XI'AN JIAOTONG-LIVERPOOL UNIVERSITY
西 交 利 物 浦 大 学

REMOTE OPEN BOOK EXAM ANSWER SUBMISSION COVER SHEET

Name	Zhou	Wenhao
Student ID Number	1824099	
Programme	Information and Computer Science	
Module Title	Introduction to Networking	
Module Code	CAN201	
Module Examiner	Fei Cheng	

By uploading or submitting the answers of this Remote Open Book Exam , I certify the following:

- I will act fairly to my classmates and teachers by completing all of my academic work with integrity. This means that I will respect the standards and instructions set by the Module Leader and the University, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance.
- I have read and understood the definitions of collusion, copying, plagiarism, and dishonest use of data as outlined in the Academic Integrity Policy, and cheating behaviors in the Regulations for the Conduct of Examinations of Xi'an Jiaotong – Liverpool University.
- This work is produced all on my own and can effectively represent my own knowledge and abilities.

I understand collusion, plagiarism, dishonest use of data, submission of procured work, submission of work produced and/or contributed by others are serious academic misconducts. By uploading or submitting the answers with this statement, I acknowledge that I will be subject to disciplinary action if I am found to have committed such acts.

Signature: Wenhao Zhou Date: 1/2 2020

CAN201Introduction to Networking

Coursework2

Routing simulation

Report

Table of Content:

1. Abstract

The main function of network layer is to send packets from source machine to destination machine locating anywhere in the internet. The mechanism behind the function includes routing in routers: routers must decide the next neighbor to drop these packets, which contains path computing. And the routing algorithms is the key to implement the function. In this project, one of the routing algorithms, Bellman-ford, is simulated by python socket programming. Each .json file represents one router in the network and their distances to each other are maintained by objects in each file. After computing the distance vector, the shortest paths to each other will converge to optimized values which are written in the output .json file. In this report, method to implement Bell-man Ford algorithm via python socket programming is introduced, and testing method along with expected results are demonstrated.

2. Introduction

2.1 Background

Bellman-Ford algorithm, a dynamic computing method, is widely applied in routing algorithms. It is a graph-based searching algorithm that is capable of finding the shortest path between local vertex and the other vertexes in the whole graph connecting to each other. It can decide the shortest path by choosing the next neighbor to drop the packets without knowing the whole layout of the entire network system [1].

The Bellman-Ford algorithm operates over an given diagram, D, containing N nodes along with distance vector Dv between the node and its neighbors. Relaxation equation is the key concept in Bellman-Ford, which works by successively computing Dv by comparing it with other known Dvs held by its neighbors. If the newly computed Dv is smaller than currently hold Dv, update it. After its convergence, the minimum distance to other nodes are found.

Below shows the relaxation equation:

```
def relax(u, v):
    if localdict[key] > peerdict[nodename] + peerdict[key]:
    localdict[key] = peerdict[nodename] + peerdict[key]
    nexthop = peernode
    distance = peerdict[nodename] + peer_dict[key]
```

Bellman-Ford algorithm is recognized as decentralized algorithm for it computes the least-cost path from source node to other peer nodes in a repeated behavior. In this algorithm, there is no need to maintain the knowledge of the distance vector of the whole network nodes, every node can only get to know the information of its neighbor nodes at the beginning. After iterating through all the neighbors' information and updating the local distance vector, local node can know the existence of other node in the network, meanwhile, it knows the direction through which neighbor the packet is to be dropped to obtain the least cost path.

2.2 Project Requirement

In this project, python socket programming is applied to implement the algorithm. Each node is represented by an application containing its own neighbor information (IP and port number) and distance vector to its neighbors. These two information are maintained in two different `.json` files which are respectively `nodename_distance.json` and `nodename_ip.json`. Each node can load its own information and communicate with other nodes via UDP socket. After connection, they can exchange their distance vector asynchronously and recompute their local distance vector if there are any shorter distance through its neighbor to any of the other nodes in the graph. After convergence of the local distance vector which means that no nearer path to other nodes are found, each node can output its optimized distance vector to a new json file called `nodename_output.json`.

2.3 Literature Review and Project Scope

In major routing algorithms, there are majorly two generics of them which are respectively decentralized and centralized algorithms. Centralized models are the routing method in which the whole network information is maintained in the routing table of routers. Every node get to know the global view of the network. On the contrary, decentralized algorithms are the ones that maintain distributed routing database. Each node communicates with other peers through exchanging their own database, and they decide which way to drop packets by judging the currently maintained routing database [2].

Centralized system is easy to protect the client information and server information physically through location virtualization. Also, it is capable of updating information taking the advantage of globally shared comprehensive information. However, centralized system cannot scale up after reaching its maximum capability of carrying out efficient calculation. Usually, a network contains thousands of routers, it is exhausted to record every information of these nodes. Thus, the performance of centralized routing system is constrained by the scale of network system. Also, its bottleneck lies when the traffic summit comes. The router has limited port numbers to which they can make connections form other nodes, as a consequence, when the number of incoming clients surpasses its workload, it suffers from packets loss.

On the other hand, decentralized system can be scaled up to a enormous volume. Each node only have to possess its own neighbor information rather than maintain the whole system information, thus, it has more autonomy to utilize more resources comparing to centralized systems. However, it is difficult to coordinate collective missions through all the nodes, due to independent behavior of each node.

Bell-man Ford algorithm is one of the decentralized algorithms. It is distributed over each node in the network, and the behavior of each node is unique. In this project, Bell-man Ford routing algorithm is simulated by using python socket programming, UDP protocol is applied to carry out communication between each node. Each node has its own neighbor distance vector information and neighbor IP information. Nodes can be started without constraint of time and sequence, finally, their local distance vector can convergent to a optimized value.

3. Methodology

3.1 Proposed functions and ideas

There are 4 functions in this project, the usage of each function is listed below:

Function name	Usage
listen*to*news*from*neighbors()	Listen to connection requests of other neighbors and update local distance vector, send news to neighbors if nearer distance is found
update*news*to*neighbors(address, this*node, dv)	Update local distance vector to peers
_argparse()	Parse parameter from external execution
main()	Read IP information and local distance information from `.json` file

Table 1: Functions

Five global variables are used in this application:

Variable name	function
local_dict	A dict containing local distance vectors to be updated
org*local*dict	A dict containing local distance vectors without updating
node_name	This node's name
neighbor_addr	A list containing neighbor's addresses
output_dict	A dict used to maintain distance vectors to other nodes

Table 2: global variables

3.2 Proposed protocols

In this application, Iterative communication between nodes is required as a consequence of distributed architecture. Once a node is online, it firstly read its own local distance vector and its own neighbor information stored in `nodeName_distance.json` and `nodeName_ip.json`. Then, it informs other neighbor that it is online and exchange its distance vector information with neighbors. Meanwhile, it listens to news from neighbors, if there are distance vector information sent from other neighbors, if firstly inspect whether there are nodes local neighbor never maintain, and it adds the foreign nodes in neighbor's distance vector to its own distance vector. After that, it uses relaxation function to decide whether the local distance vector should be updated. Following that, if updating happens during the comparison with neighbor's distance vector, it resend its updated distance vector to its neighbors and update the output dict. Finally, if the result converges or time out, which means that there are no any distance updating in the network, the program will quit and write output dict into `nodeName_output.json`.

The finite state machine is shown below:

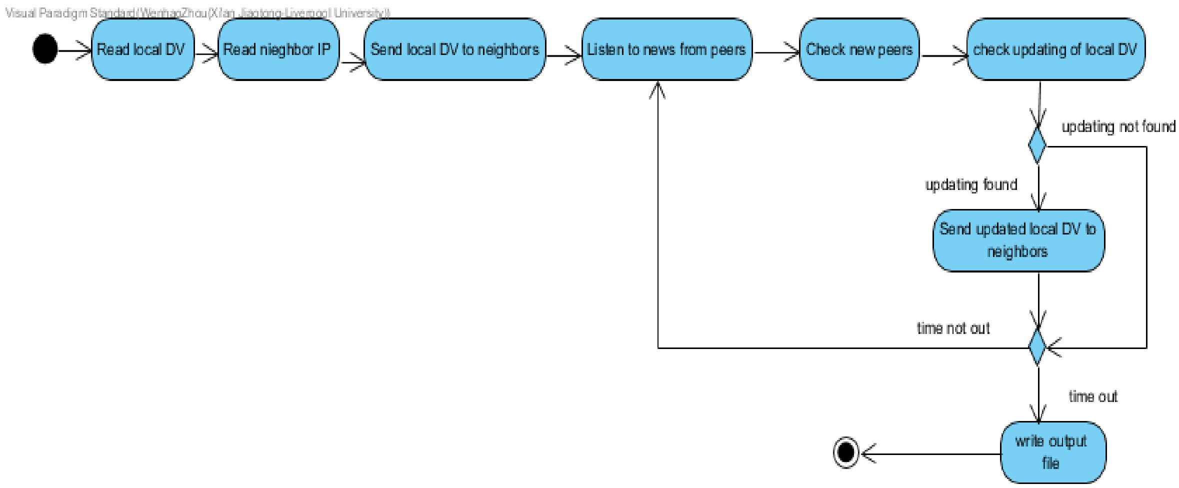


Image 1: FSM of the Program

4. Implementation

4.1 Steps of Implementation

1. File reader and information sender:

   To read in `.json` file, `json` module is imported to operate this part of function. `local_dict` is used to maintain DV information to be updated, `org_local_dict` is used to store original DV information without updating, and `neighbor_addr` is used to maintain neighbor address information. The format of `json` data is converted to dict type to perform better operation.

   To send and receive information to other nodes, one UDP socket is initialized at the beginning of the program. To fulfill the function of update news to neighbors, iteration through the neighbor address is required to send local DV sequentially to neighbors. Thus, `update_news_to_neighbors(addresses, this_node, dv)` takes in three parameters which respectively represent addresses of neighbors, name of this node and distance vector of local node.

2. Information listener

   To receive updating message coming from other neighbors and refresh local distance vector, a loop with time out constraint is required. During the time out period which is sufficient enough for updating process to complete, local DV refreshing and transmission is implemented.

   The listener tries to receive news from peers, otherwise, it enters exception part where count down is in process. If the count down has surpassed time out limit, output file will be written. If there incomes neighbor's information, it firstly parse information to get peer node name `peer_node` and peer distance vector `peer_dv`. Then, it checks if there are new node in neighbor's DV that local DV does not maintain, and add the newly found node to `local_dict` and set the distance from local node to this newly found node to infinite. After that, it iterate through all the neighbor nodes in `local_dict` to apply relaxation function to check if the distance between this node and currently checked neighbor is greater than the sum of the distance from peer node to this node and the distance from peer node to currently checked neighbor. If the constraint is met, then it updates the `next_hop` to peer node and update the distance from local node to currently checked neighbor to nearer distance via peer node. Following the updating, it then send update news to neighbors. The above demonstrated process continues until time out, finally, it will write the converged output file. The whole process is demonstrated as image 2.

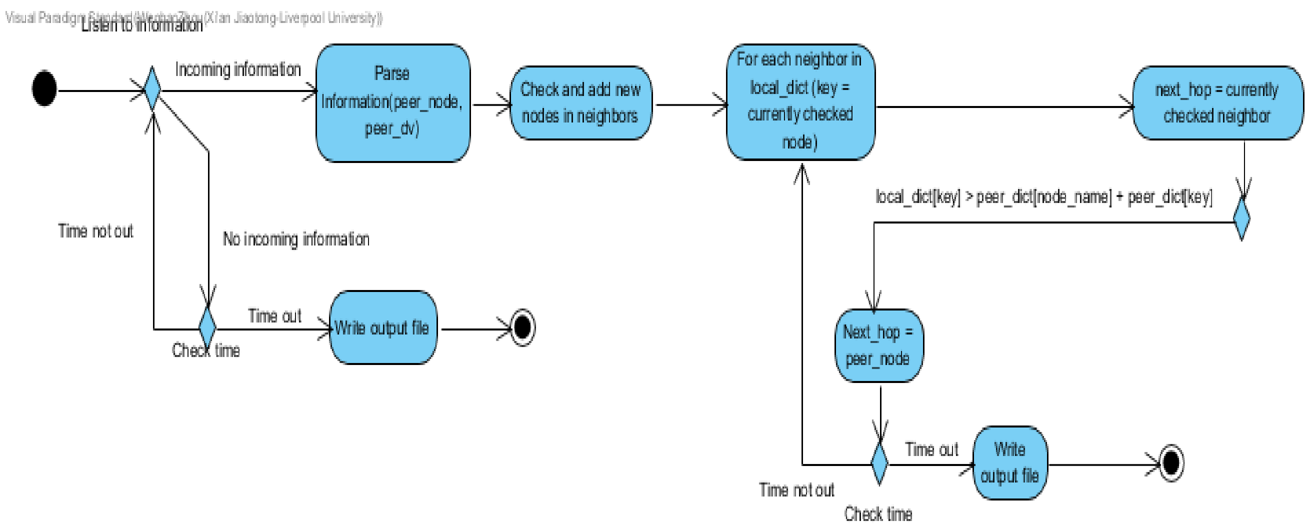


Image 2: FSM of File Listener

4.2 Programming skills

1. Distributed application

As the project required, each node only contain its own neighbor information, the nodes that are not directly connect with the local node cannot be reached at the beginning. Thus, each application is distributed on different hosts which are simulated by different port number assigned to each node's socket. After rounds of information exchange, every node get to know all the other nodes' information by its original connected nodes. In the real life, the situation is similar, each router does not know the whole structure of network, instead, they only maintain the forwarding table of its neighbors'. However, they get to know the next hop to drop packets which costs least to its destination.

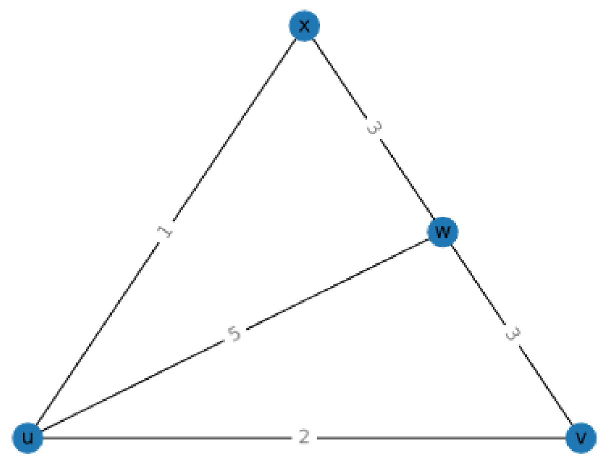
2. Recursively Information Change

Each application sends its updated local distance vector to its neighbors if there are any cheaper cost to the destination which is a recursive process rendered in the information listening process. And the convergence of distance vector is indicated by halt of the updating process.

4.3 Difficulties Met and Solutions

5. Testing and Results

1. To test the application with proper environment, Linux-based operating system (Tiny Core) and python 3.6 are required to run the code. To simulate the real word operating condition, a node network containing 4 points is firstly tested. The graph is demonstrated below:



model1

Graph 1:Test case1 four Points Network

1. one `main.py` is utilized by `simultaneously-run.py` python script to run several times and each execution acts like a real router. The content of `simultaneously-run.py` is shown below:

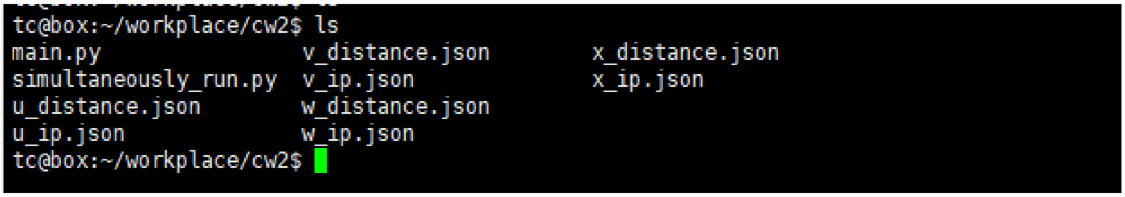
```python
import os
import threading


def run_file(file):
    os.system("python3 main.py --node " + file)


if __name__ == '__main__':
    files = ["u", "v", "w", "x"]
    for file in files:
        bellman = threading.Thread(target=run_file, args=(file,))
        bellman.start()
```

Image 3: Simultaneously-run.py

1. Then, the running environment is prepared by transferring files to virtual machine via Xftp. It is shown below by image 4.



```
tc@box:~/workplace/cw2$ ls
main.py                v_distance.json        x_distance.json
simultaneously_run.py  v_ip.json              x_ip.json
u_distance.json        w_distance.json
u_ip.json              w_ip.json
tc@box:~/workplace/cw2$
```

Image 4: Working Directory

1. After execution command:

    ```
    python3 simultaneously_run.py
    ```

    `main.py` will be executed four times with different external parameters which are respectively :

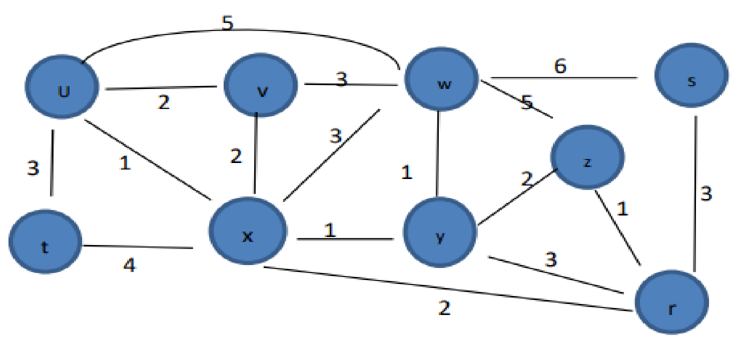
    ```
    pyhton3 main.py --node u
    pyhton3 main.py --node v
    pyhton3 main.py --node w
    pyhton3 main.py --node x
    ```

    After the convergence of result, each application's output is generated as `nodeName_output.json`. The output is demonstrated below:

```
tc@box:~/workplace/cw2$ ls
main.py                v_distance.json        w_output.json
simultaneously_run.py  v_ip.json              x_distance.json
u_distance.json        v_output.json          x_ip.json
u_ip.json              w_distance.json        x_output.json
u_output.json          w_ip.json
tc@box:~/workplace/cw2$ cat u_output.json
{"v": {"distance": 2, "next_hop": "v"}, "w": {"distance": 4, "next_hop": "x"}, "x": {"distance": 1, "next_h
op": "x"}}tc@box:~/workplace/cw2$ cat v_output.json
{"u": {"distance": 2, "next_hop": "u"}, "w": {"distance": 3, "next_hop": "w"}, "x": {"distance": 3, "next_hop": "u"}}tc@box:~/wo
rkplace/cw2$ cat w_output.json
{"v": {"distance": 3, "next_hop": "v"}, "x": {"distance": 3, "next_hop": "x"}, "u": {"distance": 5, "next_hop": "u"}}tc@box:~/wo
rkplace/cw2$ cat x_output.json
{"u": {"distance": 1, "next_hop": "u"}, "w": {"distance": 3, "next_hop": "w"}, "v": {"distance": 3, "next_hop": "u"}}tc@box:~/wo
rkplace/cw2$
SSH2   xterm   128x37   37,14   1 session   CAP  NUM
```

Image 5: Output of 4 Nodes

2. After Testing four nodes with reliable performance, a network with nine nodes is tested in the same environment to examine the reliability of the system. Another five points: r, s, t, y, z are introduced into the original graph. The graph of 9 points is shown below:



Graph 2: Test case2 nine points network

3. Then, work space of nine points is implemented by adding other distance vectors and other IP information. The updated work space is shown below:

```
tc@box:~/workplace/cw2$ ls
main.py                t_ip.json          x_distance.json
r_distance.json        u_distance.json    x_ip.json
r_ip.json              u_ip.json          y_distance.json
s_distance.json        v_distance.json    y_ip.json
s_ip.json              v_ip.json          z_distance.json
simultaneously_run.py  w_distance.json    z_ip.json
t_distance.json        w_ip.json
```

Image 6: Work Space of Nine Points

4. After starting `simultaneously-run.py` the output is shown below:

```
tc@box:~/workplace/cw2$ cat r_output.json
{"z": {"distance": 1, "next_hop": "z"}, "y": {"distance": 3, "next_hop": "y"},
 "s": {"distance": 3, "next_hop": "s"}, "x": {"distance": 2, "next_hop": "x"},
 "w": {"distance": 4, "next_hop": "y"}, "u": {"distance": 3, "next_hop": "x"},
 "v": {"distance": 4, "next_hop": "x"}, "t": {"distance": 6, "next_hop": "x"}}
tc@box:~/workplace/cw2$ cat s_output.json
{"r": {"distance": 3, "next_hop": "r"}, "w": {"distance": 6, "next_hop": "w"},
 "z": {"distance": 4, "next_hop": "r"}, "y": {"distance": 6, "next_hop": "r"},
 "x": {"distance": 5, "next_hop": "r"}, "u": {"distance": 6, "next_hop": "r"},
 "v": {"distance": 7, "next_hop": "r"}, "t": {"distance": 9, "next_hop": "r"}}
tc@box:~/workplace/cw2$ cat t_output.json
{"x": {"distance": 4, "next_hop": "x"}, "u": {"distance": 3, "next_hop": "u"},
 "v": {"distance": 5, "next_hop": "u"}, "w": {"distance": 6, "next_hop": "u"},
 "y": {"distance": 5, "next_hop": "u"}, "z": {"distance": 7, "next_hop": "u"},
 "s": {"distance": 9, "next_hop": "u"}, "r": {"distance": 6, "next_hop": "u"}}
tc@box:~/workplace/cw2$ cat u_output.json
{"v": {"distance": 2, "next_hop": "v"}, "w": {"distance": 3, "next_hop": "x"},
 "x": {"distance": 1, "next_hop": "x"}, "t": {"distance": 3, "next_hop": "t"},
 "y": {"distance": 2, "next_hop": "x"}, "z": {"distance": 4, "next_hop": "x"},
 "s": {"distance": 6, "next_hop": "x"}, "r": {"distance": 3, "next_hop": "x"}}
tc@box:~/workplace/cw2$ cat v_output.json
{"u": {"distance": 2, "next_hop": "u"}, "w": {"distance": 3, "next_hop": "w"},
 "x": {"distance": 2, "next_hop": "x"}, "y": {"distance": 3, "next_hop": "x"},
 "z": {"distance": 5, "next_hop": "x"}, "s": {"distance": 7, "next_hop": "x"},
 "t": {"distance": 5, "next_hop": "u"}, "r": {"distance": 4, "next_hop": "x"}}
tc@box:~/workplace/cw2$ cat w_output.json
{"v": {"distance": 3, "next_hop": "v"}, "x": {"distance": 2, "next_hop": "y"},
 "u": {"distance": 3, "next_hop": "x"}, "y": {"distance": 1, "next_hop": "y"},
 "z": {"distance": 3, "next_hop": "y"}, "s": {"distance": 6, "next_hop": "s"},
 "t": {"distance": 6, "next_hop": "x"}, "r": {"distance": 4, "next_hop": "y"}}
tc@box:~/workplace/cw2$ cat x_output.json
{"u": {"distance": 1, "next_hop": "u"}, "w": {"distance": 2, "next_hop": "y"},
 "v": {"distance": 2, "next_hop": "v"}, "t": {"distance": 4, "next_hop": "t"},
 "y": {"distance": 1, "next_hop": "y"}, "r": {"distance": 2, "next_hop": "r"},
 "z": {"distance": 3, "next_hop": "y"}, "s": {"distance": 5, "next_hop": "r"}}
tc@box:~/workplace/cw2$ cat y_output.json
{"w": {"distance": 1, "next_hop": "w"}, "x": {"distance": 1, "next_hop": "x"},
 "z": {"distance": 2, "next_hop": "z"}, "r": {"distance": 3, "next_hop": "r"},
 "s": {"distance": 6, "next_hop": "r"}, "u": {"distance": 2, "next_hop": "x"},
 "v": {"distance": 3, "next_hop": "x"}, "t": {"distance": 5, "next_hop": "x"}}
tc@box:~/workplace/cw2$ cat z_output.json
{"w": {"distance": 3, "next_hop": "w"}, "y": {"distance": 2, "next_hop": "y"},
 "r": {"distance": 1, "next_hop": "r"}, "v": {"distance": 6, "next_hop": "w"},
 "x": {"distance": 5, "next_hop": "w"}, "u": {"distance": 6, "next_hop": "w"},
 "s": {"distance": 9, "next_hop": "w"}, "t": {"distance": 9, "next_hop": "w"}}
tc@box:~/workplace/cw2$
```

Image 7: Output of Nine Points Network

As the outputs of each node demonstrated above, it is obvious that all the entities in the output `.json` files are the same if checking the results through the graph above. Thus, the reliability of the code is guaranteed.

6. Conclusion

In this project, Bellman-Ford algorithm is simulated through python socket programming. Each router is represented by an unique node containing its own neighbor distance vector and neighbor IP information. Apparently, nodes are distributed and decentralized into different application. They can transfer its own information with other nodes recursively and update their own distance vector by using Relaxation Function. After the convergence which means that no shorter distances are found through neighbors, the optimal choices to drop packets to all the other nodes is found.

Reference

[1]. A. Chumbley, K. Moore, E. Ross, J. Khlm. (2012, Dec. 12). *Bellman-Ford Algorithm* [Online]. Available:

https://brilliant.org/wiki/bellman-ford-algorithm/#:~:text=The%20Bellman-Ford%20algorithm%20is%20a%20graph%20search%20algorithm,be%20used%20on%20both%20weighted%20and%20unweighted%20graphs

[2]. Indika. (2011, June. 6). *Difference Between Centralized Routing and Distributed Routing* [Online]. Available:

https://www.differencebetween.com/difference-between-centralized-routing-and-vs-distributed-routing/