**Christopher Morcom**

**Seat #20**

**CS 542 – Spring 2019**

**28 April 2019**

**Link State-Routing Simulator Design and Test Report**

**Link-State Routing Protocol**

Link-State Routing is a routing algorithm in networking where all nodes in a network send information about its neighboring nodes via flooding on network updates. To create routing tables, Link-State Routing uses the Dijkstra’s Algorithm rather than the Bellman-Ford Algorithm, used in Dynamic-Vector Routing. In this project, we simulate the Open Shortest Path First (OSPF) protocol by calculating the shortest path from a source to destination, and by identifying the best router for flooding. OSPF is an implementation that uses Dijkstra’s Algorithm to find a list of all shortest routes from source to destination without having to specify any number of hops and flooding this information in the network. By implementing this with Depth-First-Search in this project, we (archaically) simulate the Time-To-Leave (TTL) field in OSPF, which prevents infinite looping of packets in a network, that is in our project, the TTL is closely approximated to the time from source to destination in the shortest path.

**Simulator Design**

The simulator uses a Basic Command-Line interface for all menu selections but creates windows in new threads as necessary to allow the user to simultaneously visualize the network and continue to perform operations on it.

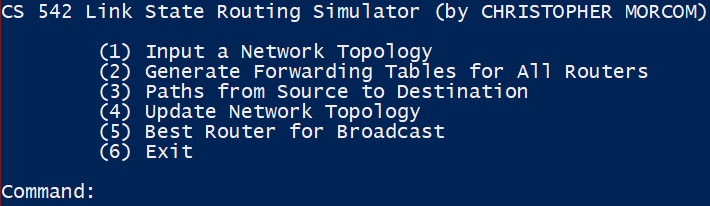
To execute the simulator, the following appropriate ***Python 3*** libraries must be installed to execute the following:

* import networkx
* import math
* import \_thread
* import msvcrt
* import collections
* import tkinter
* import matplotlib.pyplot

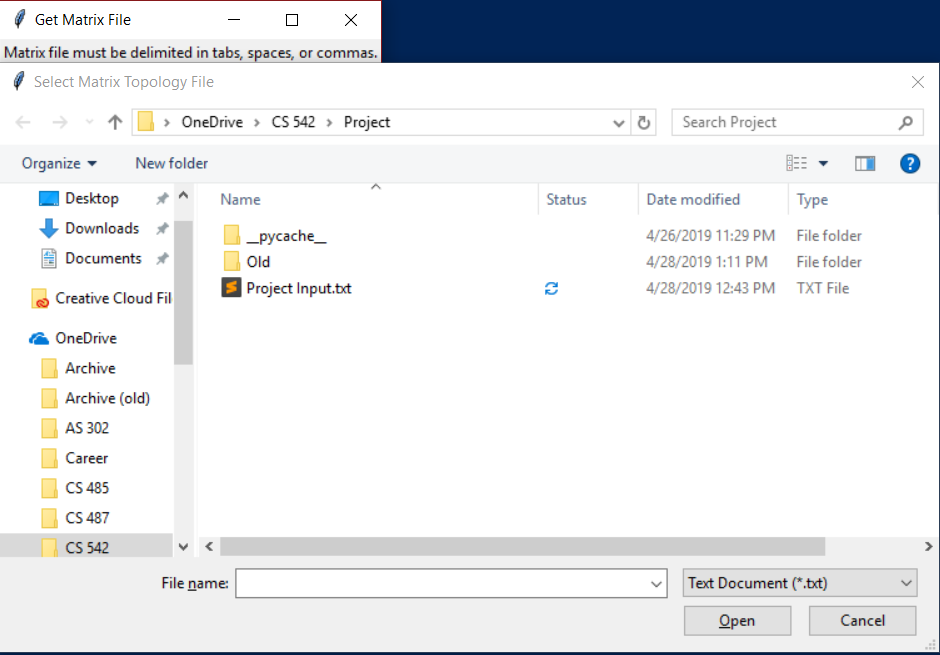
(note that \_thread is a redundant library and may be discontinued in favor of threading)

The following is a sample of each menu option in the project *(executed in win10 PowerShell)*. To access appropriate functions, the interface gets the first character typed and executes tasks. ***Note that erroneous inputs from the user are handled, but bad file inputs and creating multiple networks in the project results in crashing or other errors (see testing for explanations).***

***Main Menu:***

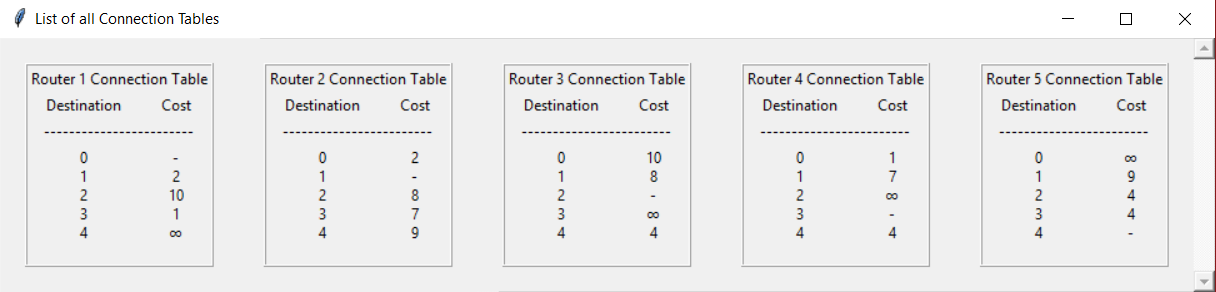


***Input Network Topology (Option 1):***

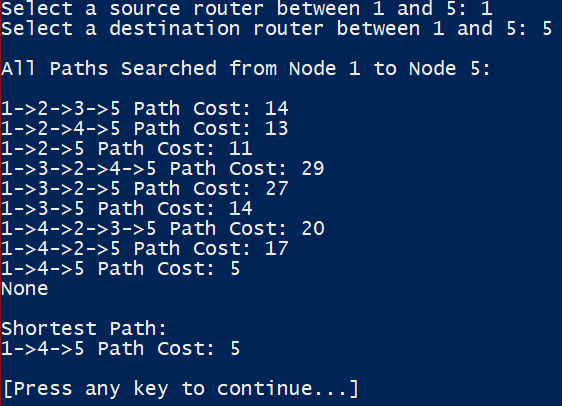


(note that any file can be imported if it is in \**.txt*, \**.csv,* or \**.tsv* format.)

***Generate Forwarding Tables for All Routers (Option 2):***

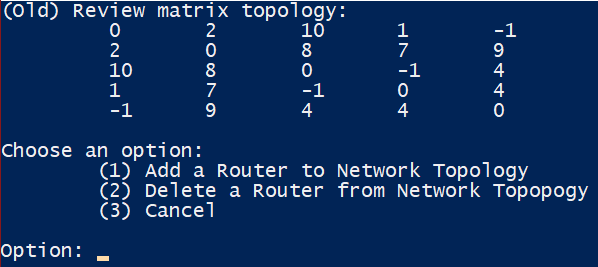


***Paths from Source to Destination (Option 3):***

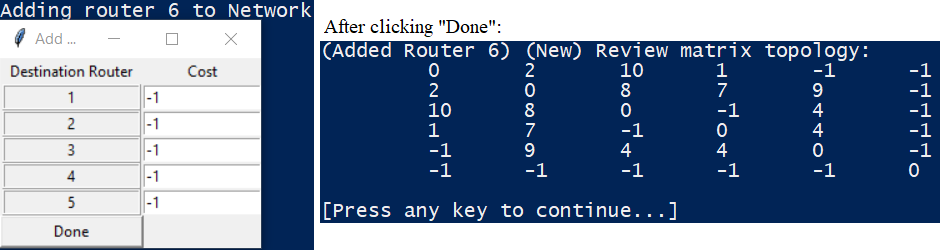


(Used input source = 1, destination = 5 for example)

***Update Paths from Source to Destination (Option 4):***

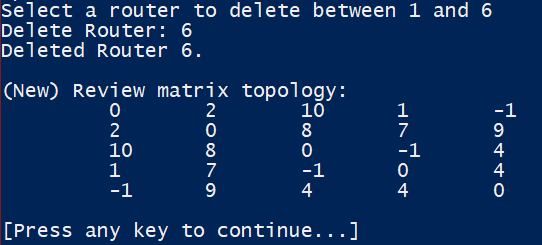


***Adding a Router:***

******

(Added a router with no connections for this example)

***Deleting a Router:***



(Deleted Router 6 used in the previous example)

***Best Router for Broadcast (Option 5):***



(only outputs a number in command line)

***Exit (Option 6):***



(This may execute with runtime errors due to multithreading errors)

**Simulator Testing and Results**

***Notes:***

This simulator has the following rules enforced in the program which allow it to work minimizing errors. As such, erroneous input files and certain user inputs may still crash the simulator or provide undesirable outputs.

*RULES (READ THIS):*

1. No operations (other than Exiting) may be selected until Option 1: *Import a Network Topology* is selected.
2. The network is **assumed** to be a graph that is weighted, but *undirected*. This means that only matrix input files where the matrix is the same as its transpose can be handled:

(i.e. MT = M for any input matrix M)

1. All edges must have an **integer weight** greater than zero.
2. Edges are not editable. You must delete the router and re-add it to the topology.
3. Deleting routers will automatically rename all other routers ahead of it. This prevents confusion and overflow. Router names are not specified by the user.
4. To make it easier for the user, after an option that *modifies* the network is selected (like adding or deleting routers), the plot will automatically update. **(The user must recreate forwarding tables after this if they want to view them!)**
5. The interface treats all negative values the same. (i.e. any negative value constitutes no connection between routers)
6. When adding a router, ***ANY*** erroneous input will be discarded, and the network will ***NOT*** update.
7. **Multiple networks can be generated in this simulator!** As such, finding the router which is best-for-broadcast will yield the lowest-cost router for the lowest cost network. (The user can solve this by creating a link with an infeasibly high cost between two sub-networks.)
8. **Threads will not suspend themselves. Please close all windows or your command line will crash and you will have to open a new one.**

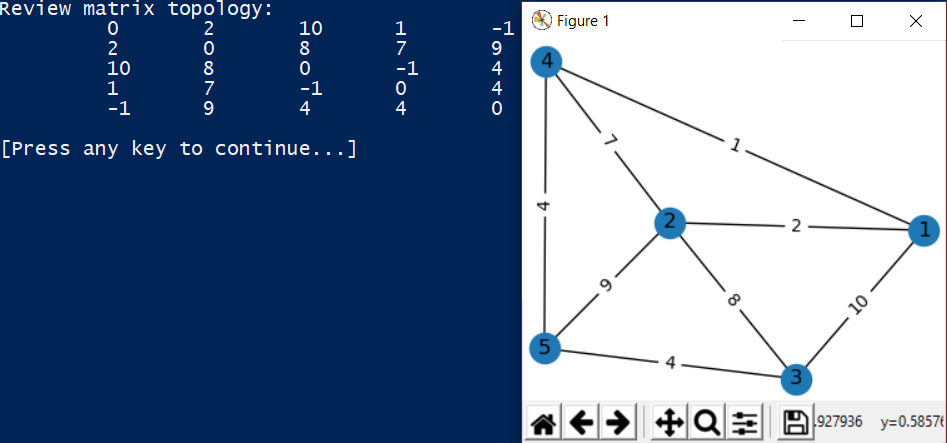
Testing has been done to establish these rules both through programming and sensible user inputs. Thread suspension and sub-network handling could still be implemented, but this is beyond the scope of this class and an undergraduate degree in computer science at IIT.

***Testing and Results:***

***Option 1:***

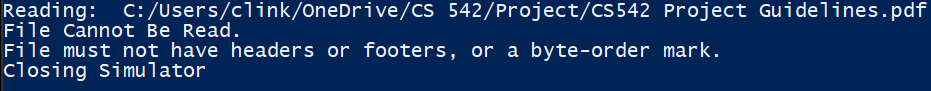
When inputting a network topology, only the upper triangular matrix is scanned. This prevents having double edges. If a text document, comma-separated values, or tab-separated values file is selected, **it must be in UTF-8 with no BOM at the beginning of the file.** (BOM is a byte-order mark). Any unreadable file will either result in a crash or output text saying that it is unreadable. Files are split with the following regex: ‘,|[A-z]|\s\*’This regex splits the file around integer values, but can only handle one delimiter at a time. For instance, two spaces between numbers will crash the program.

Sample valid output:



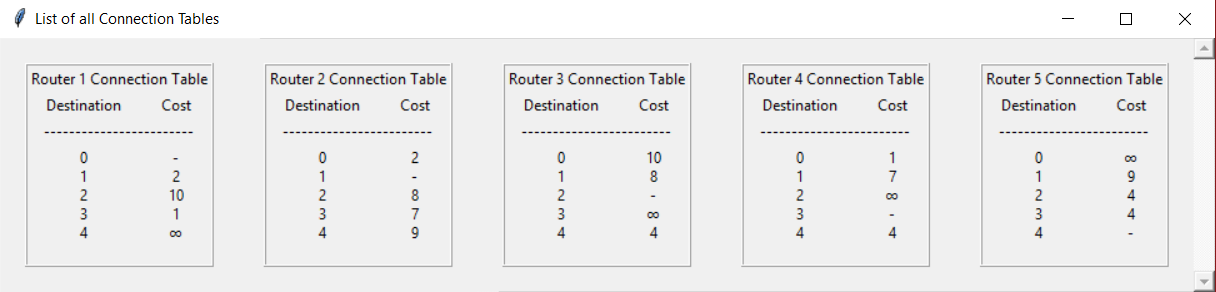
The graph display is entirely random and may not look nice sometimes. To fix, select option 1 again, but cancel the file input dialog.

Sample invalid output:



Simulator will crash to avoid threading errors on bad file input.

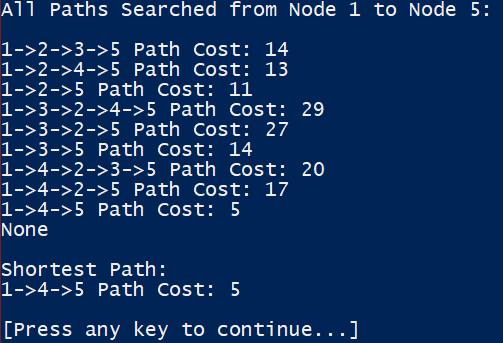
***Option 2:***



Forwarding Table Sample output with test file (listed in the next section).

Table must be recreated on close after network change.

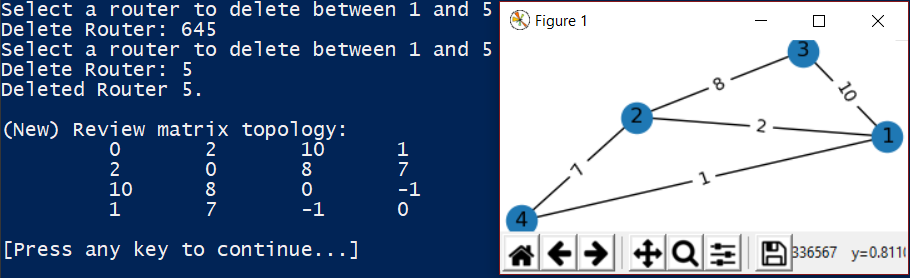
***Option 3:***



This function prompts the user for source and destination routers. The function uses a DFS (Depth-First-Search) approach to get all paths from source to destination and print the paths and their costs. The ‘None’ output is erroneous, and it is currently unclear how to get rid of that at this time. Finally, the shortest path is found using Dijkstra’s Algorithm and is printed followed by the path’s cost. This output is not cleared on resume, so leaving the graph window open or using option 1 to refresh the topology. **Having too many possible paths for a massive network may crash this simulator, but this version can handle a mesh topology with up to 40 nodes.**

See *Network Searching Algorithms* for this function’s search algorithms in detail.

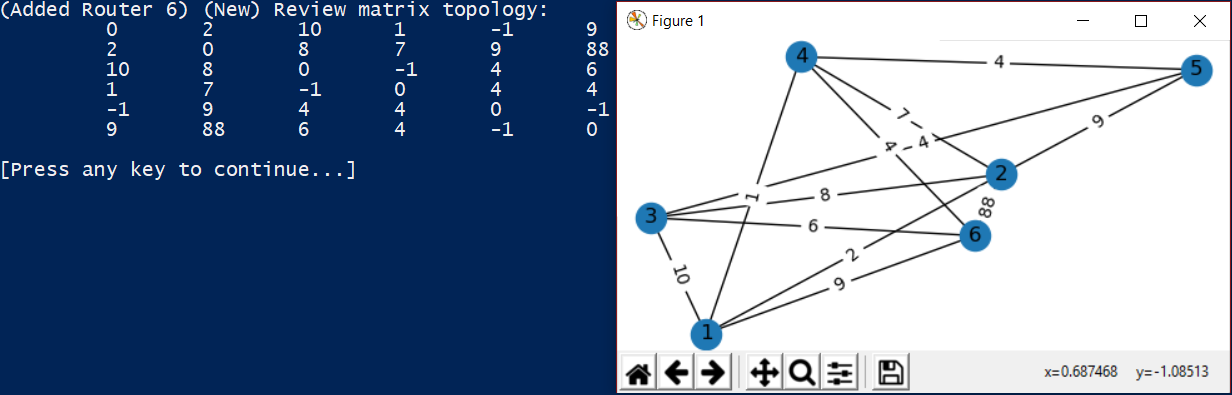
***Option 4.1:***



Sample Above deleted router 5 from original network topology. Will clear previous output and re-prompt until a valid input is entered. Refreshing the graph can be done using Option 1. Erroneous inputs from the user will be disregarded, and the user will be re-prompted until a proper integer input is given.

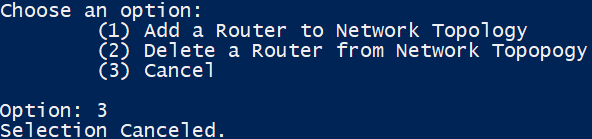
***Deleting all nodes will result in having to input a new network topology (calling option 1).***

***Option 4.2:***



Example output of adding a router to the network. Matrix topology and graph are both updated and displayed. Error inputs will not result in an update, but the current network will still be displayed. Refresh the layout by using Option 1 and cancelling the input.

***Option 4.3:***



Selecting the third option will cancel the operation and will return to the main menu. This is in case the user accidentally selects to update the topology. Erroneous selections will result in re-prompts of this menu.

***Option 5:***



This option uses Dijkstra’s Algorithm to search paths to every possible node in *n(n-1)/2* calls or in time where V is the number of nodes, and E is the number of edges. Having a large enough input set will still crash this program.

N**etwork Searching Algorithms**

It is important to note that the networkx library in python represents graphs as a dictionary of dictionaries where the keys are nodes, and the values are dictionaries of adjacent connections. This is essentially a representation of an *adjacency matrix* in python. This brings our time and space complexity to *O(V+E)* and *O(V2),* respectively. This is assigned recursively, such that any value from a key in the parent dictionary traces all paths to any other destination key.

* **Depth-First-Search (DFS)**

This DFS is a recursive method called on a graph *G* which stores the path in a tempory list called ‘path’. It clears the path variable each time it finds a path, as to avoid confusion by passing a master list. This allows us to handle a mesh topology of up to 40 nodes, without consuming too much memory and passing a massive, growing list on each call. Visited nodes are kept in a list where indices of the list correspond to the visited nodes. The list of visited nodes has the source node initialized to ‘True” as the algorithm starts from the last visited node. The pseudocode of this DFS is listed below.

*Pseudocode:*

1. Mark the current node as visited and append it to the current path.
2. If the current node is the same as the destination, print out the path and its cost (the cost is found by calculating the sums of the edges between nodes)
3. Otherwise, check if the adjacent nodes have been searched.
4. If adjacent nodes have not been searched, recursively search the next one until it prints a path.
5. If the current node has no neighbors and is an end node that cannot link to the destination node, remove (pop) it from the path and mark it as not visited. Continue searching outside of this node for all recursive DFS calls.

* **Dijkstra’s Algorithm**

Like the DFS, this also uses an adjacency matrix to achieve a time complexity of *O(V+E)*. This algorithm updates a list of weights, initialized to infinite weights between nodes, and a list of scanned nodes and nodes queued for scanning, to generate an adjacency matrix of paths. Only the shortest path, along with its weight are returned.

*Pseudocode:*

1. Queue the list of all nodes that need to be scanned (initially all of them).
2. Get the lightest node and visit it (put it in the adjacency matrix). Set it as the new source.
3. Check each neighboring node and sum its recursive path weight using the graph’s adjacency matrix to find the weights.
4. If the distance calculated is less than its corresponding weight in the list of weights, update the shortest distance to that vertex.
5. Once the dictionary of lists of all shortest paths is generated, start at the destination and backtrack to the source in the adjacency matrix and append its path. The destination node is the starting key in the adjacency matrix.
6. Since this path is reversed, correct it and find its corresponding weight in the weights list (destination node is the index of the weight in the list).
7. Return the path and its weight in a tuple.

**Test Input File:**

The following is the contents of the file: *sample input.txt*. The sample input uses whitespace as a delimiter. .csv and .tsv versions of the file have also been tested. The file was given by the TA and copied into a format that does not have a file header (which cannot be handled by Python 3).

File Contents:

0 2 10 1 -1

2 0 8 7 9

10 8 0 -1 4

1 7 -1 0 4

-1 9 4 4 0

*This file was the sample file used in the examples above.*

*Any extra/deleted nodes were the result of a demonstrated network update.*

**Running the Program:**

1. Install all python libraries (and Python 2 and 3) indicated on the first page.
2. Run the program using: python ./FinalProject.py
3. Make sure to have a matrix file that follows the specifications indicated on the first page ready.
4. Import the matrix file using option 1 and proceed to carry out any other operations.