Computer-Networks-LSR-Tips

For more notes visit

https://rtpnotes.vercel.app

For lab programs visit

https://pgmsite.netlify.app

What is LSR?

Imagine a network of routers, each one responsible for forwarding data packets to their destinations. Link State Routing allows each router to build a map of the entire network. This map shows how all the routers are connected and the cost (delay, distance, etc.) associated with sending data between them.

- **Information Sharing**: In a Link State network, each router shares information about its directly connected links with its neighbors. This information includes the cost of sending data over those links.
- **Building the Map**: Each router uses the received information to build a complete picture of the network, like a map. This map is called a Link State Database (LSDB).
- **Shortest Path Calculation**: With the LSDB, each router can independently calculate the best (shortest path) route to reach any other router in the network using Dijkstra's algorithm, which is implemented in the provided program.

Dijkstra's Algorithm in the Program

- The program calculates the shortest path from a chosen source router (`src`) to all other routers in the network. Here's how it mimics Link State Routing:
- **Nodes as Routers**: The `n` variable represents the number of routers in the network.
- **Adjacency Matrix**: The `adj` matrix represents the connections between routers. The cost of sending data between two routers is stored in the corresponding cell of the matrix.

- **Dijkstra's Algorithm**: This algorithm finds the shortest path from the source router to all other routers. It considers the cost of each link while calculating the best route.
- **Routing Table**: The program displays a routing table after each round of Dijkstra's algorithm and the final table. This table resembles the information a router would have in a Link State network, showing the best route (next hop) and the cost to reach each destination router.

Basic Algorithm for Remembering

1. Enter no of nodes
2. Enter cost between nodes
3. Initialize distances and visited array
4. Input the source node
5. Do djikstras algorithm
6. Print routing table
7. Display cost of shortest path

Steps in more Detail

1. Enter no of nodes and cost between nodes

```
printf("\nEnter the Number of Nodes: ");
scanf("%d",&n);

// Input the cost between nodes
printf("Enter the cost between Nodes:\n");
for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
                if (i == j)
                        adj[i][j] = 0;  // No self-loops
                else {
                        printf("Cost from %d -> %d: ", i + 1, j + 1);
                        scanf("%d", &adj[i][j]);
                }
        }
}
```

2. Initialise distances and visited array

```
for (i = 0; i < n; i++) {
        dist[i] = max;
        visited[i] = 0;
}
```

3. Input source node

- Here `nxhop` is an array that keeps track of the next hop for each node when constructing the shortest path tree using Dijkstra's algorithm.
- Essentially, `nxhop` helps to identify the intermediate node that should be visited next to reach a particular destination from the source node.

```
printf("\nEnter the source Node: ");
scanf("%d", &src);
src -= 1;
dist[src] = 0;

int nxhop[n];
for (i = 0; i < n; i++)
        nxhop[i] = 0;
```

4. Do Djikstras algorithm

- The outer loop runs `n-1` times because Dijkstra's algorithm requires `n-1` rounds to find the shortest path from the source to all other nodes.
- In each round, the algorithm picks the unvisited node with the smallest distance (this is done by finding `min_index`).
- The picked node is then marked as visited.
- The inner loop updates the distances to all adjacent nodes that haven't been visited yet. If a shorter path to a node `d` is found through the current node `min_index`, the distance `dist[d]` is updated.
- If `min_index` is not the source node, the `nxhop[d]` is updated to reflect the intermediate node through which the shortest path passes.

```
for (round = 0; round < n - 1; round++) {
        int min = max, min_index;

        // Find the minimum distance node from the set of unvisited nodes
        for (v = 0; v < n; v++) {
                if (visited[v] == 0 && dist[v] < min) {
                        min = dist[v];
                        min_index = v;
                }
        }

        // Mark the picked vertex as visited
        visited[min_index] = 1;
```

- `!visited[d]`: This checks if the neighboring node (`d`) hasn't been visited yet. We only care about unvisited nodes.
- `adj[min_index][d]`: This checks if there's actually a connection (edge) between the current closest node (`min_index`) and the neighboring node (`d`). Remember `adj` is likely an adjacency matrix representing connections between cities.
- `dist[min_index] != max`: This is a small safeguard to avoid overflow. It basically skips this neighbor if the distance to the current closest node (`min_index`) is already marked as infinity (max).
- `dist[min_index] + adj[min_index][d] < dist[d]`: This is the most important check. It compares the current distance to the neighboring node (`dist[d]`) with a potential shorter path.
- The potential shorter path is calculated by adding the distance to the current closest node (`dist[min_index]`) and the weight of the edge between the current closest node and the neighbor (`adj[min_index][d]`). If this sum is less than the current distance (`dist[d]`), it means there's a shorter path through the current closest node

```
// Update dist value of the adjacent vertices of the picked vertex
for (d = 0; d < n; d++) {
        if (!visited[d] && adj[min_index][d] && dist[min_index] != max
                && dist[min_index] + adj[min_index][d] < dist[d]) {
                dist[d] = dist[min_index] + adj[min_index][d];
                if (min_index != src)
                        nxhop[d] = min_index + 1;
```

```
        }
}
```

- If the previous condition is true (shorter path found), this line updates the distance to the neighboring node (`dist[d]`) with the shorter distance we just calculated.
- `min_index != src` : This check avoids setting the previous hop for the source node itself (since it has no previous hop).
- `nxhop[d] = min_index + 1` : If the current closest node (`min_index`) is not the source, then it's a previous hop on the way to the neighboring node (`d`). We store this hop information in the `nxhop` array. This information can be later used to reconstruct the actual shortest path from the source to any other node.

5. Display Routing Table

```
// Display routing table for the source node after each round
        if (min_index == src) {
            printf("\nRouting Table of Node %d", src + 1);
            printf("\nDestination\tCost\tNext Hop\n");
            for (i = 0; i < n; i++) {
                if (dist[i] == 0)
                    printf("%d\t\t-\t-\n", i + 1);
                else
                    printf("%d\t\t%d\t-\n", i + 1, dist[i]);
            }
        }
```

6. Display final routing table

```
for (i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t", i + 1, dist[i]);
        if (nxhop[i] == 0)
            printf("-\n");
        else
            printf("%d\n", nxhop[i]);
    }
```

7. Display cost

```
for (i = 1; i < n; i++)
        printf("The cost of the shortest path from router %d to %d is %d\n",
                    src + 1, i + 1, dist[i]);
```