Modifying Dijkstra's Algorithm for Use in Hybrid Access Networks

Amber Hankins, Nikhil Sharma
University of Nevada, Reno
CPE 400 Final Project
Email: amberhankins@nevada.unr.edu, nikhilsharma@nevada.unr.edu

***Abstract*—Hybrid access networks employ multiple different communication channels, allowing users to connect in a number of ways. While this can increase end-to-end transmission speeds, these networks must account for differences between distinct communication technologies. In a network composed of both wireless and wired connections, it is likely that wireless connections are less reliable than their wired counterparts. As a result, we aim to develop a modified Dijkstra's routing algorithm that is biased towards wired connections, as they are less likely to fail. Throughout this project, we create a hybrid access network in Python. We develop a modified version of Dijkstra's algorithm that is biased toward wired connections. We simulate the performance of this algorithm in our hybrid access network, with random node failure. Finally, we analyze the performance of this modified algorithm in comparison to the original Dijkstra's. Performance is measured in two metrics: loss and path length.**

I. Network Simulation

A. Network Structure

The network is initially created using Python's built-in dictionary data structure. This is denoted by the definition of "graphDictionary", using curly brackets to define a dictionary. The keys of the dictionary are represented by the various nodes on the network, stored within our list "nodes".

```
nodes = ["A", "B", "C", "D"]
graphDictionary = {}
for node in nodes:
    graphDictionary[node] = {}
```

Now that our dictionary has keys for each node, we must set values for each edge that exists on our network. These values will represent both the weight of the edge and the type of connection.

```
graphDictionary["A"]["B"] = [5, 'wireless']
graphDictionary["A"]["D"] = [4, 'wired']
graphDictionary["B"]["C"] = [3, 'wired']
```

Now our network is complete with nodes and weighted edges. In this example, we supply the edges with concrete values. However, in testing the performance of our routing algorithms we assign random values to these edge weights. Using this dictionary, we create an object using our custom Graph class. This allows us to easily pass our network as an object, and use our useful functions returnNodes(), returnEdges(), and returnValue().

The final aspect of our simulated network is node failure probability. When displaying the paths generated by Dijkstra's and modified Dijkstra's algorithms, the program randomly generates failure for each node in the path. Wireless nodes have a 1 in 5, or 20% chance of failure, and wired nodes have a 1 in 20, or 5% chance of failure. These numbers were supplied in order to simulate the real world disparity between wired and wireless connection reliability. However, the program is built flexibly so that different probabilities can be inserted here.

```
for i in range(0, len(path)-1):
    if type == "wireless":
        randomNum = random.randint(0,4)
        if randomNum == 0:
            raise Exception("Wireless  failure")
    else:
        randomNum = random.randint(0, 19)
        if randomNum == 0:
            raise Exception("Wired failure")
```

The more wireless connections a path contains, the more likely node failure is to occur. As a result, we predict that the modified algorithm biased towards wired connections will generate longer path lengths, but have a lower rate of node failure in comparison to the original Dijkstra's algorithm.

B. Network Visualization

The network graph is visualized using the pyvis library. It can be installed using the

```
pip install pyvis
```

command assuming the user has the other dependencies installed. From there the library is imported into the program to use. Using the Network interface, some common functions include:

```
add_node()
```

This function can take many parameters to specify for example value, label, color, and id. This same functionality may also be abstracted with the function

```
add_nodes()
```

This function allows for creating nodes in bulk. From here, the

`add_edge()`

function can be used to connect the nodes with edges. Similarly, a user can set the value, label, color and id as well. Finally, the

`show()`

function can be called with parameters for specifying the file name to output the graph to.

The visualization is handled in the generateVisual() function. This function takes the graph dictionary and the nodes as the parameter. From there, the program iterates through the nodes to add nodes to the graph.

```
for node in nodes:
        net.add_node(node, label=str(node))
```

Another nested iteration is done to add the edges between the nodes.

```
for edge in graphDictionary:
    for data in graphDictionary[edge]:
        net.add_edge(str(edge),
        str(data), value=2,
        label=str
        (graphDictionary[edge][data][0])
        + ' ' +
        graphDictionary[edge][data][1])
```

This process has been abstracted to handle any user graph. Finally, the graph is shown in a file called nodes.html.

```
net.show('nodes.html')
```

After copying the path to the generated file, the user may open the file in any web browser without the need of a live web server. The library allows for a user friendly graphic for the graph.

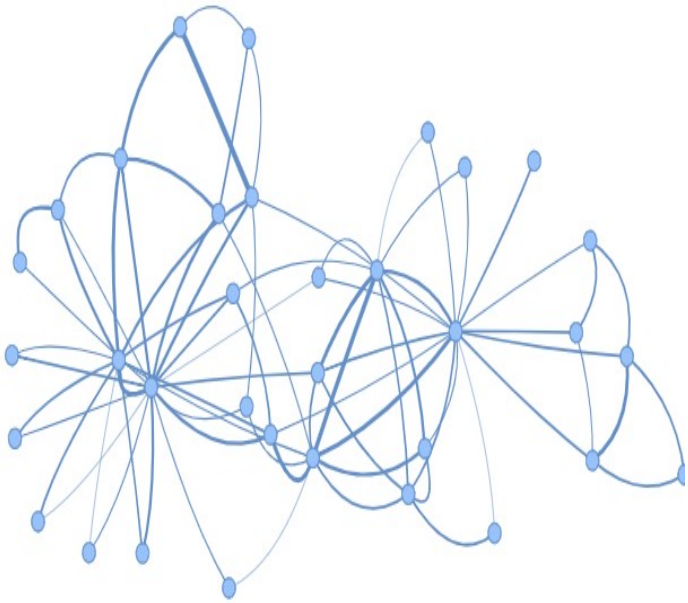

Fig. 1. Generic graph generated using the pyvis library.

II. Dijkstra's Algorithm

A. Algorithm Creation

The program simulates Dijkstra's Algorithm within the dijkstra() function. The goal of this algorithm is to return the shortest path to each node, given a starting node. It begins by initializing every node's path to the highest possible value, then initializing the path to the starting node as 0.

```
for node in unvisitedNodes:
    shortestPathValues[node] = sys.maxsize

shortestPathValues[firstNode] = 0
```

While there are still nodes to visit in the network, the program determines the current closest node. The program checks the path length to each of this node's neighbors. If this new path traveling through the closest node is shorter than the previously calculated shortest path, the new path replaces the old one. Finally, we pop the closest node off the stack in order to move down through the network.

This dijkstra() function returns two different dictionaries. One dictionary yields the path lengths to each node from the starter node. The other yields the previous node in the path to each node. Both of these dictionaries, yielded from the graph constructed in the previous section, are shown below. The dictionaries represent the graph pictured in Fig. 2.

Path Length: ['A': 0, 'B': 5, 'C': 8, 'D': 4]
Previous Node: ['B': 'A', 'D': 'A', 'C': 'B']

In this example, the first dictionary tells us that the path from A to A is 0, the path from A to B is 5, the path from A to C is 8, and the path from A to D is 4. The second dictionary tells us that the paths to B and D come directly from A, and the path to C goes through node B.

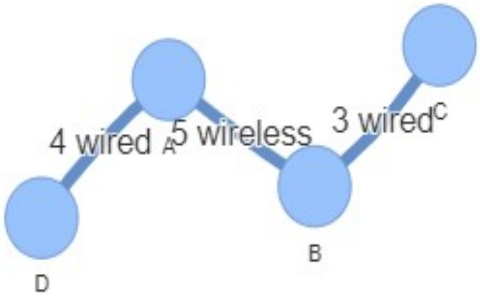

Fig. 2. Simple graph defined in Section I.

B. Algorithm Modification

The program modifies Dijkstra's Algorithm within the dijkstraMod() function. The goal of this algorithm is to return the shortest path to each node, given a starting node while also accounting for wired and wireless nodes. It begins similarly to Dijkstra's by initializing every node's path to the highest possible value, then initializing the path to the starting node as 0.

While there are still nodes to visit in the network, the program determines the current closest node. The program checks the path length to each of this node's neighbors. If this new path traveling through the closest node is shorter than the previously calculated shortest path and is a wired node, the new path replaces the old one. If the node was not wired, the program checks the path length to each of this node's neighbors. If this new path traveling through the closest node is shorter than the previously calculated shortest path, the new path replaces the old one. Finally, we pop the closest node off the stack in order to move down through the network.

```
if newPathLength <
    shortestPathValues[neighbor][0]
    and transmitMode == "wired":
```

This dijkstraMod() function returns two different dictionaries. One dictionary yields the path lengths to each node from the starter node. The other yields the previous node in the path to each node. Both of these dictionaries are shown below. The visualization of this graph is shown in Fig. 3.

Path Length: 'A': [0, ''], 'B': [3, 'wireless'], 'C': [3, 'wired'], 'D': [5, 'wired'], 'E': [5, 'wired'], 'F': [6, 'wired']

Previous Node: ['B': 'A', 'C': 'A', 'D': 'B', 'E': 'C', 'F': 'D']

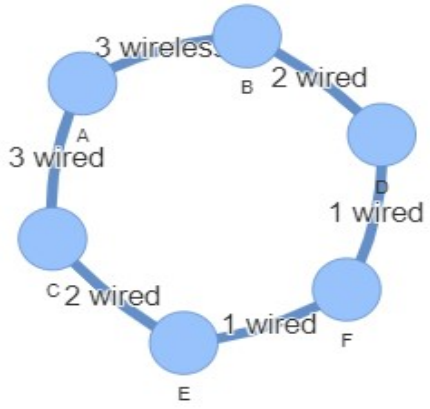

Fig. 3. Graph representing the dictionaries listed above.

In this example, the first dictionary tells us that the path from A to A is 0, the path from A to B is 3 with transmission mode wireless, the path from A to C is 3 with transmission mode wired, and the path from A to D is 5 with transmission mode wired and so on. The second dictionary tells us that the paths to B and C come directly from A, and the path to D goes through node B and so on.

III. Error Handling Scenarios

The primary use of error handling in this project was to account for node failure. Within our display() function, the failure of wireless and wired nodes is calculated randomly. Whenever node failure occurs, the program raises an exception to stop moving along the path.

```
raise Exception("Wireless node has failed")
```

This is done to prevent unnecessary computations from occurring past the point of node failure. The path is halted and does not reach it's destination. Our other case of error handling occurs in our main function. Since both algorithms have a chance of raising the previously described exception, main must handle these exceptions with a try-except statement.

The main function tries to execute display() using the path calculated by Dijkstra's algorithm. If the node failure exception is raised, main prints out a failure notice. Then it continues on to repeat this process for the path calculated by the modified algorithm. This way, if one algorithm fails, the program can continue simulating the performance of the other algorithm without error.

```
try:
    display(...)
except:
    print("Dijkstra's Algorithm Node Failure")
try:
    display(...)
except:
    print("Modified Dijkstra's Node Failure")
```

IV. Results and Analysis

We tested both the original Dijkstra's Algorithm and our modified version against a network of 8 nodes, A through H. In each trial, the weights and connection types of each edge in the network were randomized. The algorithms were tested on identical networks, with the same source node and destination node. Our hypothesis for this experiment was that the original Dijkstra's algorithm would have a much lower average path length than the modified version, but a much higher fail rate as well. We believe that the modified algorithm's avoidance of risky wireless connections may result in longer average paths, but generally more reliable communication.

Below are the results from 30 trials of random network generation. The values represent length of the path generated, and a null value indicates node failure.

Trial	dijkstra()	dijkstraMod()
1	5	5
2	4	4
3	5	5
4	10	11
5	null	null
6	5	5
7	3	3
8	null	null
9	9	13
10	11	11
11	9	11
12	null	2
13	null	5
14	null	8
15	4	4
16	8	null
17	null	null
18	1	null
19	8	10
20	1	1
21	4	null
22	3	3
23	null	12
24	2	2
25	null	10
26	null	null
27	4	4
28	3	18
29	9	9
30	null	6

The original Dijkstra's algorithm did have a lower average path length of 5.4, compared to the modified version's 7.04. The original also had a higher failure rate than the modified version, with 33% compare to 23%.

.	dijkstra()	dijkstraMod()
Avg. Path Length	5.4	7.04
Failure Rate	33%	23%

While the general trends do match our original hypothesis, we thought the difference in failure rate would be much larger. However, avoiding wireless nodes appears to not have as great an impact on reliability as we originally thought. This is potentially due to the modified algorithm's longer path lengths. The modified algorithm encounters more nodes on average than the original, since it circumvents the most direct path if it contains wireless connections. As a result, the modified algorithm may have a higher chance to encounter a faulty node due to it's indirect path.

V. Conclusion

The results of this project confirm our initial hypothesis to a certain degree. Our modified algorithm had longer path lengths on average, but generally encountered less node failure. However, the failure rate of the modified Dijkstra's Algorithm is closer to the original failure rate than we originally anticipated.