Michaela Hay
1001649623
11/20/2021
CSE 4344-001

Libraries Used: **time and numpy**

Python version: **3.8.5**

How to Compile:

With the Makefile:

make run

Without the Makefile:

python3 dv.py

For reference, I programmed this on Windows OS using Anaconda's Spyder IDE. I also ran this code in my Virtual Machine and it executed as well. This project should be able to compile on any machine as long as it has Python version 3.8.5

Possible errors that can be encountered:

If the input file of the routers skips numbers from 1 to N. Example being:

3 5 10

2 4 1

7 9 7

This skips router numbers 1, 6, and 8 so this code will likely get indexed incorrectly in my algorithm. I approach more into why this happens later in this documentation.

How I approached GUI Requirements:

These requirements a met via console output. Each node's DV table as soon as it's updated based on its neighbors' tables will be printed out to the console with its DV table denoted by its router number.

Example Output:

Michaela Hay
1001649623
11/20/2021
CSE 4344-001

```
DV for node 1:
[[ 0.  7. 16. 16.  1.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]]
```

```
DV for node 2:
[[16. 16. 16. 16. 16.]
 [ 7.  0.  1. 16.  8.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]]
```

The rows and columns of the matrix represent the number of the routers. To go into detail, for this current example, there are five columns that represent sequentially all the routers, and the same goes for the number of rows. So for node 1, its initial DV table will have 0 as the cost in column one, row one. For node 2 its DV table will have 0 as the cost for column two, row two.

This means my code depends on the routers listed in the .txt file to appear from 1 to N. I haven't tested this yet, but there is a likely chance my code can fail if the routers are random numbers instead of sequential ones. I depend on indexing a matrix in order of nodes present, so it goes by logic that it will section columns and rows by order of 1 to N for router numbers.

GUI also allows for user input.

```
computer@ubuntu-vm:~/Downloads/Computer-Network-main/Project 2$ make run
python3 dv.py
Enter name of file:
graph.txt
```

It will start by asking for the file input. Type the file name and click enter, and it will immediately make the graph for the text file if no error occurs.

```
Mode:
[1] By iteration
[2] Live output
```

It will then ask if you want to see it iteration by iteration, or if you'd rather see it spit out results live. If you go by the second choice, you won't be able to adjust the cost of links between routers.

Michaela Hay
1001649623
11/20/2021
CSE 4344-001

You will see the total time (in seconds) it took for the program to finish and how many cycles it took as well.

```
 [3. 3. 2. 0. 2.]
 [1. 5. 4. 2. 0.]]


DV for node 5:
[[0. 6. 5. 3. 1.]
 [6. 0. 1. 3. 5.]
 [5. 1. 0. 2. 4.]
 [3. 3. 2. 0. 2.]
 [1. 5. 4. 2. 0.]]


DV for node 4:
[[0. 6. 5. 3. 1.]
 [6. 0. 1. 3. 5.]
 [5. 1. 0. 2. 4.]
 [3. 3. 2. 0. 2.]
 [1. 5. 4. 2. 0.]]


Program took 0.0076215267181396484 seconds to execute.
System took 20 cycles to reach stable state.
```

For seeing it by iteration, the console looks like this:

```
|==== Print Tables ====|


DV for node 1:
[[ 0.  7. 16. 16.  1.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]]
==============
[Enter] to Continue
[1] Change Cost
[2] Stop
==============
```

Press Enter to either continue iteration by iteration, or 1 to change the cost of any graph by putting in a source and destination alongside new cost link. There's an option to Stop, but it's a little fickle and doesn't work as planned.

This is what you will see when you change link costs.

```
Enter source node: 2
Enter destination node: 1
Enter cost between nodes: 4
DV for node 2:
[[16.  4. 16. 16. 16.]
 [ 4.  0.  1. 16.  8.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]
 [16. 16. 16. 16. 16.]]
==============
```

Note that when the stable state is reached, you can no longer change the costs of the links and it will no longer accept input from the console. It will print out the stable matrices and total cycles, and the program will exit out.

DV information is passed around by neighbors by keeping track of neighboring nodes to the current node. I have an array whose indices represent each node, and in each index is a list of its neighbors. This is also why my program likely won't work with a file whose router numbers skips around randomly, leaving gaps in a sequential order.

All DVs are numpy matrices which are part of a global list called A. A has within it all the router DV tables.

Observations:

With a really simple input file, the program executes extremely fast. For the given example input file:

1 2 7

2 3 1

1 5 1

4 5 2

2 5 8

4 3 2

This only takes 20 cycles to reach a stable state.

Michaela Hay
1001649623
11/20/2021
CSE 4344-001

For changing the link cost of one node to another to be 16, this is the output:

```
Enter source node: 4
Enter destination node: 3
Enter cost between nodes: 16
DV for node 5:
[[ 0.  9.  5.  3.  1.]
 [ 9.  0.  1.  3.  5.]
 [10.  1.  0. 16.  4.]
 [ 3.  3.  2.  0.  2.]
 [ 1.  5. 16.  2.  0.]]
```

It took a few more cycles for the infinite link cost to be adjusted back to a stable number as the DV tables have to exchange communication all over again to ensure that there's a shorter route than infinity between router three and four.

```
DV for node 3:
[[ 0.  6.  5.  3.  1.]
 [ 6.  0.  1.  3.  5.]
 [ 5.  1.  0.  4.  4.]
 [ 3.  3.  2.  0.  2.]
 [ 1.  5. 16.  2.  0.]]
==============
[Enter] to Continue
[1] Change Cost
[2] Stop
==============
```

This is the table after the "fix" happens and the DVs can reconcile a shorter path.

```
System took 25 cycles to reach stable state.
```

It took five more cycles for this to be corrected.

```
Program took 0.0029137134552001953 seconds to execute.
```

This is how long it takes for the program to run when the cost link is not manipulated by the user.

For simulating a line repair this is what happens.

Michaela Hay
1001649623
11/20/2021
CSE 4344-001

```
Enter source node: 5
Enter destination node: 4
Enter cost between nodes: 16
DV for node 1:
[[ 0.  7.  5.  3.  1.]
 [ 7.  0.  1.  3.  8.]
 [16.  1.  0.  2. 16.]
 [16. 16.  2. 16.  2.]
 [ 1.  8.  4.  2. 16.]]
==============
[Enter] to Continue
[1] Change Cost
[2] Stop
==============
```

```
Enter source node: 5
Enter destination node: 4
Enter cost between nodes: 2
DV for node 2:
[[ 0.  7.  5.  3.  1.]
 [ 7.  0.  1.  3.  5.]
 [ 8.  1.  0.  2.  4.]
 [16. 16.  2.  2.  2.]
 [ 1.  8.  4.  2.  2.]]
==============
[Enter] to Continue
[1] Change Cost
```

Immediately after the repair, the infinities are almost instantaneously purged out from all the node graphs and they're quick to find the shortest path again. Versus when I left it at sixteen without a repair, I could see how the routers were quick to pass the shortest path information and propagate it to the stable state versus how leaving it at infinity made it perform a little slower. On a grander scale this would be an extremely noticeable discrepancy in performance. "Good" information passes quicker than "bad" information. Bad information being a line break of 16.