COMP9331 Assignment Report

Tian Liu

Z5230310

Development environment: Python 3.7.2

Test environment: Python 3 on CSE machine

1. Run script:

```
xterm -hold -title "configA" -e "python3 Lsr.py configA.txt" &
xterm -hold -title "configB" -e "python3 Lsr.py configB.txt" &
xterm -hold -title "configC" -e "python3 Lsr.py configC.txt" &
xterm -hold -title "configD" -e "python3 Lsr.py configD.txt" &
xterm -hold -title "configE" -e "python3 Lsr.py configE.txt" &
xterm -hold -title "configF" -e "python3 Lsr.py configF.txt" &
```

2. Brief discussion of LSR protocol:

In this assignment, I create many functions to implement the LSR protocol. The main functions in my code are:

read_file: read *CONFIG.TXT* file and return **routerid**, **protium**, **forward_graph**, **neighbour_port**, **record_times**, all values are global variable.

send_msg: send link-state packets.

send_msg_self: call function send_msg, only send own link-state packets to its neighbours.

forward_msg: call function send_msg, send link-state packets received by its neighbours.

recv_msg: receive link-state packets sent by its neighbours, not only its direct neighbours link-state packets but also its indirect neighbours forwarded by its direct neighbours.

update_graph: once receive packets, update the **global variable updating_graph**.

dijkstra: parameters are the **updating_graph** and **start-node**(i.e. routerid in my code), and return two dictionaries one indicate the distance from the start, another is the predecessor of every nodes in shortest path.

find_path: call dijkstra function to print the shortest path from routerid to every other nodes which exist in keys of the dictionary updating_graph.

And **global variable record_times** is used to deal with fail nodes by record times once receive their packets.

3. Data format and methods:

foward_graph stands for link-state packets the format is a dictionary:

{router: {link-state message}}

The key is router which is Router id(i.e. 'A' - 'F').

The value is link-state message which is also a dictionary, for example,{'B': 6.5, 'F': 2.2}, the key is neighbours, and the value is cost.

Followed is an example of link-state packets format:

```
[s-MacBook-puro-2:9331_ass skyler$ python3 Lsr.py configA.txt
{'A': {'B': 6.5, 'F': 2.2}}
```

updating_graph is used to represent the network topology:

{router1:{link-state message1},router2:{link-state message2},……}

The format is dictionary which contains many different forward_graph messages.

The picture shows below:

```
s-MacBook-puro-2:9331_ass skyler$ python3 Lsr.py configA.txt
{'A': {'B': 6.5, 'F': 2.2}, 'F': {'A': 2.2, 'D': 0.7, 'E': 6.2}, 'B': {'A': 6.5,
 'C': 1.1, 'D': 4.2, 'E': 3.2}, 'E': {'B': 3.2, 'D': 2.9, 'F': 6.2}, 'D': {'F':
0.7, 'B': 4.2, 'C': 1.6, 'E': 2.9}, 'C': {'B': 1.1, 'D': 1.6}}
I am Router A
Least cost path to router F:AF and the cost is 2.2
Least cost path to router B:AFDCB and the cost is 5.6
Least cost path to router E:AFDE and the cost is 5.8
Least cost path to router D:AFD and the cost is 2.9
Least cost path to router C:AFDC and the cost is 4.5
```

Methods 1: In my code, I use the record_times which I mentioned above to record time when I first read them in the *CONFIG.TXT,* when router received messages from my neighbours router will update record_times corresponding to this neighbour, also router will check record_times and delete neighbours if router have not receive their messages in 3 consequent HEARTBEAT which I have already set to 0.7s. Once router detected fail nodes I will also delete it from forward_graph in order to update its link-state message.

Methods 2: In my code, create a global variable forward_table to record this message is sent by who, and next time when receive same message, this router will check whether this message is received from the node that recorded in forward_table, if so this router will forward this message, otherwise router will drop it.

4. Design trade-offs considered and made:

For this assignment, I create 3 threads, first is to send my own link-state every 1 second, second is to receive link-state and update the global graph and last is the find_path which to print least-cost to every other nodes. Also the record_times can handle the problem that restart of failure nodes, once I receive message form failure neighbour, I will add them into my record_times and update my link-state.

Above are some parts that I think successful implementation, but for restricting excessive broadcasts, I think there will be a better way to eliminate such unnecessary packets, for example, use reverse path forwarding or use spanning tree. Apart from that, I use own link-state message as heartbeat message, I can change the heartbeat to an explicit heartbeat message to detect failure nodes more quickly.

It's worth talking that I will empty **updtaing_graph** after I printed the result in order to deal some simple topology(e.g. A -> B -> C), so that once B failed, A and C print nothing.

5. Reference:

Part of my dijkstra function reference from:

https://blog.csdn.net/u010558281/article/details/53905807