Spring 2022 - CSEE W4119 Computer Networks
Programming Assignment 2 - Routing Protocols Emulation

Prof. Gil Zussman

Due: 04/29/2022, 11:59pm

1 Introduction

In this assignment, you will emulate the operation of network layer protocols in a small computer network. The program you write should behave like a single node in the network. You will start several nodes (instances of your program) so that they can send packets to each other as if there are links between them. Running as an independent node, your program should implement a simple version of a Distance-Vector routing protocol as well as a Link State routing protocol.

Section 2 focuses on the implementation of a Distance Vector (DV) routing protocol, and Section 3 focuses on the implementation of a Link State (LS) (Dijkstra-based) routing protocol.

This assignment is meant to be done on your own; please do not collaborate with others as this breaches academic honesty policies and will be taken very seriously.
Please start early, and read the entire assignment! Also ensure that your code works on the Google Cloud VM!
Good luck, and have fun!

2 Distance-Vector Routing Algorithm (50pt)

2.1 Description

The objective of this part of the assignment is to implement a simplified version of a distance vector routing protocol. You are required to write a program which builds its routing table based on the distances (i.e., edge costs) to other nodes in the network. The Bellman-Ford algorithm should be used to build and update the routing tables. UDP should be used to exchange the distance vectors among the nodes in the network. The program should support changing edge costs (see Section 2.8 and should support a mechanism to cope with the count to infinity problem (see Section 2.9).

2.2 Network Model

You may assume that that all the nodes (instances of your program) run on the same machine and have the same IP address. Each node can be identified uniquely by a (UDP listening) port number, which is specified by the user. The port numbers must be > 1024. The maximum number of nodes to support is 16. The links among the nodes in the network and the costs (non-negative integer) between two directly connected nodes are specified by the user upon the activation of the program. You can assume that the user-entered link distances will be the same for either direction (e.g. between node A and B, $\text{Distance}_{A\to B} = \text{Distance}_{B\to A}$.)

2.3 Protocol

We will use UDP to exchange the routing information among the nodes. Each node will send its most recent distance vector (list of nodes and distances to them) by building a UDP packet. The destination port is the listening port of the node to which the packet is being sent. The source port is an arbitrary UDP port the node is using to send the packet. The data field of a packet should include 1) the sending node UDP listening port number (to identify the sending node to its neighbor) and 2) the most recent distance vector. Upon receipt of distance vectors from its neighbors, a node should update its own distance vector and routing table (includes next hop for each destination).

2.4 Routing Information Exchange

You are free to design the format and structure of the routing table kept locally by each node and the distance vector exchanged among neighboring nodes.

1. Upon the activation of the program, each node should construct the initial routing table and distance vector from the command line information and keep it locally.
2. Once the link and distance information for all the nodes are specified, the distance vector information will be exchanged among network neighbors. Each node should send its distance vector information to its neighbors *at least once* (see the command syntax section for more detail on how the process starts). This is the base case before there are any table updates.
3. Using the Bellman-Ford algorithm, each node will keep updating its distance vector and routing table as long as neighboring nodes send their updated distance vectors.
4. If there is any change in the distance vector, a node should send the updated information to its neighbors.

Due to the nature of UDP, packets may be lost or delivered out of order to the application. Therefore, you may consider adding some kind of *time stamp* or *sequence* information to each packet (i.e., each distance vector) so that each node can update its own routing table accordingly. Note that if this process is implemented properly, the routing information exchange should converge (stop) as soon as all nodes in the network obtain the routing tables with the shortest paths information.

2.5 Command Syntax

The program name should be `routenode dv`. **The program can be run in 2 modes: regular and Poisoned Reverse.** The user can specify argument `r` for regular mode and `p` for Poisoned Reverse (more on Poisoned Reverse mode in section 2.9). For each node in the network, the command syntax is:

```
$ routenode dv <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ...
              [last][<cost-change>]
```

`routenode`	Program name.
`dv`	run the distance vector algorithm
`r/p`	Mode of program - regular or Poisoned Reverse.
`update-interval`	Not used (will be used in the LS section) - can be any value.
`local-port`	The UDP listening port number (1024-65534) of the node.
`neighbor#-port`	The UDP listening port number (1024-65534) of one of the neighboring nodes.
`cost-#`	This will be used as the link distance to the `neighbor#-port`. It is an integer that represents the cost/weight of the link.
`last`	Indication of the last node information of the network. The program should understand this arg as optional (hence represented in brackets above). Upon the input of the command with this argument, the routing message exchanges among the nodes should kick in.
`cost-change`	Indication of the new cost that will be assigned to the link between the last node and its neighbor with the highest port number (see Section 2.8). The program should understand this arg as optional (hence represented in brackets above).
`ctrl+C (exit)`	Use ctrl+C to exit the program.

2.6 Status Messages

The following status messages should be displayed at least for the following events. The example of the messages are also shown below.

1. Distance Vector message sent:
   ```
   [<timestamp>] Message sent from Node <port-xxxx> to Node <port-vvvv>
   ```
2. Distance Vector message received:
   ```
   [<timestamp>] Message received at Node <port-vvvv> from Node <port-xxxx>
   ```
3. Routing table (computed every time after a message is received and for the initial routing table):
   ```
   [<timestamp>] Node <port-xxxx> Routing Table
   - (<distance>) -> Node <port-yyyy>
   - (<distance>) -> Node <port-zzzz> ; Next hop -> Node <port-yyyy>
   - (<distance>) -> Node <port-vvvv>
   - (<distance>) -> Node <port-wwww> ; Next hop -> Node <port-vvvv>
   ⋮
   ```

Notice that while distances are exchanged between nodes as part of the distance vector, next-hop information is not exchanged between the nodes.

2.7 Example

The example below illustrates the command inputs, message exchanges and the computation of the routing table at each node in the regular mode, given the network configuration described in Figure 1.

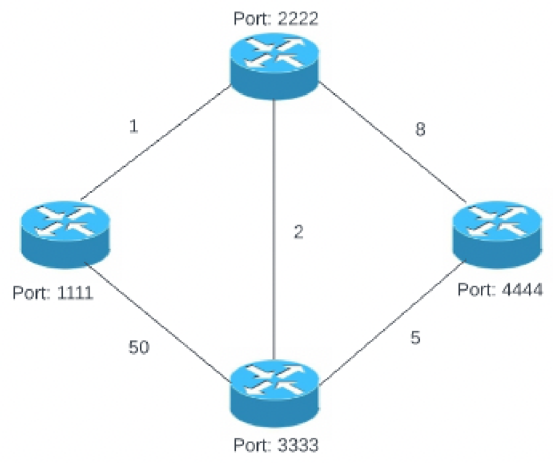

Figure 1: Network Configuration Example

The command to start a program is:

C
```
$ ./routenode dv <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ...
              [last][cost-change]
```
Java
```
$ java routenode dv <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ...
              [last][cost-change]
```
Python
```
$ python3 routenode.py dv <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ..
              [last][cost-change]
```

From the command line prompts type:

```
$ ./routenode dv r [any-num] 1111 2222 1 3333 50
$ ./routenode dv r [any-num] 2222 1111 1 3333 2 4444 8
$ ./routenode dv r [any-num] 3333 1111 50 2222 2 4444 5
$ ./routenode dv r [any-num] 4444 2222 8 3333 5 last
```

As soon as the command for node 4444 with keyword `last` is entered, the following process should kick in.

1. Node `4444` sends its routing information to node `2222` and `3333`.
2. Node `2222` and `3333` update their routing tables according to the routing information sent from node `4444`. Since both node `2222` and `3333` have not sent their routing information to anyone, node `2222` sends its routing information to node `1111`, `3333` and `4444`. Node `3333` will send its routing information to node `1111`, `2222` and `4444`.
3. Node `1111` should update its routing table according to the message received either from node `2222` or `3333` (whichever came first). Node `1111` will send its updated routing information to node `2222` and `3333`. At this point, all nodes have sent their routing information to their neighbors at least once.
4. Each node will update its routing table according to the messages received from its neighbors. However, it will send messages only if its routing table is updated.
5. The process converges as each node has the most updated routing table which includes the shortest distances to all the nodes in the network and the next hop on the shortest path to each node.

Once the algorithm converges, there should be no more status messages. The last status message from the nodes will be something like what is shown below (note: each node will print its own table and it should be ordered in increasing order of port numbers):

```
[1353035852.173] Node 1111 Routing Table
- (1) -> Node 2222
- (3) -> Node 3333; Next hop -> Node 2222
- (8) -> Node 4444; Next hop -> Node 2222

[1353035852.192] Node 2222 Routing Table
...

[1353035852.239] Node 3333 Routing Table
...

[1353035852.287] Node 4444 Routing Table
- (8) -> Node 1111; Next hop -> Node 3333
- (7) -> Node 2222; Next hop -> Node 3333
- (5) -> Node 3333
```

2.8 Triggered Link Cost Change

To test the convergence of the protocol in **both modes (regular and Poisoned Reverse, discussed below)**, the program at the "last" node (node who received input value `[last]` will trigger a link cost change after 30 seconds. Specifically, it should set a timer for 30 seconds when it is initiated. When the timer expires, the program should change the cost of the link associated with the neighbor with highest port number to `<cost-change>` (which is an input argument at that node). As outlined below, this update may trigger distance vector exchanges between some of the nodes.

As an additional status message, the neighbor and the new cost should be printed.

```
[<timestamp>] Node <port-xxxx> cost updated to <value>
```

A special control message which is **not part of the routing protocol** should be sent to the neighbor (with the `<cost-change>` value) and that neighbor should also increase the link cost back to the "last" node to `<cost-change>` and print a message indicating the update. Status messages indicating that this special exchange took place should be printed in both nodes, as outline below.

Link value change message sent:

```
[<timestamp>] Link value message sent from Node <port-xxxx> to Node <port-vvvv>
```

Link value change message received:

```
[<timestamp>] Link value message received at Node <port-vvvv> from Node <port-xxxx>
```

After sending (or receiving) the messages above, both nodes should update their routing tables (if needed). If the link cost change results in a change in the distance vector, the distance vector should be sent to neighbors and may trigger other updates.

2.9 Poisoned Reverse

Distance vector routing can result in the count to infinity problem and routing loops in cases of link failures/cost changes (see Section "Distance-Vector Algorithm: Link-Cost Changes and Link Failures" in the textbook, p. 394). In this section, we will focus on another mode of program - Poisoned Reverse, wherein if node z routes through node y to get to destination x, then z will advertise to y that its distance to x is infinity. Notice that this is not the actual cost from z to x. In all the above examples (in the "regular" mode), Poisoned Reverse is assumed to be disabled.

When the program runs in Poisoned Reverse mode, the command syntax is:

```
$ routenode dv p <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ... [last][<cost-change>]
```

Except for the Poisoned Reverse functionality, the requirements for routing information exchange are similar to those described above. All the status messages should still be provided.

In addition, when a node advertises a distance of infinity to a neighbor, along with status message 1 (see Section 2.6) an additional message should be printed of the format:

```
[<timestamp>] Message sent from Node <port-zzzz> to Node <port-yyyy> with distance to Node <port-xxxx> as inf.
```

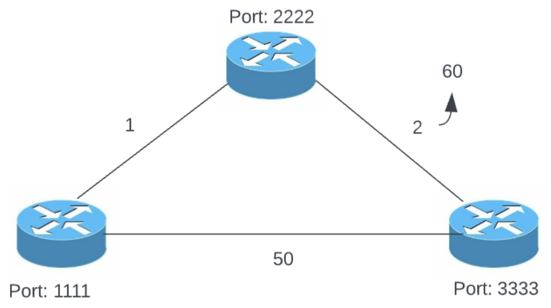

Figure 2: Network Configuration Example with Link Cost Change (corresponding to the example in the textbook)

As an example, consider the network topology in Figure 2, for which below commands would be entered:

```
$ ./routenode dv <r,p> [any-num] 1111 2222 1 3333 50
$ ./routenode dv <r,p> [any-num] 2222 1111 1 3333 2
$ ./routenode dv <r,p> [any-num] 3333 1111 50 2222 2 last 60
```

After the timer expires, the link cost associated with the link connecting Nodes 3333 and 2222 will be updated to 60. In case of regular mode, this cost update would result in a "count-to-infinity" and routing loop between Node 1111 and Node 2222 until Node 1111 computes minimum cost through Node 2222 which exceeds 50. However, in case of Poisoned Reverse mode, Node 1111 will poison the reverse path from Node 3333 by informing Node 2222 that the cost from Node 1111 to Node 3333 is inf. In this case, the routing tables should quickly converge without count-to-infinity taking place.

The final status messages in both modes are below (the convergence to these should be with a smaller number of iterations with Poisoned Reverse):

```
[1353035852.173] Node 1111 Routing Table
- (1) -> Node 2222
- (50) -> Node 3333


[1353035852.192] Node 2222 Routing Table
- (1) -> Node 1111
- (51) -> Node 3333

[1353035852.239] Node 3333 Routing Table
- (50) -> Node 1111
- (51) -> Node 2222
```

2.10 Cases

Your program should deal with the following cases:

Distance Vector	Routing tables and their updates should follow DV algorithms	(15pt)
DV Updates	DV updates should be sent out to all neighbors once the routing table is changed	(10pt)
Convergence	Routing tables should eventually converge (both initially and after the link cost change) and all output should cease	(15pt)
Poisoned Reverse	Link cost change that results in count to infinity under the regular mode should not result in count to infinity under Poisoned Reverse	(10pt)

3 Link State Routing Protocol (50pt)

3.1 Description

The objective of this part is to implement a simplified version of a Link State routing protocol. The program running at each node should construct the network topology based on link state advertisements that are flooded in the network. Based on the topology, each node should construct the routing table via computation of a least-cost path to all other nodes in the network. The program should be able to support changing edge costs

3.2 Network Model

The Network model for the Link-State Protocol should follow the same format and structure defined in section 2.2. Similarly to Section 2.2 when a node is initially started, it is only aware of the costs to its directly connected neighbors. At that point, the node does not have the knowledge about the entire network topology.

3.3 Protocol

We will use UDP to exchange the link-state advertisements among the nodes. The formal definition of the link-state advertisement (LSA) is a packet generated by a router that lists its neighbors and the distances to them. Each node will periodically (and when link costs change) send its LSA to its neighbors in a UDP packet and their neighbors will flood it throughout the network. As in Section 2.2, for every packet sent the destination port is the listening port of the node to which the packet is being sent and the source port is an arbitrary UDP port the node uses to send the packet. The data field of a link-state packet should include (i) the sending node UDP listening port number (to identify the sending node to its neighbor), (ii) the most recent LSA (list of neighbors and costs of the links to them), and (iii) uniquely assigned sequence number for each packet (see section 3.5).

3.4 Routing Information Exchange

You are free to use similar table formats and structures to the ones that you implemented for section 2.

1. Upon the activation of the program, each node should construct the initial Link State table from command line info, and keep it locally. Based on that, each node should also generate its own LSA packet.
2. Once the link and distance information for all the nodes are specified (indicated by the `[last]` parameter), the last node (the node with `[last]` parameter) should be the first node to start sending LSAs to all direct neighbours.
3. Once the LSA is received by a neighbor, that node should update its network topology table and forward the LSA packet to its own neighbors (except the node from which it received this LSA). The flooding mechanism will ensure that each LSA packet is propagated through the entire network (for more details, see section 3.5). It will also ensure that the first LSA sent by the “last” node will activate all the nodes (see below).
4. The receipt of the first LSA is a signal for ‘node activation’. Namely, once a node receives the first LSA, it should start broadcasting its own LSA to its neighbors.
5. Following sending out the first LSA, a node should broadcast an LSA periodically. The length of the period should be `UPDATE_INTERVAL` (a command line input) seconds, to which a random variable between 0 and 1 seconds should be added at every node (to avoid all nodes sending LSAs at the same time). An LSA should also be sent by the node if a link’s cost changes (see Section 3.9 below).
6. Once a node receives LSAs from all other nodes, it can infer the network topology and then compute the routing table. The routing table should be based on using Dijkstra’s algorithm to compute the least-cost paths to all the other nodes. With every receipt of new LSA, a node should check if the network topology should be changed.

7. Each node should wait for a `ROUTING_INTERVAL` (the default value is 30 seconds) since start-up and then execute Dijkstra's algorithm. After that, the Dijkstra algorithm should be invoked every time the topology table changes.

   Note: `ROUTING_INTERVAL` is not a command line input. Make sure your variable name matches exactly, as we may change it during grading.

3.5 Flooding Mechanism - PI Algorithm

Under a naïve broadcast strategy, each node retransmits every LSA packet that it receives, resulting in unnecessary broadcasts and increased overhead. To avoid such a situation, we will use the PI protocol discussed in the class.

For example, consider the example topology shown in Figure 1. The LSA packet created by node `4444` will be sent to its direct neighbors node `2222` and `3333`. Each of these two nodes will in turn broadcast this LSA packet to their own neighbors. Consider node `2222`, which broadcasts node `4444`'s LSA packet to Node `3333` and `1111`. Note that node `3333` has already been informed of node `4444`'s LSA packet once (when it received a broadcast message directly from node `4444`). Node `3333` has now received this same LSA packet via Node `2222`. Under the PI protocol, there should be no need for Node `3333` to broadcast this packet again.

With this example and class note, you should have enough information to implement the PI protocol that will support the LSA flooding. In order to adapt the PI protocol to the current setting, (i) for every node, each new LSA packet should have a unique sequence number (sequence number should be increasing). (ii) `(Node port,seq)` pair should be used by a node receiving an LSA packet to identify duplicate LSA packets (or old and irrelevant LSA packets, with sequence number lower than a packet recently received). **You have to describe your method of implementation in your README file.**

3.6 Command Syntax

The program name should be `routenode ls`. **The program should only run in 1 mode: regular** and should handle link cost change without additional functions. To be consistent with the input command from Section 2, you should use the same parameters as in section 2.5: `r` for regular mode and `p` is not used in this case. For each node in the network, the command syntax is:

```
$ routenode ls <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ...
       [last][<cost-change>]
```

`routenode`	Program name.
`ls`	Run the link state algorithm
`r/p`	Mode of program - regular.
`update-interval`	Node LSA broadcast period (1-5) seconds.
`local-port`	The UDP listening port number (1024-65534) of the node.
`neighbor#-port`	The UDP listening port number (1024-65534) of one of the neighboring nodes.
`cost-#`	This will be used as the link distance to the `neighbor#-port`. It is an integer that and represents the cost/weight of the link.
`last`	Indication of the last node information of the network. The program should understand this arg as optional (hence represented in brackets above). Upon the input of the command with this argument, the routing message exchanges among the nodes should kick in.
`cost-change`	Indication of the new cost that will be assigned to the link between the last node and its neighbor with the highest port number (see Section 3.9). The program should understand this arg as optional (hence represented in brackets above).
`ctrl+C (exit)`	Use ctrl+C to exit the program.

3.7 Status Message

The following status messages should be displayed at least for the following events. The example of the messages are also shown below.

1. link-state packet sent:
   ```
   [<timestamp>] LSA of Node <port-xxxx> with sequence number <xxx> sent to Node <port-vvvv>
   ```
2. link-state packet message received:
   ```
   [<timestamp>] LSA of node <port-xxxx> with sequence number <xxx> received from Node <port-xxxx>
   ```
3. Duplicate link-state packet received:
   ```
   [<timestamp>] DUPLICATE LSA packet Received, AND DROPPED:
   - LSA of node <port-xxxx>
   - Sequence number <xxx>
   - Received from <port-xxxx>
   ```
4. Network topology (each link should be listed only once with lower port to higher port format, and the table should be ordered by increasing port numbers). The topology should be printed: (i) when the node wakes up, (ii) every time a valid LSA packet is received by the node and changes the topology, and (iii) when link cost change detected):
   ```
   [<timestamp>] Node <port-xxxx> Network topology
   - (<distance>) from Node <port-yyyy> to Node <port-zzzz>
   - (<distance>) from Node <port-vvvv> to Node <port-wwww>
   ⋮
   ```
5. Routing table obtained using Dijkstra's algorithm (computed after `ROUTING_INTERVAL` and with every change to the network topology):
   ```
   [<timestamp>] Node <port-xxxx> Routing Table
   - (<distance>) -> Node <port-yyyy>
   - (<distance>) -> Node <port-zzzz> ; Next hop -> Node <port-yyyy>
   - (<distance>) -> Node <port-vvvv>
   - (<distance>) -> Node <port-wwww> ; Next hop -> Node <port-vvvv>
   ⋮
   ```

3.8 Example

The example below illustrates the command inputs, message exchanges, and the computation of the routing table at each node given the network configuration described in Figure 1.
The command to start a program is:

C	`$ ./routenode ls <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ... [last][<cost-change>]`
Java	`$ java routenode ls <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> ... [last][<cost-change>]`
Python	`$ python3 routenode.py ls <r/p> <update-interval> <local-port> <neighbor1-port> <cost-1> <neighbor2-port> <cost-2> .. [last][<cost-change>]`

From the command line prompts type:
```
$ ./routenode ls r 2 1111 2222 1 3333 50
$ ./routenode ls r 2 2222 1111 1 3333 2 4444 8
$ ./routenode ls r 2 3333 1111 50 2222 2 4444 5
$ ./routenode ls r 2 4444 2222 8 3333 5 last
```

As soon as the command for node 4444 with keyword `last` is entered, the following process should kick in.

1. All nodes print their topology table (the set of links connected to them)
2. Node `4444` start the timer for both `UPDATE_INTERVAL` (plus a random number) and `ROUTING_INTERVAL`. It sends its LSA packet to node `2222` and `3333`.

3. Node `2222` and `3333` update their network topology tables according to the LSA sent from node `4444`. Both nodes forward Node `4444`'s LSA packet to their neighbors. In this case, Node `2222` forwards it to Node `3333` and Node `1111`. Node `3333` forwards it to Node `2222` and Node `1111`.
4. Nodes `2222` and `3333` send their own LSAs to their neighbors (these will be propogated according the the PI protocol) and start the timers for both `UPDATE_INTERVAL` and `ROUTING_INTERVAL`.
5. Once Node `1111` received Node `4444`'s LSA, it should update its topology table, send its own LSA to its neighbors and start the timer for both `UPDATE_INTERVAL` and `ROUTING_INTERVAL`.
6. All duplicate LSAs received by each node should be dropped and not forwarded to other nodes (see Section 3.5).
7. Once the `ROUTING_INTERVAL` expires at a node, it should compute the shortet path to all nodes (using the Dijkstra's algorithm) and print the routing table.
8. The program should execute forever. In other words, each node should keep broadcasting LSA packets every `UPDATE_INTERVAL` and Dijkstra's algorithm should be executed and printed every time the topology table changes.

We will wait a little more than the duration of `ROUTING_INTERVAL` after running your program for the routing table to appear. As indicated earlier, we will restrict the size of the network to 16 nodes in the test topologies. The default value of 30 seconds is more than sufficient for all the nodes to receive LSA packets from every other node and update the the network topology. An example of the routing table output is identical to the one shown in Section 2.7.

3.9 Triggered Link Cost Change

You should use the same function that you implemented for section 2.8 with one additional feature. Link cost change should only happen after the all nodes have computed their first routing table via Dijkstra's algorithm. Therefore, the timer for Triggered link cost change should be greater than `ROUTING_INTERVAL`. Please use 1.2×`ROUTING_INTERVAL` as the timer.

After sending (or receiving) the control messages discussed in 2.8, both nodes should update their LSAs and flood them to the network. They should also update their network topologies and routing tables if needed. Every node that will receive the LSAs should update its network topology and routing table (if needed). Notice that unlike in the DV case, any link cost change should result in correct new routing tables as soon as the LSAs are received.

3.10 Cases

Your program should deal with the following cases:

LSAs	Correct LSAs are generated and sent out to neighbors every `UPDATE_INTERVAL` and proper message are printed for topology update	(10pt)
PI Flooding	Mechanism to restrict link-state broadcasts (flooding) operates correctly	(15pt)
Routing Table	Correct operation of the link state protocol where routing tables are computed after `ROUTING_INTERVAL`	(15pt)
Link cost change	Appropriate handling of link cost change where the least-cost paths are updated to reflect the change in topology	(10pt)

4 Program Structure

The program should run an independent process per node. Within one process, there are two ways to operate:

- *single thread* operation - The process should listen to the port, receive an incoming packet, parse the packet data, and take actions (to update the local routing table and send out packets to its neighbors, or to check the sequence number and send back an ACK). Once all the actions are completed, the process goes back to port listening mode. The downsize of this approach is that normally there is a limited size input queue associated with each UDP port. This implies that while the process is busy taking above actions, incoming packets may be queued up and some of them may be lost. If you select this approach, you have to verify that the routing tables still converge.

- *multi-threading* - The main thread within a process is always listening to its port. Once a UDP packet arrives to the port, a sub-thread is created to parse the message and take actions. By doing this, the main thread is always listening to the port so that you will not miss any incoming packet (there are many ways to assign tasks to different sub-threads).

The program structure is left for you to design and implement. Make sure you test it thoroughly.

5 Submission Instructions

Please use C, Java, or Python for developing the emulation programs. You should develop on Ubuntu 18.04 LTS, which is available on the Google Cloud platform.
Before you begin programming, install the required packages for your choice of the languages above. Before proceeding, be sure to run `sudo apt-get update`.
If using C, use this command in the terminal to download gcc: `sudo apt-get install build-essential`

If using Java, use `sudo apt-get install default-jdk` to install the Java JDK. Please do not use Java 8.

If using Python, the Google Cloud Ubuntu 18.04 LTS images already have Python 2.7 as well as Python 3.5 installed. If using Python 2, clearly state so in the README. Otherwise, it will be assumed that Python 3 is used.

Make sure that your program compiles and runs on a Google Cloud Ubuntu 18.04 LTS instance before you submit the package. Your final submission package should include the following deliverables:

- `README`: The file contains the project documentation, program features, usage scenarios, brief explanation of algorithms or data structures used, and the description of any additional features/functions you implemented. This should be a **text file.** *You will lose points if you submit a PDF, .rtf, or any other format.*

- `Makefile`: This file is used to build your application. If you have written the programs in C (Unix), the output file names should be `routenode`. If you used Java, the file names should be `routenode.class`. You do not need a Makefile if you used Python, but please name your files `routenode.py`.

- Your source code.

- `test.txt`: This file should contain some output samples on several test cases. This would help others to understand how your programs work in each test scenario. Specifically, for the DV part , please provide an example of a network (set of commands), different from the one above, for which count to infinity occurs under the regular mode but does not happen in Poisoned Reverse mode. In addition to the separate file, you can also include this as part of your `README` document.

The submission should be made via Courseworks. Zip all the deliverables mentioned above, and name the zip file as `<your UNI>_PA2.zip` (e.g. `gz2136_PA2.zip` for Professor Zussman). The zip file should expand to **one** directory containing all deliverables. Upload the file in the Courseworks, under `Assignments -> Programming Assignment 2`.
It is highly recommended after submitting to download your submission, unzip it, and test it on Google Cloud Ubuntu 18.04 again!

In the grading of your work, we will take the following points into account.

- The documentation clearly describes your work and the test result.

- The source code is complied properly by using the `Makefile` and generate appropriate output files.

- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display necessary status messages.
- The programs follow the command line formats exactly, and that the deliverables are presented in the format specified by the assignment.

Happy Coding and Good luck!!