OLSR Simulator

Zeeshan Sajid

CPE 400

11/21/2015

Introduction:

This project is on Optimized Link State routing, a complex IP routing protocol for mobile ad-hoc networks. Essentially, at its base, it is Link State routing, in which the the network calculates the shortest path through the routers within the network using some sort of shortest path algorithm typically Dijikstra. The key difference between LSR and OLSR is the idea that LSR packets will be not have duplicate transmission from one router to another, instead it uses Multipoint relay an approach to make sure no duplicates are sent during the initial flooding operation from one router to each other. This is utilized to discover the topology of the network and to store said topology of said network to each router while still maintaining good performance by removing unnecessary data.

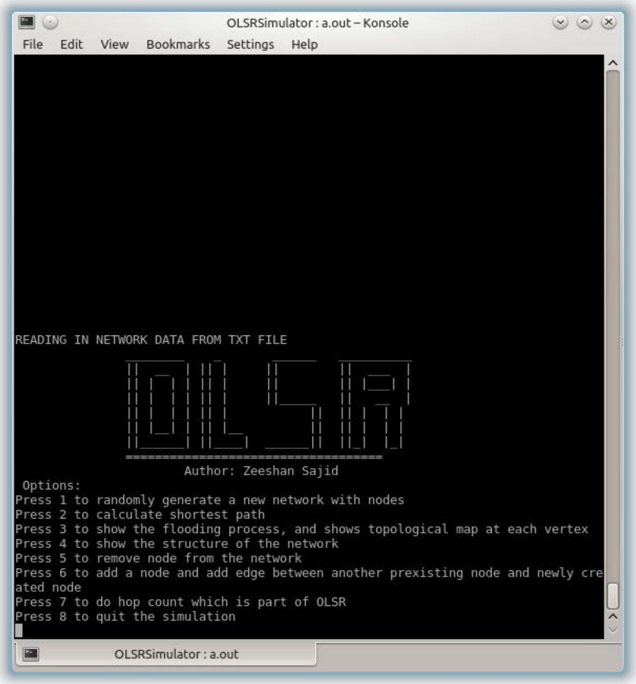
The motivation for choosing Optimized Link State routing is due to the fact that it holds one key idea and that is finding the shortest path for a router within said network. Essentially, it is significant for ad-hoc as it removes the duplicate packets from the routers, and thus improving performance. At the end of the day, however, the motivation of doing this project is to learn the basics of shortest path and some idea of how to solve it, while at the same time learning of improving performance for it as well, and of course learning to simulate a network by creating a graph data structure.

Contribution

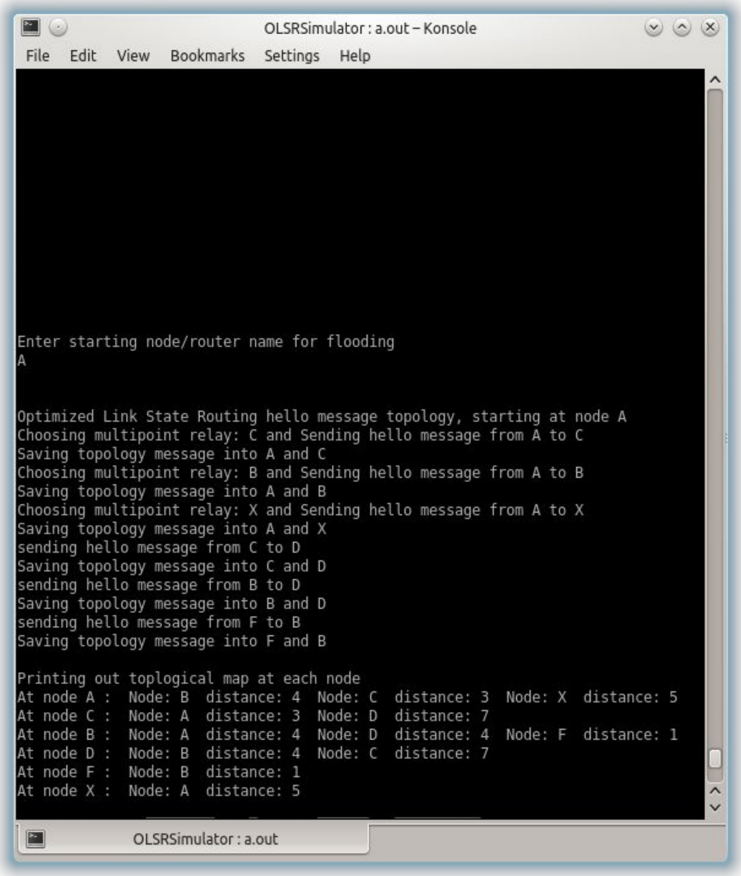
This project is split into essentially two features, in which Link State Routing and Optimized Link State Routing essentially both have and that is to find the shortest path within the network. Secondly, both protocols utilize topology checking, and storing said topological data into each vertex. However, aside from that, both protocols are reliant on individual node structures and rely heavily on a graph data structure, in which any node from said graph can be removed without really breaking the protocol or from finding shortest path.

For the program/protocol that I implemented, I have implemented all the different features and simulated them. The program focuses on Link State Routing, in which it creates a graph data structure, storing each and every individual node from a file, and then calculating shortest path. In terms of Optimized Link State routing, the program focuses on sending topological information to each router within the network, this is done using a map data structure, in which storing said data, and forwarding said data by choosing specific routers to forward the data from. Essentially, printing out the data for the user, in which the user can see that no duplicate data was sent at all during this flooding process, and additionally allowing the user to delete or adding any sort of router without disturbing the overall networking.

Picture 1: Demonstrates the menu options the user has after data is read in

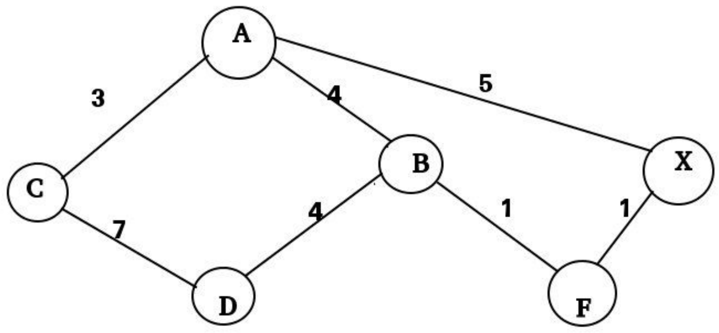


```
READING IN NETWORK DATA FROM TXT FILE

              ===================================
                       Author: Zeeshan Sajid
 Options:
Press 1 to randomly generate a new network with nodes
Press 2 to calculate shortest path
Press 3 to show the flooding process, and shows topological map at each vertex
Press 4 to show the structure of the network
Press 5 to remove node from the network
Press 6 to add a node and add edge between another prexisting node and newly cre
ated node
Press 7 to do hop count which is part of OLSR
Press 8 to quit the simulation
```

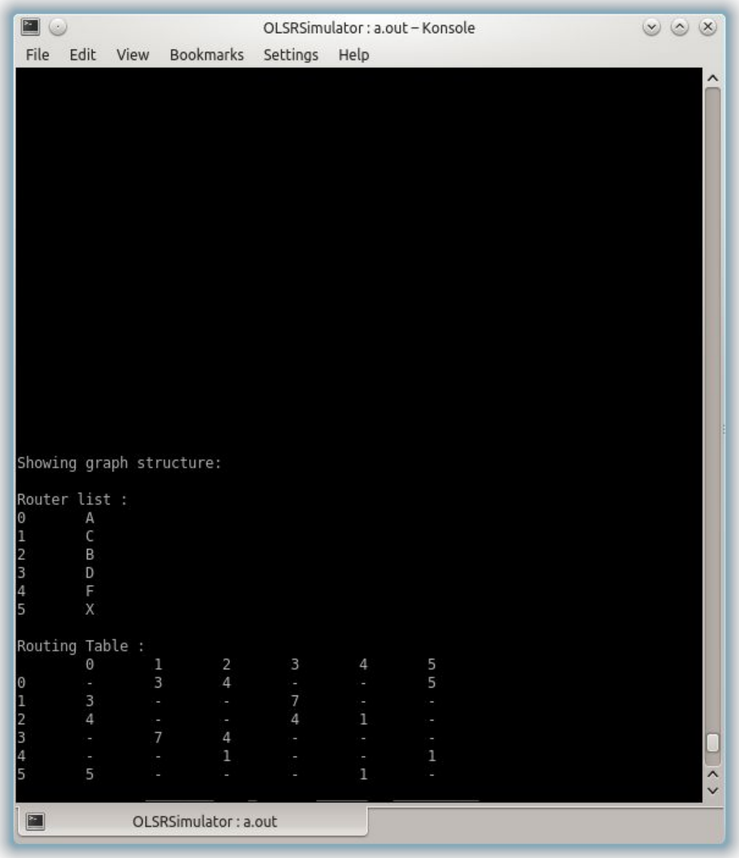

OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

```
Enter starting node/router name for flooding
A


Optimized Link State Routing hello message topology, starting at node A
Choosing multipoint relay: C and Sending hello message from A to C
Saving topology message into A and C
Choosing multipoint relay: B and Sending hello message from A to B
Saving topology message into A and B
Choosing multipoint relay: X and Sending hello message from A to X
Saving topology message into A and X
sending hello message from C to D
Saving topology message into C and D
sending hello message from B to D
Saving topology message into B and D
sending hello message from F to B
Saving topology message into F and B

Printing out toplogical map at each node
At node A :  Node: B  distance: 4  Node: C  distance: 3  Node: X  distance: 5
At node C :  Node: A  distance: 3  Node: D  distance: 7
At node B :  Node: A  distance: 4  Node: D  distance: 4  Node: F  distance: 1
At node D :  Node: B  distance: 4  Node: C  distance: 7
At node F :  Node: B  distance: 1
At node X :  Node: A  distance: 5
```

Picture 2: The picture to the left demonstrates the flooding process and hello topology process a huge feature of LSR and OLSR. The user chooses three from the menu selection option in which router/node A finds the closest neighbors, which happen to be C and B, in which they are assigned as multipoint relays, and their overall topology. Essentially, those points relay their topology to other neighbor nodes, in which the overall network topology is discovered completely as seen at the bottom of the picture. As seeable within the picture, topological node information is not duplicated, meaning A to X is only transmitted once, in which A sends B information about its topology, in which B does not try to send its topology back to A, and only forward its current topology to neighboring nodes which happen to be D and F, while still keeping A's topology information. Essentially, this feature is not called every time to keep the overall simulation clean and is used to prove it was implemented.

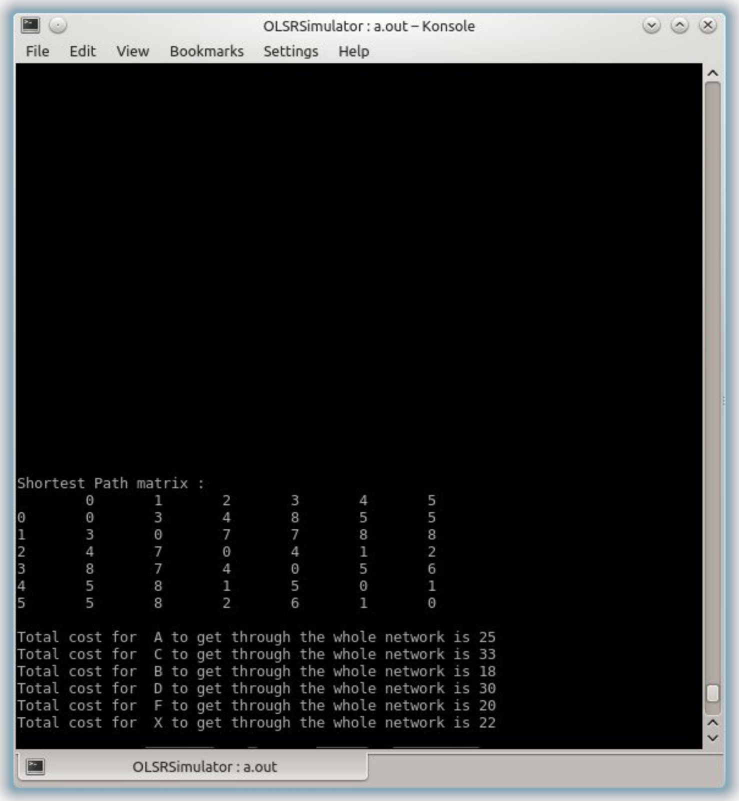


Picture 3: The picture on the left shows the graph created using the information from above, and the information from the the text file. As you can see the graph is fully connected, so essentially every router can reach each other while traveling a certain amount of distance.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

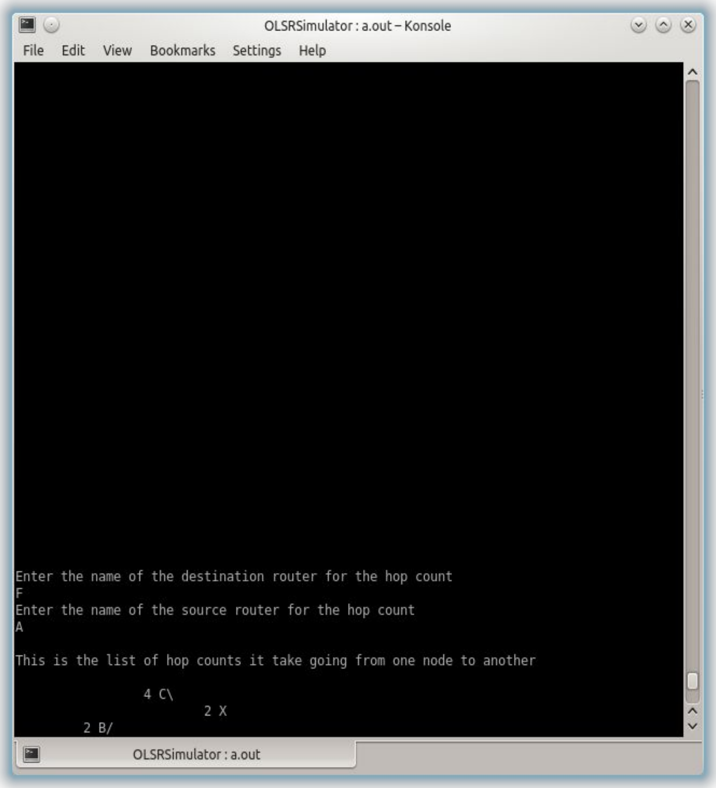
```
Showing graph structure:

Router list :
0       A
1       C
2       B
3       D
4       F
5       X

Routing Table :
        0       1       2       3       4       5
0       -       3       4       -       -       5
1       3       -       -       7       -       -
2       4       -       -       4       1       -
3       -       7       4       -       -       -
4       -       -       1       -       -       1
5       5       -       -       -       1       -
```

Picture 4: The picture to the left demonstrates another feature of the program, and that is to show the overall routers in the network and their weight that connects to each other routers. For example, to reach C from A, it would take 3, from, A to B would take 4, and from A to X it would take 5, so on. Essentially, the purpose of the picture is to show that the network is well maintained and all the information is accurately generated from start to finish.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

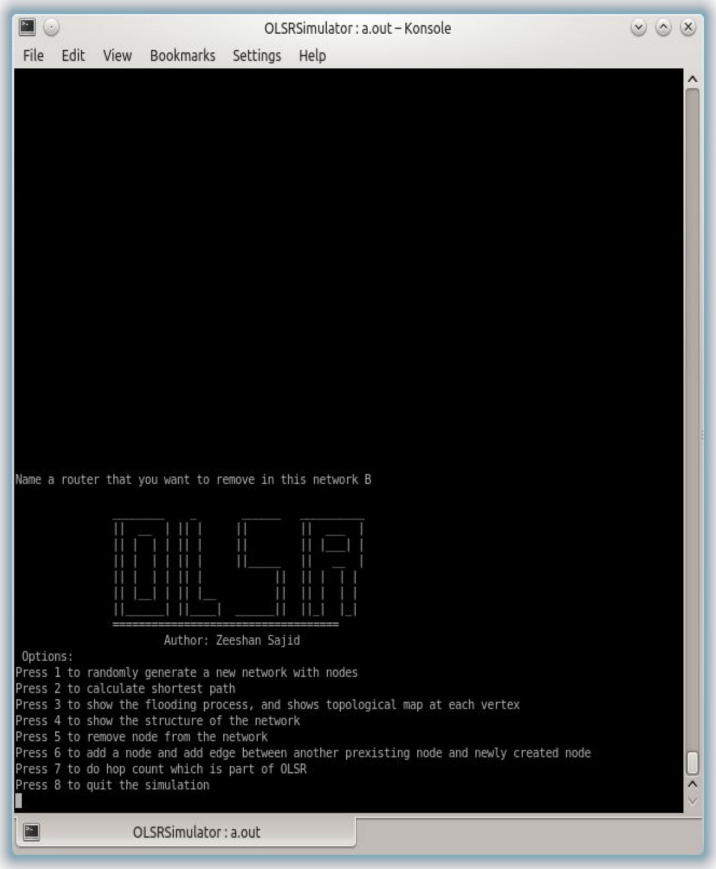
```
Shortest Path matrix :
        0       1       2       3       4       5
0       0       3       4       8       5       5
1       3       0       7       7       8       8
2       4       7       0       4       1       2
3       8       7       4       0       5       6
4       5       8       1       5       0       1
5       5       8       2       6       1       0

Total cost for  A to get through the whole network is 25
Total cost for  C to get through the whole network is 33
Total cost for  B to get through the whole network is 18
Total cost for  D to get through the whole network is 30
Total cost for  F to get through the whole network is 20
Total cost for  X to get through the whole network is 22
```

Picture 5: This picture demonstrates one of the biggest feature of LSR and OLSR, and that is calculating the shortest path to each and every other router from one of the routers in the network, and the overall amount of path a router needs to cover to go through the entire network. Essentially, router A will need eight to go from its point of origin to D, in which it needs to go through B which has four for its weight, and to D which has another four. While, to cover the whole network it would take 25.



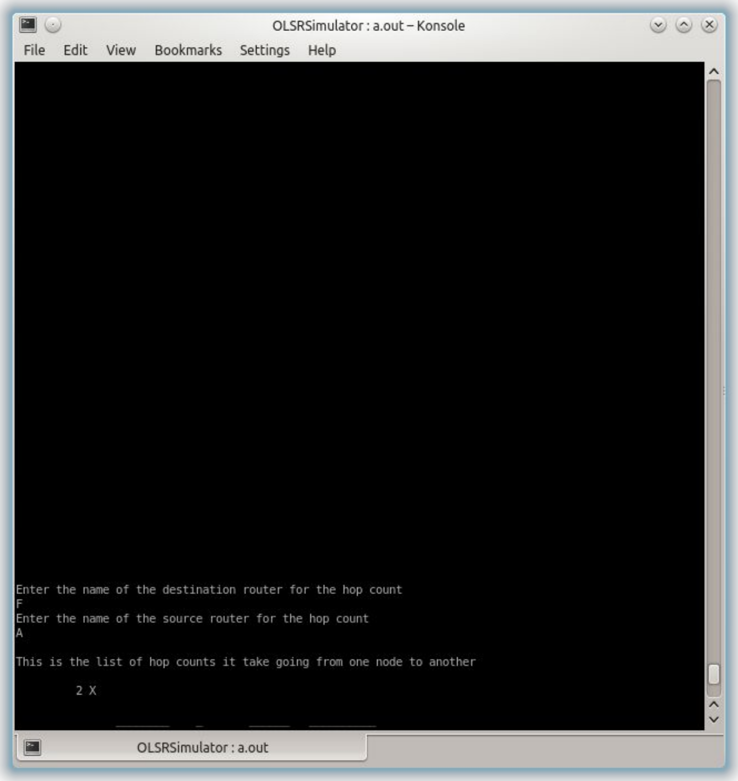
```
Enter the name of the destination router for the hop count
F
Enter the name of the source router for the hop count
A

This is the list of hop counts it take going from one node to another

                 4 C\
                         2 X
         2 B/
```

Picture 6: The picture to the left demonstrates a major feature of OLSR and that is hop counting, which it does instead of actually seeing the distance from one router to another, it counts the number of routers it needs to jump to get to its destination. The picture demonstrates when the user presses 7, it ask the user for the destination of the hop, and ask for the source, in which it list out the cases possible to reach it destination. In the particular case from above, the router F can be reached from A by going through C, which is not optimal at all, or it can go through B or X to reach F. Essentially, the hop count shows the user the different choices they have, of course in reality it would always choose the least number of hop counts to reach the destination, however, for simulation purposes, the algorithm for my hop count shows the user all of the different possibilities.



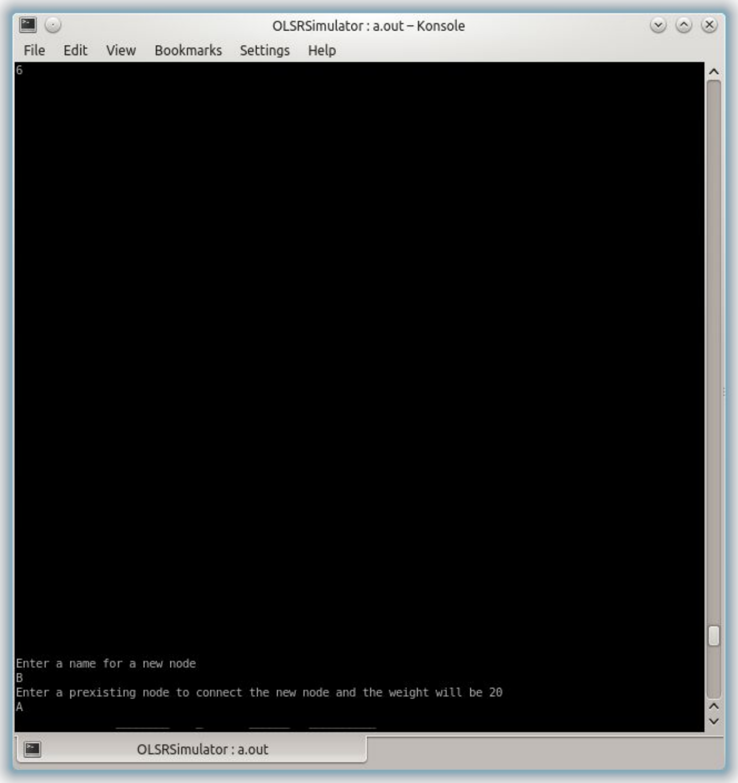
```
Name a router that you want to remove in this network B


             _______  _      _____   _________
             ||  _   |||    ||      ||       |
             || |  | |||    ||      || |___| |
             || |  | |||    ||_____ ||  ___  |
             || |  | |||         || || |  | |
             || |__| |||___      || || |  | |
             ||______||____| _____|| ||_|  |_|
             ===================================
                    Author: Zeeshan Sajid
 Options:
Press 1 to randomly generate a new network with nodes
Press 2 to calculate shortest path
Press 3 to show the flooding process, and shows topological map at each vertex
Press 4 to show the structure of the network
Press 5 to remove node from the network
Press 6 to add a node and add edge between another prexisting node and newly created node
Press 7 to do hop count which is part of OLSR
Press 8 to quit the simulation
```

Picture 7: If the user wants to remove a vertex from the network, they can press 5, in which it ask for the router to be removed. Hypothetically, a router can be removed or more so a node can be removed any time from any sort of networking, but for this simulation, the user can decide what they want to remove at what time to demonstrate that the different features of LSR and OLSR still work flawlessly regardless of whatever is removed.



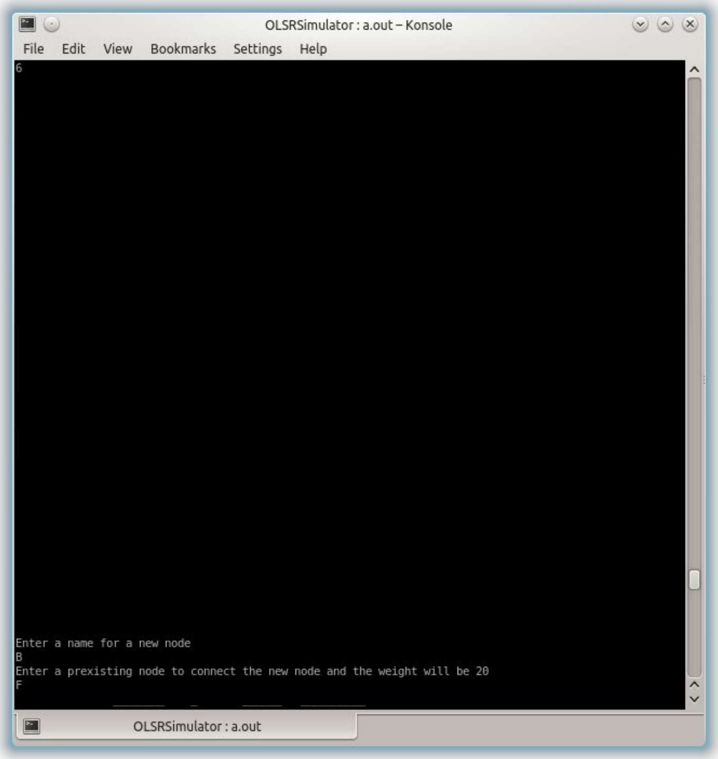
```
Enter the name of the destination router for the hop count
F
Enter the name of the source router for the hop count
A

This is the list of hop counts it take going from one node to another

        2 X
```

Picture 8: This picture this demonstrates a feature of OLSR in which if a node/router is removed from the network, the hop counts will be recalculated based on the new data, as opposed as to the old one, and still provide the different ways to get to it, and since B is removed, F does not need to go through C, and cannot go through B, so it has to go through X, and that requires one hop from F to X and from X to A.



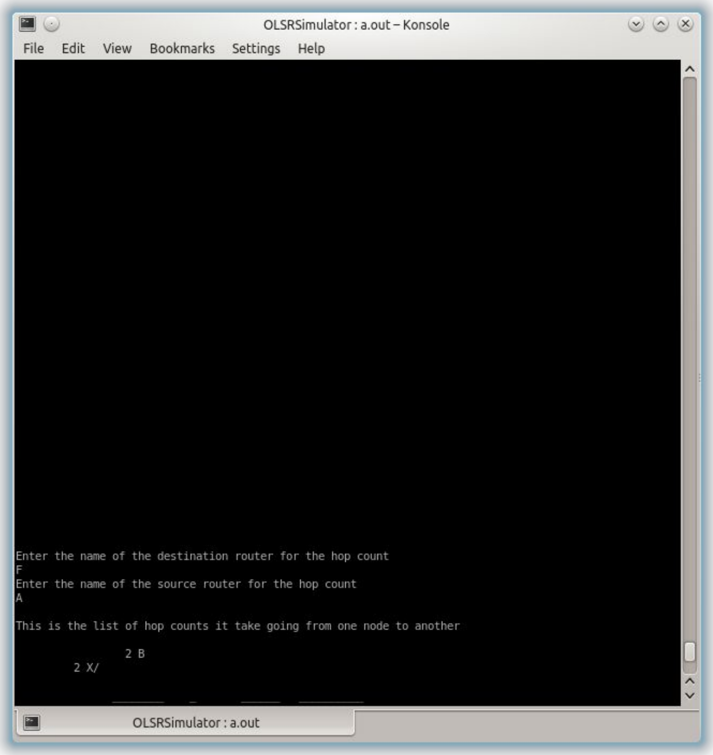
```
6

Enter a name for a new node
B
Enter a prexisting node to connect the new node and the weight will be 20
A
```

Picture 9: Demonstrates a feature of the program, in which it adds another router to the network, essentially the user can create and remove anything from a network like it happens in a real network, however, it only connects B to A, and does not construct one from B to F like the original graph had.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

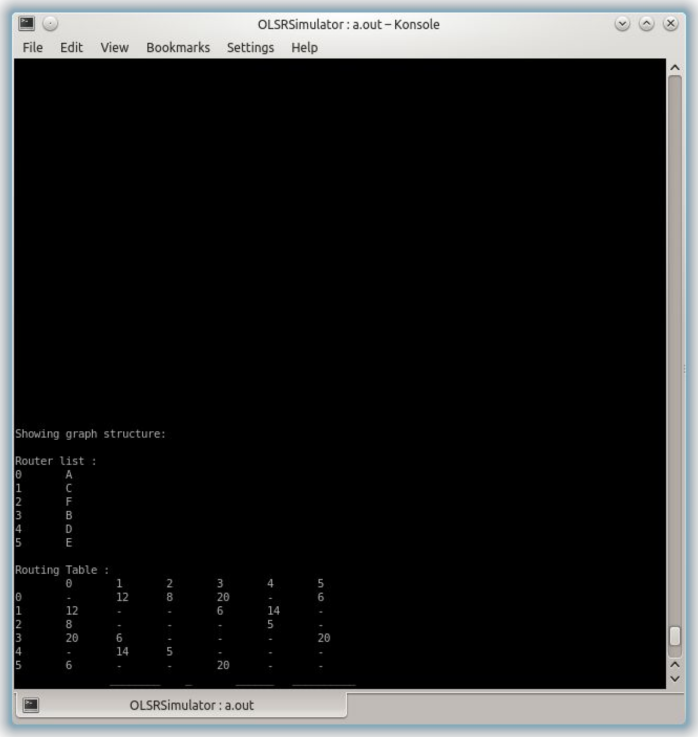
```
6

Enter a name for a new node
B
Enter a prexisting node to connect the new node and the weight will be 20
F
```

Picture 10: This picture demonstrates the same thing as above, however, this time, it's creating another link from F to B, in which it's recreating the original graph, except with weights of 20, minus the link from B to D.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

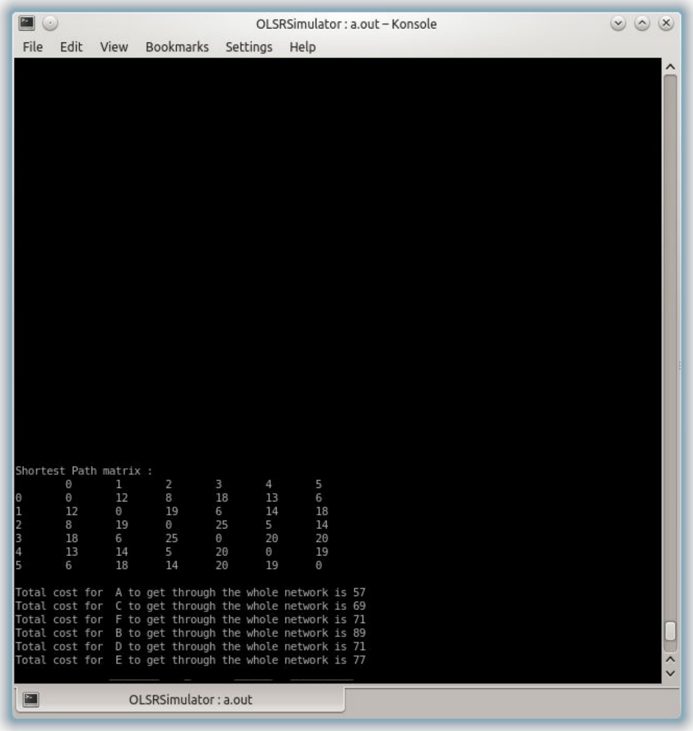
```
Enter the name of the destination router for the hop count
F
Enter the name of the source router for the hop count
A

This is the list of hop counts it take going from one node to another

                 2 B
        2 X/
```

Picture 11: This picture demonstrates the number of hops from F to A after recreating the old. Essentially, since A B and F are created, F goes to B and to A, which is to hops, and does this for F to X to A as well. Since B and D are no longer linked, it does not show the scenario if F wanted to go from B to D to C and then back to A like the original graph did.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

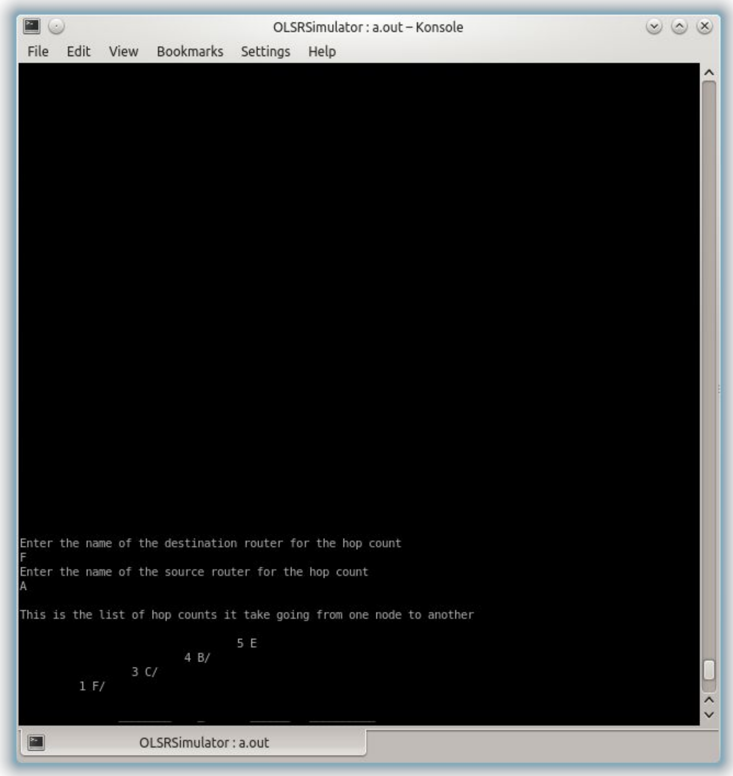
```
Showing graph structure:

Router list :
0       A
1       C
2       F
3       B
4       D
5       E

Routing Table :
        0       1       2       3       4       5
0       -       12      8       20      -       6
1       12      -       -       6       14      -
2       8       -       -       -       5       -
3       20      6       -       -       -       20
4       -       14      5       -       -       -
5       6       -       -       20      -       -
```

Picture 12: This picture demonstrates a feature of the program as opposed to a feature of the protocol. Essentially, if the user clicks on option one, which is creating a random network, it essentially creates a random number of routers holding a random number of weights in between said routers, however, keeping those number of routers small. Essentially, this picture shows the newly created network and all of the different weights from router X to router Y.


OLSRSimulator : a.out – Konsole
File Edit View Bookmarks Settings Help
OLSRSimulator : a.out

```
Shortest Path matrix :
        0       1       2       3       4       5
0       0       12      8       18      13      6
1       12      0       19      6       14      18
2       8       19      0       25      5       14
3       18      6       25      0       20      20
4       13      14      5       20      0       19
5       6       18      14      20      19      0

Total cost for  A to get through the whole network is 57
Total cost for  C to get through the whole network is 69
Total cost for  F to get through the whole network is 71
Total cost for  B to get through the whole network is 89
Total cost for  D to get through the whole network is 71
Total cost for  E to get through the whole network is 77
```

Picture 13: This picture demonstrates the same feature of shortest path calculations that was already shown above, however, this time, it shows it with the dynamic and random network that was generated from the program. Essentially, for each of the routers above, it calculates the shortest path to reach every other router. For example, for going from C to F, it would need 19.



```
Enter the name of the destination router for the hop count
F
Enter the name of the source router for the hop count
A

This is the list of hop counts it take going from one node to another

                                 5 E
                         4 B/
                 3 C/
         1 F/
```

Picture 14: This picture demonstrates a feature of OLSR and that is to calculate hop count, however, the only difference between this one and the one from above is that this is done on the randomly generated network, and very accurate. There are other pictures depicting the same features on the randomly generated network, but no reason to fluff up this document.

Code Explanation

Platform:

This program is meant to run on a Linux environment, for my environment, it was Kubuntu, a variant of Ubuntu, in which preferably g++ compiler should be installed.

Extraction:

Open sajid.zip, extract the files into a folder, such as CPE400Sajid in a home directory.

Compilation:

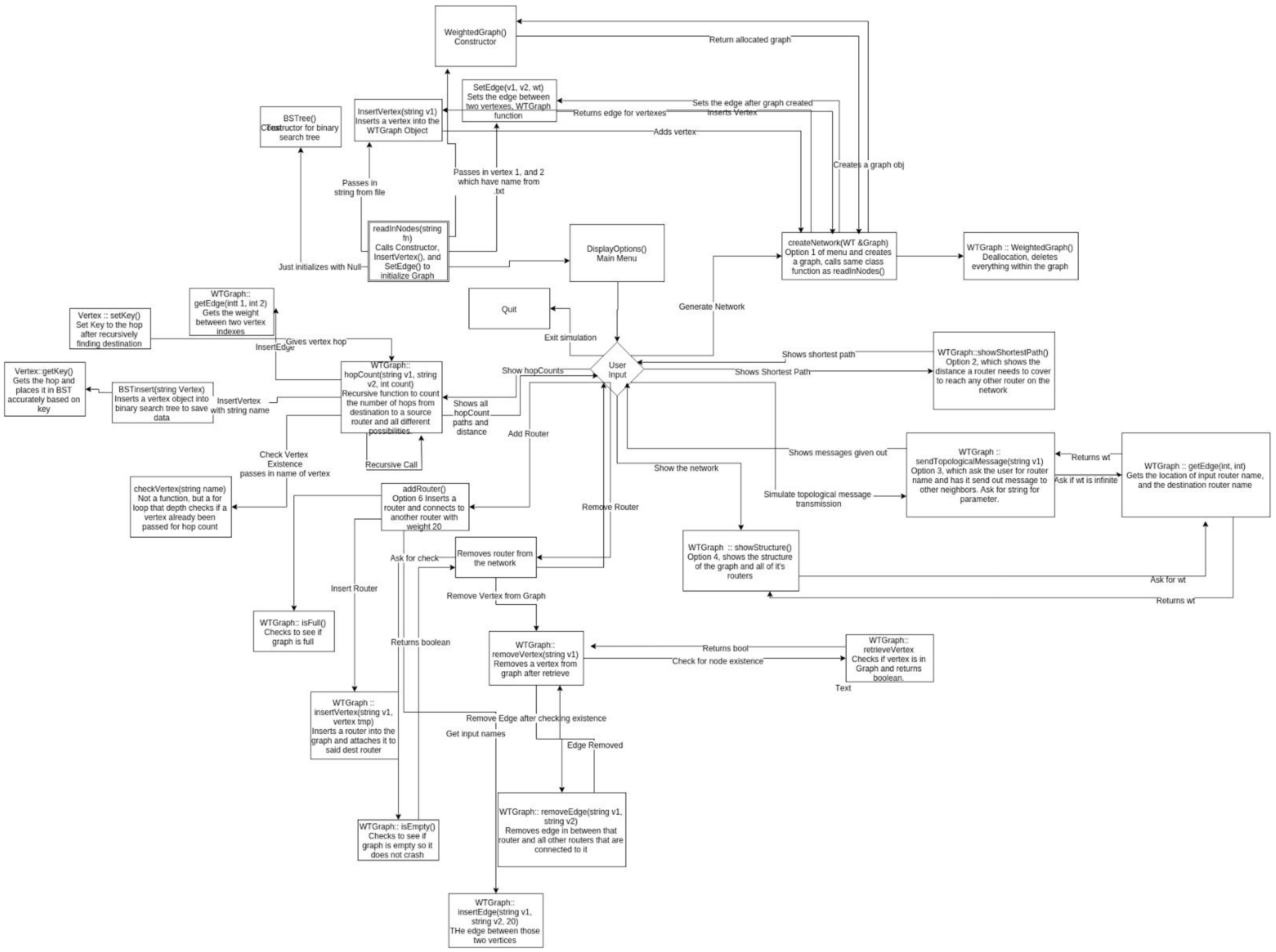
For compilation, change directory from the terminal to said folder, and then type g++ OLSR.cpp, which will compile the program. In order to run it, type ./a.out
Specific commands to run from terminal, the name of the directory/ folder may be different and cd will not guide you to the location of files for compilation if the subdirectories name are different, in which case, just cd to through all those subdirectories till reaching the files with my program

```
cd ~/CPE400Sajid
g++ OLSR.cpp
./a.out
```

Restrictions:

This program can take a single letter as a name for the routers, so for option 5 and 6, typing in F would remove router F from the network for option 5. For router 6, to add a router, type only one letter as well, and it would be good to type in capital but it should not pose a problem otherwise.

Block Diagram:


WeightedGraph() Constructor
Return allocated graph
SetEdge(v1, v2, wt) Sets the edge between two vertexes, WTGraph function
Sets the edge after graph created
Returns edge for vertexes
Inserts Vertex
InsertVertex(string v1) Inserts a vertex into the WTGraph Object
Adds vertex
BSTree() Constructor for binary search tree
Creates a graph obj
Passes in vertex 1, and 2 which have name from .txt
Passes in string from file
readInNodes(string fn) Calls Constructor, InsertVertex(), and SetEdge() to initialize Graph
Just initializes with Null
DisplayOptions() Main Menu
createNetwork(WT &Graph) Option 1 of menu and creates a graph, calls same class function as readInNodes()
WTGraph :: WeightedGraph() Deallocation, deletes everything within the graph
WTGraph:: getEdge(intt 1, int 2) Gets the weight between two vertex indexes
Vertex :: setKey() Set Key to the hop after recursively finding destination
Quit
Generate Network
Exit simulation
InsertEdge
Gives vertex hop
Vertex::getKey() Gets the hop and places it in BST accurately based on key
BSTinsert(string Vertex) Inserts a vertex object into binary search tree to save data
InsertVertex with string name
WTGraph:: hopCount(string v1, string v2, int count) Recursive function to count the number of hops from destination to a source router and all different possibilities.
Show hopCounts
User Input
Shows shortest path
Shows Shortest Path
WTGraph::showShortestPath() Option 2, which shows the distance a router needs to cover to reach any other router on the network
Shows all hopCount paths and distance
Add Router
Check Vertex Existence passes in name of vertex
Recursive Call
Shows messages given out
WTGraph :: sendTopologicalMessage(string v1) Option 3, which ask the user for router name and has it send out message to other neighbors. Ask for string for parameter.
Returns wt
Ask if wt is infinite
WTGraph :: getEdge(int, int) Gets the location of input router name, and the destination router name
Show the network
Simulate topological message transmission
checkVertex(string name) Not a function, but a for loop that depth checks if a vertex already been passed for hop count
addRouter() Option 6 Inserts a router and connects to another router with weight 20
Remove Router
WTGraph :: showStructure() Option 4, shows the structure of the graph and all of it's routers
Ask for check
Removes router from the network
Ask for wt
Returns wt
Insert Router
Remove Vertex from Graph
WTGraph:: isFull() Checks to see if graph is full
Returns boolean
WTGraph:: removeVertex(string v1) Removes a vertex from graph after retrieve
Returns bool
Check for node existence
WTGraph:: retrieveVertex Checks if vertex is in Graph and returns boolean.
Text
WTGraph :: insertVertex(string v1, vertex tmp) Inserts a router into the graph and attaches it to said dest router
Remove Edge after checking existence
Get input names
Edge Removed
WTGraph:: removeEdge(string v1, string v2) Removes edge in between that router and all other routers that are connected to it
WTGraph:: isEmpty() Checks to see if graph is empty so it does not crash
WTGraph:: insertEdge(string v1, string v2, 20) THe edge between those two vertices

Code explanation:

From the block diagram, my program starts out at readInNodes(), where it has already intialized BST object for the program and WeightedGraph object. Inside the function, it creates the vertexes by using the names it gained from the file as the names of the vertexes and the numbers from the file as the weight between those vertexes.

Essentially, it moves down to the main menu, which the user is to interact with. The user has up to 8 options, the last one being quit. These are the different options and everything to explain my code.

Option 1: Option one is to create a network from scratch, in other words, it deallocates the current graph object and all of its entries. It passes in a graph object to the function known as createNetwork, in which it loops for x amount of times, extracts two vertex name from the file, passes those names to insertVertex from the WeightedGraph, and then extracts a weight, in which it calls insertEdge to put the weight between those two vertices.

Option 2: Option two, shows shortest path by calculating each and every router to find a quicker path to another router, essentially a lot of for statements, and at the end just prints the values.

Option 3: Option 3 gets input from the user , which is then passed into sendTopologicalMessage, in which that specific router finds multipoint relays and has them broadcast more messages. It utilizes getEdge to make sure that the edge is not infinite, it also, creates variables to store router names that have already been accessed in which multiple broadcast are not done.

Option 4: Option 4 shows the user the structure of the network that includes the router. The '-' marks represent infinite weight. It gives indexes to getEdge, a class function for WeightedGraph, in which it returns the weight between those two vertices/routers.

Option 5: Option 5 removes a router by asking the user what they want to remove, in which that cin operation is passed into the WeightedGraph removeVertex() that requires a string for a parameter, in which it then calls removeEdge that finds every single vertex it is connected to by using its own name and have it remove it. If the network is empty, this does not let the user break the simulation.

Option 6: Option 6 adds a router to the network by asking the user two things, a new routers name and a router that already exist within the network, in which it then joins the two router by adding a distance of 20. The router names are passed by parameter to addVertex, class function for WeightedGraph. This option also calls isFull(), that checks if the graph cannot take anymore data.

Option 7: Option 7 checks for hopCounts. It does so by using a recursive algorithm that finds the hop count from one path to another, and essentially place that value into a BSTree Object. Essentially the BST holds all of the different ways for the source to reach the destination. It calls getEdge that passes the source and destination indices, and returns the weight to see if it is existent.

Option 8: Just quits the simulation

Results:

The results of my program is that it is able to calculate shortest path for each router to other routers. It is able to calculate hop count. It is able to remove and add routers to a network. It is also able to entirely simulate storing topological data so far for each router and storing. Essentially, my result is the capability to simulate OLSR.

Conclusion:

For the future, I would like to use all of the algorithms that I utilized for this program on real routers to see if it worked the way I simulated in this enclosed environment.