



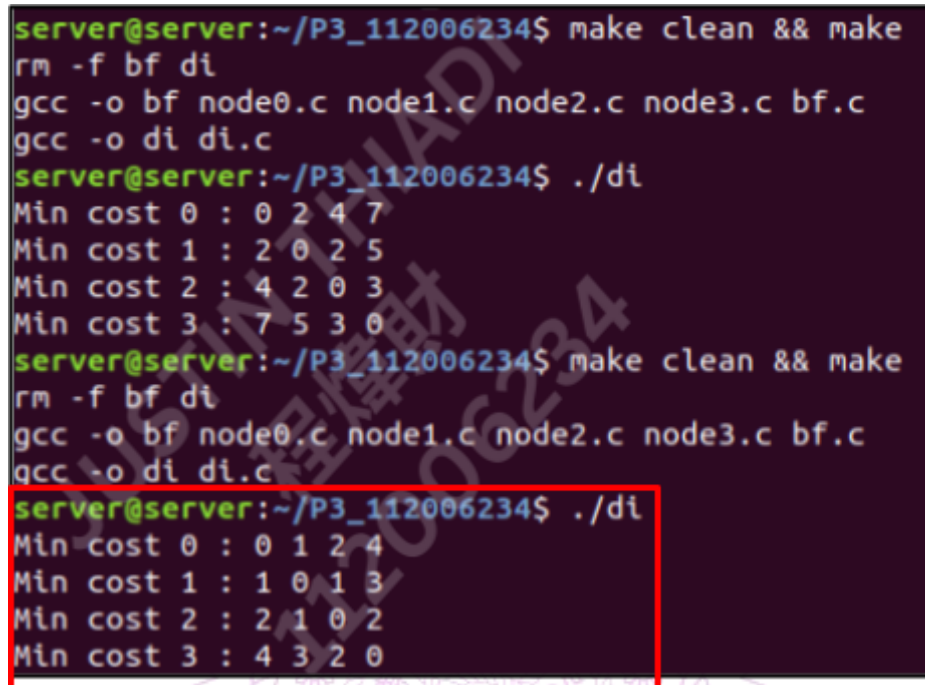
EECS3020 P3

[112006234] [Justin Thiadi 程煒財]

Task 1: Link State Routing with Dijkstra's Algorithm.....	2
1.1. Screenshots of Program.....	2
1.2. Source Code Explanation.....	3
Task 2: Distance Vector Routing with Bellman-Ford Algorithm.....	5
2.1. Screenshots of Program.....	5
2.2. Source Code Explanation.....	7
Task 3: Comparison of Dijkstra and Bellman-Ford Algorithms.....	14
3.1. Question 1.....	14
3.2. Question 2.....	15
3.3. Question 3.....	16
3.4. Question 4.....	16
3.5. Question 5.....	17

Task 1: Link State Routing with Dijkstra's Algorithm

1.1. Screenshots of Program



```
server@server:~/P3_112006234$ make clean && make
rm -f bf di
gcc -o bf node0.c node1.c node2.c node3.c bf.c
gcc -o di di.c
server@server:~/P3_112006234$ ./di
Min cost 0 : 0 2 4 7
Min cost 1 : 2 0 2 5
Min cost 2 : 4 2 0 3
Min cost 3 : 7 5 3 0
server@server:~/P3_112006234$ make clean && make
rm -f bf di
gcc -o bf node0.c node1.c node2.c node3.c bf.c
gcc -o di di.c
server@server:~/P3_112006234$ ./di
Min cost 0 : 0 1 2 4
Min cost 1 : 1 0 1 3
Min cost 2 : 2 1 0 2
Min cost 3 : 4 3 2 0
```

▲ figure 1.1. Screenshot After Implementing Dijkstra's Algorithm

To match the sample output in the PDF, I changed the value in the di_config.txt file:

```
0 1 3 7
1 0 1 999
3 1 0 2
7 999 2 0
```

So that its distance table (which will be represented by table[4][4] in the code) will be:

from \ to node \ node	0	1	2	3
0	0	1	3	7
1	1	0	1	999
2	3	1	0	2
3	7	999	2	0

1.2. Source Code Explanation

The code is based on the pseudocode given in the book which is:

```

1 Initialization:
2  $N' = \{u\}$ 
3 for all nodes  $v$ 
4 if  $v$  is a neighbor of  $u$ 
5 then  $D(v) = c(u,v)$ 
6 else  $D(v) = \text{infinity}$ 
7
8 Loop
9 find  $w$  not in  $N'$  such that  $D(w)$  is a minimum
10 add  $w$  to  $N'$ 
11 update  $D(v)$  for each neighbor  $v$  of  $w$  and not in  $N'$ :
12  $D(v) = \min(D(v), D(w) + c(w,v))$ 
13 /* new cost to  $v$  is either old cost to  $v$  or known
14 least path cost to  $w$  plus cost from  $w$  to  $v$  */
15 until  $N' = N$ 

```

1. Initialization

```
nodes[id].dist[id] = 0;
```

The distance from the source node to itself is set to 0, this corresponds to line 2 of the pseudocode.

For line 3-6:

We should note that, in the init function which will be called in the beginning of the main() function:

- **All other distances have already been initialized to infinity** (999)
- For this current node, no nodes have been visited yet (so `nodes[id].visit[] = 0` for all nodes)

In part of the loop, after the distance table (`table[i][j]`) is populated upon reading the `di_config.txt` file,

`table[min_index][k] != infinity` specifically ensures that k is a neighbor of min_index and if it is, then it updates to the direct path cost **IF** it is a shorter path than before

2. Loop

In line 15 of the pseudocode, the loop in `dijkstra()` runs until all nodes are visited, so the outer loop runs `nodes_num` times, ensuring all nodes are processed:

```

for (i = 0; i < nodes_num; i++) {
    ....
}

```

Then, in line 9, the `min_distance` variable finds `w` with the smallest distance which is unvisited before. Note that `min_index` holds the index of `w`, which is the node to be processed next:

```

for (j = 0; j < nodes_num; j++) {
    if (!nodes[id].visit[j] && nodes[id].dist[j] < min_distance) {
        min_distance = nodes[id].dist[j];
        min_index = j;
    }
}

```

After that, after selecting `w` (`min_index`), it is marked as visited (line 10):

```

nodes[id].visit[min_index] = 1;

```

(Line 11-14) For each neighbor `k` of `w` (`min_index`), the distance is updated:

```

for (k = 0; k < nodes_num; k++) {
    if (!nodes[id].visit[k] && table[min_index][k] != infinity &&
        nodes[id].dist[min_index] + table[min_index][k] < nodes[id].dist[k]) {
        nodes[id].dist[k] = nodes[id].dist[min_index] + table[min_index][k];
    }
}

```

What it does here is:

- Check if `k` is unvisited (`!nodes[id].visit[k]`)
- Check that there is a path (`table[min_index][k] != infinity`)
- Updates $D(v)$ if the new path through `w` is shorter than the previously known path.

Task 2: Distance Vector Routing with Bellman-Ford Algorithm

2.1. Screenshots of Program

```
server@server:~/P3_112006234$ make clean && make
rm -f bf dl
gcc -o bf node0.c node1.c node2.c node3.c bf.c
gcc -o dl dl.c
server@server:~/P3_112006234$ ./bf
Enter LINKCHANGES:0
Enter TRACE:0
Min cost 0 : 0 1 2 4
Min cost 1 : 1 0 1 3
Min cost 2 : 2 1 0 2
Min cost 3 : 4 3 2 0
Total Packet: 152
Total time: 38.417034 s
```

▲ figure 2.1. Screenshot After Implementing Bellman-Ford Algorithm Without Link Cost Change

To match the sample output in the PDF, I changed the value in the bf_config.txt file:

```
0 1 3 7
1 0 1 999
3 1 0 2
7 999 2 0
20
1
```

The first 4 lines represent the link costs (distances) between four nodes in a network. The indices correspond to routers (nodes), and 999 indicates no direct connection between those nodes:

from \ to node \ node	0	1	2	3
0	0	1	3	7
1	1	0	1	999
2	3	1	0	2
3	7	999	2	0

20 and 1 represent changes in the link cost between nodes 0 and 1 at different times during the simulation, where 20 is the new cost after the first link change, and 1 is the cost restored (to the original one), the implementation is shown below:

```

server@server:~/P3_112006234$ make clean && make
rm -f bf dl
gcc -o bf node0.c node1.c node2.c node3.c bf.c
gcc -o dl dl.c
server@server:~/P3_112006234$ ./bf
Enter LINKCHANGES:1
Enter TRACE:0
Min cost 0 : 0 1 2 4
Min cost 1 : 1 0 1 3
Min cost 2 : 2 1 0 2
Min cost 3 : 4 3 2 0

Change cost between Node 0 and Node 1 to 20
Min cost 0 : 0 4 3 5
Min cost 1 : 4 0 1 3
Min cost 2 : 3 1 0 2
Min cost 3 : 5 3 2 0

Change cost between Node 0 and Node 1 to 1
Min cost 0 : 0 1 2 4
Min cost 1 : 1 0 1 3
Min cost 2 : 2 1 0 2
Min cost 3 : 4 3 2 0
Total Packet: 268
Total time: 20017.087891 s

```

▲ figure 2.2. Screenshot After Implementing Bellman-Ford Algorithm With Link Cost Change

For verifying the correctness of routing table updates and making sure the packet transmission is correct, we can set TRACE to 1, which will output the information about packet exchanges before giving the final minimum cost:

```

server@server:~/P3_112006234$ ./bf
Enter LINKCHANGES:1
Enter TRACE:1
MAIN: rcv event, t=0.947, at 3 src: 0, dest: 3, contents: 0 1 3 7
MAIN: rcv event, t=0.992, at 0 src: 1, dest: 0, contents: 1 0 1 999
MAIN: rcv event, t=1.209, at 3 src: 2, dest: 3, contents: 3 1 0 2
MAIN: rcv event, t=1.642, at 2 src: 0, dest: 2, contents: 0 1 3 7
MAIN: rcv event, t=1.871, at 1 src: 0, dest: 1, contents: 0 1 3 7
MAIN: rcv event, t=2.166, at 2 src: 1, dest: 2, contents: 1 0 1 999
MAIN: rcv event, t=2.407, at 0 src: 2, dest: 0, contents: 3 1 0 2
MAIN: rcv event, t=2.421, at 2 src: 3, dest: 2, contents: 7 999 2 0
MAIN: rcv event, t=2.811, at 1 src: 2, dest: 1, contents: 3 1 0 2
MAIN: rcv event, t=2.917, at 1 src: 0, dest: 1, contents: 0 1 3 7
MAIN: rcv event, t=3.082, at 3 src: 0, dest: 3, contents: 0 1 3 7
MAIN: rcv event, t=3.293, at 2 src: 3, dest: 2, contents: 7 8 2 0
MAIN: rcv event, t=3.867, at 1 src: 0, dest: 1, contents: 0 1 2 7
MAIN: rcv event, t=4.063, at 2 src: 3, dest: 2, contents: 7 8 2 0
MAIN: rcv event, t=4.104, at 0 src: 3, dest: 0, contents: 7 999 2 0
MAIN: rcv event, t=4.330, at 0 src: 3, dest: 0, contents: 7 8 2 0
MAIN: rcv event, t=4.518, at 3 src: 0, dest: 3, contents: 0 1 2 7
MAIN: rcv event, t=4.578, at 1 src: 0, dest: 1, contents: 0 1 2 7
MAIN: rcv event, t=4.756, at 1 src: 2, dest: 1, contents: 3 1 0 2
MAIN: rcv event, t=4.945, at 2 src: 3, dest: 2, contents: 7 8 2 0
MAIN: rcv event, t=5.458, at 1 src: 2, dest: 1, contents: 3 1 0 2
MAIN: rcv event, t=5.804, at 2 src: 0, dest: 2, contents: 0 1 3 7
MAIN: rcv event, t=6.163, at 0 src: 3, dest: 0, contents: 7 8 2 0
MAIN: rcv event, t=6.229, at 0 src: 3, dest: 0, contents: 7 8 2 0
MAIN: rcv event, t=6.300, at 3 src: 0, dest: 3, contents: 0 1 2 7
MAIN: rcv event, t=6.423, at 1 src: 2, dest: 1, contents: 3 1 0 2
MAIN: rcv event, t=6.491, at 1 src: 2, dest: 1, contents: 2 1 0 2

```

▲ figure 2.3. Screenshot After Implementing Bellman-Ford Algorithm With Link Cost Change & Trace

2.2. Source Code Explanation

The core functionality of each node is basically:

1. Initializing its distance table and broadcasts its direct costs to neighbors.
2. Recalculating minimum costs upon receiving packets and updates any changes to neighbors
3. (if necessary, particularly for node 0 and 1 in our case) Adjusting for changes in link costs and notifying neighbors.

To look into it with more details:

- Node 0

- **rtinit0()**

- What it does: Initializes node 0's distance table and sends its initial minimum cost to its neighbors
 - Steps:
 - Initializes the distance table (dt0) with high costs (999)
 - Sets the direct costs to itself (dt0.costs[0][0], etc.) based on the input file
 - Updates neighbor0 to indicate which nodes are direct neighbors of node 0 (determined by non-infinite costs)
 - Sends a packet (struct rtpkt) to its direct neighbors containing its minimum cost to all nodes
 - Code:

```
extern void rtinit0()
{
    int i;
    struct rtpkt send;
    send.sourceid = 0;
    default0.sourceid = 0;

    for(i = 0; i < 4; i++){
        send.mincost[i] = dt0.costs[i][i];
        default0.mincost[i] = dt0.costs[i][i];
        if(i != 0) neighbor0[i] = 1;
    }

    for(i = 0; i < 4; i++) {
        if(neighbor0[i]) {
            send.destid = i;
            tolayer2(send);
        }
    }
}
```

- **rtupdate0(struct rtpkt *rcvdpkt)**

- What it does: Updates the distance table when a packet is received and notifies neighbors if the minimum cost changes

■ Steps:

- Identifies the source of the received packet (rcvdpkt->sourceid)
- Checks if any entry in the distance table (dt0.costs) can be updated using the received minimum cost and its own cost to the source
- If updated, recalculates the new minimum cost for each destination
- Sends updated minimum cost to all its neighbors if changes occur

■ Code:

```
extern void rtupdate0(struct rtpkt *rcvdpkt)
{
    int src = rcvdpkt->sourceid;
    int distance = dt0.costs[src][src];
    int i, j;
    int min_cost;

    if(distance == 999) return;
    for(i = 0; i < 4; i++){
        if(dt0.costs[i][src] != rcvdpkt->mincost[i] + distance){
            dt0.costs[i][src] = rcvdpkt->mincost[i] + distance;
            min_cost = dt0.costs[i][i];

            for(j = 0; j < 4; j++){
                min_cost = min_cost > dt0.costs[i][j] ? dt0.costs[i][j] : min_cost;
            }

            default0.mincost[i] = min_cost;

            for(j = 0; j < 4; j++){
                if(neighbor0[j]){
                    default0.destid = j;
                    tolayer2(default0);
                }
            }
        }
    }
}
```

○ linkhandler0(int linkid, int newcost)

- What it does: Handles changes in the cost of the direct link between node 0 and linkid

■ Steps:

- Updates the direct cost in the distance table (dt0.costs[linkid][linkid] = newcost)
- Recalculates the minimum cost and sends updated costs to its neighbors

■ Code:

```
extern void linkhandler0(int linkid, int newcost)
/* called when cost from 0 to linkid changes from current value to newcost */
/* You can leave this routine empty if you're an undergrad. If you want */
/* to use this routine, you'll need to change the value of the LINKCHANGE */
/* constant definition in prga.c from 0 to 1 */
{
    int i;
    default0.mincost[linkid] = newcost;
    dt0.costs[linkid][linkid] = newcost;
    for(i = 0; i < 4; i++){
        if(neighbor0[i]){
            default0.destid = i;
            tolayer2(default0);
        }
    }
}
```


- Node 1

- **rtinit1()**

- What it does: Initializes node 1's distance table and broadcasts initial minimum costs to its neighbors

- Steps:

- Similar to rtinit0(), it initializes the distance table (dt1) and determines its neighbors (neighbor1)
 - Node 1 considers only nodes 0 and 2 as neighbors based on predefined logic
 - Sends its initial minimum cost packet to these neighbors

- Code:

```
extern void rtinit1()
{
    int i;
    struct rtpkt send;
    send.sourceid = 1;
    default1.sourceid = 1;

    for(i = 0; i < 4; i++){
        send.mincost[i] = dt1.costs[i][i];
        default1.mincost[i] = dt1.costs[i][i];
        if(i == 0 || i == 2) neighbor1[i] = 1;
    }

    for(i = 0; i < 4; i++) {
        if(neighbor1[i]) {
            send.destid = i;
            tolayer2(send);
        }
    }
}
```

- **rtupdate1(struct rtpkt *rcvdpkt)**

- What it does: Updates node 1's distance table when it receives a packet and propagates changes if necessary

- Steps:

- Identifies the source node and calculates the distance to it
 - Updates the table if the new cost (from the source) provides a better path
 - If the minimum cost for any destination changes, sends updated minimum cost packets to neighbors

■ Code:

```
extern void rtupdate1(struct rtpkt *rcvpkt)
{
    int src = rcvpkt->sourceid;
    int distance = dt1.costs[src][src];
    int i, j;
    int min_cost;

    if(distance == 999) return;
    for(i = 0; i < 4; i++){
        if(dt1.costs[i][src] != rcvpkt->mincost[i] + distance){
            dt1.costs[i][src] = rcvpkt->mincost[i] + distance;
            min_cost = dt1.costs[i][i];
            for(j = 0; j < 4; j++){
                min_cost = min_cost > dt1.costs[i][j] ? dt1.costs[i][j] : min_cost;
            }
            default1.mincost[i] = min_cost;
            for(j = 0; j < 4; j++){
                if(neighbor1[j]) {
                    default1.destid = j;
                    tolayer2(default1);
                }
            }
        }
    }
}
```

○ linkhandler1(int linkid, int newcost)

- What it does: Adjusts for changes in the cost of a direct link from node 1 to linkid

■ Steps:

- Updates the distance table with the new cost
- Recomputes the minimum costs and broadcasts them to neighbors

■ Code:

```
extern void linkhandler1(int linkid, int newcost)
/* called when cost from 1 to linkid changes from current value to newcost */
/* You can leave this routine empty if you're an undergrad. If you want */
/* to use this routine, you'll need to change the value of the LINKCHANGE */
/* constant definition in prog3.c from 0 to 1 */
{
    int i;
    default1.mincost[linkid] = newcost;
    dt1.costs[linkid][linkid] = newcost;
    for(i = 0; i < 4; i++){
        if(neighbor1[i]) {
            default1.destid = i;
            tolayer2(default1);
        }
    }
}
```

● Node 2

○ rtinit2()

- What it does: Initializes node 2's distance table and communicates initial costs to neighbors

■ Steps:

- Initializes the table (dt2) with costs from the input file
- Determines its neighbors (neighbor2) as all nodes except itself (based on non-infinite costs)
- Sends a packet to all neighbors with the initial costs

■ Code:

```
extern void rtinit2()
{
    int i;
    struct rtpkt send;
    send.sourceid = 2;
    default2.sourceid = 2;

    for(i = 0; i < 4; i++){
        send.mincost[i] = dt2.costs[i][i];
        default2.mincost[i] = dt2.costs[i][i];
        if(i != 2) neighbor2[i] = 1;
    }

    for(i = 0; i < 4; i++) {
        if(neighbor2[i]) {
            send.destid = i;
            tolayer2(send);
        }
    }
}
```

○ rtupdate2(struct rtpkt *rcvdpkt)

- What it does: Updates node 2's distance table upon receiving a packet and shares changes if necessary

■ Steps:

- Checks if the received packet provides a better route
- Updates the table and recalculates the minimum costs for all destinations
- If changes occur, broadcasts new costs to neighbors

■ Code:

```
extern void rtupdate2(struct rtpkt *rcvdpkt)
{
    int src = rcvdpkt->sourceid;
    int distance = dt2.costs[src][src];
    int i, j;
    int min_cost;

    if(distance == 999) return;
    for(i = 0; i < 4; i++){
        if(dt2.costs[i][src] != rcvdpkt->mincost[i] + distance){
            dt2.costs[i][src] = rcvdpkt->mincost[i] + distance;
            min_cost = dt2.costs[i][i];
        }

        for(j = 0; j < 4; j++) {
            min_cost = min_cost > dt2.costs[i][j] ? dt2.costs[i][j] : min_cost;
        }

        default2.mincost[i] = min_cost;

        for(j = 0; j < 4; j++) {
            if(neighbor2[j]) {
                default2.destid = j;
                tolayer2(default2);
            }
        }
    }
}
```

● Node 3

○ rtinit3()

- What it does: Sets up node 3's distance table and sends its costs to its direct neighbors
- Steps:
 - Initializes the table (dt3) with input values
 - Sets up neighbor3 to consider only direct neighbors (nodes 0 and 2 in our case)
 - Sends the minimum cost to its neighbors
- Code:

```
extern void rtinit3()
{
    int i;
    struct rtpkt send;
    send.sourceid = 3;
    default3.sourceid = 3;

    for(i = 0; i < 4; i++){
        send.mincost[i] = dt3.costs[i][i];
        default3.mincost[i] = dt3.costs[i][i];
        if(i == 0 || i == 2) neighbor3[i] = 1;
    }

    for(i = 0; i < 4; i++) {
        if(neighbor3[i]) {
            send.destid = i;
            tolayer2(send);
        }
    }
}
```

- **rtupdate3(struct rtpkt *rcvdpkt)**
 - What it does: Updates node 3's distance table when it receives new cost information
 - Steps:
 - Updates the table if the new route (through the source node) provides a lower cost
 - Recalculates the minimum costs and shares them with neighbors

■ Code:

```
extern void rtupdate3(struct rtpkt *rcvdpkt)
{
    int src = rcvdpkt->sourceid;
    int distance = dt3.costs[src][src];
    int i, j;
    int min_cost;

    if(distance == 999) return;

    for(i = 0; i < 4; i++){
        if(dt3.costs[i][src] != rcvdpkt->mincost[i] + distance){
            dt3.costs[i][src] = rcvdpkt->mincost[i] + distance;
            min_cost = dt3.costs[i][i];

            for(j = 0; j < 4; j++){
                min_cost = min_cost > dt3.costs[i][j] ? dt3.costs[i][j] : min_cost;
            }

            default3.mincost[i] = min_cost;

            for(j = 0; j < 4; j++){
                if(neighbor3[j]){
                    default3.destid = j;
                    tolayer2(default3);
                }
            }
        }
    }
}
```

Together, these functions implement the Bellman-Ford algorithm:

- Routers (represented by nodes in our case) maintain a local distance table representing the known shortest paths
- Routers exchange their distance vectors with neighbors to update their understanding of the network topology
- Changes propagate through the network until all routers reach a consistent view of the shortest paths

Task 3: Comparison of Dijkstra and Bellman-Ford Algorithms

3.1. Question 1

What is the time complexity of Dijkstra's algorithm? Please briefly describe the steps of the algorithm to justify the time complexity.

Steps of the Algorithm:

1. Initialization:
 - a. Assign a tentative distance of infinity (∞) to all nodes except the source, which gets a distance of 0.
 - b. Mark all nodes as unvisited.
2. Main Loop:
 - a. Find the minimum distance node: Search through all unvisited nodes to find the one with the smallest tentative distance. This takes $O(n)$ time in the first iteration, since all n nodes are unvisited.
 - b. Update neighbors: For the selected node, update the tentative distances of its unvisited neighbors if a shorter path is found through the selected node.
 - c. Mark the selected node as visited.
 - d. Repeat until all nodes are visited or the shortest paths to all reachable nodes are determined.

Therefore, in terms of time complexity:

- In the first iteration, we search through all n nodes to find the minimum distance node. This costs $O(n)$
- In the second iteration, one node is already visited, so we search through $n-1$ nodes. This costs $O(n-1)$
- In the third iteration, $n-2$ nodes remain unvisited, and so on.
- The total number of searches over all iterations is:

$$n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n+1)}{2}$$

which simplifies to **$O(n^2)$** as the n^2 term dominates for large n

However, we can minimize this if we use a min-heap/priority queue:

Steps:

1. Initialize the same way as above.
2. Use a priority queue to efficiently find the node with the smallest distance.

3. When a node is processed, update the distances to its neighbors in the priority queue (using $O(\log n)$) operations for insertion/deletion).

Therefore in terms of time complexity:

- Extracting the minimum distance node n times takes $O(n \log n)$.
- Updating distances for all edges collectively takes $O(m \log n)$, where m is the number of edges.
- Therefore, the total complexity is **$O((n+m) \log n)$**

3.2. Question 2

What is the time complexity of Bellman-Ford algorithm? Please briefly describe the steps of the algorithm to justify the time complexity.

Steps of the algorithm:

1. Set the source node's distance to 0 and all other nodes to infinity, which takes $O(V)$, where V is the number of vertices
2. For $V-1$ iterations, traverse all edges and update the distance to the adjacent vertices if a shorter path is found, which requires $(V-1) \times E$, where E is the number of edges. For large V , the -1 term becomes negligible compared to V , so the expression simplifies to $O(V \times E)$
3. Check all edges one more time to verify that no further distance updates are possible, which requires $O(E)$

Therefore, the time complexity is **$O(V \times E)$** since the second step is the most computationally expensive

3.3. Question 3

How does the distance vector routing algorithm send routing packets? (To all nodes or only to neighbors)

The distance vector routing algorithm sends routing packets **only to its directly connected neighbors**. How it works:

1. Each node **periodically sends its routing table** to its neighbors.
 - a. If a change occurs in a node's routing table (due to a link cost change or node failure), the node immediately sends an update to its neighbors
2. The updates contain the node's **cheapest-known** distances to all other nodes in the network. Then after receiving an update:
 - a. Neighbors update their own routing tables if the information is better than their previous routes

- b. These updates then propagate through the network in 'rounds'
3. The process keeps on repeating until all nodes have a consistent view of the best paths

3.4. Question 4

How does the link state routing algorithm send routing packets? (To all nodes or only to neighbors)

The link state routing algorithm sends routing packets **to all nodes** in the network. How it works:

1. Each node **discovers its directly connected neighbors** and **measures the cost of** the links to them. It then creates a **link state advertisement (LSA)** containing this information.
2. The **LSA is sent to all of the node's directly connected neighbors**
 - a. Each neighbor forwards the LSA to their own neighbors, except the one from which they received it
 - b. This process keeps on repeating until the LSA has reached every node in the network
3. Each node **keeps a database of LSAs received** from all nodes. This database represents the network's topology
4. Applying **Dijkstra's algorithm** to the topology database to let each node compute the shortest path to all other nodes

3.5. Question 5

When a link cost changes, which steps does the distance vector algorithm take?

1. A node **detects the change** in the cost of one of its directly connected links (which could happen due to a failure, congestion, or dynamic cost change like the one in our assignment)
2. The node **updates its routing table** to reflect the new cost of the link to the directly connected neighbor
 - a. If the new cost affects the shortest path to any destination, the node updates the cost to those destinations in its table
3. The node **sends its updated routing table entries (or changes) to its directly connected neighbors**
4. **Each neighbor** receiving the update **compares the new information with its current routing table**:

- a. If the new information provides a shorter path to a destination, the neighbor updates its routing table.
 - b. The neighbor then sends its updated routing table to its own neighbors.
5. The process **repeats** as each node propagates updates to its neighbors **until all nodes have consistent routing tables reflecting the new link cost**
6. Note that if a link fails or its cost increases significantly, routing updates may keep on looping, causing the count-to-infinity problem. If this happens, then it needs to be handled with a certain method like poisoned reverse which is explained in the book

