Simulation of a Wireless Hybrid Minimal Infrastructure Ad-Hoc Network With Dynamic Spectrum Allocation

Written by Shane Lester

Major contributions made by Shane Lester and Jimmy Yuan

December 21, 2019

Abstract

Traditional wireless network architectures with static spectrum allocation and complex static architectures are having trouble meeting the quality of service and quality of experience of the variety of applications and their traffic loads. This is leading to new innovations in wireless networking technologies, such as using underutilized spectrum dynamically for secondary subscribers, ad-hoc networks with mesh routing to make on the fly networks without having to pay for expensive infrastructure up-front, and centralized control of networks via software defined networking controllers to decouple data and routing.

To better understand these ideas, we built a software simulation of a hybrid ad-hoc minimal infrastructure wireless network with dynamic spectrum access. By ad-hoc routing, we mean that the devices can only communicate with each other within their own transmission radius and they are created on the fly. By minimal infrastructure, base stations are used to run the channel allocation algorithm and find the optimal path with the optimal spectrum allocation. Note that this is just a model, in the internals of the software the base stations don't actually do anything. In theory, they would allow for a more centralized routing protocol then traditional ad-hoc networks use, so here we replace dynamic source routing typically used in a pure mesh network with a minimization of dynamic spectrum allocation protocol using the minimal infrastructure base stations.

Our simulation allows us to first build a random graph to model our hybrid ad-hoc minimal infrastructure network partitioned across a 10 by 10 grid. We can have between 3 and 40 user devices across the grid. We then choose between 4 and 10 wireless channels that model spectrum leased to primary subscribers. Secondary users can use those channels while primary users aren't using them. We assign a random exponential inter-arrival rate per channel which to describe the probability of a primary user accessing that channel. We then use heuristics to find the near optimal path between two devices by finding the best possible allocation of spectrum along a path. After we query a route, we gain data points to feed into our data dashboard.

The user interface of the application begins with a home page. The user can either generate manual queries between devices on the grid through an interactive routing map or they can automate 840 random queries to generate data fast. Lastly there is a metrics data dashboard that has two views: a high level view and a frequency view of the data.

1 Roles, Contributions, and Development

In developing this system, we began by partitioning the tasks into three major roles: backend, frontend, and metrics (UX).

The system design, algorithm design, and implementation were made by myself using Python and Django. I built the backend system, designed all the algorithms, and led the specs requirement. I also acted as the communication medium between the advisor on this project and our group. I built the mockups for all required frontend features to match the specifications requested to us by our advisor and to make sure the backend data was properly represented.

Jimmy Yuan took the lead for the presentation layer of the application, which he did in React and Javascript. He implemented a toggleable metrics view data dashboard for data visualizations with a frequency view and a high level view. He also implemented a random graph form, and a queryable graph complete with interactive routing. He also translated my UX specifications

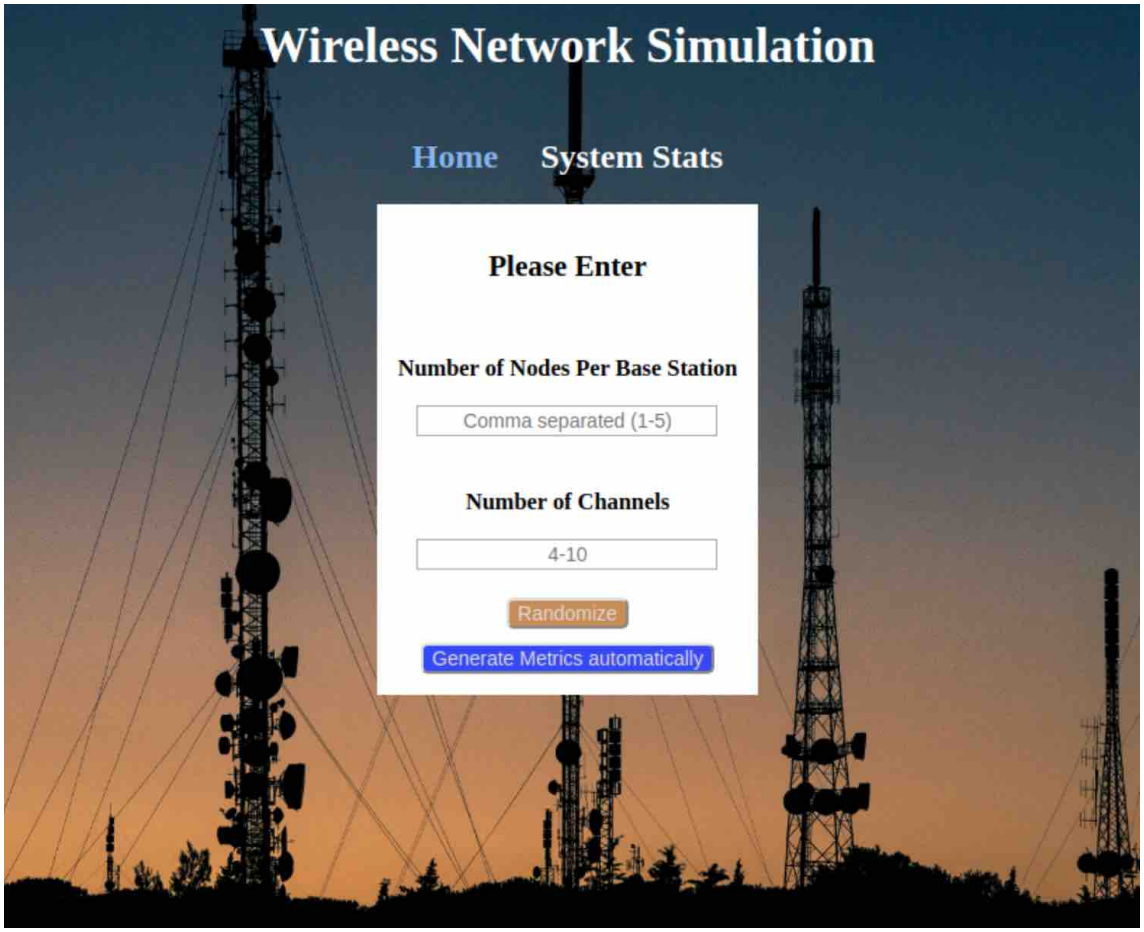

Figure 1: The homepage allows for a few different options to run our simulation.

into real design. He styled the application using CSS and found the images. His application also had good performance under high loads which was a major requirement for the graph and metric visualizations.

2 Understanding Core Features Through The User Interface

The application begins by allowing the user a few options depending on how they want to use the simulation. One is to specify the random network topology and run manual queries to generate data points. This is done by filling in the form and selecting randomize. Another is to automate 840 random queries to quickly generate lots of data points. This is done by selecting generate metrics automatically. The third option is to view the charts which we have populated using the previous two methods by selecting system stats.

2.1 Configuring the Network Topology

Configuring the network topology and then running manual queries is at the heart of the simulation. The purpose of this simulation system is to better understand hybrid minimal infrastructure ad-hoc networks and this experience does just that. First we will discuss how the form in the home page allows us to control the general clustering of the randomized network topology.

This is done by the Number of Nodes Per Base Station field. This allows for 1-8 comma separated values, to which each value is from 1-5. Each number in the list is the placement of infrastructure. The value at that position will be the number of devices generated around that infrastructure. This is important conceptually because each device can only communicate within its transmission radius, and the base station is responsible for giving the devices the information of where to route packets to. Therefore the base station must communicate with the devices and no

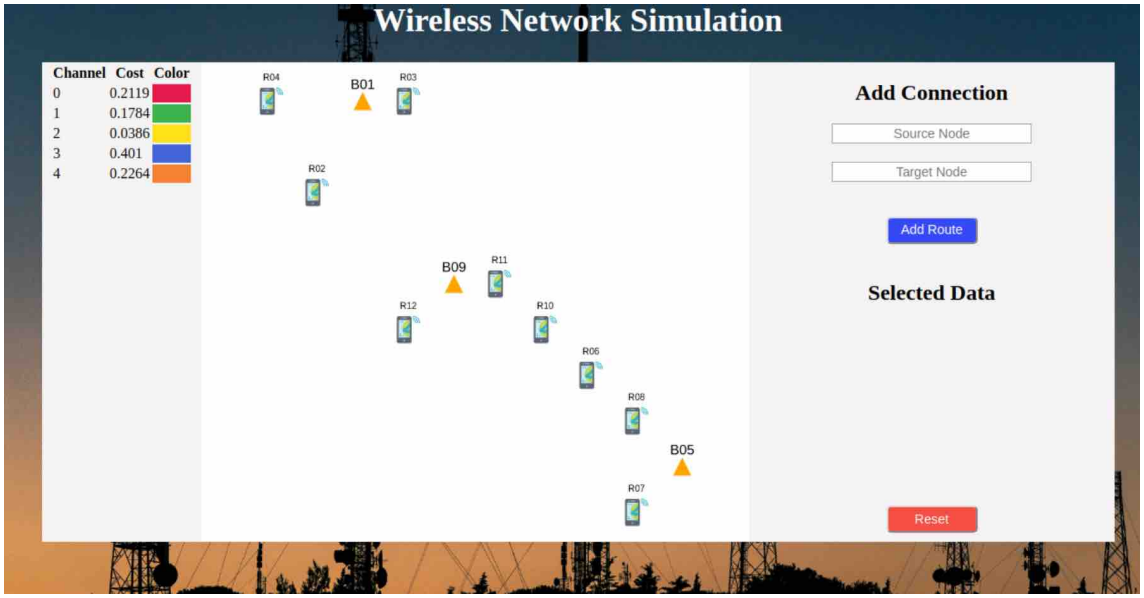

Figure 2: Topology of 3,3,3.

device should be outside of the transmission radius of a base station. In a minimal infrastructure environment, this allows the network creator to save cost on infrastructure by having to develop much less infrastructure, because they have to setup fewer base stations (or any control kind of infrastructure) and can allow the ad-hoc devices to be placed around them cheaply and be in charge of doing the physical routing.

Below we have images of several network topologies, more specifically the (3,3,3), (1,1,5,5), (1,1,1,1,1) and the maximum capacity (5,5,5,5,5,5,5,5) in Figures 2 through 5. In particular, notice how the comma separated list input allows for the clustering of the user devices and therefore some control of the topology. This is similar to real ad-hoc networks, where the devices are meant to be placed in somewhat strategic locations and not in a purely random order.

2.2 Running Manual Queries

Recall that a device can only communicate with devices within their own transmission radius. Also recall that we can route data between far away devices by allowing non-destination ad-hoc devices pass packets to the next device on the path to the destination device. By clicking on a device, all reachable devices, the coordinates of the device, and the base station it associates with (conceptually to receive the path from when it is the source device) all show up. We can see this in Figure 6. Note that conceptually to know all reachable nodes the device would probably send a multicast packet to all nodes in its transmission radius who are then tasked to do the same thing. The response would be the reachable nodes.

To run a query between two devices, we would use the source and target form. Once we do, we allocate a channel to each edge along the path. This models dynamic spectrum allocation, where the cost are inter-arrival rates of primary subscribers who have priority over secondary users. This simulation is to route data for secondary users, so the goal is to minimize the inter-arrival rate of the entire path. That being said, channels that are used within the transmission radius of a device can't be used by any device. This means that if device R11 uses channels 3 and 5 in its previous route, and device R10 is in the transmission radius of R11 and uses channel 1 on its previous route to R09 (which isn't in the transmission radius of R11) and no other route has been established, the next packet trying to route through R09 can't use channels 3, 5, and 1 because of co-channel interference and can't use channels 0, 2, 4, and 6 because of adjacent channel interference.

Co-channel interference, is where two devices are attempting to communicate across the same channel. Adjacent channel interference is two channels adjacent to a used channel also have interference. In our simulation, there is no way to delete a route being established. Therefore, once a channel is allocated between two devices those channels have interference for the duration of that graph's lifetime. This models something similar to a voice over IP ad-hoc network, which

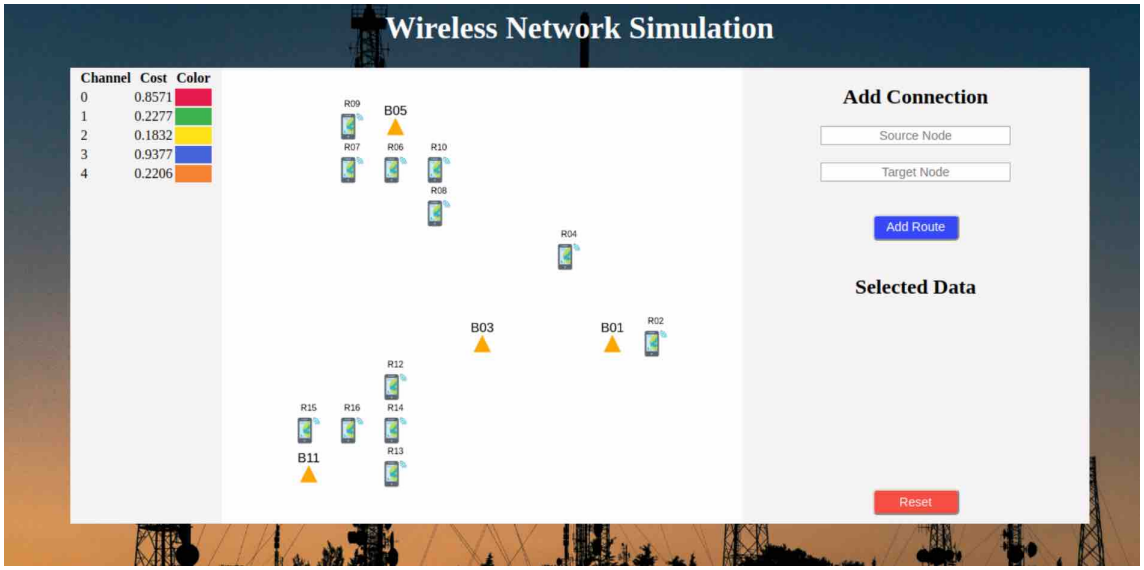

Figure 3: Topology of 1,1,5,5.

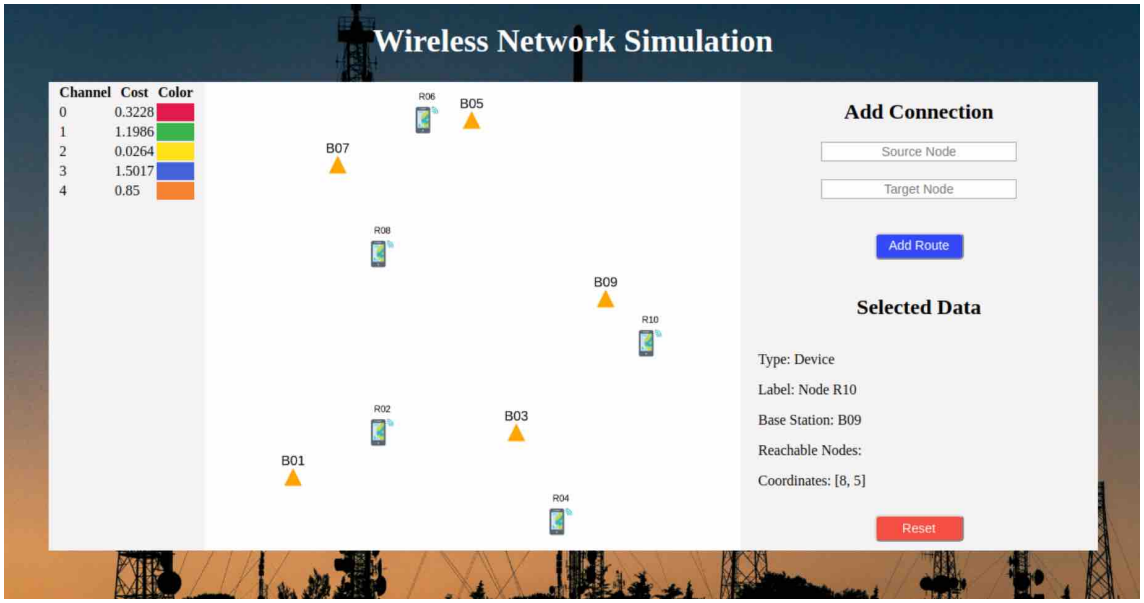

Figure 4: Topology of 1,1,1,1,1.

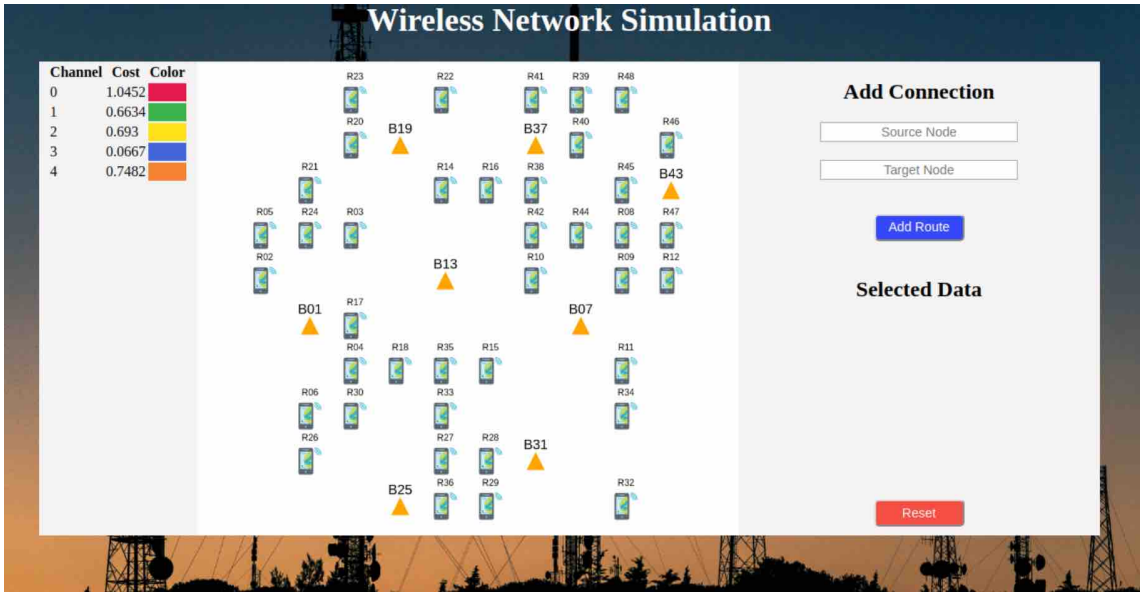

Figure 5: Topology of 5's at capacity.

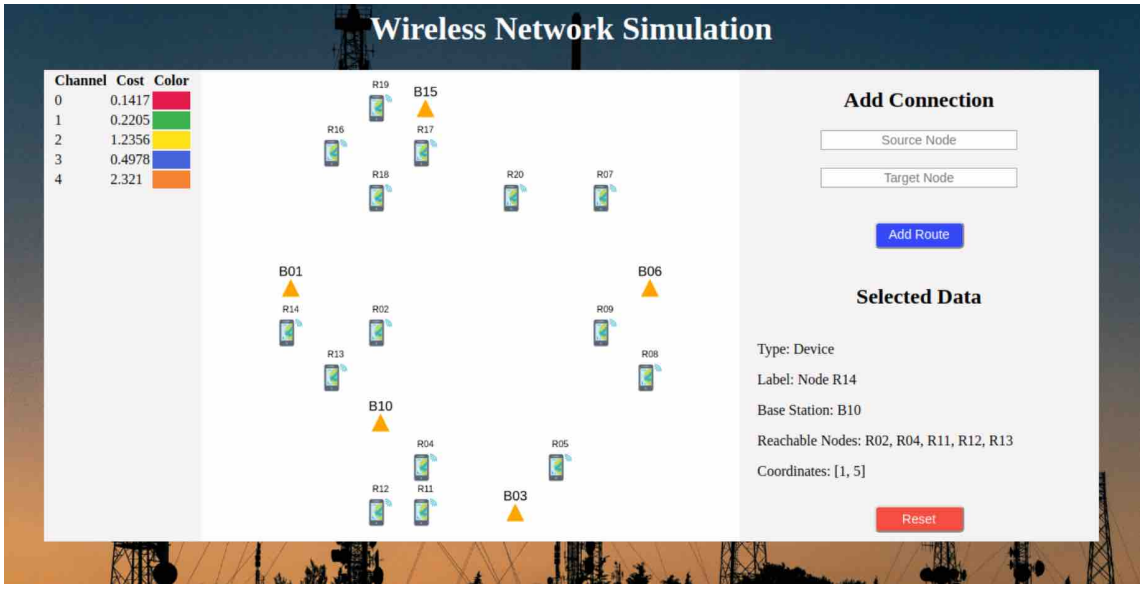

Figure 6: Lots of useful data shows up by clicking on a device, the most useful being which devices are reachable.

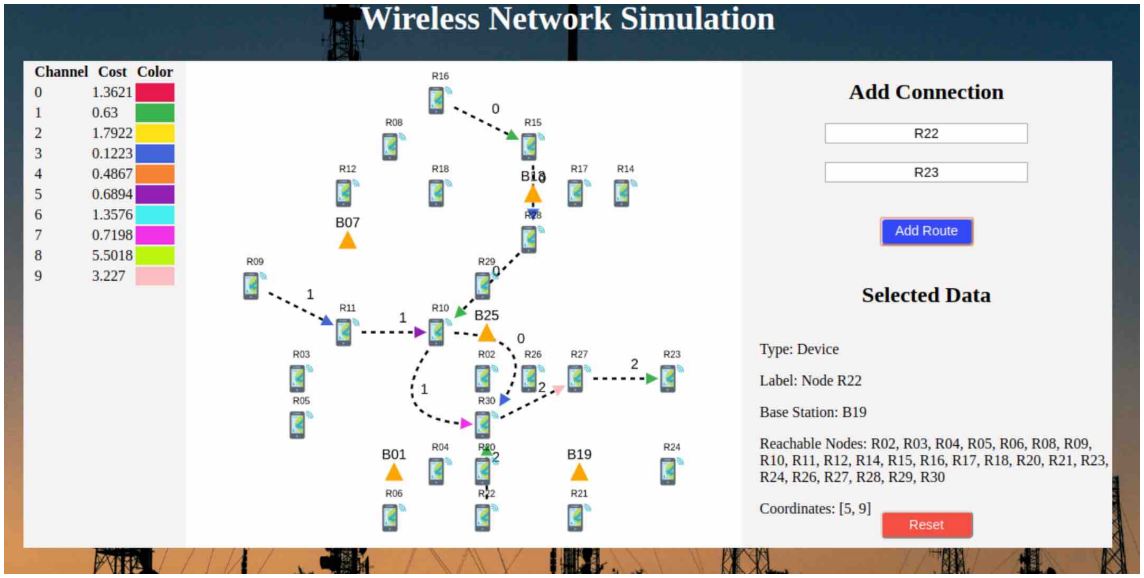

Figure 7: Note that the colors at the tip of the arrows correspond to the channels used and the numbers correspond to a single route.

establishes connections that would reserve channels for the duration of a call. In our case, we would assume that once a call is made, the connection is reserved for the duration of the graph's lifetime. Figure 7 shows multiple routes being established.

2.3 Running Automatic Queries

Running queries automatically runs 840 randomized queries for network topologies (3), (3,3), ..., (5,5,5,5,5,5,5,5). It runs each of those for 4,5,...,10 channels. For each topology and channel, it runs 5 random queries, choosing first a random edge and then a random reachable node. Due to channel interference, this generates around 550 data points out of 840 queries. The algorithm typically takes about 1-3 minutes to complete. During that time, a loading modal pops up on the screen and the user must wait until the operation completes before performing any other kinds of operations.

Typically this operation is used when we need to generate lots of data points to see trends. From a conceptual standpoint, a network system doesn't achieve steady state statistical values with the first few queries run due to different behavior of initializing systems and of systems that are in the middle of a workload. So by generating lots of data values before reviewing metrics, we can view trends that conceptually model steady state networking systems.

2.4 Data Dashboard

Each time our system runs a query it keeps track of 4 variables. On the x axis the number of nodes and number of channels are recorded and on the y axis the number of channel switches and the number of hops are recorded. The channel switches means how many different channels were used on a particular route and the number of hops is the total number of edges. We interchange the two variables to generate four charts. On each chart we select from a drop down menu the values to hold constant. This is the high level data view. The high level data view shows an x variable to a y variable and it shows which values appeared in that query. However, it doesn't count the frequency of that value to that query. Therefore if 1 hop appeared as a y value and 3 hops did as well, but 1 occurred 10 times and 3 did once, it would look the same in the high level data view. The data view can then be toggled to the frequency data view. The frequency holds a x variable constant as well and counts the frequency as the y value. Figure 8 and 9 show the high level data view and the frequency view respectively.

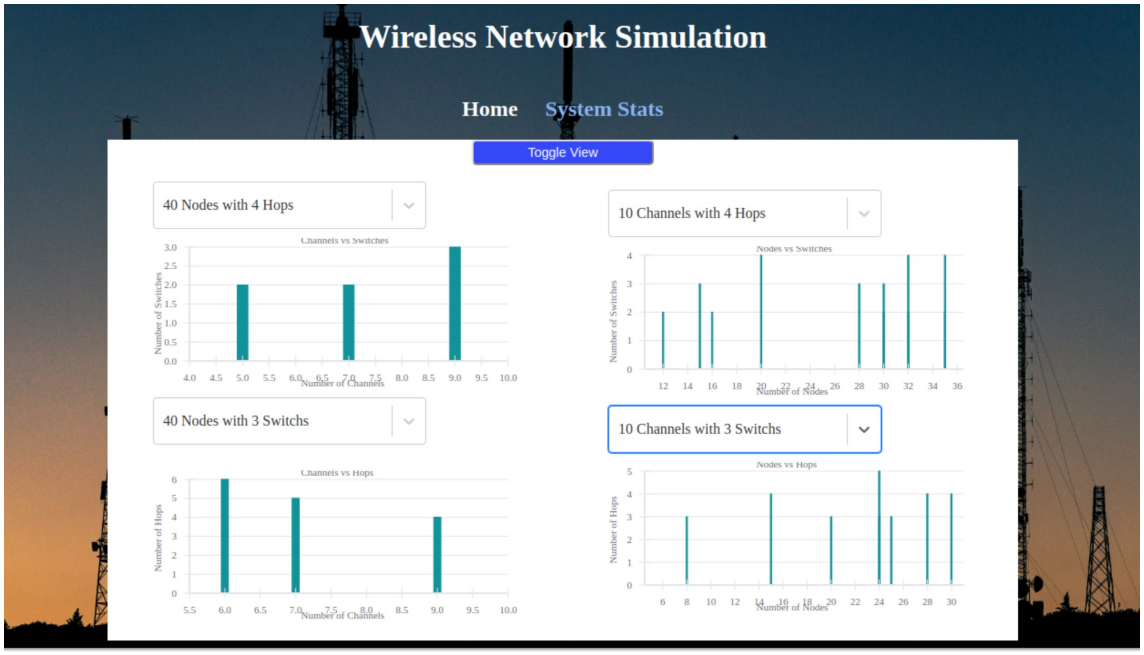

Figure 8: The drop down menu above each chart selects values to be held constant.

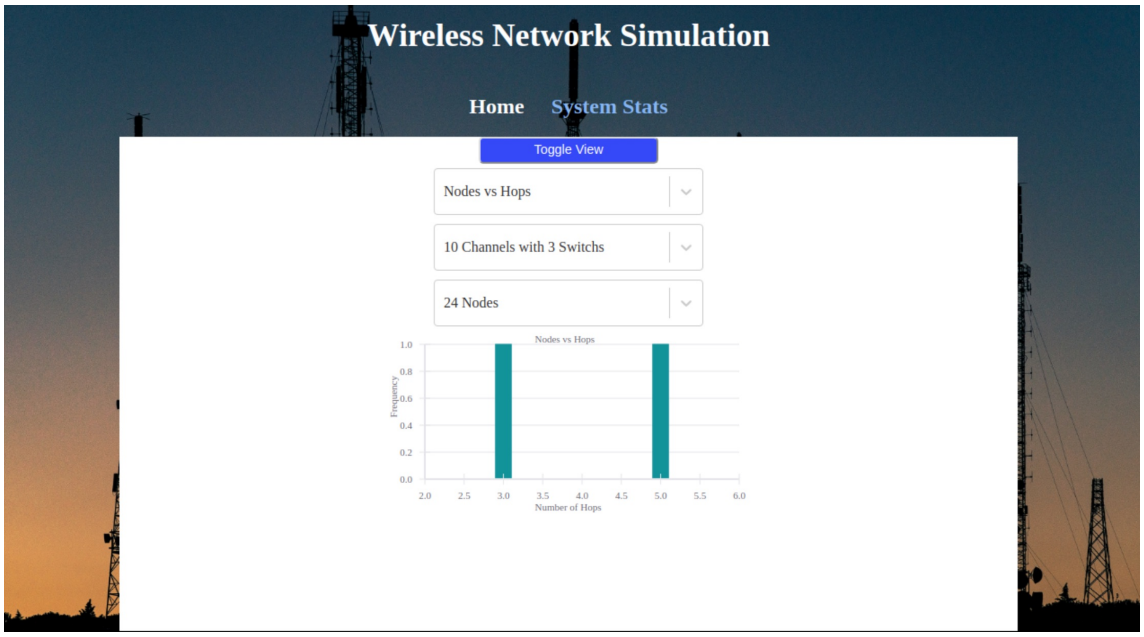

Figure 9: This view allows us to get a closer look to see how many times a particular y value occurred.

3 Core Algorithms

The presentation layer mentioned in section two contained only view logic. All the core logic occurred in the backend Django application. The backend system was built around four core algorithms, a random graph algorithm, a depth first search to enumerate and weigh all the possible paths between a source and destination node, a backtracking algorithm that enumerated all trees that could map possible channel ids to edges, and a metric generation algorithm to randomize 840 queries. The metric generation algorithm also acts as a test for robustness and performance. If it can be run fast and never hits an exception, we can have a high degree of confidence that our system performs well under heavy load. It runs in roughly 1 to 3 minutes, which is extremely fast for our use case. Additionally it has never hit an exception and it was run at least 50 times prior to the writing of this article.

The routing algorithm was built around the depth first search path ranking algorithm and the channel id tree generation backtracking algorithm. Heuristical methods were used to trade always perfect but slow performance for almost always perfect with very fast performance. The heuristic has never been seen giving incorrect values, and if it did receive a non optimal value, it would be the second or third best path rather then the very best. Even that would be rare and never came up for us. That heuristic is the reason for the very fast user experience in the manual query user input mode and for the fast automatic metric generation mode.

3.1 Network Topology Semi-Randomization: The Random Graph Algorithm

This is the algorithm that allowed for our frontend to be able to have control over the clustering of our ad-hoc network. Given a list of 1 to 8 values of 1 to 5, this algorithm could generate a randomized graph over a 10 by 10 grid that had user devices around base stations. The major requirements of this algorithm were that we wanted the grid to have lots of spacing so that all the devices didn't clump into one area, for the user to have general control over the clustering of the network topology, for devices to have other devices in their transmission radius often for a rich query experience, and to always have a base station within a devices transmission radius as it's assigned infrastructure which to us was conceptually important. The algorithm is listed below:

1. Create a grid of size 10 by 10 of strings with " $---$ " as the value and receive an input list of 1-8 values with number 1-5 as the values
2. Iterate across the list, choose a random x and y coordinate on the grid. If that is a free space and there isn't another base station in the transmission radius of that coordinate, add that base station to the grid with "$B < id >$ ".
3. Given the value x of the array at that index, randomly generate x user devices within the transmission radius of that base station as "$R < id >$."
4. Go back to list item 2 until the list stops iteration.
5. Once that is done, output a data structure that can easily be made into an adjacency list

Note that this algorithm is typically fast and creates a great semi-random graph that is perfect for our use case. It does have one flaw regarding performance. It can act as the bottleneck in the metric generation algorithm and can lead to the upper bound time of 3 minutes until completion. It sometimes becomes impossible to find a free random space subjected to the transmission radius requirements. When that is the case the algorithm must be restarted. This occurs more frequently when we create topologies close or at maximum capacity, like (4,4,4,4,4,4,4,4) and (5,5,5,5,5,5,5,5). This is also what made us reach the practical limit of 40 nodes on the grid.

3.2 Modified Depth First Search Heuristic With Predictive Power: Generate and Weigh All Possible Paths Between a Source and Destination Node

This algorithm is what allowed us to achieve such fast performance for route queries. A typical depth first search to enumerate all the paths with limiting the size of paths to no more then 7 (to improve performance and 6 hops turned out to be a good practical limit for our use case) would be

fast but because this algorithm generates input for the channel id mapping algorithm that models the dynamic allocation of channels, placing all the input into the backtracking algorithm would lead to slow performance. This issue is solved by weighing each path as we enumerate them and ranking paths here that will most likely have the best allocation of channels. This allows us to only have to run the channel allocation backtracking algorithm for each path only until the first correct value is found rather then trying all the values. Below is the depth first search algorithm to enumerate all paths between two nodes. Notice that it is pretty much identical to any other depth first search algorithm. However, the performance boost and power of the application comes from the weighing and storing the output into a priority queue. This allows us to choose the close to minimum path with O(1) access after the algorithm is finished. The algorithm is:

1. Create a stack for depth first search, put the first node in a path and add it to the top of the stack. Create an empty priority queue to store output.
2. Check if the stack is empty, if so return the minimum value in the priority queue (or nothing), if not move to the next step.
3. Pop the top of the stack and check if the path is less or equal to size 7, if so move to the next step, else move to step 2.
4. Check if the last value in the path is the destination, if so weigh it and store it in the priority queue and move to step 2. Else move to the next step.
5. Enumerate across all the connected edges to the last node in the path and if the node isn't already in the current path, append it to the end of the path and store it in the priority queue. Move back to step 2.

The key to understand why the weight assignment for all practical purposes for our use case seems to always chooses the optimal path when we ran lots of manual queries and checked by hand is to first realize that we can either have a path with no interference in any channel or we can have a path with channel interference. If there is no interference then for any given amount of hops the allocation will always be the same. Therefore with no interference across any channel in the path, there is statelessness. Statelessness meaning that regardless of any other factor, a path with two hops will always allocate the same two channels. This means that any two paths of equal amount of hops will be just as good as each other. Therefore with no channel interference across a path, any minimal amount of hop path will always produce the optimal channel allocation.

The other case is that there is interference across the path. Typically speaking, for a fixed number of hops a path only has certain options for channel selection. If we want the best options to select, we would want to have the set of channels with the worst (highest) inter-arrival rates to be the interfered channels. Then we can use the best channels for our allocation. So we sum the inter-arrival rates of the interfered channels (both co-channel and adjacent channel interference) and we can be pretty certain that we have the best options to choose from. We can also be fairly sure that the lower weighted sum will give us better channel choices, and that the worse options to choose from would give us worse channel allocation choices.

Note here that this is not always correct, as it isn't always true that the absolute best possible channel allocations given two sets of channels will be the one with the lowest overall sum. Yet although this isn't always true, in practice it almost always gives the optimal channel allocation. And if it didn't it would be the second or third best. This is because there usually isn't lots of variability amongst summed channel weights, because most nodes within the transmission radius of each other tend to have similar channels clogged. So, due to the nature of the data and the argument above, when we ran queries and checked by hand we never found a time where the heuristic chose non-optimal results. Also note that this equation would be very easy to tune if we put the effort in and chose a less naive approach then summing the inter-arrival rates.

The ideas of statelessness and the minimum number of hops leading to an always optimal allocation and that choosing the best possible options for channels using overall sum will lead to almost always optimal channel allocation leads us to the following weight equation that powered the fast performance of our application. If w is path weight, $sum(Cr)$ is the sum of all inter-arrival rates of channels with interference, and h is the number of hops along a path:

$$w = h + (1 - sum(Cr)/100000)$$

Note that the 100000 in the equation is just a large number that will greatly overpower any sum of inter-arrival rates. Because the weight is just to establish an ordering and doesn't have any intrinsic value, 100000000 would have worked just as well or 69789879.

Also note that even though this heuristic has passed the eye test and manual testing, it could be further optimized by first tracking lots of cases and getting real evidence of how often it produces the optimal value and when it doesn't, how optimal it actually is. After that it can either be further optimized, or, if it performs well, can be kept as is. But even if it is modified, an approach like this should work, but perhaps with a slightly different equation.

Lastly, it is important to consider when discussing the trade-offs of improving this equation the cost of a non optimal route being chosen. Conceptually speaking it seems to have very low cost if a secondary user has a second or third most optimal path and a very high cost if latency is extremely high due to having to enumerate all possible channel mappings. So it isn't certain that improving this equation would be worth the effort put in, as it would take doing a more formal quantitative study on it.

3.3 Dynamic Spectrum Allocation: Find the Minimum Possible Correct Mapping of Channel Ids to Graph Edges

Recall that this algorithm takes as input the output from the algorithm in section 3.2. The goal is given a set of paths, enumerate all mappings of channel id's to route edges such that the sum of the inter-arrival rates are minimized. Because of the weighting heuristic, we assume that the first correct mapping of channel ids to edges is optimal or near optimal. This allows us to only have to use this backtracking algorithm once per query usually or for as many as it takes for us to find a mapping. However note that in practice this is actually very fast, because if the first mapping doesn't work then typically all the channels have interference and no other path exists. In this case, the entire search space is pruned out because of external channel interference, and there is no performance penalty.

Lets now consider how we pruned the search space and our intentions behind our decisions. First consider that a typical modern processor can process about 1 billion instructions per second and it takes roughly a few hundred instructions to perform a basic search, therefore we can search about one million values per second [Skiena, P.230]. Now consider that in the worst case we have 10 channels, no global channel interference, and we have 6 edges to map channel id's to. Consider also that each node is in only the transmission radius of the next node in the path. Lastly consider that the minimal inter-arrival rates are at index 0 and 9, so we can only eliminate 2 channels for the next node rather then three.

In the above situation, we would have 10 choices for the first item selected and 8 choices after that. This situation would be the absolute slowest case for our backtracking algorithm. This would lead us to have $10 * 8^5 = 327680$ total items to search per operation. This is roughly a third of a million, so we can expect to run the worst case query in about a third of a second. In practice the routing algorithm feels instantaneous to the end user, so these numbers gave excellent results to our use case.

Because of our optimization in section 3.2, we choose the first correct mapping. We assumed that there is no global interference, so this situation would clearly have a mapping and therefore would be selected as the minimal path. Therefore in this worst case situation, this algorithm would produce output in about $1/3$ of a second.

Below is the specifics of the algorithm to select channels for one path:

1. Receive as initial input data structures of an empty output priority queue to store candidates, a list of channels from previously blocked queries, coordinates of the path we are mapping channels to, and an empty path called the current path.

2. Check if we have searched the entire search space (meaning we have no more recursive calls left to perform our backtrack). If there is none left, exit here and pop the minimum value from the output priority queue Else move to the next step.

3. Check if the current path is equal in size to the coordinate path given as a static input. If so, sum the inter-arrival weights and store them in the output queue. Return to step 2. If not, move to the next step.

4. Enumerate across all the possible orderings of channels, eliminating channels that have interference from either global interference or from other channels selected in the current path. This is the enumerate and prune step.

5. Backtrack: add channel to path, call recursive function (move back to step 2) then remove it from the path. This will generate all possible combinations of available channels to edges.

3.4 Metric Generation: Systematically Call Automatic Random Queries to Generate Data Points Fast For Analysis

The purpose of this algorithm is to generate automatic queries for analysis and to provide a test for robustness and performance. There was no unit testing for this application, so this test also acts as a regression test for our use case if we were to add new features. It guarantees a certain expected level of performance and fault tolerance.

In practice it can generate 840 queries in 1-3 minutes, which is great for our use case. Out of those 840 queries, about 550 produce a data point and are saved to be used in our data dashboard. The ones that don't are because of channel interference.

This algorithm tests all three other core algorithms under heavy load as well as non core algorithms like the algorithm to reach all other nodes from a source which was mentioned in section 2.2. This runs the random algorithm 168 times and for each of those random graphs we run 5 random queries. Figure 10 shows the server output after the random query algorithm halts.

Here is the algorithm that generates the random queries:

1. Generate a list of random graph inputs $[3], [3, 3], ..., [5, 5, 5, 5, 5, 5, 5, 5]$ letting us test network topologies with 3 to 40 rout-able devices. Also create a list of $[4, ..., 10]$ for the amounts of channels to allocate.

2. Select a base station at index i and a channel at index j and create a network topology.

3. Select a random node from that graph with at least one connected edge. If this can't be done, we will accept failure for this query and continue iteration.

4. Run a query from source and destination. If this query fails because of channel interference, we will accept failure and continue iteration. Else save it to the system stats.

5. Continue steps 1 to 4 until the iteration is stopped. Then output performance numbers that are the number of successful queries, the attempted queries, and the speed it took to finish this operation with the system stats generated in this run to the user.

4 Conclusion

The purpose of building this simulation on our part was to better understand future trends in networking systems. That was definitely acquired by building this simulation system. I believe this tool could also be used to teach others about the continuum of wireless hybrid networks and the trade-offs associated with the degree of infrastructure and it's relatedness to the chosen protocols typical to pure ad-hoc networks. This is because by having both base stations and ad-hoc devices around the base stations, it shows how protocols must be adjusted, as in control going to the minimal infrastructure base station rather then a dynamic source routing protocol.

This simulation also is consistent with the trend of software defined networking, where we see a distributed complex static architecture trending in the direction of centralization and using networks with lower amounts of infrastructure to reduce cost. The base stations in this simulation act as a software defined networking controller, which allows for the devices to only be accountable for routing data.

This system could be extended in a few ways other then just being used as a teaching tool. By defining a layer to collect some kind of data that measures how good quality the channel is between two devices, the randomly selected exponential arrival rates mentioned in section 3.3 could be replaced by that data and the random graph algorithm could be replaced by real infrastructure. That would take adding two layers to the system design: an interface that abstracts away the hardware and a data collection module that pipes in the latest data. The core logic described

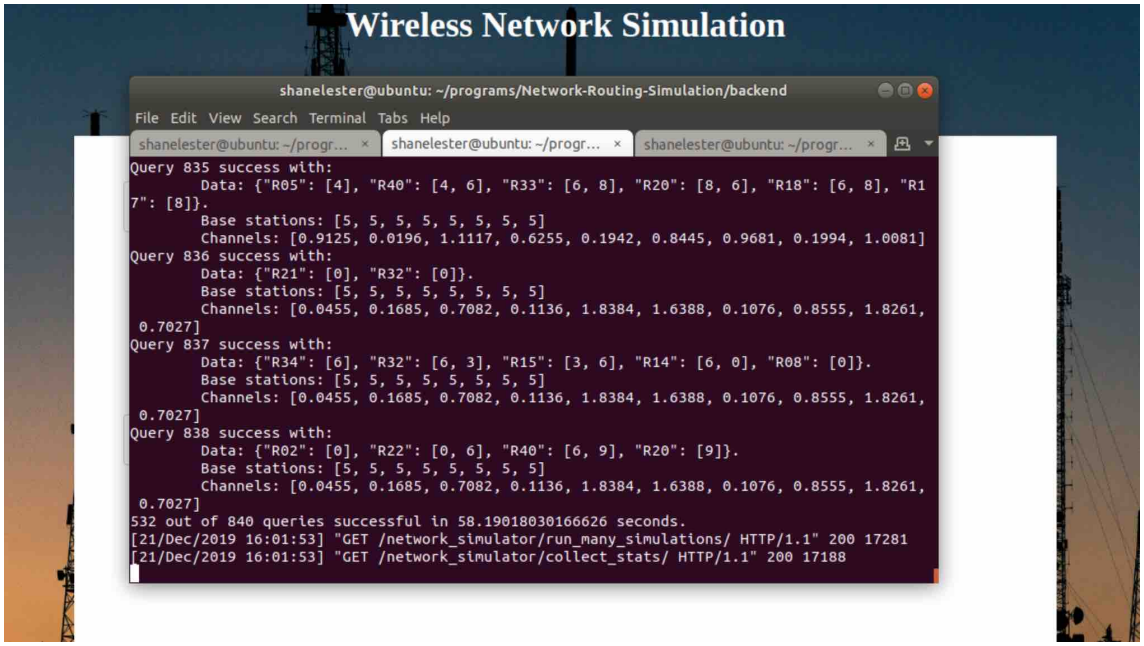

Figure 10: This proves robustness and performance as features are added, making it also perform a similar function to a regression test.

in this paper could be then used as a software defined networking controller that uses the data collection abstraction to feed current data into the system and uses the hardware abstraction to automate the changing of routing tables based on that data. That could be a batch processing job that is done on a scheduler to allow for dynamic routing in a local access network where centralized control is possible.

If that is successful a further approach could be replacing the channel allocation backtracking algorithm with a machine learning algorithm to choose optimal route selection. This would allow for streaming updates of routing tables giving us real time best routing (because backtracking is much slower then machine learning) and routing that gets better through time. This would also match real world trends in smart networking like coginitive radios.

A completely different approach that we could take is to keep this system theoretical but study the real metrics behind these kinds of networks, and tweak the algorithms until it matches real metrics and then publish a paper of what insight that brought us about hybrid wireless networks.

References

[1] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag London Limited 2008.