CS 4450: Software-Defined Network Traffic Engineering Report

Team: Networksfinalprojectteamname
Members: Ariela Gettig, Samuel Qian, Paul Sachs, Kelvin Wang

Goal

Software-defined networking is a paradigm that separates a network's control plane operations from its data plane. Often, control plane tasks, such as routing, are implemented using distributed algorithms such as distance-vector routing or link-state routing and are computed across all nodes in a network. Software-defined networking instead centralizes these control planes with a single controller program that determines how to route traffic. This makes the network easier to update. As a result, the network has less congestion and packet loss, and is able to better balance the traffic load across its respective links. This ensures the reliability and efficiency of the overall network.

The goal of this project was to implement such a controller program. Given a network topology and a set of traffic demands, we were to implement programs to allocate traffic flow within the network to optimize certain metrics. To solve these traffic flow problems, we used linear programming. We were able to break down our traffic flow problem into a set of variables and constraints, and then use the commercial optimizer Gurobi to optimize our variables so that flow could be allocated optimally. To this end, we implemented two different algorithms. The first was to maximize total throughput through the network, and the second was to minimize the maximum link utilization. We used constraints taken from the research paper linked to us and our prior knowledge in graph algorithms to bound our optimization.

Methodology

For this project, we decided to use the network topology and traffic demand matrix for the Sprint network. This was due to the ease with which the network could be visualized, and the completeness of the data. To visualize our network and allocated traffic flows, we used the python "networkx" library, with nodes representing routers and hosts and the edges representing links. We decided to use a circular layout in order to prevent the graph from becoming too cluttered. For every pair of nodes within the network, we used the "networkx" module to determine all paths between them. For both of the algorithms, we used the Gurobi

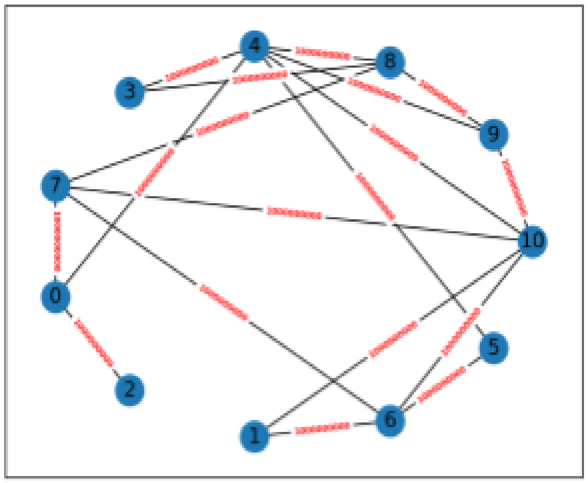

Figure 1. Sprint Network Topology; Links are labeled with capacities

optimization software to produce traffic flow allocations, and this required us to formulate both algorithms as linear programming problems.

Starting with an algorithm to maximize total throughput in the network, we chose to formulate our Gurobi model in terms of paths rather than edges. For each unique, non-cyclical path in the graph we created a decision variable. Formally, we represent the kth path from node *i* to node *j* as the variable $P_{i \to j,k}$. The value of this decision variable is the amount of traffic flow from the source node to the destination along this path.

We then added several key constraints to the Gurobi model for these path variables. For each edge *e* in the network there is a capacity *c(e)*, limiting the amount of traffic flow through the link that edge represents. The amount of traffic flow through an edge *e*, *f(e)*, is the sum of flows for all the paths through that edge. Thus, the sum of all the path variables that represent a path through *e* must be less than or equal to the capacity of *e*. This way the link is never overloaded. Mathematically, for the set of nodes *N* and set of edges *E,* these constraints are:

$$\forall e \in E, \sum_{u,v \in N,\, k} 1[e \text{ is on the } k\text{th path from } u \text{ to } v\,]\, P_{u \to v,\, k} = f(e) \leq c(e)$$

We also want to ensure we do not over-allocate traffic. Thus, for each path from node *u* to *v,* we require that the traffic flow is less than the demand between those nodes, *d(u,v)*. Note that the network gave demands for pairs of nodes at multiple time steps. To simplify this, we just set *d(u,v)* to be the maximum demand for the pair *(u,v)* for all time steps. Note that the These constraint can be represented as:

$$\forall u, v \in N, \forall k,\ P_{u \to v,k} \leq d(u, v)$$

Finally, with these constraints, the model can optimize our variables to maximize our throughput. To do so, we set the objective of the model to maximize the sum of all path flows, i.e. our path variables. All flow is represented by a path variable, and so to maximize total flow/throughput, we just need to maximize our sum. Formally our objective is:

$$max \sum_{u,v \in N,\, k} P_{u \to v,\, k}$$

As for the algorithm to minimize the maximum link utilization, the model is much the same. We declare path variables for each path in our network, and implement the capacity and demand constraints as described. However, we defined a new variable in our model, MLU, representing the maximum link utilization in the network. Link utilization is defined as the ratio between the flow through an edge/link *e*, *f(e)* and the capacity of *e*, *c(e)*. Since the MLU is the maximum link utilization, we add constraints that force this variable to be greater than or equal to the link utilization of all links in the network. Formally, these constraints are:

$$\forall e \in E, \frac{\sum_{u,v \in N, k} 1[e\ is\ on\ the\ kth\ path\ from\ u\ to\ v] P_{u \to v, k}}{c(e)} = \frac{f(e)}{c(e)} \leq MLU$$

For our optimization, we then want to minimize the *MLU* variable. However, the easiest way to minimize link utilization is just to not send any flow at all. In order to avoid this trivial case, we added a second objective to maximize throughput. Since Gurobi can only minimize or maximize, we sought to minimize both the *MLU* and the negative total throughput:

$$min\ MLU - \sum_{u,v \in N, k} P_{u \to v, k}$$

Note that this simplistic objective is only successful when there are multiple ways to maximize throughput, as the model will then choose the one with the lowest MLU. We could then use Gurobi to optimize these two models. After the software had finished for each model, we iterated through the links of the network, and summed the optimized values of the path variables that passed through the corresponding edges. This in turn gave us the traffic flow through each link.

In addition to these two algorithms, we also ran tests to see how the Gurobi optimization software scaled with topology size. Using the “networkx” library, we randomly generated graphs with set numbers of nodes and random edge placements and capacities. Then, for each pair of nodes in each graph, we randomly generated a value for the demand. We then ran the Gurobi software, measuring the amount of the time it took to optimize the model.

Results

Using the edge flows returned from each algorithm, we were able to visualize the corresponding graphs with "networkx" to display the allocated flows.

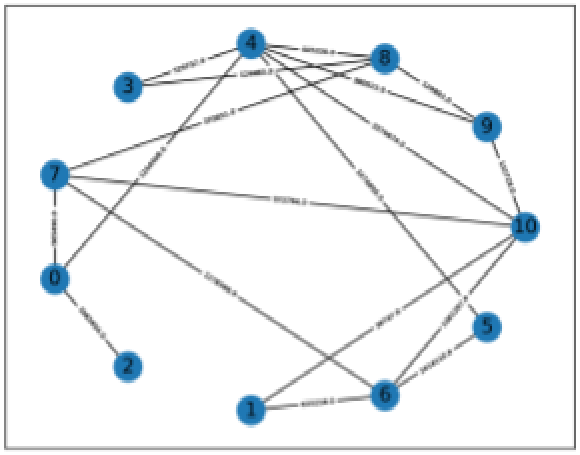

Figure 2. Maximized Throughput Traffic Flow

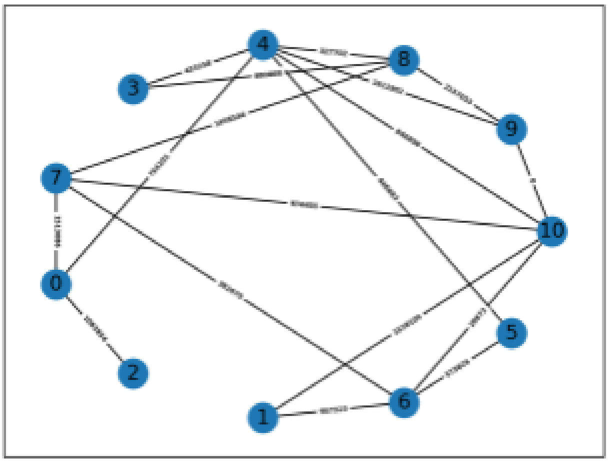

Figure 3. Minimized MLU Traffic Flows

Then, using these flow allocations, we can graphically compare the amount of flow applied to each edge by the two algorithms. Looking at figure 4, we see that when maximizing throughput alone, there is a single edge from node 4 to node 10 that is given 3000000+ Gbps, being an outlier. For the MLU allocations, the traffic is more evenly distributed over links. This is because by minimizing the MLU, we prevent any single link from being allocated an exorbitant amount of flow.

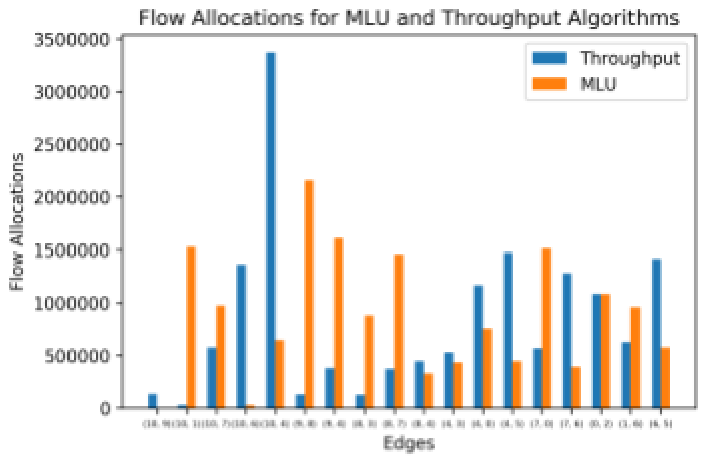

Figure 4. Flow Allocations Chart

This in turn makes minimizing the MLU more resilient to failure. With the maximize throughput allocations, there is a single link with lots of flow. If this link between nodes 10 and 4 happened to fail, it would be catastrophic with large amounts of traffic being lost and heavy re-routing would be required. However, by minimizing the MLU, we

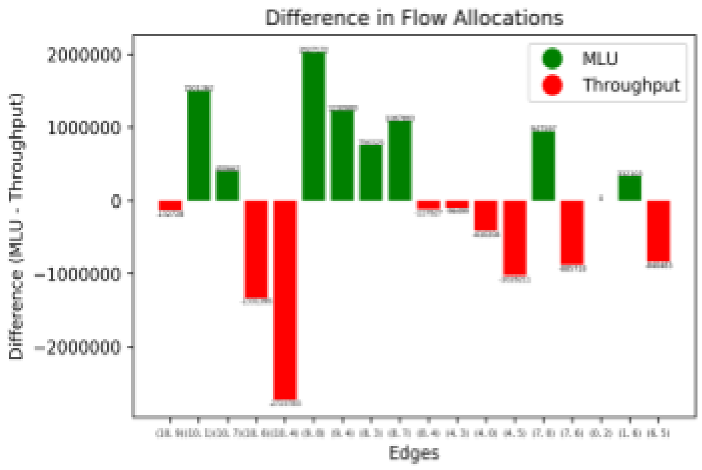

Figure 5. Differences between flow allocations for edges

avoid this. While a link failing will still cause a large amount of traffic to be lost, it will not be nearly as bad as that single link failing in the other case. By spreading out traffic, there is less potential for a single link failure to devastate the network, making this method of flow allocation more resilient in such cases.

Note that there are some benefits of the throughput method of allocation however. Simply because we are not constrained by having to minimize the MLU, maximizing the throughput can allow for a variety of different configurations. The configurations can have more freedom in the paths that they choose to direct flow. Thus it is possible that in the case of maximizing throughput alone, shorter paths can be taken, which in turn would allow for lower latency in the network, a metric that was not optimized in this project.

As for topology scaling, we measured the time it took for the Gurobi software to optimize randomly generated graphs of various sizes for both of our optimization problems. Initially, we ran our algorithms just as we did for the Sprint network implementation.

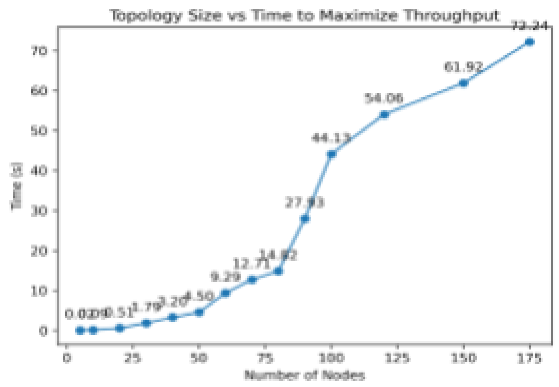

Figure 6. Time to maximize throughput, multiple paths per node pair

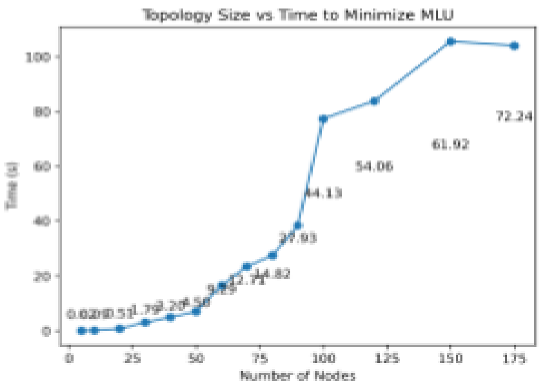

Figure 7. Time to minimize MLU, multiple paths per node pair

Here we see that there is a non-linear, strictly increasing trend in how long it takes for Gurobi to optimize a network with respect to topology size. As your network grows, the time it takes for Gurobi to optimize a flow problem will grow as well, and at a much faster rate. Note that this is only the time it took for Gurobi to optimize, and does not include the time it took to create all of the constraints. In fact, it took over an hour to get the two data points for the 175-node network, due to the sheer number of paths that needed to be considered.

Conclusion

In completing this project, we have implemented basic algorithms for software-defined traffic engineering, and have seen how they differ in their allocations of traffic flow. While maximizing the total throughput of the network is simple on its own, it can lead to a network that has heavily loaded links and is not resilient to failure. Minimizing maximum link utilization can fix this by spreading out network traffic, but it is also more restrictive and can lead to trivial allocations if not paired with another objective.

In addition, we discovered one of the main limitations of software-defined traffic engineering: it struggles to scale. As your network topology increases, so does the complexity of the linear program that needs to be solved. More nodes and edges mean more paths, and more paths mean more constraints and variables. The amount of computation needed to optimize a network will quickly bloat, far faster than the rate at which the network is growing. This makes it difficult to quickly calculate flow allocations with changing demands, making it somewhat infeasible to use with larger networks. We conclude that it is better to apply these algorithms for small networks, or at higher levels of abstraction such as between autonomous systems, as there are less nodes and fewer variables that would need to be optimized.

References:

Gurobi Optimization. "Documentation." *Gurobi Optimization*, 24 Aug. 2022, https://www.gurobi.com/documentation/.

Rachee Singh, Nikolaj Bjørner, and Umesh Krishnaswamy. 2022. Traffic Engineering: From ISP to Cloud Wide Area Networks. In The ACM SIGCOMM Symposium on SDN Research (SOSR '22), October 19–20, 2022, Virtual Event, USA. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3563647.3563652

"What Is Software-Defined Networking (SDN)?: Vmware Glossary." *VMware*, 9 Aug. 2022, https://www.vmware.com/topics/glossary/content/software-defined-networking.html.