CPE 400: Computer Communication Networks

Hybrid Dynamic Routing Mechanism Design

Joseph Trierweiler

December 6th, 2021

Abstract

The goal for this experiment was to implement a routing protocol that would be able to dynamically route a network for increasing throughput and longevity. The way we were meant to demonstrate this is via a mobile sensor network and a Command and Control base (C&C). The protocol that immediately came to mind was Open Shortest Path First (OSPF). OSPF uses Dijkstra's algorithm to find the shortest path from whatever node we start in, which would be useful for finding the highest throughput. The next component I came up with was implementing power-saving states to each sensor, where upon hitting a predefined threshold the sensor would lower its rate of throughput. The rationale being that if the sensor lowers the rate of its throughput via underclocking itself, it would reduce power consumption. This novel component added to the system ensures that Dijkstra's algorithm will always choose the fastest transmitting node, which also happens to be a node that isn't running low on power. As a result, I found that this system had lasted ~37% longer using Dijkstra's algorithm than if the system had each node acting independently.

Implementation

To simulate this protocol, I created a network of 7 nodes in MATLAB. Nodes 2 through 7 represent a mobile sensor network, while node 1 represents a command and control center. The visual representation of such can be found in the left image of Figure 1 below. During the simulation "active nodes", or the node that is broadcasting, will turn red as shown in the right image of Figure 1.

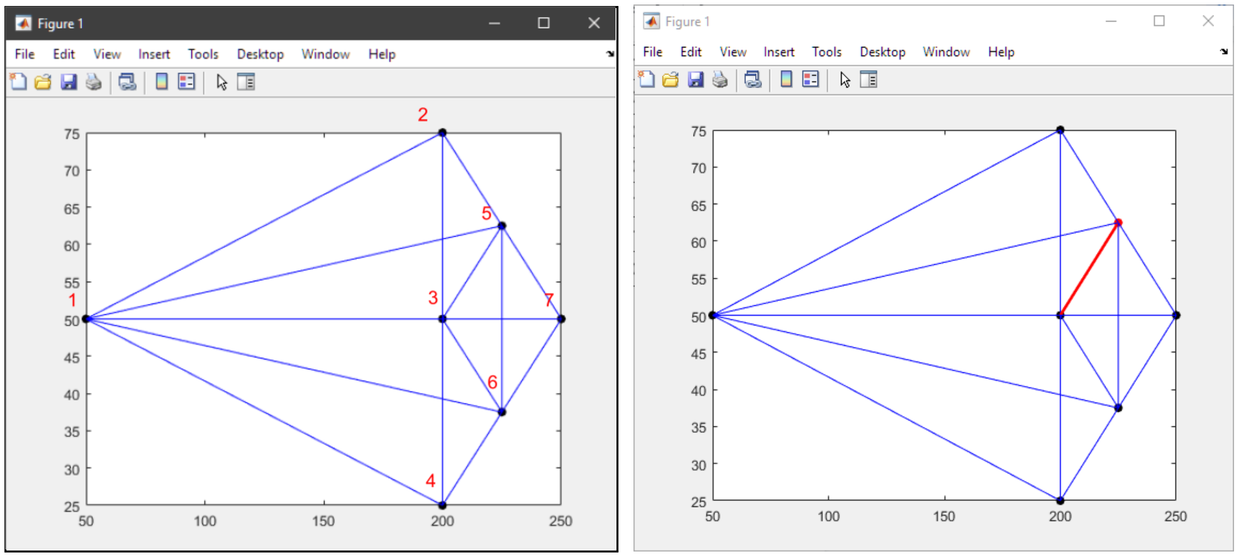
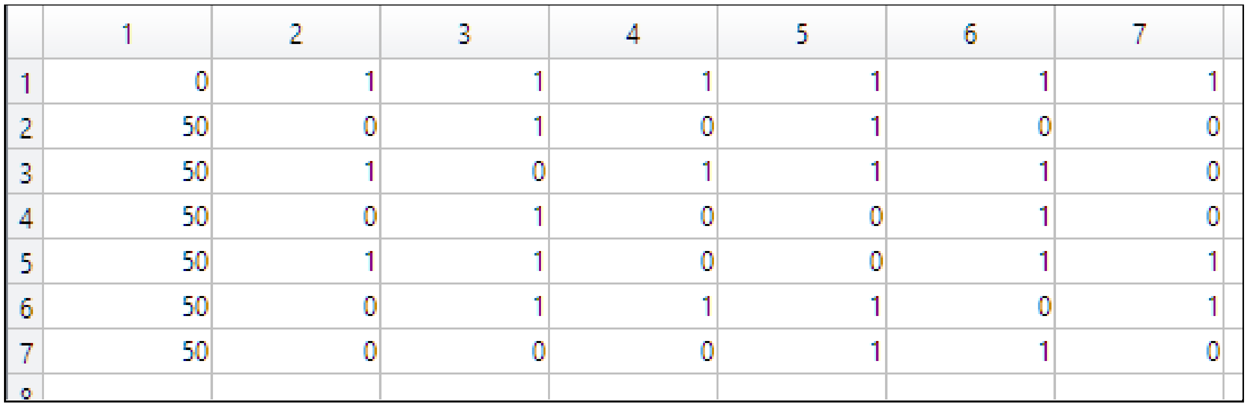


Figure 1: The visual representation of the node placement (left) and simulation transmission behavior (right)

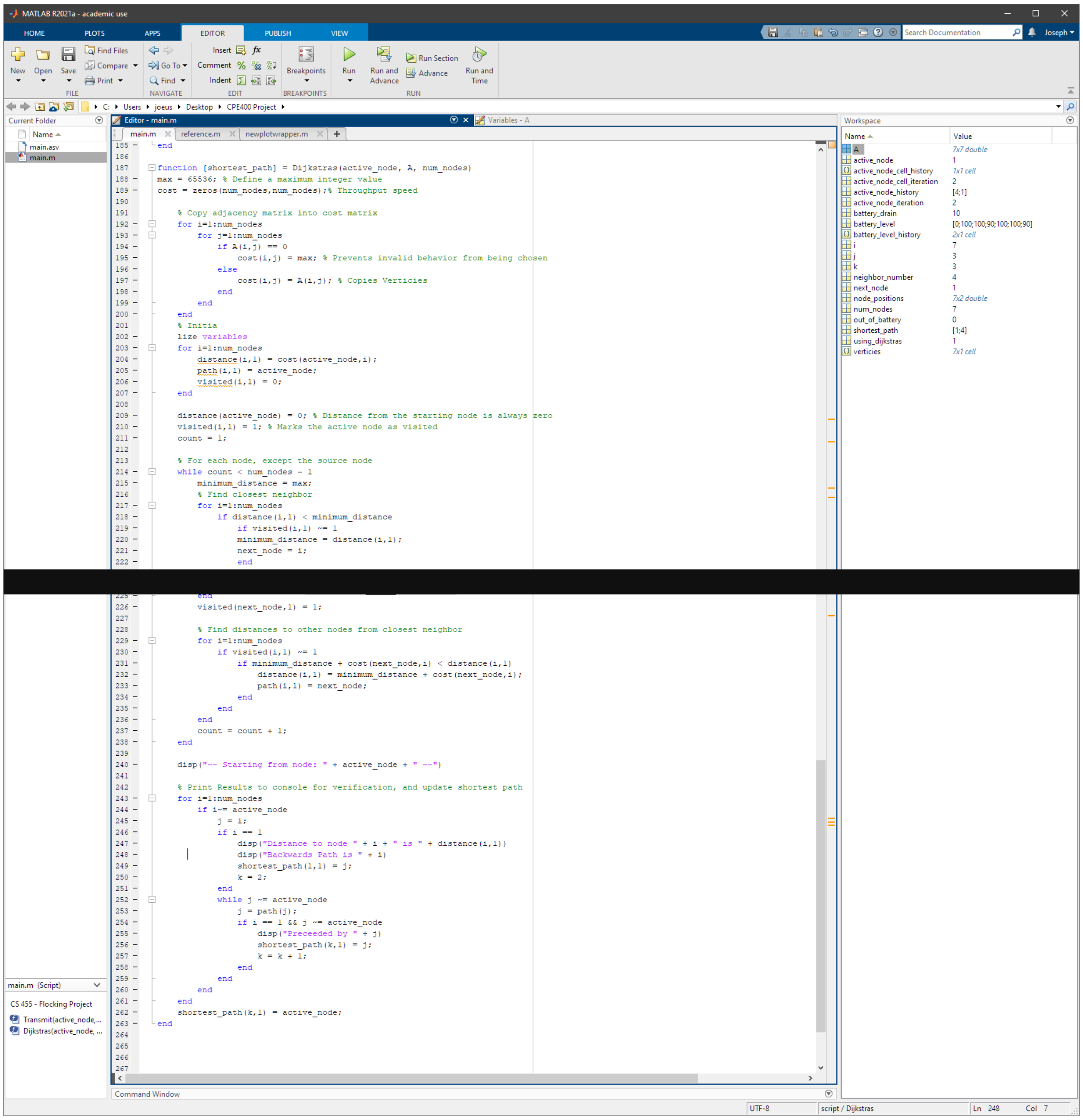
The placements of the nodes are intended to be arbitrary, as their placements are out of the scope of this project. What's important to notice is the vertices connecting the mobile sensor network. These vertices can be better understood with an adjacency matrix such as the one shown in Figure 3.



	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	50	0	1	0	1	0	0
3	50	1	0	1	1	1	0
4	50	0	1	0	0	1	0
5	50	1	1	0	0	1	1
6	50	0	1	1	1	0	1
7	50	0	0	0	1	1	0

Figure 3: The weighted adjacency matrix used for calculating transmission times

When two adjacent nodes transmit between one another, the corresponding vertice in the simulation will turn from blue to red (as seen in figure 1). You will also notice that in Figure 3, all vertices connected to node 1 are weighted. That is because our project assumes that the control and command center is farther away from the mobile sensor network than the nodes in the network are from one another. Each vertice weight can be interpreted as the rate of the transmission. These transmission rates are what our OSPF protocol will use to select a path back to the command center. The code for my OSPF protocol can be found below in Figure 4.

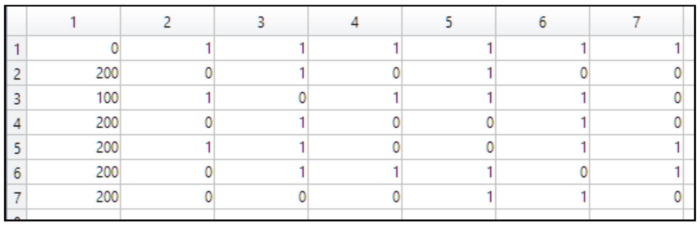


```
    end

function [shortest_path] = Dijkstras(active_node, A, num_nodes)
max = 65536; % Define a maximum integer value
cost = zeros(num_nodes,num_nodes);% Throughput speed

    % Copy adjacency matrix into cost matrix
    for i=1:num_nodes
        for j=1:num_nodes
            if A(i,j) == 0
                cost(i,j) = max; % Prevents invalid behavior from being chosen
            else
                cost(i,j) = A(i,j); % Copies Verticies
            end
        end
    end
    % Initia
    lize variables
    for i=1:num_nodes
        distance(i,1) = cost(active_node,i);
        path(i,1) = active_node;
        visited(i,1) = 0;
    end

    distance(active_node) = 0; % Distance from the starting node is always zero
    visited(i,1) = 1; % Marks the active node as visited
    count = 1;

    % For each node, except the source node
    while count < num_nodes - 1
        minimum_distance = max;
        % Find closest neighbor
        for i=1:num_nodes
            if distance(i,1) < minimum_distance
                if visited(i,1) ~= 1
                minimum_distance = distance(i,1);
                next_node = i;
                end
[redacted]
        end
        visited(next_node,1) = 1;

        % Find distances to other nodes from closest neighbor
        for i=1:num_nodes
            if visited(i,1) ~= 1
                if minimum_distance + cost(next_node,i) < distance(i,1)
                    distance(i,1) = minimum_distance + cost(next_node,i);
                    path(i,1) = next_node;
                end
            end
        end
        count = count + 1;
    end

    disp("-- Starting from node: " + active_node + " --")

    % Print Results to console for verification, and update shortest path
    for i=1:num_nodes
        if i~= active_node
            j = i;
            if i == 1
                disp("Distance to node " + i + " is " + distance(i,1))
                disp("Backwards Path is " + i)
                shortest_path(1,1) = j;
                k = 2;
            end
            while j ~= active_node
                j = path(j);
                if i == 1 && j ~= active_node
                    disp("Preceeded by " + j)
                    shortest_path(k,1) = j;
                    k = k + 1;
                end
            end
        end
    end
    shortest_path(k,1) = active_node;
end
```

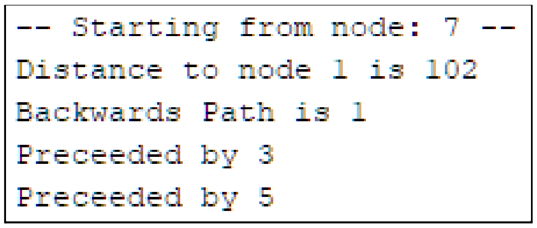
Name	Value
A	7x7 double
active_node	1
active_node_cell_history	1x1 cell
active_node_cell_iteration	2
active_node_history	[4;1]
active_node_iteration	2
battery_drain	10
battery_level	[0;100;100;90;100;100;90]
battery_level_history	2x1 cell
i	7
j	3
k	3
neighbor_number	4
next_node	1
node_positions	7x2 double
num_nodes	7
out_of_battery	0
shortest_path	[1;4]
using_dijkstras	1
verticies	7x1 cell

Figure 4: The code implementation for Dijkstra

Generally how a Open Shortest Path First (OSPF) protocall works, is that each node creates a link state database. The database is updated during every iteration so that the entire network has the same information, making it useful for a closed network where we know all nodes in the system. The link state database advertises the shortest path to get to each node in the network and is generated via Dijkstra's algorithm. Dijkstra's algorithm calculates the shortest path by taking in a source node and an adjacency matrix. For demonstration purposes, look to figure 5 below.



	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	200	0	1	0	1	0	0
3	100	1	0	1	1	1	0
4	200	0	1	0	0	1	0
5	200	1	1	0	0	1	1
6	200	0	1	1	1	0	1
7	200	0	0	0	1	1	0



```
-- Starting from node: 7 --
Distance to node 1 is 102
Backwards Path is 1
Preceeded by 3
Preceeded by 5
```

Figure 5: The weighted adjacency matrix input into Dijkstras and the shortest path to node 1

From figure 5, we can see our source node is node 7 and that we need to navigate through the adjacency matrix to node 1. The first step of Dijkstra’s algorithm is to create a copy of the adjacency matrix that initializes all unconnected nodes to have the cost of infinity, and our source node to be zero. We then initialize all nodes as unvisited except for our source node. Next we create a distance matrix that stores the cost of visiting each vertex. We then select an unvisited node with the lowest cost, recalculating the shortest distance of the remaining vertices from the source. Once all nodes have been visited, we should know which path contains the lowest cost from node 7 to node 1. Such a path is calculated in Figure 5, and you can visualize the results of that output in Figure 6 shown below:

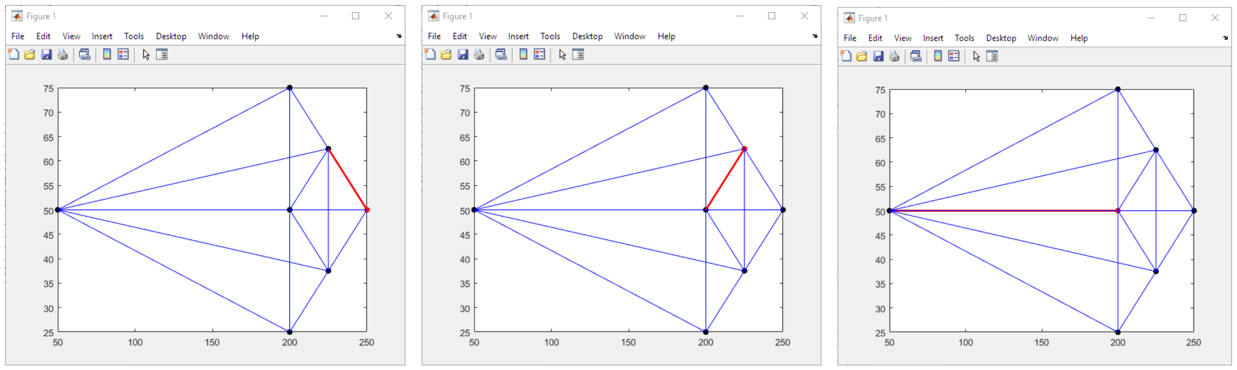
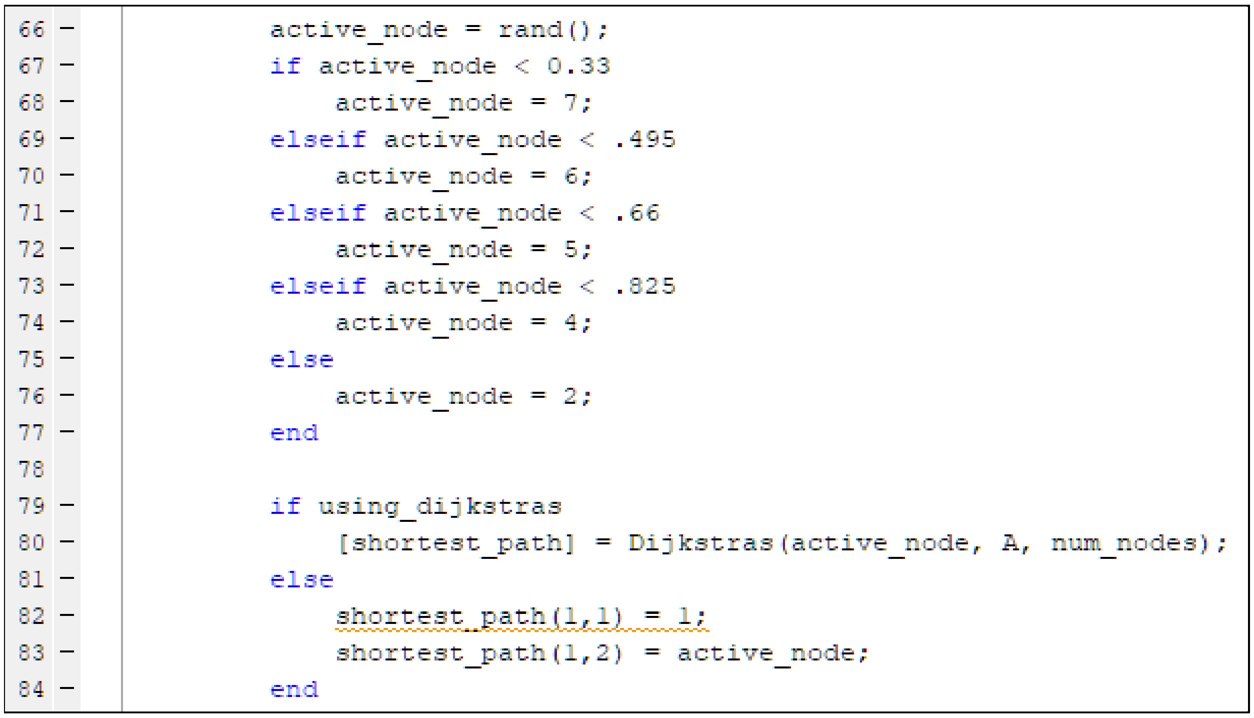


Figure 6: The visual representation of the simulation performing Dijkstra’s algorithm

The source node in our case would be whatever node in the mobile sensor network that has data to transmit to the command and control center. For this simulation, I used a function for generating a random float between 0 and 1 to determine which node has data to transfer. The code for this behavior can be found in figure 7 shown below:

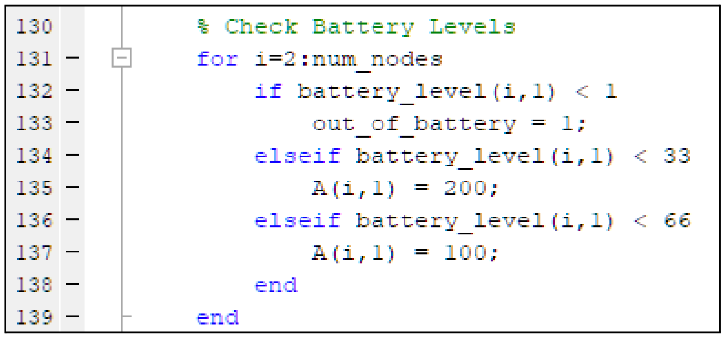


```
active_node = rand();
if active_node < 0.33
    active_node = 7;
elseif active_node < .495
    active_node = 6;
elseif active_node < .66
    active_node = 5;
elseif active_node < .825
    active_node = 4;
else
    active_node = 2;
end

if using_dijkstras
    [shortest_path] = Dijkstras(active_node, A, num_nodes);
else
    shortest_path(1,1) = 1;
    shortest_path(1,2) = active_node;
end
```

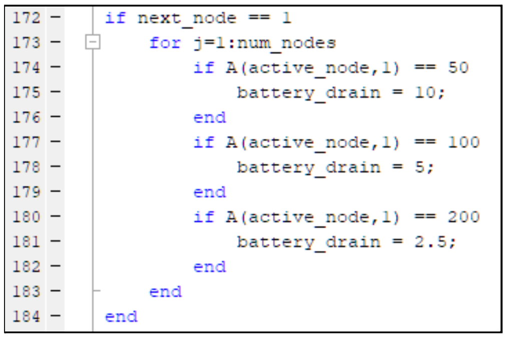
Figure 7: The code that handles the initial active node and the logic for what algorithm to use

After a random number is picked, I use a probability logic to determine which node found data worth transmitting. If you look back to the node placements in Figure 1, you will see node 7 is leading the mobile sensor network if the network were to be moving to the East. This simulation assumes that the network is flocking together in a ‘V’ shape formation east of the command center. If that were the case, then node 7 would be most likely to encounter data to transmit first, and the subsequent nodes connected to node 7 would be about half as likely. Node 3 would be the exception, as anything transmitted by node 3 would be redundant information provided by node 7. While the formation of this network is not the focus of this project, it is important for demonstrating the strengths of our routing design when we look at the results.

For the implementation of the power saving states, I simply assigned a battery level to each node in the network. Each battery level was initialized at 100 to represent the percentage of battery left for the sensor. I then implemented logic to simulate current consumption which can be found in Figure 8.



```
        % Check Battery Levels
        for i=2:num_nodes
            if battery_level(i,1) < 1
                out_of_battery = 1;
            elseif battery_level(i,1) < 33
                A(i,1) = 200;
            elseif battery_level(i,1) < 66
                A(i,1) = 100;
            end
        end
```



```
if next_node == 1
    for j=1:num_nodes
        if A(active_node,1) == 50
            battery_drain = 10;
        end
        if A(active_node,1) == 100
            battery_drain = 5;
        end
        if A(active_node,1) == 200
            battery_drain = 2.5;
        end
    end
end
```

Figure 8: The code that handles power consumption states (left) and the code that handles scaling power drain to the throughput speed (right)

The logic for current consumption is such that for each pass between nodes in the mobile sensor network only 0.5 percent is subtracted from the active node's battery level. The battery consumption for transmitting back to the command center is variable based on what power saving state the node is in. The power saving state is determined by whether or not the node's battery level has dropped below the threshold 66 percent, or 33 percent. When a node's battery level drops below these thresholds the node's transmission rate slows down and it drains less current. These changes are reflected in the weighted adjacency matrix used for Dijkstra's algorithm, tying the throughput algorithm to the energy consumption logic. The goal of which is to keep the network alive as long as possible and to increase throughput.

Results and Analysis

In order to analyze the result of our protocol I compared it to the transmission logic found in Figure 7, where nodes simply transmit directly to the command center when they become the active node. The results of which can be found below in Figure 9.

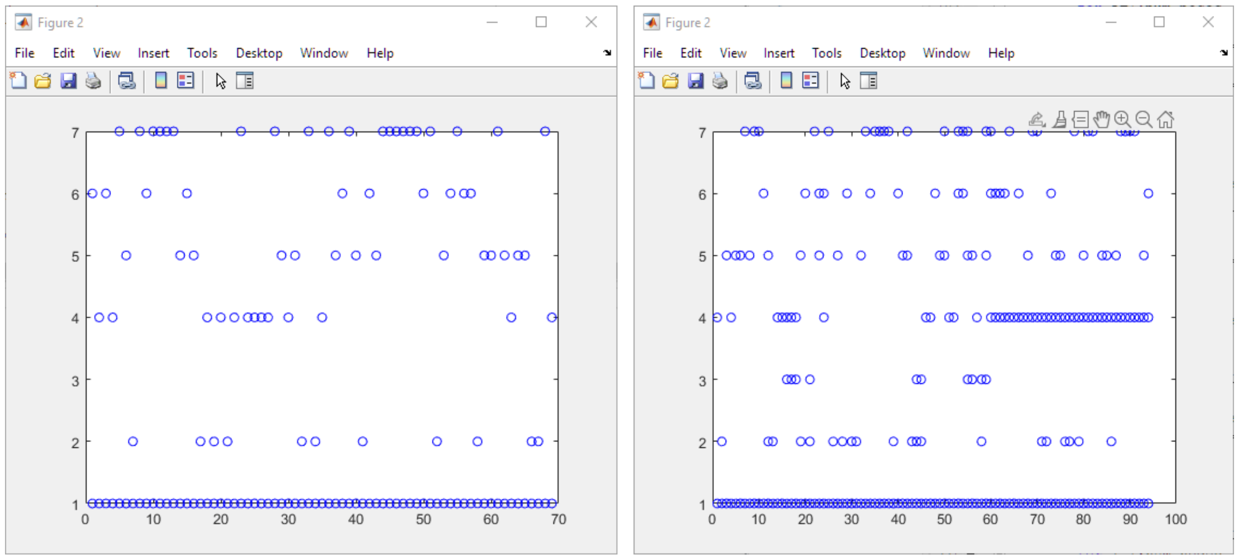


Figure 9: A plot of node activity selection comparing no networking protocol (left) to Dijkstra's (right)

The action selection index plot shows the history of nodes that were active during each iteration of the simulation. The simulation iterates until a node in the mobile sensor network has its battery level drop below zero. As you can see from the X axis of the two action selection plots, the OSPF protocol had more throughput than the simulation that used no networking protocol. Another interesting point to take from Figure 9, is that node 3 was never selected in the simulation without a network protocol. This can hint us as to why the OSPF network performed better. OSPF leveraged all the nodes despite node 3 never being a source node. After analyzing the node battery levels found in Figure 10 it should become obvious why this was an advantage.

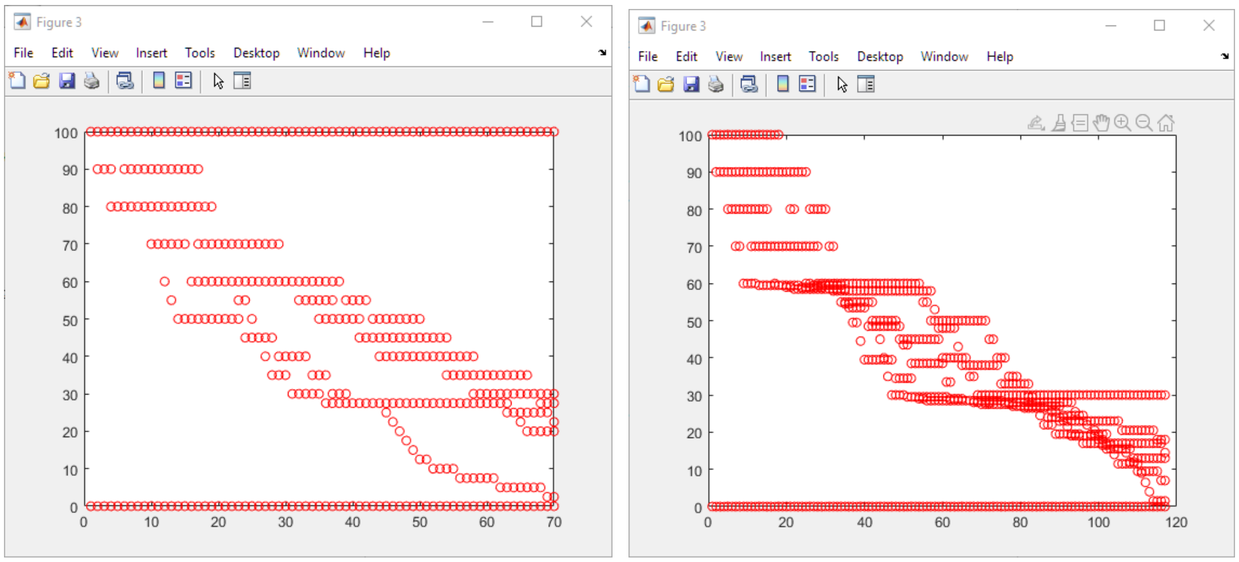


Figure 10: Node battery levels comparing no networking logic (left) to Dijkstra's (right)

The battery levels for each node are plotted in Figure 10. On the left, you'll notice that one node's battery was never touched, that node would be node 3, which never had data to transmit to the command center. The battery level plot for our protocol utilizes every node, and as a result, you will see its X axis also reflects that it outperformed the simulation with no protocol. Also notice the effect of the power saving state logic, where each node is brought to the same threshold before battery levels drop lower. This ensures wear-leveling so that the mobile sensor network wastes less power than it would without that logic.

<u>Novel Contribution</u>

The OSPF protocol component combined with the novel power saving state is what allows this protocol to achieve the goal of maximizing throughput and longevity. Without the power state component, the OSPF protocol would have the same exact behavior as the simulation with no protocol. This is because the shortest path would always be to directly transmit back to the control center.

Error Handling Scenarios

There can be a logic error if the link state database hasn't been updated for all nodes. This error would be handled by this project by having the active node transmit directly to the command center. This would be sub-optimal, but the error would be self correcting due to the wear leveling from the power saving state component. Another source of error could be a node failing to transmit back to the command base, however this error should be handled by our OSPF protocol. Once the protocol updates the link state database, the path from the failing node will be marked as infinite. OSPF will then find the next shortest path to the command base and the mobile sensor network will continue as normal.