Modeling TCP Performance using Graph Neural Networks

Benedikt Jaeger, Max Helm, Lars Schwegmann, Georg Carle
{jaeger|helm|schwegmann|carle}@net.in.tum.de
Chair of Network Architectures and Services, Technical University of Munich, Germany

ABSTRACT

TCP throughput and RTT prediction are essential to model TCP behavior and optimize network configurations. Flows adapt their sending rate to network parameters like link capacity or buffer size and interact with parallel flows. Especially the elastic behavior of TCP congestion control can vary, even when only slight changes in the network occur. Thus, existing analytical models for TCP behavior reach their limits due to the number and complexity of different algorithms. Machine learning approaches, in contrast, are often fixed to specific network topologies.

This paper presents a TCP bandwidth and RTT prediction approach that can handle different algorithms and topologies. For this, we utilize Gated Graph Neural Networks and simulated network traffic. We evaluate different encodings of the input data into graphs and how network size, number of flows, and TCP algorithms influence prediction accuracy. Additionally, we quantify the impact of different input features on our models. We show that Graph Neural Networks can be used to model TCP behavior. The resulting models can predict RTT with a median relative error of 2.29 % and throughput with an error of 13.31 %.

CCS CONCEPTS

• **Networks → Network performance modeling**.

KEYWORDS

TCP modeling, congestion control, throughput, round-trip time, graph neural networks

ACM Reference Format:
Benedikt Jaeger, Max Helm, Lars Schwegmann, Georg Carle. 2022. Modeling TCP Performance using Graph Neural Networks. In *Graph Neural Networking Workshop (GNNet '22), December 9, 2022, Roma, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3565473.3569190

1 INTRODUCTION

The Transmission Control Protocol (TCP) is a widely used protocol on the Internet due to its multiple features. Besides offering a reliable, in-order byte stream, it provides built-in flow and congestion control (CC). The latter results in an elastic behavior of TCP's sending rate influenced by network parameters and traffic. Even in simple dumbbell topologies, this can lead to chaotic behavior when only one CC algorithm is involved [23], and it becomes even worse when multiple different ones are in use [13]. This complex behavior makes modeling and predicting performance metrics like Round-trip Time (RTT) and rate increasingly challenging. Early analytical approaches [14] have limitations due to newer, more complex algorithms and topologies. However, neural network approaches can provide solutions to this problem. Unlike analytical models, neural networks are data-driven, extracting important features from given data. There exist neural network approaches which utilize the way input data is structured. For example, convolutional neural networks, as used for image recognition, utilize that an input image is structured in a matrix of pixels. While this approach suits the given task, it is less applicable to computer networks that contain more complex structures. Graph Neural Networks (GNNs) have gained traction over the past years [22], also in the computer networking domain. Conventional neural networks, as applied for image recognition, use matrices as input, while GNNs operate on graphs. Graphs are defined as $G = (V, E)$ with nodes V and edges E, where each can be assigned attributes. This structuring of input data allows a higher expressiveness since not only node attributes are used as input but also the structure of the graph. For example, the graph could contain nodes for all end hosts and edges between all interconnected nodes in a computer network. Besides, the graphs are not limited to physical nodes or connections but can also contain logical connections. GNNs are based on the message passing concept. There, each node in the graph is represented by a state vector which is updated according to the state vectors of adjacent nodes. Doing this spreads information through the graph, resulting in an output matrix that can be used with conventional neural networks. The possibility of encoding data as graphs also raises the question of how this should be handled. Furthermore, by using arbitrary graphs as input, GNNs can generalize even for entirely new graphs unseen during training.

In this paper, we apply a GNN approach to model two TCP performance metrics, RTT and sending rate. We generate data containing different network topologies and CC algorithms using the *ns-3* simulator. We train a GNN and evaluate its accuracy. The impact of CC algorithms, the network topology, and how data is modeled as a graph is investigated. Our contributions in this paper are to provide a baseline for performance prediction of TCP for arbitrary topologies and CC algorithms using GNNs, compare different graph encodings, and assess the models' accuracy and feature importance. We want to provide a baseline of how GNNs can be used to model TCP CC and, by this, gain a better understanding of the complex behavior.

This paper is structured as follows. Section 2 covers the basics of TCP CC and GNNs, and related work in the field of TCP modeling is presented. Implementation details regarding the generation of the used dataset and neural network are displayed in Section 3. Section 4 presents results before Section 5 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section, we present background about TCP CC algorithms and GNNs as well as related work in the performance modeling of TCP.

2.1 Background

CC was added to TCP to prevent congestion collapse in the network in the early days of the Internet. It is usually done by each sender and regulates the amount of traffic sent into the network (e. g., decrease under congestion and increase with spare network capacity). This elastic behavior of TCP makes modeling and performance prediction more complex since network parameters like link delay, capacity, and loss rate impact the protocol, and particular algorithms interact differently. End-host-driven CC algorithms rely on feedback from the network to estimate the amount of traffic to be sent. Potential feedback mechanisms are packet loss or changing connection RTTs. Today, several approaches exist to implementing CC, i. e., congestion detection and potential reactions. A broad overview of TCP CC algorithms is presented by Afanasyev et al. [1]. In the following, different classes of algorithms used in this paper are briefly summarized.

Loss-based CC: The earliest approach to implementing CC was based on the assumption that packet loss is only caused by congestion. Whenever packet loss is detected, the amount of inflight data is reduced and increased otherwise. Inflight data, i. e., data sent but not acknowledged yet, is limited using a congestion window. Popular loss-based algorithms are *Reno*, *Bic*, and *Cubic*. *Reno* increases its congestion window linearly with the connection's RTT and reduces it by 50 % on packet loss. *Cubic* improves on weaknesses of *Reno* and increases its window following a cubic function independent of the RTT and only decreases the window by 30 % [9]. These algorithms always fill up the buffer at the bottleneck link leading to increased queuing delay and suffer from stochastic packet loss caused by reasons other than congestion.

Delay-based CC: This class of algorithms relies on the measured RTT of the connection to estimate congestion. Whenever a delay increase is detected, the congestion window is reduced. While this approach allows operating with the minimum possible RTT, it has fairness problems when operating beside a loss-based algorithm. Examples of delay-based algorithms are *Vegas* [2] and *Ledbat*. *Ledbat* is designed to utilize available resources and reduce its rate when other flows are present. This targets, for example, background bulk transfers (see RFC 6817).

Hybrid CC: Besides the classes above, hybrid approaches exist, which utilize packet loss and delay for deciding the congestion window size. Examples of this class are *Veno* [7] and *Illinois* [16]. In general, loss-based algorithms are robust with other flows on the path but fill buffers, increasing queuing delay. Delay-based algorithms achieve low rates when running in parallel and can operate with minimal RTT. The optimal operation point for delivery rate and RTT is with exactly one bandwidth-delay product (BDP) of data in flight.

BBR [4] recently introduced a new approach for CC by modeling the whole network path and setting the congestion window accordingly. We do not use this algorithm in this paper since we could not produce consistent results with the available *ns-3* implementation. At seemingly random times, the sending rate of *BBR* flows dropped to zero for the remaining time of the simulation.

Graph Neural Networks: In general, neural networks are designed to simulate neurons communicating with each other, inspired by the human brain. They consist of different layers of neurons, and each layer is fully connected with its adjacent ones. Deep neural networks consist of multiple such layers, for which training became faster and more feasible, with the possibility to offload computation to the GPU. Depending on the application, different neural networks have been proposed, like convolutional neural networks for image classification and recurrent neural networks for speech recognition.

A more recent neural network approach are GNNs [21]. They receive a graph $G = (V, E)$ as input consisting of a set V of nodes and a set E of edges between those nodes. Each node or edge can be assigned attributes. The input features for the GNN consist of those attributes as well as the structure of the graph. Computations and training on this input can then be done as follows.

Each node is initialized with a hidden state vector h, for which the node attributes can be used. The central concept behind GNNs is called *message passing* which considers the graph structure during computation. Each node's hidden state vector is updated during message passing according to functions f and g. For example, the updated hidden state vector h^{t+1} can be computed with

$$h^{t+1} = f(h^t, g(h_i^t, \dots, h_j^t))$$

using the previous H of the node and all its adjacent nodes $i, \dots, j$ as input. However, there exist different approaches to how this update function can be defined. In this paper, we use a simple sum $\sum$ for g and a Gated Recurrent Unit (GRU) for f. Message passing is done for a fixed number of steps and results in information passing through the graph. The output of the message passing phase can then be further processed or directly used for regression or classification. A more general explanation of the message passing step can be found in [22].

2.2 Related Work

Modeling latency and transmission rate of TCP in networks has already been done before. Early analytical approaches for modeling are usually limited to the simple TCP *Reno* algorithm. They use parameters like RTT and loss probability and are typically limited to simple problem sizes regarding the number of flows and topology size [5, 11, 14].

Other machine learning approaches exist, utilizing methods like support vector machines [18], long short-term memory cells [10], and neural networks [17]. However, all of the above are either limited to fixed topologies or only applied to basic TCP *Reno*.

Recent research showed that applying GNNs for network traffic modeling is promising. This paper follows the graph modeling approach by Geyer [8], who makes throughput predictions on arbitrary topologies with TCP *Reno*. The current baseline regarding delay prediction using GNNs is set by RouteNet [6, 20]. There, traffic is modeled following packet size and inter-arrival time distributions while we consider the TCP CC behavior. Also, they use link capacities of up to 100 kbit/s whereas we increase this to up to 100 Mbit/s. An overview on applications of GNNs in the networking domain is given in [22].

Parameter	Distribution	Unit
# Routers	$\mathcal{U}(2, 10)$	
Link rate	$\mathcal{U}(10, 100)$	Mbit/s
Link latency	$\mathcal{U}(5, 50)$	ms
Buffer size	$\mathcal{U}(1, 5)$	BDP
# Flows	$\mathcal{U}(1, 10)$	
Algorithms	*Reno*, *Cubic*, *Vegas*, *Bic*, *Ledbat*, *Illinois*, *Veno*	

Table 1: Parameters for the dataset generation

3 IMPLEMENTATION

This section covers the generation of the used dataset, the encoding into graphs, and the design of the GNN.

3.1 Dataset Generation

As a dataset for the neural network, we generated network configurations and then simulated them using *ns-3*. Network topologies are created using a random number of routers. For simplicity, we restricted the topologies to trees, so we do not need to handle any routing. However, the approach would allow arbitrary topologies. A random *Prüfer sequence* is generated, which can be mapped unambiguously to a tree. Routers in the network topology are linked according to the generated tree. Each link is assigned a link rate and link latency as the one-way propagation delay. Then flows are generated by randomly selecting a source and destination router. Source and destination servers are connected to the selected routers for each flow. Eventually, all network nodes not traversed by any flows are pruned. All components are randomly parameterized as given in Table 1 with $\mathcal{U}(x, y)$, meaning uniform distribution with minimum x and maximum y.

Based on these parameter distributions, other relevant characteristics of the generated dataset can be derived (cf. Table 2). Path rate is the minimum link rate on each flow's path, and path RTT corresponds to twice the sum of all link latencies on the path. On average, there are more than two flows on each link between the routers, i. e., running in parallel.

We simulate the generated topologies using *ns-3* [19] for 60 s and collect metrics. For each socket, we compute the average transmission rate as $\frac{\text{transmitted bytes}}{\text{duration}}$ and take the average RTT from the internal RTT estimation of the socket. Based on the number of simulated flows and the maximum transmission rates for the flows, the simulation took, on average, 260 s per topology and core on an *AMD EPYC 7542*.

We created 100 k data samples for training and another 10 k for validating using identical parameter distributions. The dataset was further split during training into 80 % train and 20 % test sets. The validation dataset was only used for the evaluation of the models.

The dataset is available along with the code [12].

Figure 2 shows the distributions of the simulation results for the RTT, rate, and queuing delay (as RTT − path latency) split into the different CC algorithms. Note that the queuing delay contributes about 10 % to the RTT.

3.2 Graph Representation

The generated data is encoded into a graph to be used as input for the GNN. This graph contains the topology of the data samples and all other information used as input features. We refer to this mapping as Graph Representation (GR). In this paper, we compare three different GRs.

Parameter	min	median	mean	max	std
Path rate	10	20	26.10	100	17.28
Path RTT	30	210	223.65	790	88.96
Path length	4	5	5.07	12	1.23
Flows per link	1	2	2.76	10	1.76

Table 2: Properties of the dataset

GR1: Each outgoing interface of a network node (routers and servers) is modeled as an interface node. For each link in the network topology, two interface nodes are added with the link latency, rate, and buffer size as attributes. Flows are modeled as flow nodes connected to each interface node they traverse. This allows the logical connection between the network nodes and traffic flows to be incorporated into one graph. Additionally, we assume for this graph that flows only interfere with each other if they share at least one outgoing interface. For example, in Figure 1b, F_1 and F_2 are connected via an interface node, while F_2 and F_3 are unconnected.

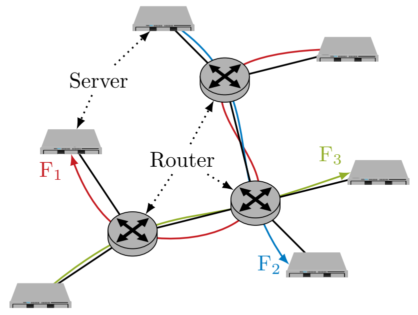
GR2: Since TCP features a feedback channel containing the acknowledgments (Ack), this information should also be modeled in the graph. This GR contains an additional Ack node for each flow node connecting all interface nodes from the acknowledgment path (i. e., the reverse flow path). In Figure 1b, F_1 and F_3 are now connected via the Ack node even though they do not share any nodes on the data path.

GR3: In this GR, additional path nodes are added between each flow and interface node, encoding its position on the path (e. g., first, second, …). This node is then connected to the flow-node / Ack-node and the interface node. The position on the path is a one-hot encoded attribute that encodes if the node is the first, second, …network interface on the flow's path.

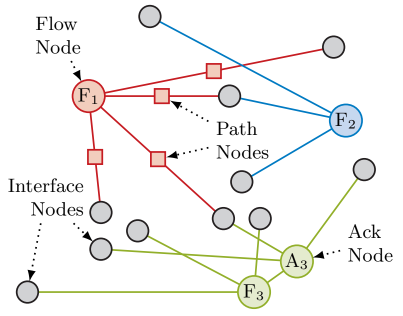
Examples for these GRs can be seen in Figure 1b. For simplification, Ack nodes are only shown for F_3 and path nodes only for F_1. We expect the third GR to be the most suitable since it includes the most information. Input features and output values are encoded as node attributes (Table 3), while edges only connect nodes and are otherwise not parameterized.

Matrix Conversion: This graph is then converted into a matrix of size $N \times F$, with N being the number of nodes in the graph and F being the length of the feature vector of each node (see Figure 1c). Categorical features are one-hot encoded (e. g., the node type, the CC algorithm, and the position on the path). The length of the feature vector varies between the different GRs. For GR1, it consists of 14 values (one for each of link rate, link RTT, queue size, and one-hot encoded three for the node type, and eight for the CC algorithm. GR2 increases the vector's size by one since an additional node type is added. For GR3, the size is 32, with one additional node type and the path order as one-hot encoded of length 16. Note that all rows have the same feature vector dimensions even though not all features are assigned to each node. These values are set to zero in the matrix.

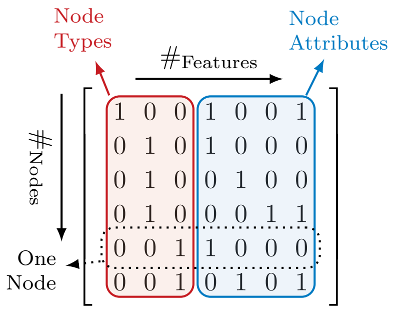
This conversion from a graph to a matrix can be applied to all kinds of graphs and is not limited by network size or the number of flows.



(a) Network topology with three flows



(b) Graph Representation of the topology



$$\begin{array}{ccc|cccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}$$

(c) Matrix representation

Figure 1: Conversion of the network data into an input matrix for the GNN

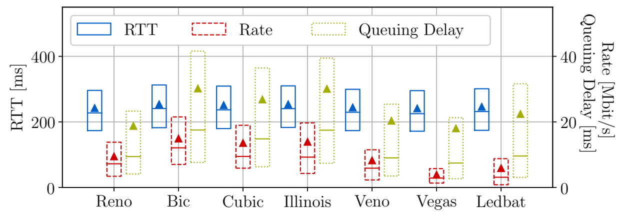


Figure 2: Simulation results for different algorithms. The horizontal line denotes the median, ▲ the mean.

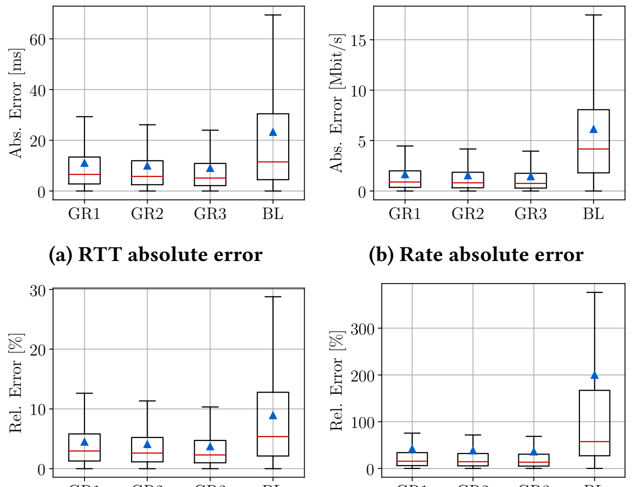
Node type	Node Attributes
Flow node	CC algorithm*, **flow RTT**, **flow rate**
Interface node	latency, rate, buffer size
Ack node	–
Path node	position on the path*

Table 3: Node attributes in the Graph Representation (attributes marked with * are one-hot encoded, bold attributes are used as output for the models)

3.3 Neural Network Design

For modeling, we follow the approach from [8] using a Gated Graph Neural Network (GGNN) [15] consisting of both a GRU and a Feed-Forward (FF) network.The input matrix of size $(N \times F)$ is first fed into a FF network resulting in an output of size $(N \times H)$ with H being the configurable hidden state size. Then the GRU component computes the hidden state updates during message passing, which does not change the shape of the matrix. Like a Long Short-Term Memory (LSTM) cell, a GRU implements gate mechanisms to store sequenced data like the message passing steps. Afterward, data is passed through another FF layer to process the final output, i. e., the flow RTT or rate. Eventually, the output matrix has a shape of $(N \times 1)$, and the output vectors can be interpreted depending on the row number of the input nodes. We train two independent models for the RTT and rate. Thus only one output value is provided for each node.

For training, we use the mean squared error (squared L2 norm) as the loss function. Only rows containing flow nodes are considered for predicting the flow RTT and rate by the loss function during the training process.



(a) RTT absolute error (b) Rate absolute error

(c) RTT relative error (d) Rate relative error

Figure 3: Comparison of the prediction error for different graph representations (GR) and the baseline (BL)

We use *nni*[1] with a subset of the training data for hyperparameter tuning. The implementation is realized using *PyTorch* and *PyTorch Geometric* and is available along with the dataset [12].

4 EVALUATION

This section presents results and insights gained from analyzing and applying the trained models. For the evaluation, we use the absolute and absolute relative error:

$$\text{Abs. error} = |y - y'| \qquad \text{Rel. error} = \frac{|y - y'|}{y}$$

with y being the actual value and y' the prediction. The shown boxplots contain the median as a horizontal line, the mean as ▲, the quartiles Q_1 and Q_3, and the whiskers show the 1.5 *interquartile range* $(Q_3 - Q_1)$.

4.1 Comparison of Graph Representations

First, we compare the three GRs and assess their prediction quality for RTT and rate.

[1]https://github.com/microsoft/nni

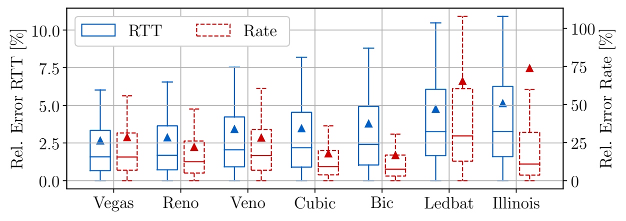


Figure 4: Relative error for different algorithms

As expected, adding more information to the graph decreases errors for the RTT estimation. However, less improvement is visible for the bandwidth estimation. This evaluation shows that the different GRs impact the prediction accuracy depending on how much information is added as input features. Adding an Ack node, which includes the return path (GR2), decreases the relative error for the RTT model, both for the median and quartiles (Figure 3c). However, it has less impact on the rate model (Figure 3d). We argue that for the RTT model, the return path must be included in the input features, i. e., the GR, since the RTT considers both ways on the network path. Contrary, for the rate model, the reverse path is less critical since for congestion on the links, mainly the sent data is responsible and less the acknowledgments.

For comparison, we added simple analytical models as a baseline. For the rate baseline, we determine the bottleneck link for each flow as the link with the minimal value for $\frac{\text{link rate}}{\text{\# flows on the link}}$, assuming that all flows share bandwidth evenly. This model does not include differences in CC algorithms. The baseline for the RTT model is simply summing up twice the link latencies on the flow's path ignoring the queuing delay. While the RTT baseline's error roughly mirrors the queuing delay in the dataset (cf. Figure 2), the rate baseline is less accurate. The baseline does not include the different behavior of CC algorithms and their interaction.

4.2 Differences in Algorithms

One contribution in this paper is that we include multiple CC algorithms as input features. Figure 4 shows the models' prediction accuracy split into different algorithms. For the RTT model, the differences are only marginal between the algorithms. One reason is that queuing delay only has a minor impact on the overall RTT compared to the propagation delay. As seen in Figure 2, all flows in the training dataset share a similar RTT distribution independent of the algorithm. The loss-based algorithms *Reno*, *Cubic*, and *Bic* have the lowest relative error. Though, the relative error is more significant for *Ledbat* and *Illinois*. This is most likely due to the more complex behavior of the algorithms. For the rate model, the same effect can be observed.

4.3 Feature Importance

To better understand the models' internal behavior, we analyzed how different input features contribute to the training process and to the models' output. We use the permutation feature importance, which measures how the permutation of one feature in the training or test data influences the prediction accuracy [3].

We shuffled the columns in the input matrix which belong to the given feature and leave all other columns untouched. For example, we shuffled only the column representing the link delay across all rows for the importance of the link delay. For one-hot encoded features, all columns used for encoding are shuffled likewise. The permutation breaks the correlation between this particular feature and the model's output. With the permuted data we trained a model for each input feature and evaluated them against the validation data. We compare the median relative error of the shuffled data with the original one as baseline as an indicator for importance – the larger the difference, the more important the feature.

In Figure 5, the importance of the different features is depicted. The y-axis shows the shuffled data's relative error minus the unshuffled model's relative error. For simplicity we min-max normalized the values to $[0, 1]$, with 0 being less important and 1 more. Thus, the figure only shows qualitative importance for each model and does not allow a quantitative comparison between them.

For the RTT model, the link delay feature is naturally considered important. TCP algorithm and the buffer size are less impactful. We argue that for the used data the queuing delay is considerably smaller than the propagation delay making both input features less important for the RTT (cf. Figure 2). The link rate feature is the second most important feature for the RTT.

The algorithm and link rate features are considerably more impactful for the rate model. It is important for the sending rate of a TCP algorithm if it has a greedy (e. g. loss-based) or a conservative (e. g. delay-based) behavior. The same applies to competing flows on the same network path.

Overall, the feature importances are consistent with our expectations.

4.4 Scalability

This section presents the impact of the number of flows and network size on the prediction accuracy. The datasets contain between one and ten flows uniformly distributed. Figure 6 shows the relative errors for the different number of flows. The RTT model is hardly

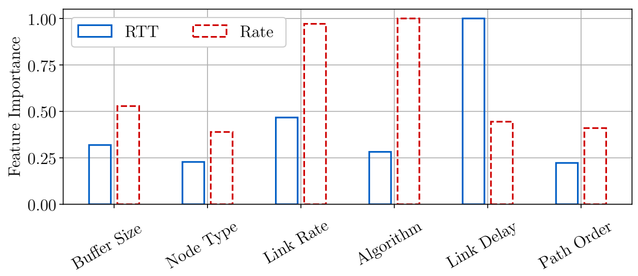


Figure 5: Feature importance for the models

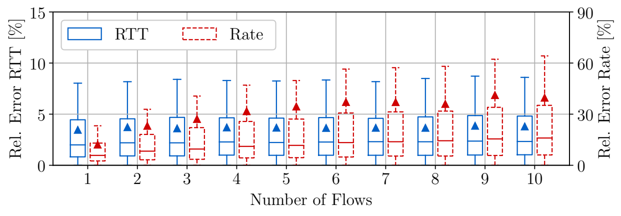


Figure 6: Relative error for different numbers of flows

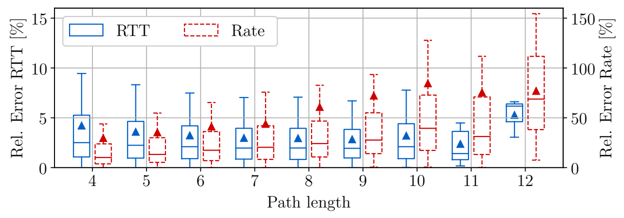


Figure 7: Relative error for different path lengths

influenced by the number of flows. Mainly for two reasons: First, the most significant part of the RTT comes from the propagation delay, which is independent of the number of flows. Second, flows sharing the same links have the same RTT when even one loss-based flow fills the buffer at the bottleneck link. However, the rate model shows an increase in error with increasing number of flows. The more flows contained in the network, the higher the chance that more links are shared by more flows making the behavior more complex. The topologies in the datasets consist of a tree containing two to ten routers; each flow has source and destination servers connected to two different routers. This results in a minimum path length of four and a maximum of 12.

In Figure 7, the relative errors for flows of different path lengths are compared. Rate prediction becomes less accurate with increasing path length. Flows on longer paths might share links with more and different flows resulting in higher rate fluctuations. The RTT model shows the lowest error for a path length of seven which increases slightly with shorter and longer paths. Note, for path lengths larger than ten only a few samples are contained in the dataset.

5 CONCLUSION

This paper shows how GNNs can be utilized for TCP rate and RTT prediction. The presented approach can handle arbitrary network topologies and TCP CC algorithms. We investigate how the encoding into different graphs impacts the prediction accuracy, analyze how vital the distinct input features are, and check if the network size, number of flows, or algorithms impact the model results. We found differences in the prediction accuracy depending on the algorithm, and the feature importance tests showed that the used TCP algorithm is a significant input feature. With the trained models, we achieved a median relative error of 2.29 % for the RTT and 13.31 % for the rate. However, inference with the models can be done within 5 ms on a CPU or 7 ms on a GPU which can be further accelerated with batching, while simulating a topology took on average 260 s. The used datasets, scripts for training, and the trained models are available along with scripts to reproduce results and plots [12].

ACKNOWLEDGMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101008468 and No. 101079774. Additionally, we received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. This work is partially funded by Germany Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK001K) and 6G-ANNA (16KISK107).

REFERENCES

[1] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. 2010. Host-to-host congestion control for TCP. *IEEE Communications surveys & tutorials* 12, 3 (2010), 304–342.
[2] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. 1994. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*. 24–35.
[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[4] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue* 14, 5 (2016), 20–53.
[5] Neal Cardwell, Stefan Savage, and Thomas Anderson. 2000. Modeling TCP latency. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, Vol. 3. IEEE, 1742–1751.
[6] Miquel Ferriol-Galmés, Krzysztof Rusek, José Suárez-Varela, Shihan Xiao, Xiang Shi, Xiangle Cheng, Bo Wu, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2022. Routenet-erlang: A graph neural network for network performance evaluation. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2018–2027.
[7] Cheng Peng Fu and Soung C Liew. 2003. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications* 21, 2 (2003), 216–228.
[8] Fabien Geyer. 2017. Performance Evaluation of Network Topologies using Graph-Based Deep Learning. In *Proc. 11th EAI International Conference on Performance Evaluation Methodologies and Tools*. Venice, Italy. https://doi.org/10.1145/3150928.3150941
[9] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.
[10] Desta Haileselassie Hagos, Paal E Engelstad, Anis Yazid, and Carsten Griwodz. 2019. A deep learning approach to dynamic passive RTT prediction model for TCP. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–10.
[11] Qi He, Constantine Dovrolis, and Mostafa Ammar. 2005. On the predictability of large transfer TCP throughput. *ACM SIGCOMM Computer Communication Review* 35, 4 (2005), 145–156.
[12] Benedikt Jaeger et al. 2022. *Code and Data Publication*. https://gitlab.lrz.de/gnnet-2022/code
[13] Benedikt Jaeger, Dominik Scholz, Daniel Raumer, Fabien Geyer, and Georg Carle. 2019. Reproducible measurements of TCP BBR congestion control. *Computer Communications* 144 (2019), 31–43.
[14] Inas Khalifa and Ljiljana Trajkovic. 2004. An overview and comparison of analytical TCP models. In *2004 IEEE International Symposium on Circuits and Systems (ISCAS)*, Vol. 5. IEEE, V–V.
[15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).
[16] Shao Liu, Tamer Başar, and Ravi Srikant. 2006. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. 55–es.
[17] Albert Mestres, Eduard Alarcón, Yusheng Ji, and Albert Cabellos-Aparicio. 2018. Understanding the Modeling of Computer Network Delays Using Neural Networks. In *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks* (Budapest, Hungary) *(Big-DAMA '18)*. Association for Computing Machinery, New York, NY, USA, 46–52.
[18] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. 2007. A machine learning approach to TCP throughput prediction. *ACM SIGMETRICS Performance Evaluation Review* 35, 1 (2007), 97–108.
[19] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.
[20] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2020. RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2260–2270.
[21] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The Graph Neural Network Model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.
[22] José Suárez-Varela, Paul Almasan, Miquel Ferriol-Galmés, Krzysztof Rusek, Fabien Geyer, Xiangle Cheng, Xiang Shi, Shihan Xiao, Franco Scarselli, Albert Cabellos-Aparicio, et al. 2022. Graph Neural Networks for Communication Networks: Context, Use Cases and Opportunities. *IEEE Network* (2022).
[23] Andras Veres and Miklos Boda. 2000. The chaotic nature of TCP congestion control. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, Vol. 3. IEEE, 1715–1723.