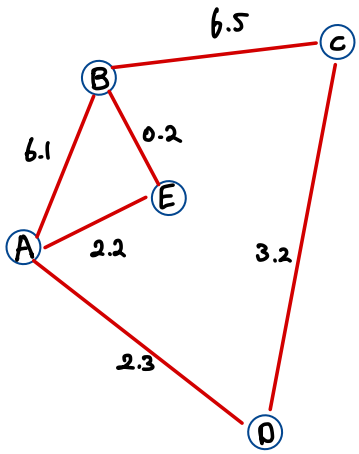
1. Test results:

Here is the topology that I have tested with. It has five nodes. This gives a brief insight into how the algorithm is implemented and how it works. My algorithm also works well with the net topology that was also given in the spec sheet as well as the numerous test that I have conducted with different test cases. I chose to use this specific model since it's very intuitive and easy to understand than complex topology. (Again my code works well with the topology that has more than 10 nodes too)




```
❯ python3 COMP3221_DiVR.py A 6000 config/Aconfig.txt
I am node A
Least cost path from A to B: AB: 6.1
Least cost path from A to D: AD: 2.3
Least cost path from A to E: AE: 2.2
Least cost path from A to C: ABC: 12.6

I am node A
Least cost path from A to B: AEB: 2.4000000000000004
Least cost path from A to D: AD: 2.3
Least cost path from A to E: AE: 2.2
Least cost path from A to C: ADC: 5.5
```

```
❯ python3 COMP3221_DiVR.py B 6001 config/Bconfig.txt
I am node B
Least cost path from B to A: BA: 6.1
Least cost path from B to C: BC: 6.5
Least cost path from B to E: BE: 0.2
Least cost path from B to D: BAD: 8.399999999999999

I am node B
Least cost path from B to A: BEA: 2.4000000000000004
Least cost path from B to C: BC: 6.5
Least cost path from B to E: BE: 0.2
Least cost path from B to D: BEAD: 4.7
```

```
❯ python3 COMP3221_DiVR.py C 6002 config/Cconfig.txt.txt
I am node C
Least cost path from C to B: CB: 6.5
Least cost path from C to D: CD: 3.2
Least cost path from C to A: CBA: 12.6
Least cost path from C to E: CBE: 6.7

I am node C
Least cost path from C to B: CB: 6.5
Least cost path from C to D: CD: 3.2
Least cost path from C to A: CDA: 5.5
Least cost path from C to E: CBE: 6.7
```

```
❯ python3 COMP3221_DiVR.py D 6003 config/Dconfig.txt.txt
I am node D
Least cost path from D to A: DA: 2.3
Least cost path from D to C: DC: 3.2
Least cost path from D to B: DAB: 8.399999999999999
Least cost path from D to E: DAE: 4.5

I am node D
Least cost path from D to A: DA: 2.3
Least cost path from D to C: DC: 3.2
Least cost path from D to B: DAEB: 4.7
Least cost path from D to E: DAE: 4.5
```

```
❯ python3 COMP3221_DiVR.py E 6004 config/Econfig.txt.txt
I am node E
Least cost path from E to A: EA: 2.2
Least cost path from E to B: EB: 0.2
Least cost path from E to C: EBC: 6.7
Least cost path from E to D: EAD: 4.5
```

The image at the left is the net topology for the mini test that I have conducted. Four images at the right are the result of printing the least-cost path from each different node. It prints out the result whenever there is a change to the routing table. Each node will be communicating with each other every desired frequency denoted as global variable FREQUENCY.

2. General implementation idea

First, the program will receive arguments from the user same way as the spec sheet. The first argument is the node id, the second is the port number, and finally, the file location of its neighboring nodes. Then, it will initialize each node such that it has its port number, routing table, list of neighboring nodes, and other class instances that we need when implementing the algorithm. After the initialization, it will create three different threads. Each for listening, sending, and printing the routing table.

The listening thread will detect a connection from its neighboring nodes and create a thread for each of their connection. Then it will update the routing table using the route() function which is implemented in class Node. Since there are multiple threads, I used mutex lock to guard the critical section which is the 'routing table'. Here is an example. Let's say Node A opens listening threads for B, E, and D. If Node B sends its

routing table to Node A, it will lock Node'A routing table such that node E and D cannot modify Node A's routing table.

Sending thread will send its routing table information to its neighbors. It will send its information in every desired frequency which is a global variable named FREQUENCY. When sending its information it will create a thread for each neighboring node. For Node C it will create threads for B and D and send its routing table in every desired second. It will send its routing table by converting it into JSON format and encoding. This made the whole process of sending and receiving easy since it's just sending the actual routing table without having to go through a heavy string modification.

Lastly, the printing thread will print the routing table when there is a change to the routing table. When there is no change it even though nodes are still communicating it will not print out its routing table.

3. Routing algorithm:

I used the bellman-ford algorithm to calculate the routing table for each node. The algorithm is implemented inside the class Node with the name 'route()'. The route function will be called only if the node receives the routing table from its neighbors. For example, the function "route()" will be called for Node A when B, D, E sends its routing table to node A. A will listen from these neighboring nodes from a separate thread. Therefore it was necessary to use a lock since it's changing the same routing table of Node A.

When node A receives another node's routing table via listen() method, it will compare their routing table if there is an alternate path cost that leads to lower path cost than the previous path. If so it will update its own routing table. Also, this will be marked as a change which will be printed later on by printing thread. (One of the three initial thread that was created)

4. Problem encounters

There were multiple problems that I have encountered during implementing this assignment.
First, it's when multiple threads access the same area of memory and tries to read and write at the same time. Again as I have already mentioned I have solved this by using mutex locks to lock the critical section.

Another problem that I have encountered is when there is a connection lost from the node. This was a pretty easy fix since you can create another external while loop where it can catch the error and adopt the routing table when the error has risen. Since I have used the TCP protocol to communicate between different threads it was easy to notice when some of the nodes were not available.