Assignment 3

Tuguldur Tserenbaljir (鐵特德) 109006271

1) Part 1: Link state routing with Dijkstra’s algorithm

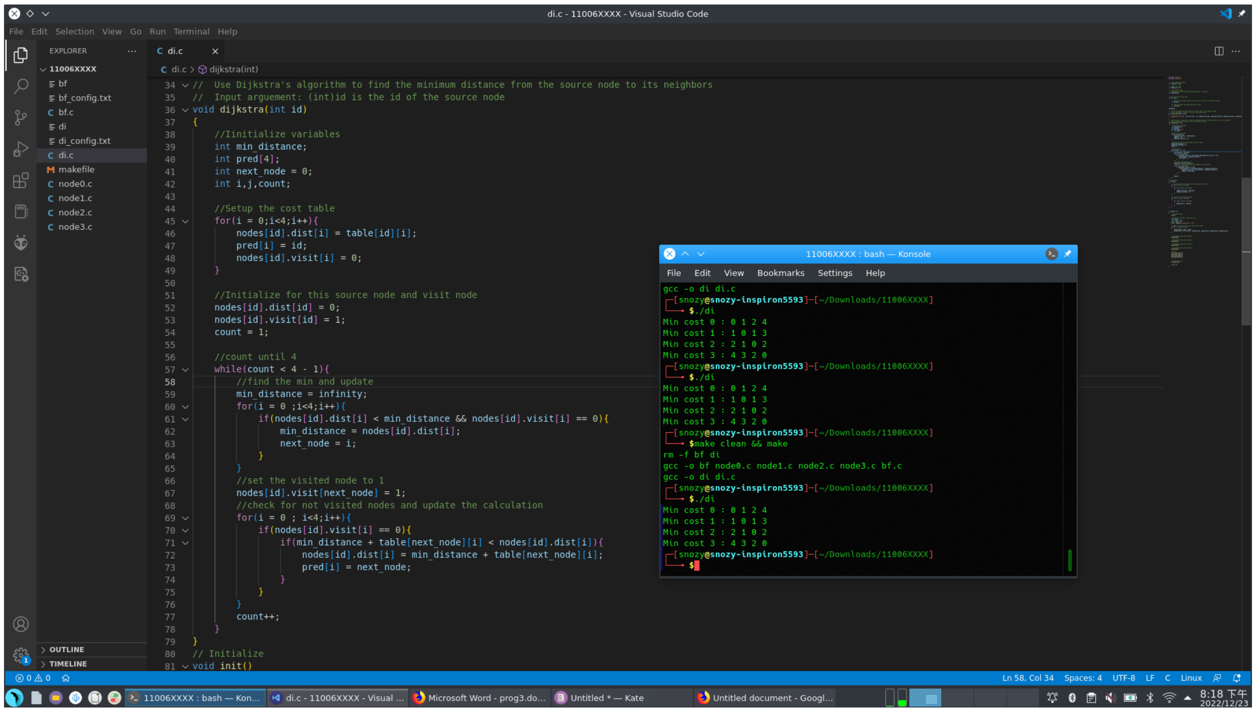
For part 1, the objective was to complete the Dijkstra function in the “di.c”. From previous projects on “Hackthebox Academy”, I had already setup my environment to Parrot OS. So, I used Parrot OS as my environment instead of Ubuntu. Thus, I will explain my implementation down below for Dijkstra's algorithm.

Before starting to program the function, I first reread the Dijkstra algorithm in the book and did some research online (my source is mentioned below). The source node that uses the “void dijkstra” function will be the starting vertex and all the other paths will be assigned infinity values to have an easier time calculating the minimum distance to each other. Then, from the starting node it will go to each neighboring vertex , and update the distance between them. If a new path length is more than the path length of the adjacent neighboring vertex, it will not update it. To avoid updating paths for the already visited paths, I will mark them with 1’s and 0’s (later on in the C implementation). After every count, the program will visit the unvisited vertices with the least path length that are saved as 0’s in the code. The function will repeat until all of the vertices have been visited.

To code first, I set up the variables for saving the minimum distance (**mind_distance**), increments (**i,j,count**), next node (**next_node**), and previous distance (**pred**). Then, in the implementation, set the distance from the source node to its neighbors by the distance in the **table** given to us. Thus, inside the first for loop , I set the distance from source to visiting nodes to **0** and save the value of the distance to the variable **pred**. To finish the initialization, I set for this source node to 0 and to tell the program that it is visiting this node, I set the value of visit to 1.

After initialization, the first for loop in the while loop finds the minimum value by checking if the distance for this source node is smaller than the infinity value that is initially set for the minimum distance while it is not visited yet. If so, we update the minimum value and increment and increment the next node. After, my algorithm will set the distance for the next visiting node to 1 because we already visited from this source node. Thus, the final for loop is to update the distance from this source node to the visiting node by checking. First it will check if we already visited this node then if not it will update the minimum distance.

The condition for the big while loop is 3 because I already know that the max size will be a 4 by 4 matrix.



```c
//  Use Dijkstra's algorithm to find the minimum distance from the source node to its neighbors
//  Input arguement: (int)id is the id of the source node
void dijkstra(int id)
{
    //Iinitialize variables
    int min_distance;
    int pred[4];
    int next_node = 0;
    int i,j,count;

    //Setup the cost table
    for(i = 0;i<4;i++){
        nodes[id].dist[i] = table[id][i];
        pred[i] = id;
        nodes[id].visit[i] = 0;
    }

    //Initialize for this source node and visit node
    nodes[id].dist[id] = 0;
    nodes[id].visit[id] = 1;
    count = 1;

    //count until 4
    while(count < 4 - 1){
        //find the min and update
        min_distance = infinity;
        for(i = 0 ;i<4;i++){
            if(nodes[id].dist[i] < min_distance && nodes[id].visit[i] == 0){
                min_distance = nodes[id].dist[i];
                next_node = i;
            }
        }
        //set the visited node to 1
        nodes[id].visit[next_node] = 1;
        //check for not visited nodes and update the calculation
        for(i = 0 ; i<4;i++){
            if(nodes[id].visit[i] == 0){
                if(min_distance + table[next_node][i] < nodes[id].dist[i]){
                    nodes[id].dist[i] = min_distance + table[next_node][i];
                    pred[i] = next_node;
                }
            }
        }
        count++;
    }
}
// Initialize
void init()
```

```
gcc -o di di.c
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $./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
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $./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
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $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
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $./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
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $
```

(Figure 1.1 Screenshot of my result)

2) Part 2: Distance vector routing with Bellman-Ford algorithm

For part 2 was to fill in the function for each node. All of the nodes are similar to each other, so will only explain the first node which is "node.c" functions. There are two functions in the "node0.c" file which are "**rtinit0()**" and "**rtupdate0()**".

For the first function, the purpose is to send minimum cost paths to all other network nodes. First, I update the costs of each network with respect to the topology given to us in the PDF file. (Example:

```
 dt0.costs[0][0] = 0;
 dt0.costs[1][0] = 1;
 dt0.costs[1][1] = 1;
///...
```

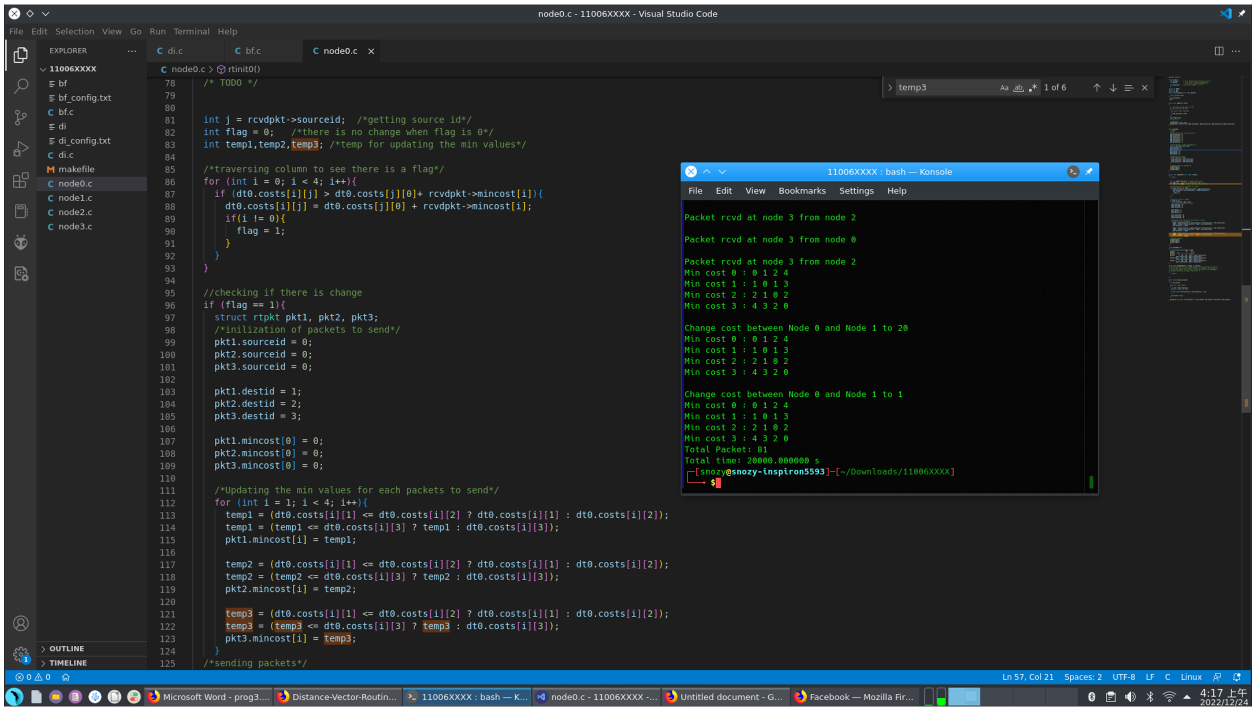
)

Then make packets to send (For this node, there is packet1,packet2,packet3) with respect to our design and set their destination (Example: packet 1 destination is 1 and packet 2 destination is 2). Next, updated the costs of those packets to send to each destination in a for loop. Finally, sent using the **tolayer2()** function mentioned in the Appendix.

The second function to implement is **rtupdate0().** First we get the received packet source id and store it in a variable **j** to check if the matrix is inverted and to see if the algorithm needs to update it. I set a variable called flag to see if it is inverted or not. Then later on, check if it is inverted or not with a if statement. If it is not inverted the function ends. Otherwise, my algorithm creates new packets to send. To update their value and cost, I check them in a for loop and update their minimum value. Finally, send them using the **tolayer2()** function.

Similarly, all the other note c program files have the same implementation with the one I mentioned above.

Since I am an undergraduate I left the linkhandler function empty.



```c
/* TODO */


int j = rcvdpkt->sourceid;  /*getting source id*/
int flag = 0;   /*there is no change when flag is 0*/
int temp1,temp2,temp3; /*temp for updating the min values*/

/*traversing column to see there is a flag*/
for (int i = 0; i < 4; i++){
  if (dt0.costs[i][j] > dt0.costs[j][0]+ rcvdpkt->mincost[i]){
    dt0.costs[i][j] = dt0.costs[j][0] + rcvdpkt->mincost[i];
    if(i != 0){
      flag = 1;
    }
  }
}

//checking if there is change
if (flag == 1){
  struct rtpkt pkt1, pkt2, pkt3;
  /*inilization of packets to send*/
  pkt1.sourceid = 0;
  pkt2.sourceid = 0;
  pkt3.sourceid = 0;

  pkt1.destid = 1;
  pkt2.destid = 2;
  pkt3.destid = 3;

  pkt1.mincost[0] = 0;
  pkt2.mincost[0] = 0;
  pkt3.mincost[0] = 0;

  /*Updating the min values for each packets to send*/
  for (int i = 1; i < 4; i++){
    temp1 = (dt0.costs[i][1] <= dt0.costs[i][2] ? dt0.costs[i][1] : dt0.costs[i][2]);
    temp1 = (temp1 <= dt0.costs[i][3] ? temp1 : dt0.costs[i][3]);
    pkt1.mincost[i] = temp1;

    temp2 = (dt0.costs[i][1] <= dt0.costs[i][2] ? dt0.costs[i][1] : dt0.costs[i][2]);
    temp2 = (temp2 <= dt0.costs[i][3] ? temp2 : dt0.costs[i][3]);
    pkt2.mincost[i] = temp2;

    temp3 = (dt0.costs[i][1] <= dt0.costs[i][2] ? dt0.costs[i][1] : dt0.costs[i][2]);
    temp3 = (temp3 <= dt0.costs[i][3] ? temp3 : dt0.costs[i][3]);
    pkt3.mincost[i] = temp3;
  }
/*sending packets*/
```

```
Packet rcvd at node 3 from node 2

Packet rcvd at node 3 from node 0

Packet rcvd at node 3 from node 2
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 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 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: 81
Total time: 20000.000000 s
┌─[snozy@snozy-inspiron5593]─[~/Downloads/11006XXXX]
└──╼ $
```

3) **Part 3: Comparison of Dijkstra and Bellman-Ford algorithms**

Question 1: What is the time complexity of Dijkstra's algorithm?

Using the pseudocode for the Dijkstra algorithm (Figure 3.1). Shown in the figure, the first while loop is for visiting all the vertices and the next for loop is for each vertex from the while loop it's neighbor vertex path processing and updating the path, if found any. The time complexity of the Dijstra's algorithm is

O(V)*O(V) = O(V^2).

V is the number of vertices.

Time for visiting all of the vertices is O(V).

Time for vertex processing is O(V).

In all it is O(V)*O(V) = O(V^2).

However, we can reduce the time complexity for the Dijstra's algorithm to O((V+E) * logV). We can use list representation of the graph and a min-heap to store all of the unvisited vertices.

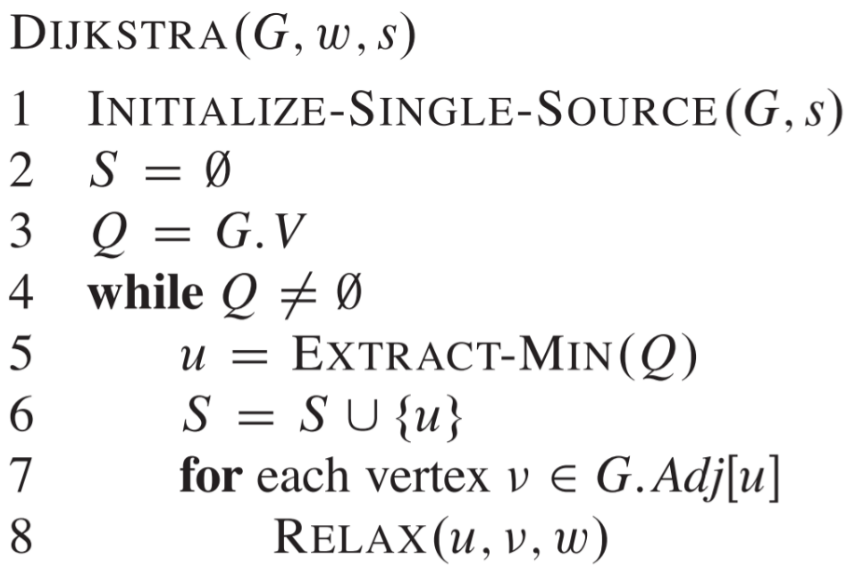
V is the number of vertices.

E is the number of edges.

Time for visiting all of the vertices is O(V+E).

Time for vertex processing is O(logV).

In all it is O(V+E) * O(logV) = O((V+E) * logV).



```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

(**Figure 3.1.1 Dijkstra pseudocode**)

Question 2:What is the time complexity of Bellman-Ford algorithm?

Where V is the number of vertices and E is the number of edges.

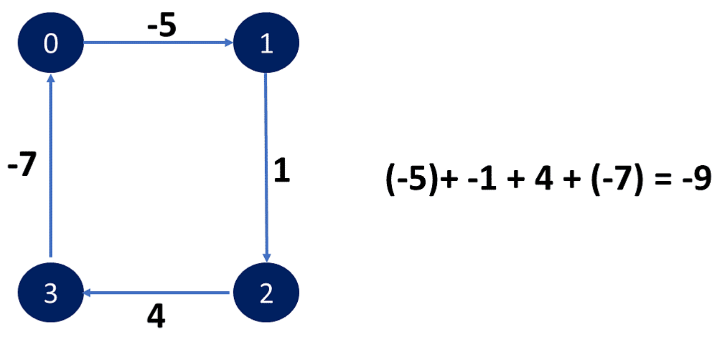
Worst case **(Figure 3.2.1)** happens when the algorithm encounters a negative cycle in the graph.

Average case **(Figure 3.2.2)** happens when the algorithm stops when there are no changes made to the path values (meaning fewer iterations).

Best case happens **(Figure 3.2.3)** when the algorithm if the edges occur from left to right where the algorithm needs to do one job to find the shortest path.

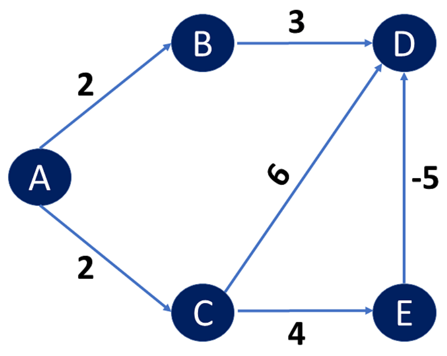
Time complexity for Bellman-Ford algorithm:

Worst case - O(V^3)



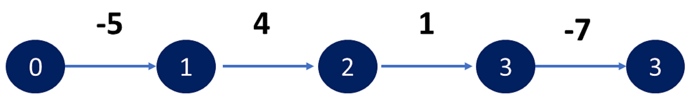
(Figure 3.2.1 Worst Case)

Average case - O(V * E)

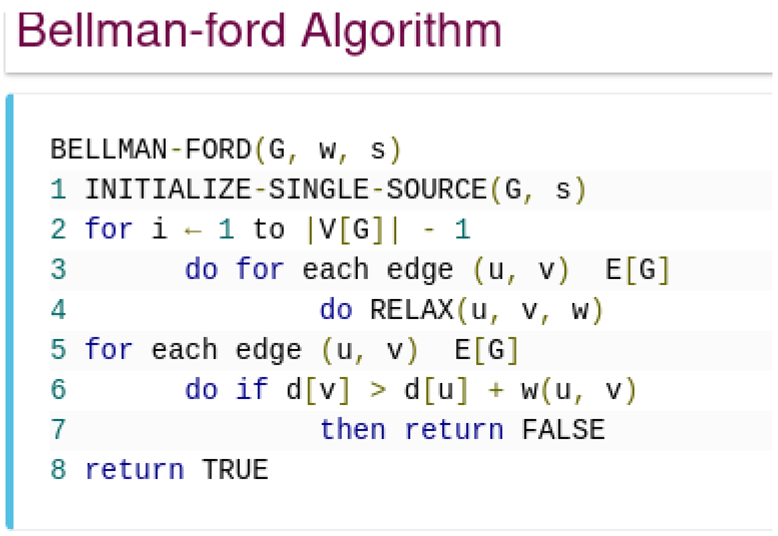


(Figure 3.2.2 Average Case)

Best case - O(E)



(Figure 3.2.3 Best Case)



Bellman-ford Algorithm

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i ← 1 to |V[G]| - 1
      do for each edge (u, v)  E[G]
            do RELAX(u, v, w)
for each edge (u, v)  E[G]
      do if d[v] > d[u] + w(u, v)
            then return FALSE
return TRUE
```

(**Figure 3.2.4 Bellman-Ford Pseudocode**)

Question 3: How does the distance vector routing algorithm send routing packets?

The distance vector routing algorithm uses Bellman-Ford's algorithm to send routing packets. From the source to the preferred destination, every route will share its routing table with their neighbor where the distance and routers in the network are known. After all of the information in the table is updated it is ready for the next step. From the source to the preferred destination, hops to the neighbors by looking at the distance vector table, sharing the packet and shortest path to the destination .Then the neighbors forward the shortest path to the destination and the packet to its neighbor until it reaches the preferred destination. With the help of Bellman-Ford's algorithm, each node sends its own distance vector estimate to neighbors and when they receive the distance vector they update their own.

Question 4: How does the link state routing algorithm send routing packets?

The link state routing algorithm uses Dijstra's algorithm to send routing packets. The link state will have a complete map of the entire network where the map from the source to the preferred destination is included. First, the router looks up the destination network from the map and calculates the shortest distance with the Dijstra's algorithm. Then, the router shares the packet to the next hop alongside the shortest distance until it reaches the preferred destination.

Question 5: When a link cost changes, which steps does the distance vector algorithm take?

When a link cost changes, the distance vector takes 4 steps. First, the updated route will send information to its neighbors about the cost change. Then, each neighbor will update their own distance vector table respect to the source. Next, it checks if to change the path. If so, the neighbor will send information about the change. Finally, this process will go for all the routers until they get the new distance vector table with the new link changes.

4) Problems I faced

-Understanding the difference between Dijstra's and Bellman-Ford's algorithms.

-Coding the nodes for part 2 was challenging.

-Tried to do linkhandler function, but was not able to.