LSR REPORT

*Note: This assignment is written and run in Python 2

Program Design

The program I wrote utilises a multi-thread environment. Each of the threads do the following tasks:

- Thread 1: Broadcasts a packet containing information of the current router it's on and its neighbours. It's set to send a packet every second if certain criteria are met

- Thread 2: This thread (checkPulse) checks the heartbeat of each router to determine whether it is still alive or not. Dead router won't show up in the output messages as they are removed by this thread. The thread is run every 3 seconds.

- Thread 3: This final thread outputs the final message. It calls the Dijkstra's algorithm function which figures out the shortest path to each living router. It does this once every 30 seconds as per the assignment specification.

Features include:

- Classes are used to store data of each router.
  - The Router ID and Port number are stored as string variables
  - Neighbours are stored in a list within the class
- Link State Packets are broadcasted every 1 second
  - Data is sorted into the same format as the input file before being sent
  - It is essentially a bunch of strings that will be unpacked at the next router using a function I wrote called "sort_input_file". The function extracts the info into the senders' ID, port number, and neighbours.
- A Heartbeat function is able to check whether router is dead or alive
  - The check_pulse function is run every 3 seconds.
- Function "sort_input_file" is able to easily extract info from input in the format of the input file.
  - This is why I just decided to send the packets in the same format as the input file we got

Data Structure of Network Topology

- Dictionaries are used to store the results from Dijkstra's algorithm
  - The previously traversed router of each existing router is stored in a dictionary ("previous")
  - The cost to each respective router is stored in another dictionary
- The shortest routes to each known router is then calculated using the "getItinerary" function
  - This is calculated right before we give the output so it isn't stored anywhere, just printed and tossed during the final for loop

Ability to restrict excessive link rebroadcasts

- Doesn't rebroadcast packet back to sender:
  - The code reads the ports numbers and make sure to not send the packet to the port number it received the original packet from.
- Doesn't send to any of the senders' neighbours:
  - The original router is responsible for sending packets to its own neighbours.
  - If neighbours are shared between two routers, the second router won't send an extra packet to neighbour, reducing overhead

```
for neighbour in initial_router.neighs:
    if neighbour.target_router.port != address[1]: #Makes sure to not send back to sender
        if neighbour.target_router.ID not in recv_neighs: #Doesn't send packet to shared neighbours
            socket.sendto(received_packet, ("localhost", neighbour.target_router.port))
```

Figure 1: Excerpt from Lsr.py which shows how I restrict excessive broadcasts

Dealing with Router Failures

- There is a Boolean value for each known router within the network. As long as packets are continually received from these routers, the Boolean value will remain true.
    - All living routers with the 'True' value is kept in the "known_routers" list which is fundamental for the rest of the code working
- However, once a router is dead, its Boolean Value will be 'False'. Once this happens, the checkPulse function will remove the router from the "known_routers" list and updates the information.
- Routers don't have a problem re-joining the network when they come back to life

Design Trade-offs and Performance

- My code mainly uses dictionaries to store data. This is quite efficient and made it quite easy for me to follow while I was writing it.
- Storing the router information as a class made the code a lot cleaner and way easier to access.
- I believe that transmitting packets in the format of the input file is quite efficient as I only have to write a single function to extract information from the packets.
- Heartbeat:
    - The assignment spec recommended to wait for 3 missing heartbeat messages to be received before declaring a node dead
    - I found it more efficient and easier to implement to just simply update the heartbeat every 3 seconds which pretty much achieves the same result.
-

Sources that helped me complete my code:

- Dijkstra's Algorithm:
    - https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
        - The Wikipedia article was very helpful in helping me understand Dijkstra's
    - https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
        - The python implementation shown in geeksforgeeks helped me implement it Dijkstra's in my own code
- Threading in Python
    - https://realpython.com/intro-to-python-threading/
        - I've never used threading before so this gave me a basic understanding of how it works
    - https://www.youtube.com/watch?v=ecKWiaHCEKs
        - This video by 'Engineer Man' gave a me concise visual explanation of how threading worked