COMP3331/9331 Computer Assignment Report

Name: Vishaal Vageshwari

zID: z5161415

1 – Brief Discussion of LSR protocol implementation

The implementation of the LSR protocol in my assignment is a multi-threaded object-oriented (although not strictly adhering to oo-principles) implementation written in Python and tested on Python3 version Python 3.7.3. The implementation uses a ConfigParser module to extract information from the config files for a Router object which is created in each process. The main function then calls runLSR on the Router object which starts the three thread of execution used in the implementation. One thread is set to broadcast link state advertisements (LSAs) over a UDP socket and the other to receive LSAs, store relevant details in its network map and forward LSAs to neighbours. The LSAs sent contain a sequence number that is iterated on each broadcast, the id and address of the origin router, the number of neighbours it has and details on its neighbours including whether they are 'dead' or 'alive'. When a router receives an LSA in the receiver thread it extracts information from it and decides whether to store its information in the network map and forward the LSA by comparing its sequence number to ones it received before. It also removes information of dead routers from is network map. The main thread of execution is left to run Dijkstra's algorithm on a copy of the network map and print the relevant info every ROUTER_UPDATE_INTERVAL (30 secs) using a NetworkDijkstra object.

List of Implemented Features:

- Sending, receiving and forwarding link state advertisements (LSA flooding) using two additional treads (multi-threaded)
- Having routers independently conduct Dijkstra's algorithm every 30seconds to find lowest cost paths to all other routers in the topology
- Restricting excessive broadcast by preventing forwarding outdated or duplicate packets
- Handling multiple node failures by using the regular LSAs as heartbeat messages
- Handling re-entry of dead routers into the topology

There weren't any features in the specifications that I did not implement.

2.1 – Data structure of network topology

The network_map field in the Router object is my data structure representing the network topology. It is adjustment to common way of representing a dictionary graph in python ({EdgeID1: {EdgetID2: cost}}). It is a dictionary of dictionaries of dictionaries where the keys for the first and second dictionary are the ids of the routers and the third contain the address of the neighbour and weight/cost of that link. An example of one router and its neighbours in this map would be:

{'A': {'B': {'address': ('127.0.0.1', 5001), 'weight': 6.5}, 'F': {'address': ('127.0.0.1', 5005), 'weight': 2.2}}}

2.2 – How node failures are dealt with

When a router is first initialised, it has a dictionary entry for each neighbour with key = 'last_received' which it set to the current time using the time function from the time module. Whenever a router receives an LSA from its neighbour it updates last received to the current time when it received it. Then whenever

an LSA is created it checks whether the difference in the current time to the time it last received a LSA from its neighbour is greater than (MAX_HEARTBEATS * UPDATE_INTERVAL) + LAST_LSA_CUSHION (3 * 1 + 0.01 = 3.01 seconds) and if it is that router is considered dead till its neighbours receive another LSA from it. The reason UPDATE_INTERVAL is used is because we are doubling up our LSAs as control messages for whether routers are dead or alive. MAX_HEARTBEAT is just the maximum number of LSAs missed before considered dead and using the recommendation from the specifications that's three. The LAST_LSA_CUSHION is extra time given for last LSA to possibly be received. This was chosen by testing the times LSAs are sent and received and using those times to determine an appropriate buffer (0.01 seconds plenty of time but not enough to delay recognition of a failed router) for that last LSA to be received. The neighbours then create LSAs that are forwarded to other nodes in the network informing them of the dead router and they then remove instances of that router from their network map using the remove_dead_router function. If the dead router re-joins its neighbour will mark it as alive and it'll stop getting removed allowing it to be considered back in the topology.

2.3 – How the implementation restricts excessive link-state packets

When LSA is received we record the sequence number of the LSA from the origin router in a dictionary field called received_seq. An example entry would be {'A': 3} where the origin of the LSA received was A and the sequence number was 3. When LSAs are received if there is no entry for that router in received sequence or the entry in the received sequence has already received a LSA with a higher sequence number then there is no need to forward because this router has already received and forwarded a LSA for that router that is either the same or more recent. Therefore, duplicates are detected and prevented from being forwarded restricting excessive link-state packets circulating in the topology.

3 – Discuss design trade-offs, what is special about your implementation and improvements/extensions

There were a few design trade-offs made to makes my implementation either more extensible or simpler. One example is that I added addresses in my network map even though for this specific assignment the output only requires the IDs and the costs. This is so that it would be easy to see the lowest cost path in terms of addresses if one wished. This however made my network map slightly more memory inefficient. The implementation also does not use a priority queue for Dijkstra's algorithm which would increase the efficiency of Dijkstra's algorithm to O(|E| + |V|log|V|). The reason for this is just because it was more effort then I deemed necessary to write an entire module just for the priority queue (binary heap). One of the things that is special about my implementation is because of the use of object-oriented programming my implementation provides more abstraction and encapsulation of tasks. This should make it easier for another person to navigate through the implementation. The implementation also makes extensive use of python dictionaries which make use of robust hash function to allow for getting an items to have an average time complexity of O(1). They are also relatively space efficient. The implementation is also multithreaded making use of three thread to accomplish the various tasks. These two things make my implementation relatively efficient.

Improvements/extensions:

1. As previously stated, I could have implement Dijkstra's algorithm using a priority queue by implementing another module for the priority queue. This would increase the average efficiency of my implementation of Dijkstra's.
2. Another extension I could include some method to also display shortest path by addresses. This could be achieved simply by including addresses in my retrace of predecessor structure in Dijkstra's.

3. Handling new nodes joining the network. This could be done if the implementation was extended to include the steps necessary to establish a neighbour connection.
4. Handling node failures that create partitions in the topology. I think this could be handled by having a router detect when they have no links to the rest of the topology and finding a method to establish a connection to some router in the topology.

4 – Indicate any segments of code that you have borrowed from the Web or other books

The implementation of Dijkstra's algorithm I used was adapted from pseudo code from https://brilliant.org/wiki/dijkstras-short-path-finder/.