IITM-CS6040 : Router Architectures and Algorithms
Assignment 1

Release Date: Aug 9, 2024
Due Date : Aug 23, 2024

Roll No: CS23S018
Name: Nikhil Shinde
Collaborators: None
References/sources:
[1] https://networkx.org/documentation/stable/index.html

1. Comparison of 12-node NSFNET and 20-node ARPANET topologies for custom connection requests of size 100, 200, 300.

Solution:

- Terminology:
  1. Link-disjoint paths: The two shortest paths do not share a common link from source to destination.
  2. Non link-disjoint paths: The two shortest paths share at-least one common link from source to destination.
  3. Blocking probability: The number of blocked connections to total connections.
- NSFNET-12 node topology: The left table in Table 1 gives the blocking probabilities for non link-disjoint paths, and the right table gives the blocking probabilities for link-disjoint paths.

  It can be observed that for both type of shortest paths implementations, the blocking probabilities are similar with respect to a configuration and number of connections.

  In case of both non link-disjoint paths and link-disjoint paths for NSFNET-12 topology, the change in configuration parameters does not cause a significant change in the probabilities.
- ARPANET-20 node topology: The left table in Table 2 gives the blocking probabilities for non link-disjoint paths, and the right table gives the blocking probabilities for link-disjoint paths.

  It can be observed that for ARPANET-20 topology link-disjoint shortest paths implementation has lower blocking probability than non link-disjoint shortest paths implementation for 100 connection requests. However, as the connection requests grow, the blocking probabilities are similar for both implementations.

  In both implementation cases, hop metric has lower blocking probability than distance based metric.

-	Number of connections		
flag/p	100	200	300
h/0	0.32	0.43	0.56
h/1	0.34	0.47	0.59
d/0	0.31	0.43	0.57
d/1	0.34	0.45	0.61

-	Number of connections		
flag/p	100	200	300
h/0	0.32	0.43	0.56
h/1	0.34	0.47	0.59
d/0	0.32	0.45	0.55
d/1	0.34	0.47	0.59

Table 1: NSFNET-12. **a) Left.** Non link-disjoint shortest paths **b) Right.** Link-disjoint shortest paths

-	Number of connections		
flag/p	100	200	300
h/0	0.06	0.14	0.33
h/1	0.07	0.17	0.35
d/0	0.15	0.28	0.42
d/1	0.17	0.29	0.45

-	Number of connections		
flag/p	100	200	300
h/0	0.01	0.17	0.37
h/1	0.01	0.19	0.39
d/0	0.09	0.26	0.41
d/1	0.12	0.28	0.41

Table 2: ARPANET-20. **a) Left.** Non link-disjoint shortest paths **b) Right.** Link-disjoint shortest paths

A Algorithms

Algorithm 1 Virtual Circuit Switching Simulation

Input: topo_file, conn_file, rt_file, ft_file, path_file, flag, approach
Output: Forwarding Table, Paths Table

Function `Main()`:
    Initialize Exercise object with parameters: topo_file, conn_file, rt_file, ft_file, path_file, flag, approach
    Load topology from topo_file and build network nw
    Build routing table using `build_routing_table()`
    Load connection requests from conn_file
    Process connections using `process_connections()`
    Write forwarding table to ft_file
    Write paths table to path_file
    Display statistics about connection requests and failures

Function build_routing_table():
    for each node in network nw **do**
        Get paths from node to all other nodes using `get_paths()`
        Write paths to routing table file rt_file

Function get_paths(src_node):
    Initialize empty paths dictionary for all nodes
    for each node in network nw **do**
        Copy network nw to nw_copy
        Find shortest paths from src_node to node using `yenKSP()`
        Store paths and their metrics

Function yenKSP(nw, src, dst, K=2):
    Initialize empty lists for paths and their lengths
    Find the first shortest path using Dijkstra's algorithm
    Find additional paths using Yen's algorithm
    return paths and their lengths

Function process_connections():
    for each connection request **do**
        Try to admit the connection on path1
        if Path1 fails **then**
            Try to admit the connection on path2
        if Connection admitted **then**
            Assign VCIDs and update forwarding table
            Store connection path and metrics in paths table
        else
            Count the failed connection

Function get_new_vcid(node, flag):
    Generate a random VCID
    Ensure VCID uniqueness in forwarding table
    return VCID