CS4150 Computer Networks Laboratory – Assignment 4

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

1. ***Write a program that uses Dijkstra's algorithm to compute the least cost path from a given node to every other node in the network. Your program should take 3 arguments: the first argument is n – number of nodes in the network, the second argument is s – the source node, and the third argument is is the name of the file which contains the cost (a positive integer less than 10,000) between every pair of nodes in the network. Note that every node need not be connected to every other node in the network. Your program should have a run-time complexity of at-most O(n2), and should output the least cost path (along with the path price) from s to every other node in the network.***

Dijkstra's algorithm is a single source shortest path (SSSP) algorithm which is a greedy algorithm that finds the shortest path from all nodes to the give source node.
The program takes three arguments

i. Number of nodes (n: Integer): This tells the number of nodes in the graph. Given an integer n as input, a graph is create with nodes from 1 to n.
ii. Source node (k: Integer, $1 \leq k \leq n$): The start node from which the shortest distances are calculated.
iii. Input file (f: string): The input file from which the edges are read. Each line is of the form "u v c", where u-v is the undirected edge and c is the cost of the edge.

The *dijkstra* function is called with source as the input argument and the working of the function is as follows:

1. Initialise a *dist* array to *infinity* in which the node at index *i* is at a distance of *dist[i]* from the source node.
2. Set the distance of the source to itself as 0 i.e. *dist[source] = 0.*
3. Initialise a priority queue with priority as the distance of the node from source and key as the node itself.
4. Now until all nodes are assigned a distance, we do the following:
   4.1. Get the node with the minimum distance from source i.e. pop from the priority queue. Let us say that this node is M.
   4.2. For the min-cost node (obtained in the previous step) M, we go through all its neighbours and do the following:
      4.2.1. If its neighbour (N) has a shorter path through the current min-cost node (obtained in 4.1.), we update the shortest path of the current neighbour (N) as *dist[M] + cost* where *cost* is the weight (distance) of the edge between the nodes M and N.
      4.2.2. If the distance was updated in the previous step, then we add the current neighbour to the priority queue (with priority *dist[N]* and key N) since the neighbours of the current neighbour might benefit from the current shortest path.

A high level overview of this algorithm is that it picks a node and for each of its neighbours, it tries to see if the neighbour can have a shorter path to the source through the current node, if this check is positive, it updates the distance of the neighbour and puts it in the priority queue so that its neighbours can benefit from this shorter path.

2. ***Write a program to simulate the Distance Vector (DV) routing algorithm. Your program should take 2 arguments: the first argument is n – number of nodes in the network, and the second argument is the name of the file which contains the cost (a positive integer less than 10,000) between every pair of nodes in the network. Note that every node need not be connected to every other node in the network. Note that your program should be able to demonstrate the possibly random local routing table updates that can happen in DV, and should terminate and print the local routing table at every node after convergence has occurred.***

The distance vector routing algorithm is similar to the Bellman-Ford algorithm which is a dynamic programming based approach to the shortest path between all pairs in a graph.
The program takes two arguments

i Number of nodes (n: Integer): This tells the number of nodes in the graph. Given an integer n as input, a graph is create with nodes from 1 to n.
ii Input file (f: string): The input file from which the edges are read. Each line is of the form "u v c", where u-v is the undirected edge and c is the cost of the edge.

Some important variables to understand:

1. *converged:* It stores a list of nodes, that have converged to their minimum distance to all nodes. The algorithm stops when each node have converged to the minimum distance to all the other nodes.
2. *updated:* It tells whether the current step in the DV routing algorithm has produced any updates to current node.

The algorithm flow is as follows:

1. Initialize *converged* to an empty set (we need to store a node only once and sets server this purpose).
2. Until the *converged* set has all nodes, do the following:
   2.1. Generate a random node (this simulates the random updates in the routing table) from the set of all nodes. Let this node be M.
   2.2. Now the DV routing algorithm is run for one step with the current node (M) as the main focus.
   2.3. In the current step, we iterate through all neighbours of the node M
      2.3.1. For each neighbour (N), we go through all the nodes in the graph and do the following:
      For each node (V), if there is an edge between the node N and V and if the current distance between the nodes M and V is higher than the distance from node M to V via N, then we update the distance between nodes M and V to the sum of weight(distance) between the edge N-V and the current distance between the nodes M and N.
   2.4. If there was no update in the iteration in 2.3.1., then we return *updated* as false and the current node (M) is added to the set *converged.*
   2.5. If there was an update in the iteration in 2.3.1., the we return *updated* as true and the *converged* set is cleared since if there was at-least one update, then there is a possibility that other nodes might be updated in further iterations.

Figure 1: The input graph used for both the questions