# Network Topology

We use a program “graph\_generator.py” to randomly generate a network containing 10 nodes and 15-20 random edges. The program generates the corresponding config for each node and draws a graph representing this network.

We have also manually produced some special topologies where we carefully select all link-distances for use of testing.

For example, configs in config1 folder are specially produced for testing huge scale network; the configs in config3 folder are specially produced to test trinode network.

About how to use graph\_generator.py, please refer to README.md

# Routing Algorithm

Refer to week 3 lecture slide [1]

Regarding the choice of routing algorithm, we considered both the Router Information Protocol (Distance Vector Algorithm) [[2]](https://en.wikipedia.org/wiki/Routing_Information_Protocol) and Open Shortest Path First (Link-state Routing Algorithm) [[3]](https://en.wikipedia.org/wiki/Open_Shortest_Path_First).

Given that the network is decentralized, with each node initially lacking global information, and considering the network undergoes dynamic changes often (using Command Line Interface to alter the network structure), we have chosen to use distance vector algorithms.

In our design, each node sends its routing table to its neighbours every 10 seconds. Upon receiving a routing table from a neighbour, each node will compute and update its own routing table. The specific update method follows the Bellman-Ford equation: let be the cost of the least-cost path from x to y, then . If a node, after receiving information from neighbor v, discovers that , it will update its routing table, setting , and update the path information.

We also applied some techniques to address specific scenarios. For example, we implement Split Horizon, where a node, upon receiving a routing table from a neighbour, will identify which pieces of information originated from itself (sent to the neighbour and then included in the neighbour’s routing table) and disregard those pieces of information. This can effectively handle the Routing Loop Problem. Furthermore, if a node hasn't received a packet from a neighbour for 12 seconds, it will set the distance to that neighbour as infinite and modify the distances to related nodes until it receives information from them again.

# Implementation Methodology

To enable a node to simultaneously send packets, receive packets, accept commands from the CLI, and determine if a neighbor is disabled, we create multiple threads, with each thread responsible for one specific function of that node.

To facilitate communication between nodes, we use sockets. A node will encode the message it wants to send and then transmit it through the socket to the target port. Moreover, each node continuously listens to its own port.

To implement the link cost change feature, we added a Command Line instruction for each node. Upon receiving this instruction, the node sends a message to the corresponding node to notify it of the change and then updates its own config file and routing table. The corresponding node, upon receiving the message, will also update its config file and routing table. A modification to a link cost will lead to changes in the entire routing table. Therefore, the node that receives the message will delete **all** paths related to the changed link from its routing table and then rerun the routing algorithm.

To implement the disable and enable functionality, we introduced a new trigger for each node. If this trigger is activated, the node will pause its operations (but it will complete any ongoing routing before pausing). If other nodes do not receive any messages from it for 12 seconds, they will recognize that it has been disabled and set the distance to it as infinity.

# Simulation Results

## Simulation on config2

In config2, we conducted a test on a simple network with only three nodes.

图示

描述已自动生成

Once the network is stabilized, Every node’s routing table is correct.

Our primary purpose in testing this network was to see if our routing algorithm could solve the routing loop problem.

If we disable C, both A and B successfully set the distance to C to infinity (instead of B assuming it can reach C through A because A’s transmitted routing table includes C). This proves our algorithm could handle the routing loop problem.

If we set the distance between B and C to 4.

A, B, C will update their routing tables correspondingly and correctly.

## Simulation on config3

In config3, we conducted a test on the trinode network, with the network structure as shown in the diagram:

图示

描述已自动生成

Once the network stabilized, Every node’s routing table is correct.

It is evident that our algorithm can handle cases where there is a shorter distance to a certain neighbor through another path than the distance specified in the config.

Our primary purpose in testing the trinode network was to see if our routing algorithm could solve the problem of misleading.

If we disable C, both A and B successfully determine that C has been disabled and set its distance to infinity (instead of A assuming it can reach C through B because B's transmitted routing table includes C) and set the distance to each other to 5. This proves that A and B are not misled by each other.

## Simulation on example within the specification

After the network is stabilized, every node’s routing table is correct.

If we disable node C and wait for the network to stabilize for 20 seconds, then input the command "routing table" into all nodes to prompt them to print their current routing tables, the routing tables of all nodes are correct.

After conducting more experiments on the network, including disabling a large number of nodes and changing some link costs, each node is able to correctly update its own routing table.

# Reference

[1] COMP3221 Distributive System week3 lecture slide

[2] <https://en.wikipedia.org/wiki/Routing_Information_Protocol>

[3] <https://en.wikipedia.org/wiki/Open_Shortest_Path_First>