Traffic Engineering with Evolution Strategies and GraphSage

1st Kyle Thompson
Computer Science
Cal Poly San Luis Obispo
San Luis Obispo, USA
rkthomps@calpoly.edu

***Abstract*—Software Defined Networking (SDN) has enabled a smarter and more flexible internet. SDN has been has been successful enough to reignite the communities interest in Knowledge Defined Networking (KDN). The goal of KDN is roughly to provide a more autonomous internet. One way to increase the level of autonomy in the internet is though smarter routing policies. In recent years, researchers have studied methods for creating adaptable routing policies that minimize the delay of a network. This field is called Traffic Engineering (TE). One promising direction for TE is the use of Deep Reinforcement learning (DRL) to guide the Internet's routing policies. In this work, I evaluate the performance of a DRL agent that represents the network using the GraphSage algorithm. I train the agent using a recently developed method called Evolution Strategies (ES).**

***Index Terms*—Traffic Engineering, Reinforcement Learning, Graph Neural Networks**

I. Introduction

In 2003, David Clark proposed a knowledge plane for the Internet to maintain a high-level view of the Internet's goals and permit its automatic adjustment and repair [1]. However, at the time Clark made this proposal, the Internet remained distributed in nature. Nodes had little information about the network as a whole and few computing resources. Software defined networking has provided a central logical view of the network. This central view, along with increased compute at the switches of the network has enabled a new paradigm called Knowledge Defined Networking [2]. Knowledge Defined Networking uses a knowledge plane above the management and control planes to automatically configure the network according to an encompassing view of the network. Towards this end, reinforcement learning algorithms have been used to address various problems in networking [3]. TE is one such problem.

Classical reinforcement learning entails a lookup between state spaces and action spaces. However, because TE entails very complicated state and action spaces, classical reinforcement learning is infeasible. However, recent advances in DRL offer mechanisms to handle more complex state and action spaces [3]. In this work, I study a DRL agent for TE over a network of Autonomous Systems (ASes). At each node in the network, my agent creates a local representation using a popular Graph Neural Network (GNN) called GraphSage [4]. I train my agent using a recently developed method called Evolusion Strategies. I construct a real-world AS-level topology as a testbed for my agent. I evaluate my agent against various classical AS-level routing strategies.

II. Related Work

There is existing work on using DRL for TE. In any application of reinforcement learning, there must be a state space and an action space. The agent learns a policy for choosing actions over the state space [3]. Stampa et al. present the first effort to use DRL for traffic engineering [5]. For the state space, Stampa et al. use a traffic matrix which includes the bandwidth request for each source-destination pair, and for the action space they use the link-weights for all paths in the network. Xu et al. improve on this approach by showing that using existing algorithms for TE as a baseline for exploration of the action space improves training behavior [6]. They use a state space of throughput and delay between source-destination pairs. Sun et al. improve on the state of the art with the key insight that the state and action spaces can be simplified by selecting "driver nodes" according to control theory [7]–[9]. The authors then monitor and control the network solely through the driver nodes. The authors show that this technique increases the robustness of their model to changes in topology. However, they do not discuss the implications of one of the driver nodes being lost in the topology change. Suarez et al. provide an architecture that is even more robust to topology changes by representing the state space as individual paths between source and destination nodes [10]. Therefore, the state-space is invariant to the network topology because any topology has paths between the source and destination nodes.

Rusek et al. use Graph Neural Networks (GNNs) in the context of TE [11], [12]. However their GNNs are used for prediction of delay and throughput of a network instead of being used in a representation of the state space in a DRL agent. Almasan et al. offer a system that is most similar to the present work [13]. This work uses a type of GNN based on the message passing scheme presented by Gilmer et al. [14].

III. Constructing an AS-Level Topology

A. Creating Fully-Reachable Topology

I construct an AS-Level Topology using the API provided by PeeringDB[1]. Specifically, I use four of the objects provided

[1]https://www.peeringdb.com/

by the API:

- **org**: Gives information such as name, address, website, etc. about organizations on the Internet.
- **net**: Gives information about each AS on the internet including its parent organization.
- **ix**: Gives information about each Internet Exchange (IX) on the internet including its parent organization.
- **netixlan**: Gives information about a connection between a single AS and a single IX including the bandwidth of link used in the connection.

First, I use the net, ix, and netixlan objects to create a graph of the complete Internet topology reported by PeeringDB. Note that in this graph, each of an IX's neighbors is an AS, and each of an AS's neighbors is an IX. Then, I find all connected-components of the graph and select the largest. Reducing the graph to its largest connected-component enables the assumption that there exists a path between any pair of nodes in the graph. Additionally, it guarantees that the location of each node in the graph can be interpolated after some number of iterations. This guarantee is necessary for this work since I use distance as a proxy for the delay of the network. The complete Internet topology consists of 27,129 nodes and 45,472 edges. The largest connected-component consists of 14,446 nodes and 45,375 edges.

B. *Finding the Location of each Node*

Since every AS and IX reported by PeeringDB is associated with an organization, I use information from the organization to infer a latitude and longitude for each AX and IX. I use four methods to find the location of an organization.

1) PeeringDB reports the latitude and longitude of the organization (3,696 nodes).
2) PeeringDB reports the zipcode of the organization and I infer its latitude and longitude using a zipcode dataset[2]. (3,584 nodes).
3) PeeringDB reports the country of the organization and I infer its latitude and longitude using the same zipcode dataset (5,817 nodes).
4) PeeringDB reports no geographical information about the organization and I interpolate its latitude and longitude with the mean of its neighbors (1,349 nodes).

After some number of rounds of interpolation every node will have a latitude and longitude because the graph is composed of a single connected component. Fig. 1 shows the final AS-Level topology.

IV. Simulating BGP

In this work, I compare five routing inter-AS routing policies.

- **Hops**: An AS routes a packet to the neighbor with the smallest number of hops to the destination.
- **Distance**: An AS routes a packet to the neighbor with the smallest distance to the destination. I calculate distance as the sum of the link lengths on the path from the neighbor to the destination.

[2]https://www.listendata.com/2020/11/zip-code-to-latitude-and-longitude.html

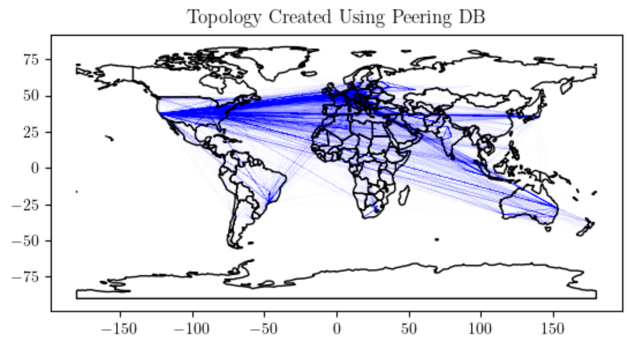

Fig. 1. Map of Resulting AS-Level Topology

- **Bandwidth**: An AS routes a packet to the neighbor with the largest average bandwidth to the destination. I calculate average bandwidth as the sum of link bandwidths on the path from the neighbor to the destination divided by the number of links on the path.
- **Distance-Bandwidth**: An AS routes a packet to the neighbor with the smallest distance-to-bandwidth ratio to the destination. I calculate distance-to-bandwidth ratio of a path as the quotient of the sum of distances and the sum of bandwidths on a path from the neighbor to the destination.
- **Weight**: An AS routes a packet to the neighbor with the largest weight. In this work, weights are assigned by a neural network.

A. *Broadcasting*

Each of the five routing policies this work compares are implemented via routing tables. Each node's routing table has an entry for every reachable node. Each reachable node's entry contains statistics about the best path to to the reachable node. For example, consider the routing table for `node1` below.

```
{
  node2: {
    hops: 2,
    distance: 18,893m,
    bandwidth: 10,323mbps,
    bandwidth-aware-hops: 5,
    distance-aware-bandwidth: 15,342mbps,
    bandwidth-aware-distance: 23,333m,
    weight: 123
 },
 node3: {
    hops: 1,
    distance: 19,333m,
    bandwidth: 11,112mbps,
    bandwidth-aware-hops: 3,
    distance-aware-bandwidth: 11,332mbps,
    bandwidth-aware-distance: 11,111m,
    weight: 11
  }
}
```

The entry “hops” in `node2` describes the minimum number of hops on a path from `node1` to `node2`. The entry “distance” is similarly interpreted as the minimum distance on a path from `node1` to `node2`. The entries “bandwidth” and “bandwidth-aware-hops” are used together to describe the maximum average bandwidth of a path from `node1` to `node2`. Because “bandwidth” and “bandwidth-aware-hops” describe the same path, they must be updated together. Similarly, “bandwidth-aware-distance” and “distance-aware-bandwidth” together describe the minimum distance-to-bandwidth ratio of a path from `node1` to `node2`. Therefore, “bandwidth-aware-distance” and “distance-aware-bandwidth” must be updated together.

Though each routing table in this work only contains readability information about the node hosting the table, it is important to note that in practice, ASes store both their own reachability information and reachability information about their neighbors. However, for purposes of simulation, it is more convenient to store a single copy of routing information accessible by pointers.

I use a broadcasting algorithm to ensure that each node is aware of all possible paths in the network. That way, each node’s routing table is guaranteed to accurately characterize network routes.

Algorithm 1: Broadcasting Algorithm

```
for $1 \leq$ iteration $\leq$ Network Diameter do
    foreach Node $\in$ Network do
        foreach Neighbor, Link $\in$ Node.Neighbors do
            foreach Entry $\in$ Node.RoutingTable do
                NewRoutingEntry =
                  ComputeRoutingEntry(Entry, Link)
                ModifyOrAdd(Neighbor.RoutingTable,
                  NewRoutingEntry)
            end
        end
    end
end
```

Each node in the network begins with a trivial routing table. That is, each node has a routing table only containing the path to itself. Such a path has 0 hops, 0 distance, 0 bandwidth and 0 distance-bandwidth. For a number of iterations equal to the diameter of the network, each node compares its routing table to the routing tables of its neighbors. At each iteration, each node modifies each of its neighbors routing tables. First, for each entry in its routing table, the node computes the entry as it would appear to the neighbor by accounting for the link distance and bandwidth between the node and the neighbor. If the destination associated with the new entry does not appear in the neighbor’s table, the node adds the newly-computed entry to the neighbor’s table. Otherwise, if the destination does appear in neighbor’s table, but there exists a better route going through the node by one of the five metrics, then the given metric is updated with the new route. Keep in mind that some pairs of metrics must be updated together.

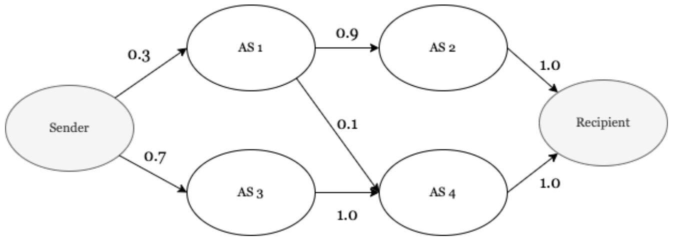

Fig. 2. A scenario in which multiple routes exist between the Sender and the Recipient. Edge labels originating from an AS denote the proportion of traffic it sends to each of its neighboring ASes.

B. Simulating Network Traffic

After broadcasting routing tables so that every node’s routing table accurately describes all paths through the network, I compare the expected delay of the five routing policies under randomly generated network traffic. Since I do not have information about how long a packet takes to traverse each link in the network or information about how long a packet takes to traverse each AS in the network, I assume that the delay of a packet is proportional to the distance travelled by the packet.

To randomly generate traffic over the network, I first randomly select pairs of nodes in the network to communicate. Any node in the network has probability p_{send} to be sending data to any other node in the network. If a node is sending data to another node in the network, the amount of data it sends is determined by a normal distribution with mean μ and standard deviation $\mu/4$[3].

Though I could simulate network traffic at the packet level, it is more efficient to simulate traffic at the flow level. To simulate traffic at the flow level, I use the network’s routing policy to generate a set of paths to from each source node to each recipient node. There can be more than one path between each source node and each recipient node because an AS can simultaneously send traffic from the same flow down multiple links. For example, consider Fig. 2. In Fig. 2, the proportion associated with a link is equal the proportion of traffic the AS sends to the link. In this scenario, there are three routes from sender to receiver. Furthermore, each route gets a particular proportion of the total traffic flowing from sender to receiver. Specifically, the routes and their proportion of traffic are as follows.

```
Sender -> AS1 -> AS2 -> Recipient; 0.27
Sender -> AS1 -> AS4 -> Recipient; 0.03
Sender -> AS3 -> AS4 -> Recipient; 0.70
```

In this work, an AS sends traffic associated with the same flow to different links if there is a tie in the metric describing the best path from the link. In this case, an equal amount of traffic is sent to each link involved in the tie.

[3] $\mu/4$ is an arbitrary value. A detailed measurement of network traffic might reveal a more accurate standard deviation

Let $\mathcal{P}_{i \longrightarrow j}$ be the set of paths from sender i to node j in the network. For each $Q \in \mathcal{P}_{i \longrightarrow j}$, the proportion of traffic from i flowing through Q to j is equal to the product outgoing link proportions along Q. Thus, the proportion of traffic from i flowing through j is the sum of the proportions of traffic from i flowing to j along each $Q \in \mathcal{P}_{i \longrightarrow j}$.

Because I can the proportion of traffic flowing through each node given a sender and receiver pair, I can calculate the total traffic flowing through each node over all sender-receiver pairs. Furthermore, this implies that I can calculate the amount of traffic that each node is attempting to send to each of its outgoing links. I define the **attempted utilization** of a link to be the sum of traffic an AS attempts to sent to the link over all communicating pairs in the network.

C. Calculating Network Delay

To estimate the expected delay of a packet flowing along path $Q \in \mathcal{P}_{\text{recipient}}$, I calculate the expected distance the packet will travel while traversing the path. Of course, the lower bound of this distance is the sum of the link lengths along the path where the length of each link is the haversine distance between the nodes at the ends of the link. However, if the attempted utilization of the link is greater than its bandwidth, then some number of packets will be dropped from the link. I estimate the probability of successfully traversing link l_i to be p_i where p_i is modeled as:

$$p_i = \frac{\text{bandwidth}_i}{\max(\text{bandwidth}_i, \text{attempted utilization}_i)} \tag{1}$$

Those packets that are dropped must traverse the network once more for another chance to cross the link. Suppose that path Q is composed of the list of links $(l_1, l_2, l_3, ..., l_n)$. Furthermore, let d_i be the length of link l_i. I calculate the expected distance of a packet traversing Q as follows.

$$\mathbb{E}[\text{Distance}(Q)] = \sum_{i=1}^{n}(\frac{1}{p_i} - 1)(\sum_{j=1}^{i-1} d_j) + d_i \tag{2}$$

Note that the term $1/p_i - 1$ is the expected number of times a packet will have to re-traverse a network to get to link l_i.

Let $P(Q)$ be the proportion of traffic sent along path Q in a session of communication from $i \longrightarrow j$. I can calculate the expected delay of a packet sent from node i to node j as

$$\mathbb{E}(\text{Distance}(i \longrightarrow j)) = \sum_{Q \in \mathcal{P}_{i \longrightarrow j}} P(Q) \cdot \mathbb{E}(\text{Distance}(Q)) \tag{3}$$

While simulating traffic over the network, there are many communicating pairs. In order to report a single metric to describe the performance of a routing policy, I sum the expected distances for all sender and receiver pairs and divide by the sum of haversine distances between all sender and receiver pairs. I call the result the **delay inflation factor**. If my assumption holds that distance traveled by a packet is proportional to the delay of the packet, then the delay inflation factor of a network describes how much slower the network is under a particular routing policy than if there was a direct link between every pair of communicating parties.

V. Assigning Weights with GraphSage

What if the optimal routing policy is not a pure strategy such minimizing the number of hops to the destination or maximizing bandwidth to the destination? Furthermore, the routing policies hops, distance, bandwidth, and distance-bandwidth do not consider the local attempted utilization information that is available to each AS in the network. To create a flexible routing policy that accounts for the utilization of an AS's outgoing links, I train a neural network to assign a weight to each routing entry in the AS-level network. The neural network first creates a **base embedding** for each AS in the topology. The base embedding for a node v is a vector $\vec{b_v}$. The vector $\vec{b_v}$ embeds information about the node and its local region in the topology. To create a base embedding for each AS, I use the GraphSage algorithm [4]. The GraphSage algorithm provides a method of updating the embedding of v with the embeddings of other nodes in its neighborhood [4]. Let K denote the number of times we update a node's embedding with the embedding of its neighborhood. Let N_v^i be a matrix whose rows are the embeddings of a node's neighbors after iteration i. Since a neighbor's embedding in N_v^i does not necessarily contain information about the link connecting v and the the neighbor, I append the appropriate link bandwidth, attempted utilization and link length to each row in N_v^i. Let A_i be a matrix of weights used to aggregate a node's neighbors at iteration i. Let W_i be a matrix of weights used to update a node's embedding at iteration i. "relu" denotes the element-wise nonlinearity relu. At each iteration $i < K$, the embedding at each node is updated as follows [4]. The colon represents the concatenation operation.

$$\begin{aligned} \vec{a_v^{i+1}} &= \text{Column-wise } \max(\text{relu}(N_v^i A_i)) \\ \vec{b_v^{i+1}} &= \text{relu}(W_i(a_v^{i+1} : b_v^i)) \end{aligned} \tag{4}$$

Of course, for this algorithm to work, each node must begin with a base embedding. The base embedding for each node is a length-one vector containing the degree of the node. For the GraphSage algorithm to perform well, I normalize all of these features. To normalize degree, I divide by the number of nodes in the network. To normalize bandwidth, I divide by the maximum bandwidth of any link in the network. To normalize length, I divide by half the circumference of the earth.

After finding a base embedding for each node in the network, I use the base embeddings to assign weights to each entry of a node's routing table. Specifically, I concatenate the base embedding with an embedding for the node's routing table. The embedding for an entry in a node's routing table is a length six vector containing hops, distance, bandwidth, bandwidth-aware-hops, distance-aware-bandwidth and bandwidth-aware-distance. Again, I normalize all values of hops by the number of nodes in the network, all distances by half the circumference of the earth, and all bandwidths by the maximum bandwidth of any link in PeeringDB. Let $\vec{e_r}$ be the embedding of entry r in a node v's routing table. Let O

be a matrix of weights. Then, the weight for entry r in node v's routing table is given as:

$$\text{weight of } r = O(\vec{e_r} : \vec{b_v}) \tag{5}$$

Given a destination, an AS uses weights to route traffic by routing to the neighbor that has the highest GraphSage-assigned weight associate with the destination in its routing table. Different from other routing policies discussed in this work, the weight-based policy accounts for utilization in the network. However, since utilization is a dynamic entity in the network that depends on the weight assignment itself, I perform iterations of weight assignment and simulation to allow for stabilization of the routing-entry weights.

VI. Training with Evolutionary Strategies

To train GraphSage to assign useful routing-entry weights, I must find a list of parameters θ for each A_i, each W_i and O such that a routing policy based on the routing-entry weights assigned under θ achieves low delay. This can be framed as a reinforcement learning problem where the action space of a network consists of assigning weights to each routing-entry in the network and the reward is the negation of the delay inflation factor of the resulting network. I use the Evolution Strategies algorithm [15] to find some θ that leads to a performant routing policy.

Evolution Strategies (ES) differs from other reinforcement learning algorithms such as REINFORCE [16] or Q-learning [17] in that ES explores the action space by adding randomness to the policy network used to select actions rather than adding randomness to the actions themselves [15]. Furthermore, ES is more simple to implement than Q-learning or Reinforce because it only requires a forward-pass through the network [15]. Also, ES is performs well in scenarios with sparse reward signals [15]. My problem consists of very sparse reward signals since the network of ASes must assign $O(n^2)$ routing-entry weights for some number of iterations before it can determine the delay inflation factor of the network.

To use ES to find some θ that leads to a low delay inflation factor, I perform a number of training iterations called epochs. In each epoch, I generate a set of **perturbations** $\mathcal{V}$ [15]. Each perturbation is composed of a list of real numbers $\mathcal{E}$ such that $|\mathcal{E}| = |\theta|$ [15]. Each perturbation yields a new list of parameters. Let σ be a hyperparameter associated with the standard deviation of the perturbations. Then, each perturbation yields a new set of parameters θ' as shown below

$$\theta' = \theta + \sigma \cdot \mathcal{E} \tag{6}$$

Then, I compute the delay inflation factor associated with using the GraphSage model parameterized by θ' to assign routing-entry weights. The reward associated with this perturbation is the negation of the resulting delay inflation factor since ES is designed to maximize reward. For perturbation $V \in \mathcal{V}$, let $\mathcal{E}_V$ be its corresponding list of real numbers and R_V be its corresponding reward. Let α be a hyperparameter

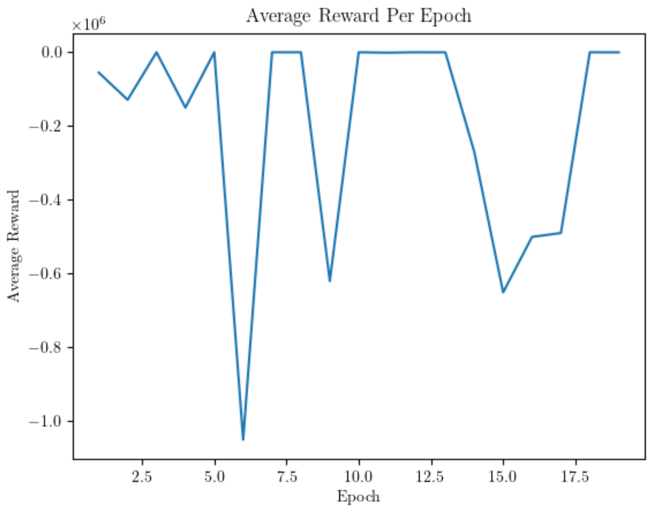

Fig. 3. Training Behavior for GraphSage Agent. The x-axis denotes the training step. The y-axis denotes the negative delay inflation factor.

called the learning rate. Then, we update θ at the end of the epoch with [15]:

$$\theta = \theta + \frac{\alpha}{\sigma|\mathcal{V}|} \sum_{V \in \mathcal{V}} \mathcal{E}_V \cdot R_V \tag{7}$$

Note that the main hyperparameters associated with ES are α, σ, the number of perturbations, $|\mathcal{V}|$, and the number of epochs.

A. *Training Behavior*

I train the final Graph Sage agent over 20 training steps. Since simulating the delay of the entire 14,446 node network is computationally infeasible, I randomly sample a 20-node connected component from the network for each training step. In each training step, I generate 16 perturbations of the network. Each perturbation uses $\sigma = 0.1$. Each parameter update uses $\alpha = 0.001$. The negative average delay inflation factors over the perturbations at each training step are shown in Fig. 3. One can see that the training behavior varies wildly. There are number of reasons why this could be the case. For one, 20 training steps is small compared to many deep learning models. Also, 16 perturbations is analogous to having a batch size of 16 which is also comparatively small. I used a small number of training steps and a small number of perturbations because of the cost associated with computing the delay inflation factor. Even with these hyperparameter settings, it took roughly 3 hours to train the model on a Macbook Pro Laptop with a 2.9 GHz Quad-Core Intel i7 Processor.

VII. Known Problems in Methodology

There are a number of oversights that I made when designing the simulation of routing policies. One such oversight is that I allow the attempted utilization of a link by an AS to be greater than the sum of the bandwidths of the incoming links to the AS. Therefore, the metrics reported in this paper report higher delay inflation factors than they should. My simulation

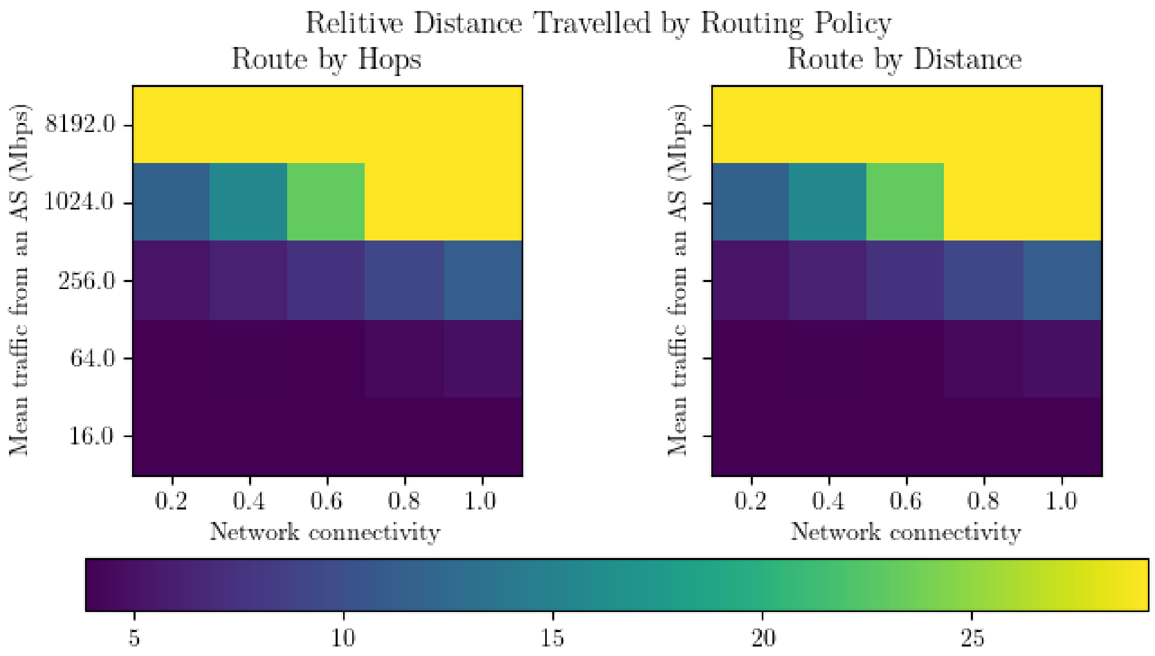

Fig. 4. Comparison of delay inflation factor between the routing policies hops and distance. P_{send} is plotted on the x-axis of each of the plots. μ is plotted on the y-axis of each of the plots. The color of each square indicates the delay inflation factor of 3 randomly-generated network topologies of the network given P_{send}

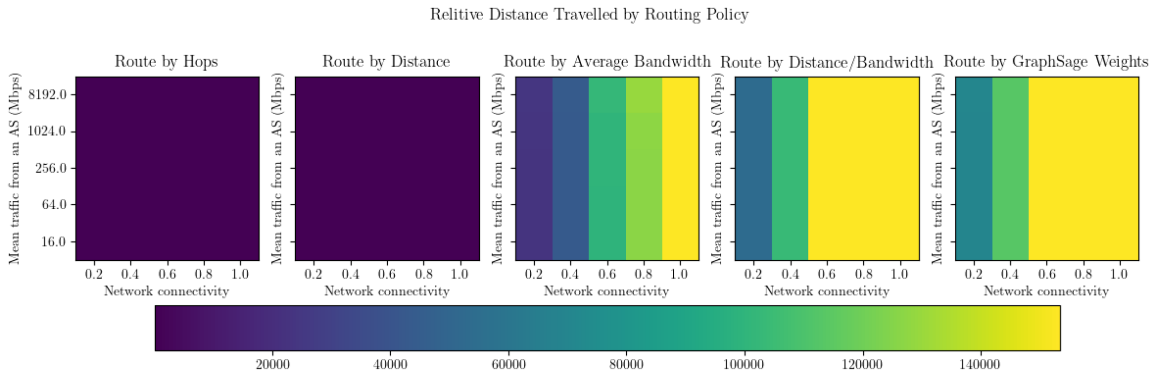

Fig. 5. Comparison of delay inflation factor between the routing policies hops, distance, bandwidth, distance-bandwidth, and GraphSage. The plots are generated in exactly the same way they are generated in Fig. 4

methodology would have to undergo major changes to account for this problem. Currently, for a single pair of communicating nodes, I am able to calculate the proportion of traffic from the pair flowing through any AS of the network *independently* of other pairs of communicating nodes as shown in Section IV-B. However, this methodology reports the correct numbers only when no links have true attempted utilization greater than their bandwidth. Otherwise, it overestimates the attempted utilization of downstream links. Addressing this problem is the most pressing area of future work for this paper.

Another flaw of this paper is in the way that the GraphSage model assigns link weights to routing entries. Currently, a node's base embedding is used in conjunction with a vector representation of a given routing entry in order to produce a weight for the routing entry. The problem is that this weight is used by a node's neighboring ASes. In the real world, neighboring ASes would not easily be able to compute this weight because they do not have direct access to the outgoing link utilizations of a node. To remedy this flaw, I could store routing entries for all of a node's neighbors at the node and use the node's base embedding to calculate a weight for each entry. However, this solution entails more strenuous memory requirements.

VIII. Results

I compare the delay inflation factor for hops, distance, bandwidth, distance-bandwidth, and the GraphSage Model. Since there is major variation in the delay inflation factors associated with these routing policies, I show them in two separate plots: Fig. 4 and Fig. 5. From Fig. 5, it is obvious that routing policies based on hops and distance alone had the lowest delay-inflation factors. From Fig. 4, one can see that hops and distance perform nearly identically at minimizing the delay inflation factor. There are a number of surprises in these results. First, it is surprising that routing according to distance-to-bandwidth ratio has a higher delay inflation factor than routing according to bandwidth alone. Before examining these results, I expected that routing according to distance-to-bandwidth ratio would perform better than distance alone because it accounts for possible congestion. Also surprising is that the performance of the GraphSage agent is nearly identical to routing by the distance-to-bandwidth ratio. As shown in Fig. 3, the GraphSage agent did not show promising training behavior so I expect that it routes almost randomly. However, it is puzzling that routing based on distance-to-bandwidth ratio also exhibits this behavior.

IX. Conclusion

In this work, I implement a new DRL agent that uses the GraphSage algorithm to create a local embedding at each node. I train this agent using ES. As a testbed for comparing my agent to existing routing strategies, I use data available on PeeringDB to create a realistic AS-level topology. I found that the computational requirements associated with calculating the delay-inflation-factor make training the DRL agent difficult. Due to the difficulty in training the DRL agent, it was not competitive against traditional routing policies such routing by hops or routing by distance. I curiously found that routing by distance-to-bandwidth ratio behaves very poorly.

There are a number of ways one could continue this work. Most importantly, one would have to address the issues highlighted in Section VII. The first problem is a flaw in my calculation of attempted utilization. To my knowledge, fixing this flaw would increase the time-complexity of calculating the delay inflation factor by at least an order of magnitude because it introduces a dependency between flows. The second problem is a flaw in the way GraphSage assigns weights to routing entries. For the current architecture to be implemented, a node must have knowledge about its neighbor's link utilizations which is an unrealistic assumption. Beyond these pressing flaws in methodology, the results of this work may be improved by calculating the delay inflation factor in a more performant programming language. Also, one could investigate the poor performance of the distance-bandwidth ratio routing policy.

References

[1] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski, "A knowledge plane for the internet," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 3–10.

[2] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett *et al.*, "Knowledge-defined networking," *ACM SIGCOMM Computer Communication Review*, vol. 47, no. 3, pp. 2–10, 2017.

[3] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[4] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[5] G. Stampa, M. Arias, D. Sánchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[6] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM 2018-IEEE conference on computer communications*. IEEE, 2018, pp. 1871–1879.

[7] P. Sun, J. Li, Z. Guo, Y. Xu, J. Lan, and Y. Hu, "Sinet: Enabling scalable network routing with deep reinforcement learning on partial nodes," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019, pp. 88–89.

[8] P. Sun, J. Lan, Z. Guo, Y. Xu, and Y. Hu, "Improving the scalability of deep reinforcement learning-based routing with control on partial nodes," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 3557–3561.

[9] P. Sun, Z. Guo, J. Li, Y. Xu, J. Lan, and Y. Hu, "Enabling scalable routing in software-defined networks with deep reinforcement learning on critical nodes," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 629–640, 2021.

[10] J. Suárez-Varela, A. Mestres, J. Yu, L. Kuang, H. Feng, P. Barlet-Ros, and A. Cabellos-Aparicio, "Feature engineering for deep reinforcement learning based routing," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[11] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, "Unveiling the potential of graph neural networks for network modeling and optimization in sdn," in *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019, pp. 140–151.

[12] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "Routenet: Leveraging graph neural networks for network modeling and optimization in sdn," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2260–2270, 2020.

[13] P. Almasan, J. Suárez-Varela, A. Badia-Sampera, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio, "Deep reinforcement learning meets graph neural networks: Exploring a routing optimization use case," *arXiv preprint arXiv:1910.07421*, 2019.

[14] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[15] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[17] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.