of dynamic protocols, we can have Interior or Exterior Gateway Protocols. EGP’s deals with routing information between different autonomous.

An example of an EGP is Border Gateway Protocol (BGP). The three routing protocols we chose to compare are IGP’s, protocols that exchange routing information within an AS. These protocols can either use distance vector (such as RIP and EIGRP) or link-state algorithms (such as OSPF) to optimize convergence times.

In this project we will compare the three dynamic routing protocols shown on the right of the hierarchy chart below: RIP, OSPF and EIGRP.



**1.2.1 Routing Information Protocol (RIP):**

The Routing Information Protocol (RIP), which is a distance-vector based algorithm, is one of the first routing protocols implemented on TCP/IP. Information is sent through the network using UDP. Each router that uses this protocol has limited knowledge of the network around it.

This simple protocol uses a hop count mechanism to find an optimal path for packet routing. A maximum number of 16 hops are employed to avoid routing loops. However, this parameter limits the size of the networks that this protocol can support.

The popularity of this protocol is largely due to its simplicity and its easy configurability. However, its disadvantages include slow convergence times, and its scalability limitations. Therefore, this protocol works best for small networks.

We can implement distance vector routing algorithm with Prims algorithm which help in finding shortest path.

**1.2.1.1Prims Algorithm**

Prim’s starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.  
A group of edges that connects two set of vertices in a graph is called [cut in graph theory](http://en.wikipedia.org/wiki/Cut_%28graph_theory%29). So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

**How does Prim’s Algorithm Work?** The idea behind Prim’s algorithm is simple. A spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

**Algorithm**  
**1)** Create a set mstSet that keeps track of vertices already included in MST.  
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.  
**3)** While mstSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in mstSet and has minimum key value.  
….**b)** Include u to mstSet.  
….**c)** Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

**1.2.2 Open Shortest Path First (OSPF)**

Open Shortest Path First (OSPF) is a very widely used link-state routing protocols. This protocol routes Internet Protocol (IP) packets by gathering link-state information from neighboring routers and constructing a map of the network.

OSPF routers send many message types including hello messages, link state requests and updates and database descriptions. Djisktra’s algorithm is then used to find the shortest path to the destination.

Shortest Path First (SPF) calculations are computed either periodically or upon a received Link State Advertisement (LSA), depending on the protocol implementation. Topology changes are detected very quickly using this protocol.

Another advantage of OSPF is that its many configurable parameters make it a very flexible and robust protocol. Contrary to RIP, however, OSPF has the disadvantage of being too complicated.

**1.2.2.1 Dijkstra Algorithm**

Dijkstra algorithm we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, and other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

**Algorithm**

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e.,

whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign

distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

….a) Pick a vertex u which is not there in sptSetand has minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all

adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge uv,

is less than the distance value of v, then update the distance value of v.

**2. RESEARCH BACKGROUND**

**2.1 Distance-vector routing protocol**

In [computer communication](http://en.wikipedia.org/wiki/Computer_communication) theory relating to [packet-switched networks](http://en.wikipedia.org/wiki/Packet-switched_network), a distance-vector routing protocol is one of the two major classes of [routing protocols](http://en.wikipedia.org/wiki/Routing_protocol), the other major class being the [link-state protocol](http://en.wikipedia.org/wiki/Link-state_protocol). Distance-vector routing protocols use the [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm), [Ford–Fulkerson algorithm](http://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm), or [DUAL FSM](http://en.wikipedia.org/wiki/DUAL_FSM) (in the case of [Cisco Systems](http://en.wikipedia.org/wiki/Cisco_Systems)'s protocols) to calculate paths.

A distance-vector routing protocol requires that a router informs its neighbors of topology changes periodically. Compared to [link-state protocols](http://en.wikipedia.org/wiki/Link-state_routing_protocol), which require a router to inform all the nodes in a network of topology changes, distance-vector routing protocols have less [computational complexity](http://en.wikipedia.org/wiki/Computational_complexity) and [message overhead](http://en.wikipedia.org/w/index.php?title=Message_overhead&action=edit&redlink=1).

The term distance vector refers to the fact that the protocol manipulates vectors ([arrays](http://en.wikipedia.org/wiki/Array_data_structure)) of distances to other nodes in the network. The vector distance algorithm was original ARPANET routing algorithm and was also used in the internet under the name of RIP(routing internet protocol).

Examples of distance-vector routing protocols include [RIPv1 and RIPv2](http://en.wikipedia.org/wiki/Routing_Information_Protocol) and [IGRP](http://en.wikipedia.org/wiki/Interior_Gateway_Routing_Protocol).

## 2.1.1 Method

Routers using distance-vector protocol do not have knowledge of the entire path to a destination. Instead they use two methods:

1. Direction in which router or exit interface a packet should be forwarded.
2. Distance from its destination

Distance-vector protocols are based on calculating the direction and distance to any link in a network. "Direction" usually means the next hop address and the exit interface. "Distance" is a measure of the cost to reach a certain node. The least cost route between any two nodes is the route with minimum distance. Each node maintains a vector (table) of minimum distance to every node. The cost of reaching a destination is calculated using various route metrics. [RIP](http://en.wikipedia.org/wiki/Routing_Information_Protocol) uses the hop count of the destination whereas [IGRP](http://en.wikipedia.org/wiki/IGRP) takes into account other information such as node delay and available bandwidth.

Updates are performed periodically in a distance-vector protocol where all or part of a router's routing table is sent to all its neighbors that are configured to use the same distance-vector routing protocol. RIP supports cross-platform distance vector routing whereas IGRP is a [Cisco Systems](http://en.wikipedia.org/wiki/Cisco_Systems) proprietary distance vector routing protocol. Once a router has this information it is able to amend its own routing table to reflect the changes and then inform its neighbors of the changes. This process has been described as ‘routing by rumor’ because routers are relying on the information they receive from other routers and cannot determine if the information is actually valid and true. There are a number of features which can be used to help with instability and inaccurate routing information.

## 2.1.2 Count-to-infinity problem

The [Bellman–Ford algorithm](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm) does not prevent [routing loops](http://en.wikipedia.org/wiki/Routing_loop) from happening and suffers from the count-to-infinity problem. The core of the count-to-infinity problem is that if A tells B that it has a path somewhere, there is no way for B to know if the path has B as a part of it. To see the problem clearly, imagine a subnet connected like as A–B–C–D–E–F, and let the metric between the routers be "number of jumps". Now suppose that A is taken offline. In the vector-update-process B notices that the route to A, which was distance 1, is down – B does not receive the vector update from A. The problem is, B also gets an update from C, and C is still not aware of the fact that A is down – so it tells B that A is only two jumps from C (C to B to A), which is false. This slowly propagates through the network until it reaches infinity (in which case the algorithm corrects itself, due to the relaxation property of Bellman–Ford).

The Bellman–Ford algorithm is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) that computes [shortest paths](http://en.wikipedia.org/wiki/Shortest_path) from a single source [vertex](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) to all of the other vertices in a [weighted digraph](http://en.wikipedia.org/wiki/Weighted_digraph)

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm.[[2]](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm#cite_note-FOOTNOTESedgewick2002-2) If a graph contains a "negative cycle", i.e., a [cycle](http://en.wikipedia.org/wiki/Cycle_(graph_theory)) whose edges sum to a negative value, then there is no cheapest path, because any path can be made cheaper by one more [walk](http://en.wikipedia.org/wiki/Walk_(graph_theory)) through the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence, but it cannot produce a correct "shortest path" answer if a negative cycle is reachable from the source.

## 2.1.3 Algorithm

Bellman–Ford is based on the principle of [relaxation](http://en.wikipedia.org/wiki/Relaxation_(iterative_method)), in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. the Bellman–Ford algorithm simply relaxes allthe edges, and does this |V | − 1 times, where |V | is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. Bellman–Ford runs in [O](http://en.wikipedia.org/wiki/Big_O_notation)(|V|·|E|) time, where |V| and |E| are the number of vertices and edges respectively.

procedure BellmanFord(list vertices, list edges, vertex source)

// This implementation takes in a graph, represented as lists of vertices and edges,

// and fills two arrays (distance and predecessor) with shortest-path information

// Step 1: initialize graph

for each vertex v in vertices:

if v is source then distance[v] := 0

else distance[v] := infinity

predecessor[v] := null

// Step 2: relax edges repeatedly

for i from 1 to size(vertices)-1:

for each edge (u, v) with weight w in edges:

if distance[u] + w < distance[v]:

distance[v] := distance[u] + w

predecessor[v] := u

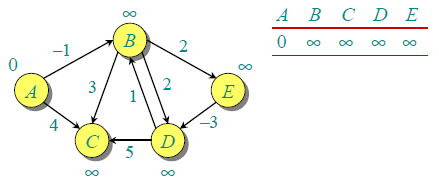
// Step 3: check for negative-weight cycles

for each edge (u, v) with weight w in edges:

if distance[u] + w < distance[v]:

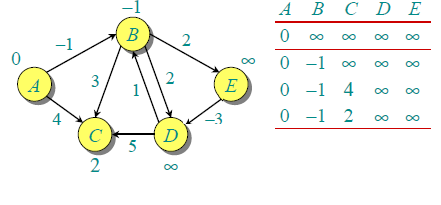
error "Graph contains a negative-weight cycle"

**Example**  
Let us understand the algorithm with following example graph. Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/bellman2.png)

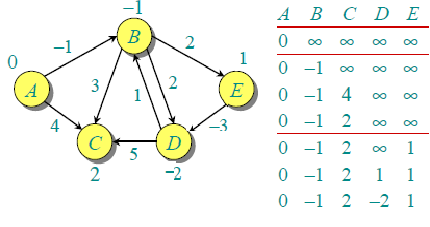
**Fig 2.1**

Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/After1stIteration.png)

**Fig 2.2**

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/seconditeration2.png)

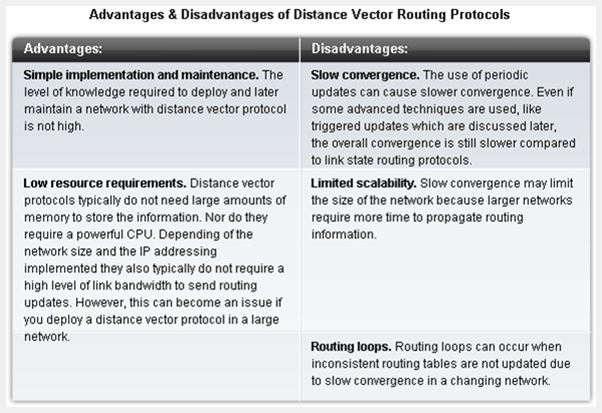
**Fig 2.3**

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

Here’s the algorithm in a condensed form:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void bellman\_ford(int s) {          int i, j;            for (i = 0; i < n; ++i)                  d[i] = INFINITY;            d[s] = 0;            for (i = 0; i < n - 1; ++i)                  for (j = 0; j < e; ++j)                          if (d[edges[j].u] + edges[j].w < d[edges[j].v])                                  d[edges[j].v] = d[edges[j].u] + edges[j].w;  } |

**2.1.4 Advantages And Disadvantages**



**Fig 2.4**

# 2.2 Link-State Routing Protocol

A link-state routing protocol is one of the two main classes of [routing protocols](http://en.wikipedia.org/wiki/Routing_protocol) used in [packet switching](http://en.wikipedia.org/wiki/Packet_switching) networks for [computer communications](http://en.wikipedia.org/wiki/Computer_communication) (the other is the [distance-vector routing protocol](http://en.wikipedia.org/wiki/Distance-vector_routing_protocol)). Examples of link-state routing protocols include [open shortest path first](http://en.wikipedia.org/wiki/Open_shortest_path_first) (OSPF) and [intermediate system to intermediate system](http://en.wikipedia.org/wiki/IS-IS) (IS-IS).

The link-state protocol is performed by every switching node in the network (i.e., nodes that are prepared to forward packets; in the [Internet](http://en.wikipedia.org/wiki/Internet), these are called [routers](http://en.wikipedia.org/wiki/Router_(computing))). The basic concept of link-state routing is that every node constructs a map of the connectivity to the network, in the form of a [graph](http://en.wikipedia.org/wiki/Graph_theory), showing which nodes are connected to which other nodes. Each node then independently calculates the next best logical path from it to every possible destination in the network. The collection of best paths will then form the node's [routing table](http://en.wikipedia.org/wiki/Routing_table).

This contrasts with [distance-vector routing protocols](http://en.wikipedia.org/wiki/Distance-vector_routing_protocol), which work by having each node share its routing table with its neighbors. In a link-state protocol the only information passed between nodes is connectivity related.

Link-state algorithms are sometimes characterized informally as each router 'telling the world about its neighbors'.

## 2.2.1 Distributing maps

This description covers only the simplest configuration; i.e., one with no areas, so that all nodes have a map of the entire network. The hierarchical case is somewhat more complex; see the various protocol specifications.

As previously mentioned, the first main stage in the link-state algorithm is to give a map of the network to every node. This is done with several subsidiary steps.

### Determining the neighbors of each node

First, each node needs to determine what other ports it is connected to, over fully working links; it does this using a reachability protocol which it runs periodically and separately with each of its directly connected neighbors.

### Distributing the information for the map

Next, each node periodically (and in case of connectivity changes) sends a short message, the [link-state advertisement](http://en.wikipedia.org/wiki/Link-state_advertisement), which:

* Identifies the node which is producing it.
* Identifies all the other nodes (either routers or networks) to which it is directly connected.
* Includes a sequence number, which increases every time the source node makes up a new version of the message.

This message is then flooded throughout the network. As a necessary precursor, each node in the network remembers, for every other node in the network, the sequence number of the last link-state message which it received from that node.

Starting with the node which originally produced the message, it sends a copy to all of its neighbors. When a link-state advertisement is received at a node, the node looks up the sequence number it has stored for the source of that link-state message. If this message is newer (i.e., has a higher sequence number), it is saved, and a copy is sent in turn to each of that node's neighbors. This procedure rapidly gets a copy of the latest version of each node's link-state advertisement to every node in the network.

Networks running link state algorithms can also be segmented into hierarchies which limit the scope of route changes. These features mean that link state algorithms scale better to larger networks.

### Creating the map

Finally, with the complete set of link-state advertisements (one from each node in the network) in hand, each node produces the graph for the map of the network. The algorithm iterates over the collection of link-state advertisements; for each one, it makes links on the map of the network, from the node which sent that message, to all the nodes which that message indicates are neighbors of the sending node.

No link is considered to have been correctly reported unless the two ends agree; i.e., if one node reports that it is connected to another, but the other node does not report that it is connected to the first, there is a problem, and the link is not included on the map.

### Notes about this stage

The link-state message giving information about the neighbors is recomputed, and then flooded throughout the network, whenever there is a change in the connectivity between the node and its neighbors; e.g., when a link fails. Any such change will be detected by the reachability protocol which each node runs with its neighbors.

## 2.2.2 Calculating The Routing Table

As initially mentioned, the second main stage in the link-state algorithm is to produce routing tables, by inspecting the maps. This is again done with several steps.

### Calculating the shortest paths

Each node independently runs an [algorithm](http://en.wikipedia.org/wiki/Algorithm) over the map to determine the [shortest path](http://en.wikipedia.org/wiki/Shortest_path_problem) from itself to every other node in the network; generally some variant of [Dijkstra's algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) is used. This is based around a link cost across each path which includes available bandwidth among other things.

A node maintains two data structures: a [tree](http://en.wikipedia.org/wiki/Tree_data_structure) containing nodes which are "done", and a list of candidates. The algorithm starts with both structures empty; it then adds to the first one the node itself. The variant of a [Greedy Algorithm](http://en.wikipedia.org/wiki/Greedy_Algorithm) then repetitively does the following:

* All neighbour nodes which are directly connected to the node are just added to the tree (excepting any nodes which are already in either the tree or the candidate list). The rest are added to the second (candidate) list.
* Each node in the candidate list is compared to each of the nodes already in the tree. The candidate node which is closest to any of the nodes already in the tree is itself moved into the tree and attached to the appropriate neighbor node. When a node is moved from the candidate list into the tree, it is removed from the candidate list and is not considered in subsequent iterations of the algorithm.

The above two steps are repeated as long as there are any nodes left in the candidate list. (When there are none, all the nodes in the network will have been added to the tree.) This procedure ends with the tree containing all the nodes in the network, with the node on which the algorithm is running as the root of the tree. The shortest path from that node to any other node is indicated by the list of nodes one traverses to get from the root of the tree, to the desired node in the tree.

### Filling the routing table

With the shortest paths in hand, the next step is to fill in the routing table. For any given destination node, the best path for that destination is the node which is the first step from the root node, down the branch in the shortest-path tree which leads toward the desired destination node. To create the routing table, it is only necessary to walk the tree, remembering the identity of the node at the head of each branch, and filling in the routing table entry for each node one comes across with that identity.

### Optimizations to the algorithm

The algorithm described above was made as simple as possible, to aid in ease of understanding. In practice, there are a number of optimizations which are used.

Most importantly, whenever a change in the connectivity map happens, it is necessary to recompute the shortest-path tree, and then recreate the routing table. Work by [BBN Technologies](http://en.wikipedia.org/wiki/BBN_Technologies) discovered how to recompute only that part of the tree which could have been affected by a given change in the map.

Also, the routing table would normally be filled in as the shortest-path tree is computed, instead of making it a separate operation.

## Failure modes

If all the nodes are not working from exactly the same map, routing loops can form. These are situations in which, in the simplest form, two neighboring nodes each think the other is the best path to a given destination. Any packet headed to that destination arriving at either node will loop between the two, hence the name. Routing loops involving more than two nodes are also possible.

This can occur since each node computes its shortest-path tree and its routing table without interacting in any way with any other nodes. If two nodes start with different maps, it is possible to have scenarios in which routing loops are created.

# 2.2.3 Dijkstra's Algorithm

Dijkstra's algorithm, conceived by Dutch [computer scientist](http://en.wikipedia.org/wiki/Computer_scientist) [Edsger Dijkstra](http://en.wikipedia.org/wiki/Edsger_Dijkstra) in 1956 and published in 1959, is a [graph search algorithm](http://en.wikipedia.org/wiki/Graph_search_algorithm) that solves the single-source [shortest path problem](http://en.wikipedia.org/wiki/Shortest_path_problem) for a [graph](http://en.wikipedia.org/wiki/Graph_(mathematics)) with non-negative [edge](http://en.wikipedia.org/wiki/Edge_(graph_theory)) path costs, producing a [shortest path tree](http://en.wikipedia.org/wiki/Shortest_path_tree).

For a given source [vertex](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined. For example, if the vertices of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. As a result, the shortest path first is widely used in network [routing protocols](http://en.wikipedia.org/wiki/Routing_protocol), most notably [IS-IS](http://en.wikipedia.org/wiki/IS-IS) and [OSPF](http://en.wikipedia.org/wiki/OSPF) (Open Shortest Path First).

Dijkstra's original algorithm does not use a [min-priority queue](http://en.wikipedia.org/wiki/Min-priority_queue) and runs in O(|V|^2) (where |V| is the number of vertices). The idea of this algorithm is also given in ([Leyzorek et al. 1957](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#CITEREFLeyzorekGrayJohnsonLadew1957)). The implementation based on a min-priority queue implemented by a [Fibonacci heap](http://en.wikipedia.org/wiki/Fibonacci_heap) and running in O(|E|+|V|\log|V|) (where |E| is the number of edges) is due to ([Fredman & Tarjan 1984](http://en.wikipedia.org/wiki/Dijkstra's_algorithm#CITEREFFredmanTarjan1984)). This is [asymptotically](http://en.wikipedia.org/wiki/Asymptotic_computational_complexity) the fastest known single-source [shortest-path algorithm](http://en.wikipedia.org/wiki/Shortest_path_problem) for arbitrary [directed graphs](http://en.wikipedia.org/wiki/Directed_graph) with unbounded non-negative weights.

## 2.2.3.1 Algorithm

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8 if this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the unvisited set.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

Let us understand with the following example:

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/Fig-11.jpg)

**Fig 2.5**

The set sptSetis initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/MST1.jpg) **Fig 2.6**

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/DIJ2.jpg) **Fig 2.7**

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/DIJ3.jpg) **Fig 2.8**

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/DIJ4.jpg) **Fig 2.9**

We repeat the above steps until sptSet doesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

[](http://d2o58evtke57tz.cloudfront.net/wp-content/uploads/DIJ5.jpg) **Fig 2.10**

## 2.2.3.2 Pseudocode

In the following algorithm, the code u := vertex in Q with smallest dist, searches for the vertex u in the vertex set Q that has the leastdist[u] value. That vertex is removed from the set Q and returned to the user. dist\_between(u, v) calculates the length between the two neighbor-nodes u and v. The variable alt on lines 20 & 22 is the length of the path from the root node to the neighbor node v if it were to go through u. If this path is shorter than the current shortest path recorded for v, that current path is replaced with this alt path. The previous array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

1 function Dijkstra(Graph, source):

2 for each vertex v in Graph: // Initializations

3 dist[v] := infinity; // Mark distances from source to v as not yet computed

4 visited[v] := false; // Mark all nodes as unvisited

5 previous[v] := undefined; // Previous node in optimal path from source

6 end for

7

8 dist[source] := 0; // Distance from source to itself is zero

9 insert source into Q; // Start off with the source node

10

11 while Q is not empty: // The main loop

12 u := vertex in Q with smallest distance in dist and has not been visited; // Source node in first case

13 remove u from Q;

14 visited[u] := true // mark this node as visited

15

16 for each neighbor v of u:

17 alt := dist[u] + dist\_between(u, v); // accumulate shortest dist from source

18 if alt < dist[v] && !visited[v]:

19 dist[v] := alt; // keep the shortest dist from src to v

20 previous[v] := u;

21 insert v into Q; // Add unvisited v into the Q to be processed

22 end if

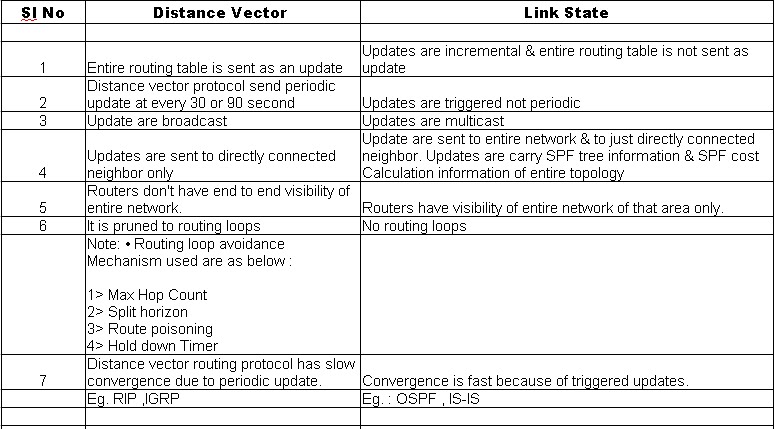
23 end for

24 end while

25 return dist;

26 endfunction

**2.3 Comparison Between Distance Vector And Link State Routing Protocols**



**Fig 2.11**

**3. RESEARCH METHODOLOGY**

In this section, we will discuss the breakdown of the project implementation from taking the user inputs for the number of nodes and costs to finding out the most efficient path according to the algorithm used.

The implementation of Dijkstra shortest path algorithm, Prims algorithm has been done using C++. They have also been implemented in MATLAB along with Bellman-Ford algorithm.

**3.1 C++ Implementation**

The C++ code explains the Dijkstra shortest path algorithm to find the shortest path between various nodes of a given graph.

It also explains the Prims algorithm to find the shortest path from one node to all the other nodes in a graph.

**3.1.1 Steps involved**

* The user is asked to enter the number of nodes in the network.
* Subsequently the user enters the cost of each path that exists. The values for a path from node to node itself are ignored.
* The graphical representation of the graph is generated along with the Adjacency matrix representation.
* The user is given a choice to choose the algorithm to be used:
* Dijkstra algorithm
* Prims algoritm
* If dijkstra is chosen the source and destination nodes are taken from the user and the shortest path is graphically generated along with computation of the cost of the path.
* If Prims is chosen only the source node is taken from the user and the system generates the shortest paths from that node to all the other nodes along with the costs.

**3.1.2 FEATURES**

* The code has been designed so as to provide a graphical visualization and tabular representation of the shortest path algorithms.
* The Applet window and the network nodes have been designed to have a one to one correspondence with their size. The network nodes are designed with relative spacing with respect to the width and height of the graphics window. Hence the it is browser independent, monitor independent and window size independent.
* The code has been tested to provide accurate results for any number of nodes and any complexity of the network.
* The code allows the implementation of both directed and bi-directional graphs.

**3.1.3 TECHNOLOGY INFRASTRUCTURE**

**C++** is a [programming language](http://en.wikipedia.org/wiki/Programming_language) that is general purpose, [statically typed](http://en.wikipedia.org/wiki/Statically_typed), [free-form](http://en.wikipedia.org/wiki/Free-form_language), [multi-paradigm](http://en.wikipedia.org/wiki/Multi-paradigm_programming_language) and [compiled](http://en.wikipedia.org/wiki/Compiled_language). It is regarded as an intermediate-level language, as it comprises both [high-level](http://en.wikipedia.org/wiki/High-level_programming_language) and [low-level](http://en.wikipedia.org/wiki/Low-level_programming_language) language features. C++ was originally named **C with Classes**, adding [object oriented](http://en.wikipedia.org/wiki/Object-oriented_programming) features, such as classes, and other enhancements to the [C programming language](http://en.wikipedia.org/wiki/C_(programming_language)).

C++ is one of the most popular programming languages and is implemented on a wide variety of hardware and operating system platforms. As an efficient compiler to native code, its application domains include systems software, [application software](http://en.wikipedia.org/wiki/Application_software), device drivers, embedded software, high-performance server and client applications, and entertainment software.

C++ introduces [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming) (OOP) features to C. It offers [classes](http://en.wikipedia.org/wiki/Class_(computer_science)), which provide the four features commonly present in OOP (and some non-OOP) languages: [abstraction](http://en.wikipedia.org/wiki/Abstraction_(computer_science)), encapsulation, inheritance, and [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_(computer_science)).

**Encapsulation**

[Encapsulation](http://en.wikipedia.org/wiki/Information_hiding) is the hiding of information to ensure that data structures and operators are used as intended and to make the usage model more obvious to the developer. C++ provides the ability to define classes and functions as its primary encapsulation mechanisms.

#### Inheritance

[Inheritance](http://en.wikipedia.org/wiki/Inheritance_(computer_science)) allows one data type to acquire properties of other data types. Inheritance from a [base class](http://en.wikipedia.org/wiki/Base_class) may be declared as public, protected, or private. This access specifier determines whether unrelated and derived classes can access the inherited public and protected members of the base class.

### Polymorphism

[Polymorphism](http://en.wikipedia.org/wiki/Type_polymorphism) enables one common interface for many implementations, and for objects to act differently under different circumstances.

**Abstraction**

**Abstraction** is the process by which [data](http://en.wikipedia.org/wiki/Data_(computing)) and [programs](http://en.wikipedia.org/wiki/Program_(machine)) are defined with a [representation](http://en.wikipedia.org/wiki/Representation_(mathematics)) similar in form to its meaning ([semantics](http://en.wikipedia.org/wiki/Semantics#Computer_science)), while hiding away the [implementation](http://en.wikipedia.org/wiki/Implementation#Computer_Science) details. Abstraction tries to reduce and factor out details so that the [programmer](http://en.wikipedia.org/wiki/Programmer) can focus on a few concepts at a time.

**3.1.4 C++ Code**

#include <stdio.h>

#include <iostream>

#include <graphics.h>

using namespace std;

#define inf 9999

#define size 10/\*Defining maximum size of the matrix\*/

#define True 1

#define False 0

int n,v1,v2,temp[size],c;

int main( )

{

int a[size][size],i,j,a1[size][2],lcost,i1,gr[size];

int dij(int[size][size],int,int,int);

int prim(int[size][size],int,int);

printf("Enter the number of nodes : ");

scanf("%d",&n);

/\*Input 0 if there is no direct edge between vertex pair\*/

printf("Enter a weighted matrix(with weights) as input :\n");

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

printf("Enter the value of a[%d][%d] : ",i,j);

scanf("%d",&a[i][j]);

}

}

printf("The entered matrix is:\n");

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

printf("%d \t",a[i][j]);

printf("\n");

}

int gd = DETECT, gm;

initgraph(&gd, &gm, "C:\\TC\\BGI");

i1=200;float n1=400;

for(j=0;j<n;j++)

{

circle(i1, n1-i1, 10);

char\* node=itoa(j,node,10);

outtextxy(i1,n1-i1,node);

a1[j][0]=i1;

a1[j][1]=n1-i1;

i1+=120;n1+=(j+1.5)\*75;

}

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

if(a[i][j]!=0 && i!=j)

{

line(a1[i][0], a1[i][1],a1[j][0],a1[j][1]);

char\* str=itoa(a[i][j],str,10);

outtextxy((a1[i][0]+ a1[j][0])/2,(a1[i][1]+a1[j][1])/2,str);

} }

}

getch();

closegraph();

char w;

w='y';

while( w=='y'||w=='Y')

{

cout<<"Enter the algorithm to be used: \n1. Dijkstra\n2. Prims\n";

int ch;

cin>>ch;

if(ch!=1&&ch!=2)

{

cout<<"invalid choice ";

}

else{

switch(ch){

case 1:

printf(" Enter starting vertex v");

scanf("%d",&v1);

printf("Enter ending vertex v");

scanf("%d",&v2);

/\*Check for validity of input vertices\*/

if(v1<0||v1>n-1||v2<0||v2>n-1)

{

printf("!!!!!ERROR!!!!!n");

printf("!!!!!Invalid vertex given!!!!!");

return 0;

break;

}

printf("Shortest path between v%d & v%d : ",v1,v2);

lcost=dij(a,n,v1,v2);

printf("\n \n Shortest cost between v%d & v%d : ",v1,v2);

printf("%d",lcost);/\*Print the shortest cost\*/

getch();

return 0;

break;

case 2:

printf("Enter starting vertex: v");

int vs;

scanf("%d",&vs);/\*Read the starting vertex from user\*/

if(vs<0||vs>n-1)/\*Checking validity of the starting vertex given by the user\*/

{

printf("!!!!!ERROR!!!!!n");

printf("!!!!!Invalid vertex given!!!!!");

}

printf("nSelected order of edges : ");

int min\_weight=prim(a,n,vs); //call the prim function

printf("\n");

printf("Minimum weight :%d",min\_weight);/\*The total weight of the minimal spanning tree is displayed in the output\*/

getch();

return 0;

break;

}

}

cout<<"Do you want to continue..... press Y if yes else press any key";

cin>>w;

}

}

int dij(int a[size][size],int n,int v1,int v2)

{

int length[size],set[size], path[size], i ,j ,s ,z ,tmp ,temp[size] ,c=0 ,f=0 ,a2[size][size] ,m ,l ,a1[size][size] ,tree[size][size];

s=v1;

z=v2;

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

tree[i][j]=0;

}

}

int srch\_min(int[],int[],int);

for(i=0;i<n;i++)

set[i]=0;

for(i=0;i<n;i++)

{

if(a[s][i]==0)/\*There is no direct edge between vertices s and i\*/

{

length[i]=inf;

path[i]=0;/\*Empty path\*/

}

else

{

length[i]=a[s][i];

path[i]=s;/\*s is immediate predecessor of i\*/

}

}

set[s]=1;/\*s is included in the set\*/

length[s]=0;/\*s is implicitly enumerated with length as 0\*/

while(set[z]!=1)/\*Iteration will be considered until final vertex z belongs to s\*/

{

j=srch\_min(length,set,n);/\*Select a vertex j with minimum label such that it is not included in the set[]\*/

set[j]=1;/\*Vertex j is included in the set[]\*/

for(i=0;i<n;i++)

{

if(set[i]!=1)

{

if(a[i][j]!=0)

{

if(length[j]+a[i][j]<length[i])/\*When exsisting label is not minimum only then replacement is done\*/

{

length[i]=length[j]+a[i][j];

path[i]=j;

}

}

}

}

}

j=0;

i=z;

while(i!=s)

{

tmp=path[i];

temp[j]=tmp;

i=tmp;

j++;

c++;

}

for(j=c-1;j>=0;j--)

{

printf("v%d--->",temp[j]);/\*Print the shortest path\*/

if(temp[j-1]){

m=temp[j];l=temp[j-1];

tree[m][l]=1;

}

if(temp[j]==z)

f=1;

}

int gd = DETECT, gm;

initgraph(&gd, &gm, "C:\\TC\\BGI");

int i1=200; int n1=400;int i2=180;

for(j=0;j<n;j++)

{

circle(i1, n1-i1, 10);

// char\* node1=itoa(j,node1,10);

// outtextxy(i1,n1-i1,node1);

a1[j][0]=i1;

a1[j][1]=n1-i1;

i1+=120;n1+=(j+1.5)\*75;

}

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

if(a[i][j]!=0 && i!=j)

{

setcolor(15);

line(a1[i][0], a1[i][1],a1[j][0],a1[j][1]);

if(tree[i][j]!=0)

{

setcolor(14);

line(a1[i][0]+5, a1[i][1]+10,a1[j][0],a1[j][1]+8);}

//char\* str1=itoa(tree[i][j],str1,10);

//outtextxy((a1[i][0]+ a1[j][0])/2,(a1[i][1]+a1[j][1])/2,str1);

}

}

}

getch();

//closegraph();

if(f!=1)

printf("v%d",z);

printf("n");

return length[z];

}

/\*This function will return a vertex with minimum label such that it is not included in set[]\*/

int srch\_min(int length[],int set[],int n)

{

int min,i,min\_index;

min=99999,min\_index;

for(i=1;i<n;i++)

{

if(set[i]!=1)

{

if(length[i]<min)

{

min=length[i];

min\_index=i;

}

}

}

return min\_index;

}

int prim(int a[size][size],int n,int vs)

{

int selected[size],tree[size][size],nv,i,j,x,y,cost=0,min,i1,n1,a1[size][size];

for(i=0;i<n;i++)

selected[i]=False;/\*Initially no vertices are selected\*/

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

tree[i][j]=0;/\*Initially spanning tree is empty\*/

}

selected[vs]=True;/\*Starting vertex is selected at first\*/

nv=1;

while(nv<n)/\*Iteration will be considered until all the vertices are selected\*/

{

min=inf;/\*min is initialized by a large value\*/

for(i=0;i<n;i++)

{

if(selected[i]==True)/\*Iteration will be considered iff i th vertex is already selected\*/

{

for(j=0;j<n;j++)

{

if(selected[j]==False)/\*Iteration will be considered iff j th vertex is not already selected\*/

{

if(a[i][j]!=0)

/\*Iteration will be considered iff there is a path between i th and j th vertex\*/

{

if(min>a[i][j])/\*Search for an edge with minimum weight\*/

{

min=a[i][j];

x=i;

y=j;

}

}

}

}

}

}

cost=cost+min;

/\*Updation of previous cost by adding it to cost of newly selected edge\*/

tree[x][y]=min;

/\*The newly selected edge is included in the minimal spanning tree\*/

selected[y]=True;

nv++;

printf("(v%d,v%d)->",x,y);

}

printf("\b \b\b ");

printf("\n");

printf("\nThe spanning tree is(with weights) : \n");

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

printf("%d\t",tree[i][j]);

printf("\n");

}

int gd = DETECT, gm;

initgraph(&gd, &gm, "C:\\TC\\BGI");

i1=200; n1=400;int i2=180;

for(j=0;j<n;j++)

{

circle(i1, n1-i1, 10);

// char\* node1=itoa(j,node1,10);

// outtextxy(i1,n1-i1,node1);

a1[j][0]=i1;

a1[j][1]=n1-i1;

i1+=120;n1+=(j+1.5)\*75;

}

for(i=0;i<n;i++)

{

for(j=0;j<n;j++)

{

if(a[i][j]!=0 && i!=j)

{

setcolor(15);

line(a1[i][0], a1[i][1],a1[j][0],a1[j][1]);

if(tree[i][j]!=0 && i!=j)

{

setcolor(14);

line(a1[i][0]+5, a1[i][1]+10,a1[j][0],a1[j][1]+8);}

//char\* str1=itoa(tree[i][j],str1,10);

//outtextxy((a1[i][0]+ a1[j][0])/2,(a1[i][1]+a1[j][1])/2,str1);

}

}

}

getch();

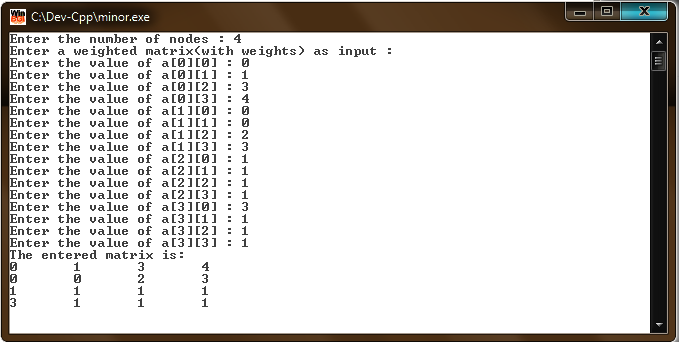
//closegraph();

return cost;

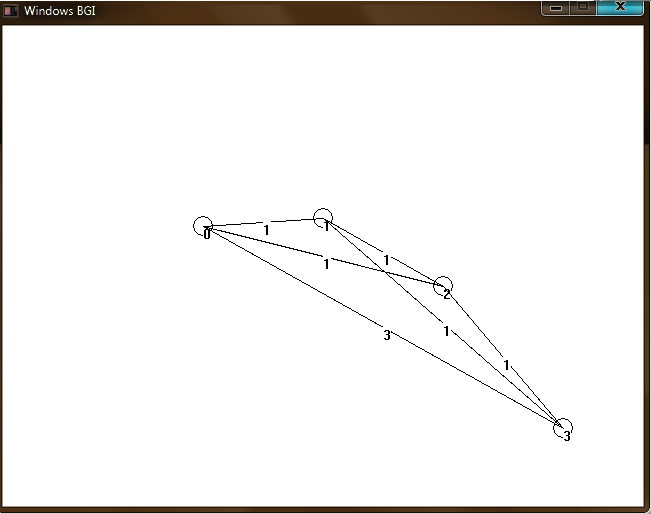
}

**3.1.5 OUTPUTS**

SCREENSHOTS

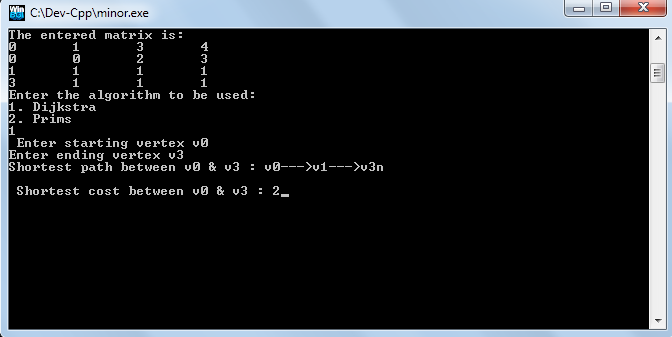


**Fig. 3.1** Input values for graph

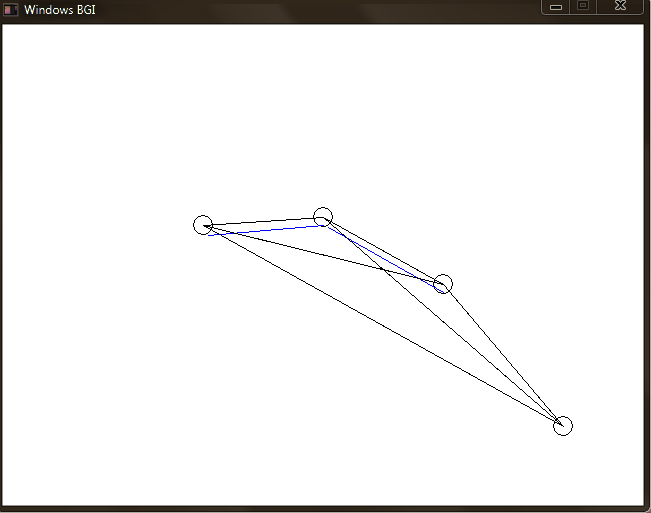


**Fig 3.2** Graph generated

DIJKSTRA SHORTEST PATH ALGORITHM

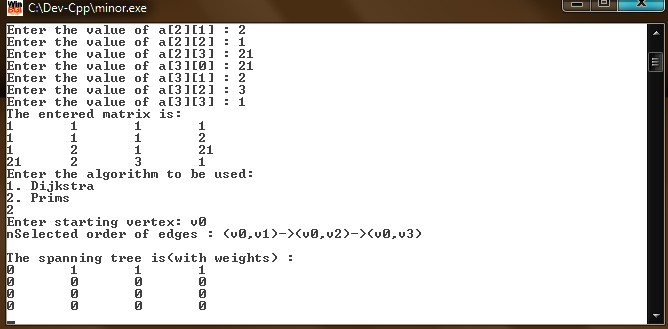


**Fig 3.3** Adjacency Matrix displayed. User asked to choose the algorithm. Shortest path and cost displayed(Dijikstra’s)

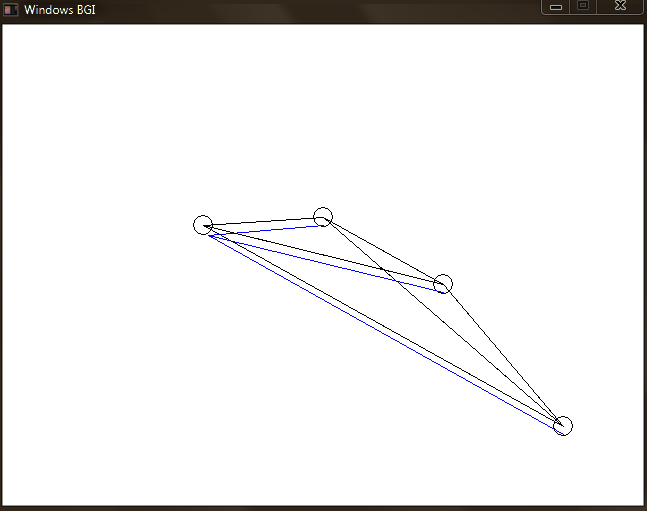


**Fig 3.4** Graph generated along with shortest path (Blue color)

PRIMS ALGORITHM



**Fig. 3.5** Input values for graph, Adjacency Matrix displayed. User asked to choose the algorithm. Spanning tree displayed(Prims)



**Fig 3.6** Graph generated along with shortest path (Blue color)

**3.2 MATLAB implementation**

The C++ code explains the Dijkstra shortest path algorithm to find the shortest path between various nodes of a given graph.

It also explains the Prims algorithm to find the shortest path from one node to all the other nodes in a graph.

**3.2.1 Steps Involved**

* The user is asked to enter the number of nodes in the network.
* Subsequently the user enters the cost of each path that exists. The values for a path from node to node itself are ignored.
* The graphical representation of the graph is generated along with the Adjacency matrix representation.
* Required inputs are taken from the user on the basis of algorithm being implemented (Dijkstra shortest path algorithm/Bellmann-Ford Routing algorithm)
* The shortest path is displayed graphically along with the cost generated.

**3.2.2 Features**

* The code has been designed so as to provide a graphical visualization and tabular representation of the shortest path algorithms.
* The graphics window and the network nodes have been designed to have a one to one correspondence with their size. The network nodes are designed with relative spacing with respect to the width and height of the graphics window. Hence it is browser independent, monitor independent and window size independent.
* The code has been tested to provide accurate results for any number of nodes and any complexity of the network.

**3.2.3 TECHNOLOGY INFRASTRUCTURE**

**MATLAB** (**mat**rix **lab**oratory) is a [numerical computing](http://en.wikipedia.org/wiki/Numerical_analysis) environment and [fourth-generation programming language](http://en.wikipedia.org/wiki/Fourth-generation_programming_language). Developed by [MathWorks](http://en.wikipedia.org/wiki/MathWorks), MATLAB allows [matrix](http://en.wikipedia.org/wiki/Matrix_(mathematics)) manipulations, plotting of [functions](http://en.wikipedia.org/wiki/Function_(mathematics)) and data, implementation of [algorithms](http://en.wikipedia.org/wiki/Algorithm), creation of [user interfaces](http://en.wikipedia.org/wiki/User_interface), and interfacing with programs written in other languages, including [C](http://en.wikipedia.org/wiki/C_(programming_language)), [C++](http://en.wikipedia.org/wiki/C%2B%2B), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)).

MATLAB's support for [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming) includes classes, inheritance, virtual dispatch, packages, pass-by-value semantics, and pass-by-reference semantics.

**Key Features**

* High-level language for numerical computation, visualization, and application development
* Interactive environment for iterative exploration, design, and problem solving
* Mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration, and solving ordinary differential equations
* Built-in graphics for visualizing data and tools for creating custom plots
* Development tools for improving code quality and maintainability and maximizing performance
* Tools for building applications with custom graphical interfaces
* Functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET, and Microsoft® Excel®

**3.2.4 MATLAB Codes**

(i) Dijkstra Shortest Path

clc

clear

n=input('Enter the no. of nodes: ');

for i = 1:n

for j = 1:n

str = ['Enter element in row ' num2str(i) ', col ' num2str(j) ': '];

a(i,j) = input(str);

end

end

display(a);

DG = sparse(a);

display(DG);

g=DG;

h = view(biograph(DG,[],'ShowWeights','on'));

so=input ('Enter the source node : ');

d=input ('Enter the Destination node : ');

[dist,path,pred]=graphshortestpath(DG,so,d,'Directed',true,'method','Dijkstra');

[dist1,path1,pred1] = graphshortestpath(DG,so,d);

display(path1);

%display(pred1);

display(dist1);

set(h.Nodes(path),'Color',[1 0.4 0.4]);

edges = getedgesbynodeid(h,get(h.Nodes(path),'ID'));

set(edges,'LineColor',[1 0 0])

set(edges,'LineWidth',1.5)

(b) Bellmann-Ford Routing Alhgoritm

clc

clear

n=input('Enter the no. of nodes: ');

for i = 1:n

for j = 1:n

str = ['Enter element in row ' num2str(i) ', col ' num2str(j) ': '];

a(i,j) = input(str);

end

end

display(a);

DG = sparse(a);

display(DG);

g=DG;

h = view(biograph(DG,[],'ShowWeights','on'));

so=input ('Enter the source node : ');

d=input ('Enter the Destination node : ');

[dist,path,pred]=graphshortestpath(DG,so,d,'Directed',true,'method','Bellman-Ford');

[dist1,path1,pred1] = graphshortestpath(DG,so,d);

display(path1);

%display(pred1);

display(dist1);

set(h.Nodes(path),'Color',[1 0.4 0.4]);

edges = getedgesbynodeid(h,get(h.Nodes(path),'ID'));

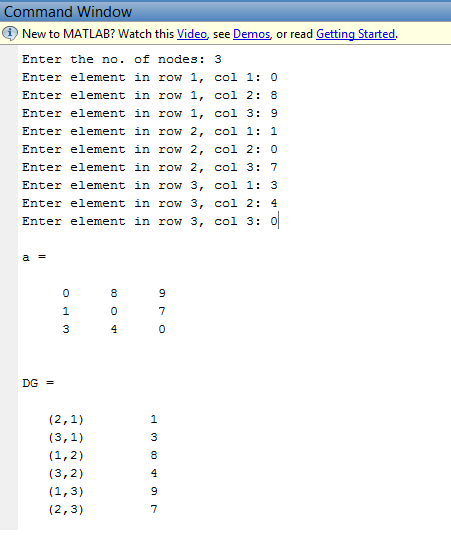
set(edges,'LineColor',[1 0 0])

set(edges,'LineWidth',1.5)

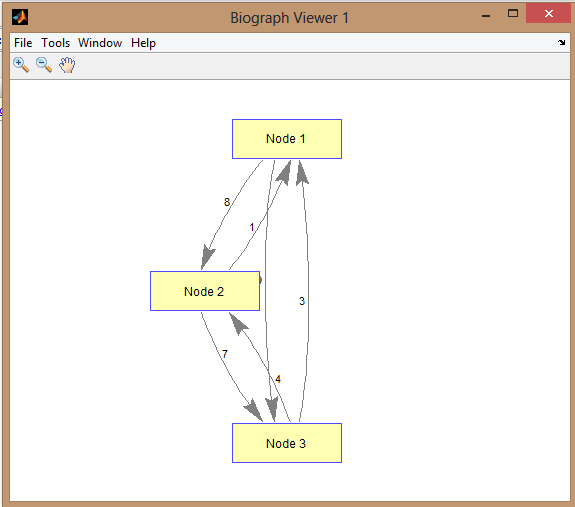
**3.2.5 OUTPUTS**

**SCREENSHOTS**

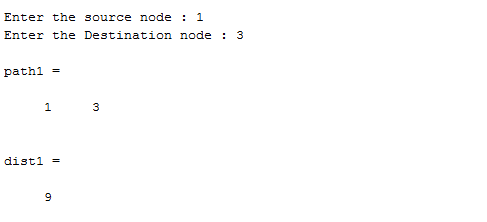
DIJKSTRA SHORTEST PATH ALGORITHM



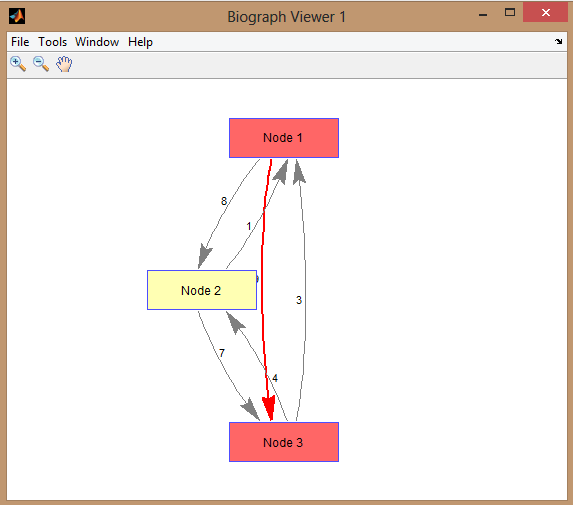
**Fig 3.7** Input values for graph



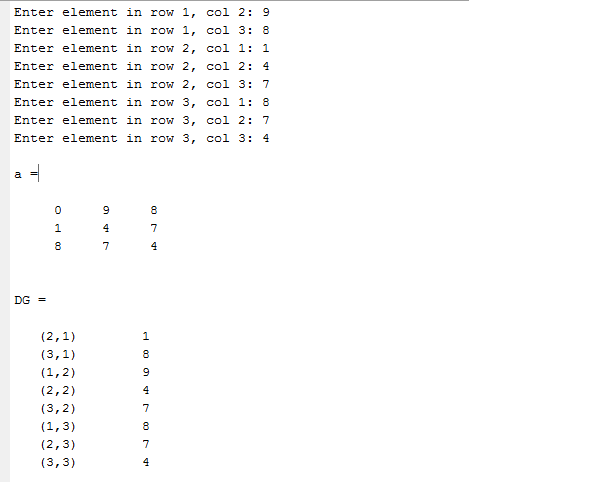
**Fig 3.8** Graph generated



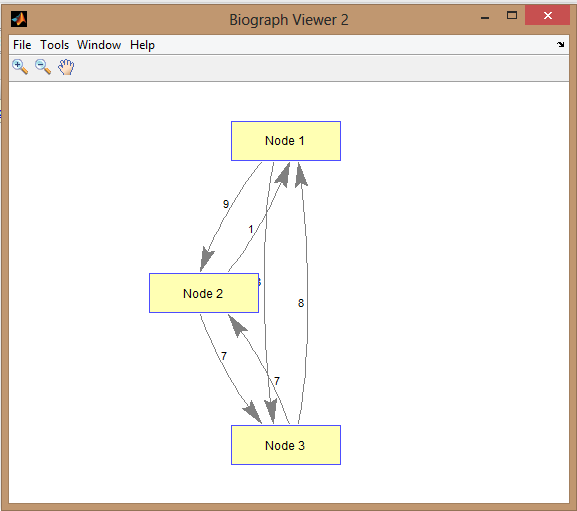
**Fig 3.9**  Input for source and destination taken and path and cost calculated



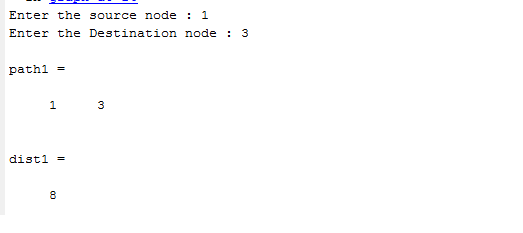
**Fig 3.10** Graph generated along with shortest path (Red color)

BELLMANN-FORD ROUTING ALGORITHM

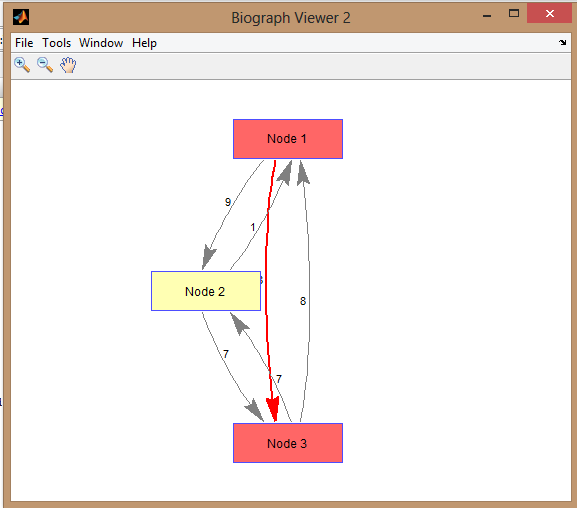
**Fig 3.11** Input values for graph



**Fig 3.12** Graph generated



**Fig 3.13**  Input for source and destination taken and path and cost calculated



**Fig 3.14** Graph generated along with shortest path (Red color)