Priority Aware Scheduling in PCNs using DL Based Model

A B.Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Astik Raj, Rohit Kumar
(190101016, 190101077)

under the guidance of

Prof. Hemangee Kalpesh Kapoor

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

April, 2023

Abstract

Payment Channel Networks (PCNs) have been proposed as a solution to the scalability issue faced in layer 1 blockchain technology. It proves to be a good solution since it promises to improve the network throughput, thus increasing the number of users and transactions day by day. Due to the simultaneous transactions, there have been issues such as throughput, latency, and congestion in the network. Much research has been done to solve these issues, but in this paper, we focus on minimizing the forwarding fee to process transactions and ensuring the throughput remains up to some value. We execute priority-aware scheduling in which the nodes transfer the transaction from itself to the other according to its priority in case there are multiple transactions ready to be forwarded to that node. The paths of all the transactions are calculated beforehand. The transaction with the high priority may be charged a higher fee. This priority allotment is done by a DL-based priority assignment algorithm which works on the graph and the sets of transactions to prioritize each node and transaction pair. In the entire paper, we assume that certain nodes have the information on graph topology and channel balances and they will help the entire network to lower the average fee per transaction. We have built a simulator to check the performances of our algorithm. The experimental results are not significant, but there has been some reduction in the average fee per transaction. We believe the results can be improved in further research using this scheme.

Contents

Chapter 1

Introduction

Blockchain technologies like Bitcoin [1], Ethereum [2], Ripple [3] etc. have been very popular for the last few years. The technology was created to enable trusted collaboration between untrustworthy parties without the need for a centralized authority or any third party. These have been used as an alternative to the traditional forms of payment. But the low throughput of blockchain brings up the scalability issue as there are restrictions placed on the blockchain's growth.

To handle this scalability issue, Payment Channel Networks (PCNs) [4] were introduced. PCN is a network of off-chain transactions made up of cryptocurrency users and payment channels. These networks help in lowering transaction latencies and improving the throughput of the system very significantly. There is no need to register each transaction on the blockchain [1] as the users can perform many off-chain transactions, except the initial and final transactions. If the two users do not have a common channel, they can send it through multiple hops of payment channels i.e. through a route between the sender and receiver user using the help of some intermediate users.

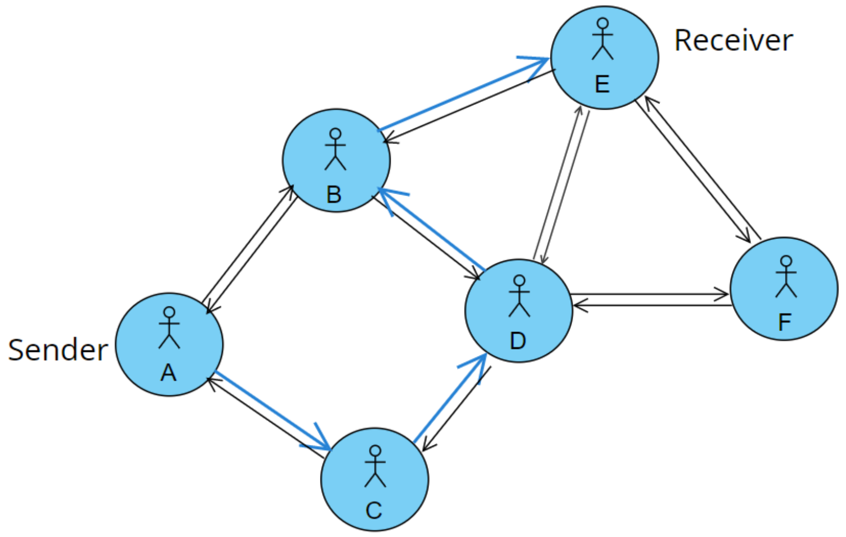


Fig. 1.1: Bitcoins getting transferred along the blue-colored path

A PCN can be understood as a weighted directed graph where the users represent the nodes and the connections(channels) between two users represent the edges of a graph. The edge weight represents the amount that can be passed through the particular channel, the balance of an edge may get exhausted after forwarding some transactions. A simple illustration of PCN is shown in Fig. 1.1.

Without using a slow pace and costly blockchain transactions, PCNs may execute immediate payments of smaller amounts. The intermediate users in the route charge fees in the money-transferring process, which are expected to be significantly smaller than the blockchain transaction fees. Therefore, the primary objective becomes improving PCN routing and ensuring payment success. There have been lots of works related to devising routing protocols aimed to improve different parameters such as latency, throughput, average fee, etc. of the whole system. Network Congestion [5] is another issue on which some researchers have worked, but they have certain limitations too. In section 2, we talk about the work which has been done related to routing protocols in PCNs.

In this report, we propose a Payment Channel Network where the transactions are forwarded to another user based on their priorities in case of multiple transactions arrive at a certain user. Each intermediate user in the entire network has a priority queue to schedule transactions as per the priorities. It may happen too, to execute the priority aware scheduling, balances of some edges get exhausted. In that case, the transaction is assumed to be canceled. Priority assigning must be finished to ascertain the routing fee of each hop before the transaction is issued because the routing fee is determined by the transaction sender and included in the transaction.

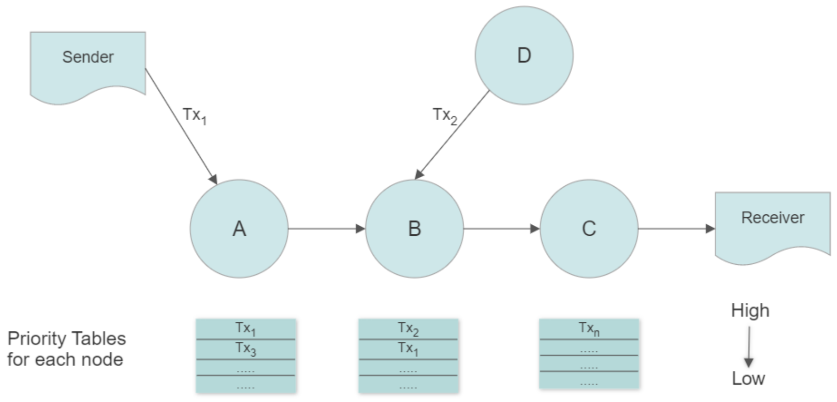


Fig. 1.2: Transactions taking place according to their priorities

In our design, the priorities get affected by the forwarding fees senders specify the priority for their transactions according to their demand. The design may charge a higher fee for a particular transaction and prioritize this particular transaction to lower the average fee of the entire network. Also, this design assumes that certain nodes in the network have the required information about the nodes, edge balances, and transactions. These nodes referred to as landmark nodes [6] in many papers, help other nodes in scheduling transactions with better throughput and fees spent at the cost of privacy. As a result, we design a deep learning model which outputs the priorities of different transactions at each node, the details of which are explained in this report. Figure 1.2 shows how the transactions get prioritized at each node.

The rest of this report is organized as follows. Chapter 2 discusses the related works in PCNs and learning to improve the various statistics of PCNs and our contributions in this field. Chapter 3 of this report is a detailed analysis of our simulator, built for model implementation and results evaluation. Chapter 4 is about the priority assignment problem formulation and our approach to solving the problem with a detailed analysis of our deep learning model. Chapter 5 presents the evaluation results and comparison of these results with the non-learning-based algorithms, while we conclude this report in Chapter 6.

Chapter 2

Related Work

In the domain of Lightning Network, there has been a lot of research most of which focuses on the routing protocols and algorithms on the main network.

2.1 Past Work

Although, the whole concept of Lightning Network relies on the fact that it is a decentralized system for instant, high-volume micropayments that remove the risk associated with giving trusted third-party custody of your money. Still, there have been certain centralized routing algorithms, namely - MPCN-RP routing [7], a general routing protocol for payment channel networks with multiple charges. Flash-efficient dynamic routing [8], an autonomous routing algorithm, Auto-Tune, optimizing the routing concerning the success rate and the routing fee and utilizing the limited channel capacity information, etc. But a lot of decentralized algorithms have been proposed since then, as these cryptocurrencies run on the idea of decentralization. Malavolta [9] researched the privacy-reserving routing issue and created SilentWhisper, a routing method based on Landmark Routing.

In the case of a decentralized system, one of the earliest routing algorithms, Flare [10], was proposed in 2016. Each node in Flare includes a routing table that lists the neighborhood of nodes that are within a few hops and paths to several beacon nodes. Speedy Murmers [11] resolved these flaws and outperformed SilentWhisper in terms of metrics including payment success ratio, delay, overhead, path length, and stabilization by using embedding-based path discovery. To properly utilize the network's capacities, Rohrer [12] drew the payment flow as a series of paths that added up. These works do not put too much emphasis on hop-based constraints. Later, to reduce the transaction charge of a payment method while taking into account the timeliness and feasibility restrictions, Zhang et al. [13] suggested CheaPay. Adding robustness to this idea, Yuhui Zhang [14] proposed RobustPay [14]. In 2020, the Ant Routing [15] algorithm for the Lightning Network was proposed in for maximal decentralization, anonymity, and potential scaling. It solves several problems of the current implementation, such as channel information update and centralization by beacon nodes. Also, there have been some queuing-based algorithms to provide congestion control methods for high throughput, e.g. Last-In-First-

Out in Spider [5].

All the works mentioned above aim to optimize parameters like throughput, latency, congestion, etc. but none of them have used the idea of Machine Learning to help them give good results. To the best of our knowledge, Learning-Based Off-Chain Transaction Scheduling in Prioritized Payment Channel Networks by Xiaofei Luo [16] is the only paper that uses Multi-Agent Reinforcement Learning to work on congestion control mechanisms. Each node acts as an agent and, they prioritize the transactions to pass forward on a pre-path calculated system for each transaction. On simulator development, CLoTH [17] has been proposed recently to simulate the execution of payments in a payment-channel network and produce performance measures such as the probability of payment success and the average payment time.

2.2 Our Contributions

As a promising technique, we use deep learning on PCNs to give priority to each transaction on every node. Our main motive is to lower the average fee of the entire network for all transactions keeping in mind that the number of successful transactions does not get affected too much. To train the model, we calculate the average fees and successful transaction rate by executing the priorities of transactions and pre-calculated path on the main PCN and then do the backpropagation. The details of training our model are in Chapter 5. We make the following contributions:

- We build our simulator to test all of the results and compare them with the one provided by the model. Our simulator has its advantages discussed in Chapter 3.
- We propose two algorithms to route transactions on the PCN, one is similar to Distributed Bellman-Ford routing [18] and the other one is a simple DFS.
- We build a learning model which gives the priorities of all transactions on every node, based on which the final execution of transactions has been done.
- We compare the results obtained by our model with the ones obtained by different non-learning algorithms under different sets of parameters such as nodes, balance upper bound, number of transactions, etc.

Chapter 3

Simulator Design

This simulator is a PCN simulator written in C++. As input, it takes certain parameters listed below to generate a graph and a set of payments. The first step is graph generation which tries to mimic the main Payment Channel Network as much as possible. In our simulator, there are mainly 3 modules, namely -

- Network: Mainly related to graph generation.
- Transaction: Transaction set generation, fee calculation and The execution of paths happen here.
- Simulator: The main cpp file to read inputs and output the required results.

The reason behind choosing this set of inputs is explained in the next subsection and also all these inputs for the actual payment channel network are available on *bitcoinvisuals.com* [19]. The following table shows the input parameters used to generate the graph and transaction set.

Input Parameter	Description
n_nodes	The number of nodes of the payment channel network.
degree*	The average degree of all nodes in the network, four types of inputs are taken related to this parameter as explained below.
capacity*	The average capacity of the channels between different nodes in the network, again four different inputs are taken.
base_fee**	The base fee constant required to calculate fee at each node, two different inputs are taken as described below.
proportional_fee**	The proportional fee constant required to calculate the fee at each node, again two different inputs are taken as described below.
imbalance_fee**	The imbalance fee constant required to calculate fee at each node, again two different inputs are taken as described below.
num_of_txn_sets	The number of transaction sets, implemented on a graph set after set.
num_of_txn	the Total number of transactions to be simulated among all the sets.
txn_amount**	The limit on transaction amount to be passed through the channels.
faulty_node_probability	The probability that a node is faulty when asked to forward a payment.

* - *It means there are four inputs corresponding to this parameter, namely - overall average, an average of top 10 percentile elements, an average of top 50 percentile elements, and that of top 90 percentile elements.*
** - *It means there are two inputs corresponding to this parameter, one for the upper bound and the other for the lower bound, the values generated by the simulator will be in between these values.*

3.1 Graph Generation

The channels between nodes are generated randomly to satisfy the given average degrees as inputs. In the actual Lightning Network, we know that *power law* [20] is followed i.e. if a node has a higher degree more nodes would want this particular node to be their direct connection as it is going to act like a hub for transaction forwarding. To achieve this, we assign the nodes to some degree randomly using a uniform distribution. Now, since the mean value is different for different types of nodes(top 90th percentile, top 50th percentile, and top 10th percentile) the eventual allotment of degrees to the nodes follows somewhat similar to the power law.

At this point, each node has a given degree, we generate a connected graph that follows the power law and try to avoid bridges as well. To ensure this, we establish channels using a customized variation of *Havel Habibi algorithm* [21].

Therefore, we start with the nodes with the highest degrees and open a channel between these nodes and the nodes with the lowest degrees. In this way, the network becomes similar to the actual network where nodes with higher degrees are the actual hubs, but there is still a problem. This may lead to the creation of multiple connected components in this graph which we tackle in this way: we check the number of connected components and then randomly connect these islands with each other by adding extra channels.

Similarly, we add the weights to each channel using the average capacity. The base, proportional and imbalance fee constants are the properties of a node which are randomly assigned too, satisfying the input.

3.2 Transaction Generation and Path calculation

To generate random transactions, say for a set, we use the num_of_txn and the lower and upper limits on the amounts to be passed from the input. So, a transaction consists of a sender, receiver and the amount to be passed. These are assigned randomly.
After we get a set of transactions, we find the paths between all the senders and receivers one by one to execute that particular transaction. There are lots of algorithms and protocols regarding this, both centralised and distributed. In our simulator, we devise two of them, they are as follows:

- **DFS related centralised Algorithm** - In this algorithm, we find any random path from sender to receiver without thinking about the shortest path. As it happens in the standard DFS algorithm, it starts at the root and explores one of its neighbour's neighbours, and then moves to the next neighbour's neighbours, and so on. The pseudocode is attached below. The time complexity of our version of DFS algorithm is *O(V+E)*.

- **Distributed Bellman-Ford Algorithm** - In this algorithm, we find our path very similar to what happens in a Bellman Ford algorithm, we tend to find the path which has the least fee. The pseudocode of our implementation is attached below. the only difference is that we start our implementation in reverse, starting from the destination node because the fee to be transferred by the sender is to be calculated. Thus, it calculates the fee depending on the path.

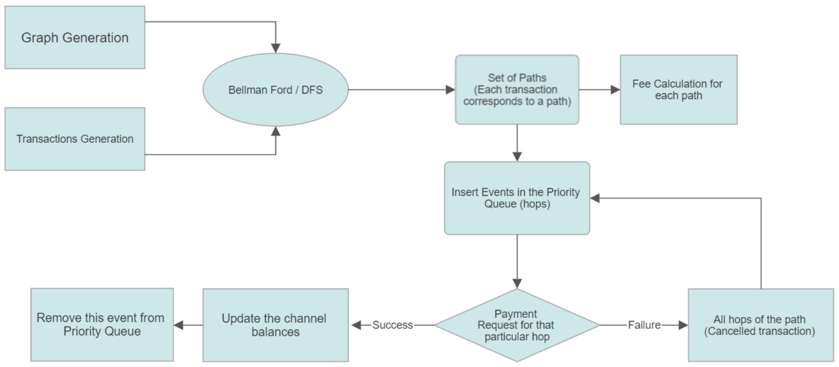
Algorithm 1 DFS Path Finding

```
function DFSPath($G, source, destination, amount$)
    $path \leftarrow$ empty list
    $visited \leftarrow$ empty set
    $dis \leftarrow amount$
    DFS($G, destination, source, path, visited, dis$)
    return $path$
end function
function DFS($G, current, destination, path, visited$)
    add $current$ to $visited$
    add $current$ to $path$
    if $current = end$ then
        return
    end if
    for $(current, v) \in E$ do
        $vBalance \leftarrow$ balance of v
        $currentBalance \leftarrow$ balance of current
        $balance\_diff \leftarrow |vBalance - currentBalance|$
        $fee\_taken \leftarrow$ CALC_FEE($current,\ v,\ balance\_diff$)
        if $v$ is not in $visited$ and $vBalance \geq dis + fee\_taken$ then
            $dis \leftarrow$ dis + distance from $current$ to $v$
            DFS($G, v, end, path, visited$)
            if $end$ is in $path$ then
                return
            end if
            $dis \leftarrow$ dis - distance from $current$ to $v$
        end if
    end for
    remove the last element from $path$
end function
```

After the above process, it may happen that some of the transactions do not get any path due to the lack of balances in some intermediate channels. The algorithm can find paths between nodes for a good fraction of transactions. After that, we calculate the fee (except for the imbalance fee as it is dynamic, thus it is calculated while the execution of paths takes place) one by one for each transaction corresponding to the given paths and associate them with that transaction. Now, all the transactions are ready to be executed on the network. The following flowchart shows the sequence of events in our simulator.

Algorithm 2 Get payment path using Bellman-Ford

```
1:  function GETPAYMENTPATHUSINGBELLMANFORD($source, destination, amount, current\_network$)
2:      $n \leftarrow$ size of $current\_network$
3:      $\forall v \in V, dis[v] \leftarrow \infty$
4:      $dis[destination] \leftarrow$ amount
5:      for i from 1 to n-1 do
6:          for $(u, v) \in E$ do
7:              $vb \leftarrow$ balance of v
8:              $ub \leftarrow$ balance of u
9:              if $dis[u] < \infty$ and $vb \geq dis[u]$ then
10:                 $balance\_diff \leftarrow |vb - ub|$
11:                 $fee\_taken \leftarrow$ CALC_FEE($u, v, balance\_diff$)
12:                 if $dis[v] > dis[u] + fee\_taken$ then
13:                     $dis[v] \leftarrow dis[u] + fee\_taken$
14:                     Backtrack to store the path p
15:                 end if
16:             end if
17:         end for
18:     end for
19:     return $p$
20: end function
```



Graph Generation
Transactions Generation
Bellman Ford / DFS
Set of Paths (Each transaction corresponds to a path)
Fee Calculation for each path
Insert Events in the Priority Queue (hops)
Payment Request for that particular hop
Success
Failure
Update the channel balances
Remove this event from Priority Queue
All hops of the path (Cancelled transaction)

Fig. 3.1: Flowchart of the simulator

3.3 Transactions Execution

Now, we have a set of transactions with the pre-processed paths. The execution of these transactions takes place in such a way that it simulates the simultaneous execution of payments on the actual network eventually. Again, it will happen that balances of some of the channels get exhausted in between after the execution took place. We consider those cases as that of canceled transactions. In that case, all the updated balances of processed hops get reverted.

To implement the above execution of transactions, we use a priority queue that stores the **events** to happen. We insert two types of events in our priority queue. One is the usual payment forwarding from one node to its neighbor node (a single hop) along the path and the other one is for the aborted transaction. Here again, a single hop corresponds to a single event. An event results in an updation of the edge balance too. Also, all the events are inserted randomly in the priority queue. The hops are scheduled to take place such that the path sequence does not change for a transaction as depicted in the figure.

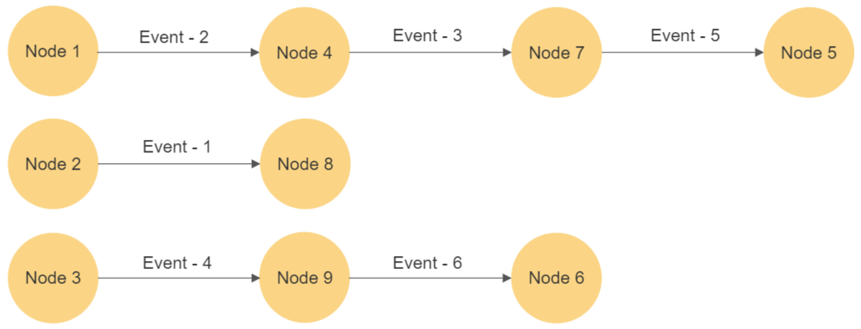


Fig. 3.2: Events timeline achieving the simultaneous execution of transactions

The figure attached above is an example of how are we achieving simultaneous execution of transactions as it happens in the actual lightning network. The above case consists of 3 different transactions with fixed paths as follows:

Transaction No.	**Path** 24
Transaction 1	1 -> 4 -> 7 -> 5
Transaction 2	2 -> 8
Transaction 3	3 -> 9 -> 6

The event numbers assigned to each hop in the figure are stored in the priority queue. So, the first hop to be processed is *2 -> 8*, after that it is *1 -> 4* and so on. Note that all the hops are processed sequentially in a transaction.

Now, since we can have cases where multiple transactions occur at a node, in that case, we have priorities for each transaction at each node. The transaction having the highest priority should be forwarded ideally. This is the priority-aware scheduling that we have been talking till now. And these are provided by the model when we use DL based algorithm. For simple DFS and Bellman-Ford routing, we have uniform priorities for each transaction at a node, any transaction can be forwarded to the next node randomly.

Chapter 4

DL Based Priority Assignment Scheme

In this chapter, we formulate our priority assignment problem and propose a DL-based transaction priority assignment scheme to improve the performance of PCNs. In short, we have a graph and a set of transactions, we find the priorities for each (node, transaction) pair, calculate the fees and throughput of the network, and backpropagate. Throughout this research, we assume that certain nodes have the required information such as channel capacities and the topology of the graph, many refer to them as landmark nodes [6]. So, all the users of the network agree to share this information because they help them route and execute the transactions such that the average fees of the system remain low.

4.1 Priority Assignment Problem Formulation

We represent a PCN as a directed graph $G = (V, E)$, where V represents the set of nodes, and E is the set of edges. Each node $v_i \in V$ represents a user and each edge $e = (v_i, v_j) \in E$represents a channel through which coins get transferred. Note that graph G is a connected directed graph, hence each node must have a channel with some other node. And since it is a directed graph, $e = (v_i, v_j) \in E$ means v_i is the sender and v_j is the receiver. A node v_i stores the base, proportional, and imbalance fee constants fb_i, fp_i, and fi_i that they use to charge the fees while forwarding the payment. Each edge $e = (v_i, v_j) \in E$ has a channel balance $b_{i,j}$ which denotes the amount of remaining balance in that particular channel. We do not consider the edges with zero balances, they are assumed to be removed from the graph.

Similarly, the payment request is represented by $P = (v_s, v_r, A)$, where v_s is the sender, v_r is the receiver and A is the amount to be transferred from the sender to the receiver. A set S of such requests will be processed simultaneously. A path p given by the routing algorithm for each such request will be a set of nodes like this: $v_0, v_1, ..., v_n$ where v_0 & v_n are the source and destination nodes. Along with this, we use a transaction fee function $F_p(s, r)$ to denote the total transaction fee from v_s to v_r for a given path and network throughput τ for a given set of transactions on that graph. Also, we have a priority table PT which stores the priorities of each transaction at each node. In case, there are multiple transactions to be forwarded, they will be scheduled according to the priority table. Thus, $PT_{i,j}$ represents the priority of transaction j at node i.

One more important thing is how we calculate the total fees to complete a transaction. We take into account the following fees:

- Base Fee (F_B): This is the flat fee charged for a payment routed.
- Proportional Fee (F_P): This corresponds to the fee that is a percentage of the amount being routed and that also reflects how much priority should the transaction be given.
- Imbalance Fee (F_I): This fee is due to the difference between the balances of edges between any two nodes having a channel.

We define the fee calculated for a payment request $P = (V_0, v_n, a)$ with the path $p = (v_0 -> v_1 -> ....v_n)$ as follows:

$$F_p = \sum_{i=1}^{n-1} F_B + \sum_{i=1}^{n-1} F_P + \sum_{i=1}^{n} F_I \tag{4.1}$$

Note that the first node does not charge any fee as he is the sender himself as well as the destination node, except for the imbalance term will be there. These terms are for the base, proportional and imbalance fees respectively. In case of a transfer from v_i to v_j, they are defined as follows:

$$F_B = f b_j \tag{4.2}$$

$$F_P = (f p_j + PT_{i,p}) * A \tag{4.3}$$

$$F_I = f i_j * (b_{j,i} - b_{i,j}) \tag{4.4}$$

After the execution of n payment requests, let us say there are f such requests which get completed, then we define the throughput of the system in this iteration as:

$$\tau = \frac{\#numberofsuccessfulrequests}{\#numberofrequests} = \frac{f}{n} \tag{4.5}$$

After all the above definitions, we come to our final motive, to minimize the total fees overall payment requests, and the throughput remains constant.

$$min.\ F = \sum_{p} F_p \tau = c \tag{4.6}$$

4.2 Model Overview

As stated earlier in this report, we perceive this problem as a priority assignment problem. On the basis of the priorities allocated, we schedule the payment requests and calculate the fee and throughput of the system. We use a deep learning model explained in this section to assign the priorities, in order to minimize the average fee.

Our proposed model is a composition of two functions, the priority prediction model, $\mathbb{F}_m$, and the transaction execution step, $\mathbb{E}_X$. We calculate the priority table defined above as follows:

$$PT = \mathbb{F}_m(G, (P_1, v_1), (P_2, v_2) \dots (P_{N_T}, v_{N_T}))$$

$$F = \sum_{p} F_p = \mathbb{E}_x(PT) = \mathbb{E}_x(\mathbb{F}_m(G,(P_1,v_1),(P_2,v_2)\dots(P_{N_T},v_{N_T})))$$

where, N_T refers to the number of payment requests and $PT \in \mathbb{R}^{V \times N_T}$ is the priority assignment table. The inputs to this prediction model consist of graph G and N_T payment requests along with the pre-calculated paths v_i. Thus, in a pair (P_i, v_i), P_i represents the payment request and v_i represents the path to that particular transaction.

Hence, we get the priority table from the priority prediction model, $\mathbb{F}_m$ and these priorities act as inputs to the Execution Step $\mathbb{E}_x$. Here, again the transactions are forwarded according to the paths they have been assigned, but in case of multiple transactions reaching a node simultaneously, they are forwarded according to the priority table given by $\mathbb{F}_m$. This step is explained in detail in Chapter 4 of this report.

Our model learns using back propagation done through the entire process, both the priority prediction model and execution by minimizing the total fee we got after the execution step.

Graph Attention Networks

Graph attention networks (GATs) by Velickovic [21] are novel neural network architectures that operate on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. They enable specifying different weights to different nodes in a neighborhood without requiring any type of expensive matrix operation (such as inversion) or depending on knowing the graph structure in advance by stacking layers in which nodes are able to attend over their neighborhoods' features.

In our case, we use our customized version of GATs, the details of which are explained in the subsection below.

Custom Graph Attention Layer

In this section, we describe the single graph attentional layer that was employed in all of the GAT layers we use to build the architecture. The number of nodes in our graph is V, say N. The input to our layer is a set of node features $h = \vec{h_1}, \vec{h_2}, ..., \vec{h_N}, \vec{h_i} \in \mathbb{R}^3$. where N is the number of nodes and the feature size is 3 because each node has the base, proportional, and imbalance fee constants stored with them which is each node's feature initially. Thus, the size of h is $\mathbb{N} \times 2$ in the beginning.
Now we expand the input features by multiplying the h with a learnable weight matrix $W \in \mathbb{R}^{3 \times f}$., which was initialized using the method explained in Xavier et al. [22]. At least one learnable linear transformation must be used to provide the necessary ability to express to convert the input features into higher-level features. Here f is the new feature size which turns out to be 40 when we tune them to get the best results, these hyperparameters are mentioned in the last part of this chapter.

We then apply self-attention on the nodes, also called the shared attentional mechanism, to compute the attention coefficients. We get a matrix a, the values of this matrix are learnable too, and do the following to get the attention matrix e.

$$Wh_1 = \langle Wh\, ,\, a_1 \rangle$$
$$Wh_2 = \langle Wh\, ,\, a_2 \rangle$$
$$e = Wh_1 \otimes Wh_2$$

The matrix $e \in \mathbb{R}^{N \times N}$ is the attention matrix, this represents the different importance to each neighbor's contribution. Thus, in our case, the attention value for each (node, node) pair represents how these nodes are important to each other. For example, if frequent transactions occur between two nodes the attention value for this pair will be high. We have used a two-headed attention mechanism in our model, a 3-headed mechanism is shown in figure 4.1.

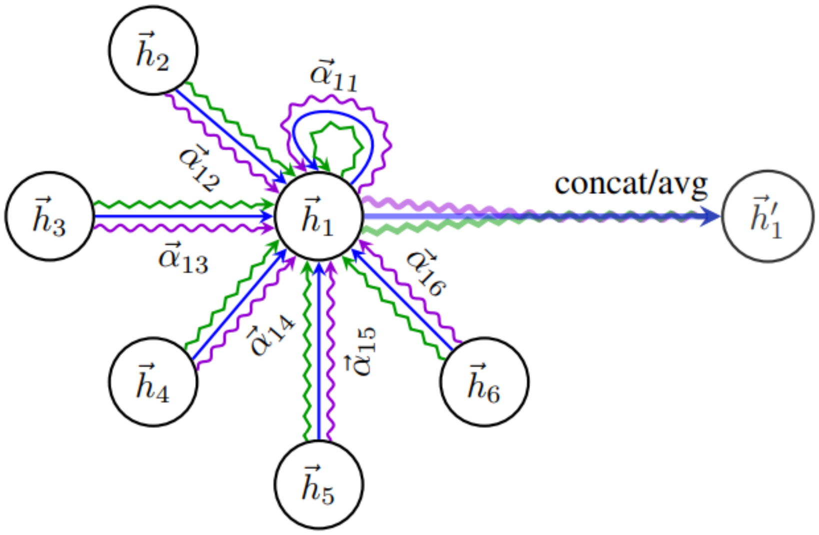


Fig. 4.1: An illustration of multihead attention (with K = 3 heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations [21].

For the two unconnected nodes, the attention value is zero, as there is no direct importance between each other. Now, we customize our Graph Attention by adding a balance list to the attention matrix to add the importance of balances between two connected nodes. This makes sense as if two connected nodes are having a higher balance between their channel, the attention matrix will give more importance to these nodes. This is the inductive bias used by us in the model.

Then, we convert the attention values as probability distributions by using the softmax function.

$$\alpha_{i,j} = \mathrm{softmax_j}(e_{i,j}) = \frac{\exp(e_{i,j})}{\sum_{k \in N_i} \exp(e_{i,k})}$$

Next, we apply the Leaky ReLu non-linearity in the attention mechanism. To enhance the node features, we multiply the attention matrix e with the node feature matrix Wh, to get a new matrix $h' \in \mathbb{R}^{N \times f}$. This is the attention-enriched node feature matrix.

Up to this, we, sort of, extracted the features of the graph and the dependence of nodes on each other. Now we bring the payment requests to this model. Thus, for each payment request, $P = (v_s, v_r, A)$ and a path $p = (v_1, v_2, ..., v_n)$, we do the following steps:

1. Firstly, we do the embedding of the numerical value of the amount using a fully connected layer to the size $(f \times 1)$.

2. In the next step, we extract all the node features of nodes in the order they appear in the path. Along with this, we do the weighted mean pooling of all the node features with attention to their weights. This makes sense as the importance of the channels used in this path are being reflected in this step as follows:

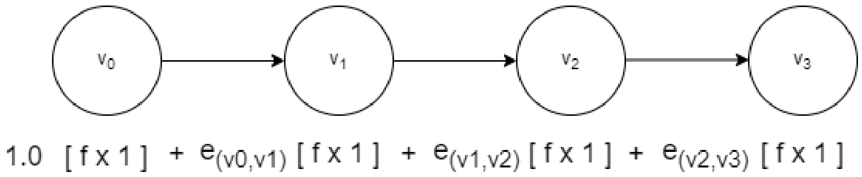

v0
v1
v2
v3
1.0 [f x 1] + e(v0,v1) [f x 1] + e(v1,v2) [f x 1] + e(v2,v3) [f x 1]

Fig. 4.2: The attention values is multiplied with the h' of each node of size $(f \times 1)$, to add each of them.

3. Now we add the feature vector we got after embedding the amount with what we got above to get a new f-size feature embedding. This new f size feature embedding is transformed using a $(f \times f)$ learnable matrix to a size of $(f \times 1)$, say F matrix.

4. Finally, we multiply the $h' \in \mathbb{R}^{N \times f}$ with the matrix F we got in the last step to obtain the $N \times 1$ node priorities of the P payment request with the path p. This represents the node priorities for this payment request for all the nodes.
[1]

After this step, we have got two important things, one the priority of this transaction on every node and the attention-enriched node feature matrix. The Priority Table (PT) gets filled up after the above set of steps repeats over all the payment requests. So, what happens inside this layer is shown in this flowchart in Fig 4.3.

[1] Ideally, the nodes which do not participate in this transaction will have the values as 0

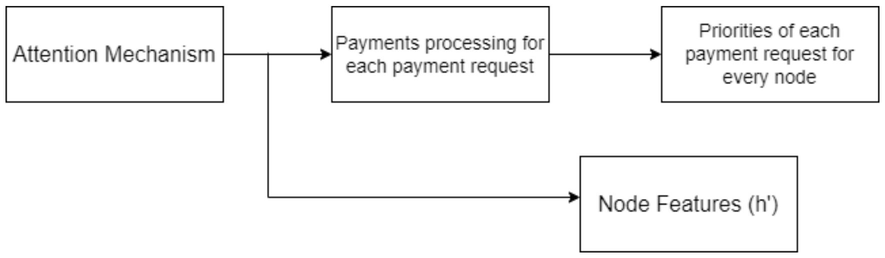

Attention Mechanism
Payments processing for each payment request
Priorities of each payment request for every node
Node Features (h')

Fig. 4.3: The processes inside Graph Attention Layer

Trans-Graph Attention Network

We use the Graph Attention layer described in the previous section and create a multi-head Graph Attention Network. We described the processes happening inside one layer of the multiple layers in the model. The same process occurs in each of the layers. Also, We add dropout in between the learnable layers to apply regularization.

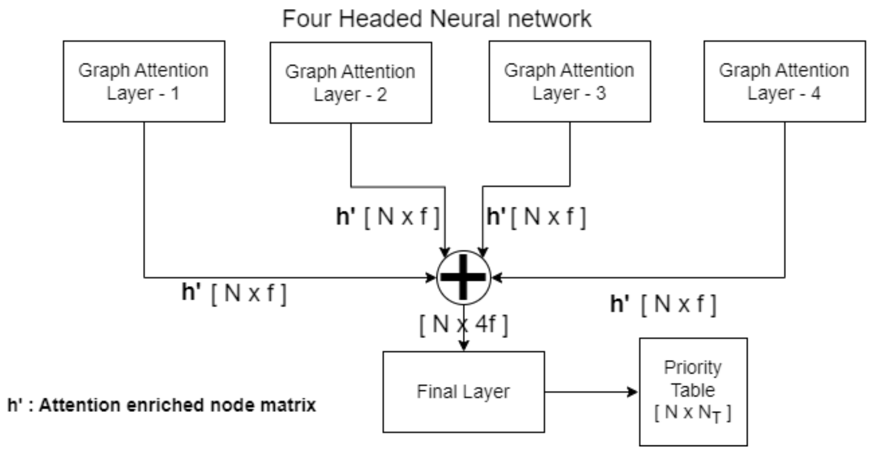

Four Headed Neural network
Graph Attention Layer - 1
Graph Attention Layer - 2
Graph Attention Layer - 3
Graph Attention Layer - 4
h' [N x f]
h' [N x f]
h' [N x f]
h' [N x f]
[N x 4f]
Final Layer
Priority Table [N x N_T]
h' : Attention enriched node matrix

Fig. 4.4: The 4-headed network which eventually outputs the node features to the final layer to get the priority table.

The model consists of N_h heads, and in each head, we give our graph G and payment requests along with the paths, (P, p), we obtain the enriched node features, $h' \in \mathbb{R}^{N \times f}$ from each head. We then concatenate the features from all the heads to obtain final node features, $N \times N_h.F$. Using this matrix as our final node features and the payment requests, we pass them as inputs to our final Graph attention layer to obtain the transaction priorities for each node i.e. the priority table.
Thus, in the initial layers (N_h heads), we make use of attention-enriched node features ignoring the priorities at that time and then in the final layer, we get our priorities ignoring h'. All these processes are shown in the flow chart of our model in Fig 4.4.

4.3 Training process and Parameters

In this section, we explain the training mechanism of our model. We use a 2-step training process in which in the first step, our model is trained on a lot of different diverse graphs. With this set of initial parameters after the first step, we train our model only on one graph repetitively that is very similar to the actual Payment Channel Network. Fig 4.3 shows how this 2-step training works.

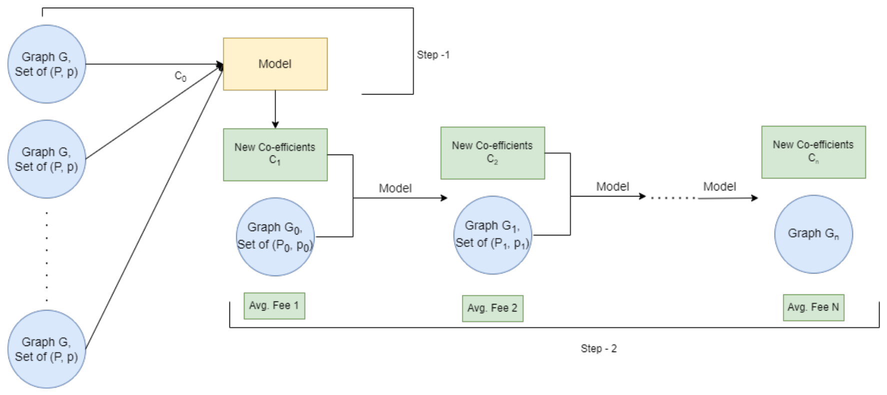


Fig. 4.5: The two-step training used by our model finally gives the output as average fee and the updated graph

Step - 1 Training

This step is mainly used by the model to have exposure to a diverse amount of data. The biases in the data sources can spread or even be amplified by algorithms. To reduce bias, a variety of data should be employed in this instance. Some examples of the extreme cases in our dataset for the first step are as follows:

1. A graph having a small number of nodes with fewer degrees and a lot of transactions with the channel balances is not big enough. In this case, most of the transactions should get canceled.
2. A graph has a lot of nodes and a few amounts of transactions with adequate channel balances. In this case, most of the time there will be transactions getting executed successfully.
3. A graph having high fee constants at nodes whose channels have good balances but low fee constants at nodes whose channels have low balances. In this case, to decrease the average fee, the path length will be much more.

There are other cases as well of which we have taken care of while giving the inputs.

Step - 2 Training

In this step, we overfit our model $\mathbb{F}_m$ to a single graph topology with various balance edges which corresponds to a real-world situation. After step 1 training, we have got the

initial values of the learnable parameters or coefficients which we use at the beginning of Step 2. As shown in the diagram, the initial coefficients C_1 and network parameters are given as inputs to the model. Then, we get the updated coefficients as it happens after backpropagation. Note that, the topology of graph G_1 is the same as that of G_0, just the channel balances have been updated in the new graph as they should after the execution of transactions.
This step happens several times to output the final average fee, which is the minimum fee according to the model. We compare the statistics of step 2 for a trained and untrained model in the next chapter. In the untrained model, the priorities of all the transactions would be the same and there will be nothing like coefficients.

Hyperparameters and Parameters

The values of hyperparameters and the parameter size(Hidden feature size) after tuning our model to get the best results are provided in this table:

Hyperparameters	**Values**
Hidden feature size	40
Number of heads	4
Dropout rate	0.2
Epochs	60 (Step 1), 1 (Step 2)
Learning rate	1e-2 (Step 1), 1e-3 (Step 2)
Number of Graphs generated	90
Simulation params	24

The repository contains all of the source code and associated files for our project. It is hosted on GitHub and can be accessed at the following link: Click here.

Chapter 5

Experimental Evaluation

In this chapter, we show the results after running different simulations on our simulator. Note that the simulator is capable of executing the transactions using three different routing algorithms, namely - DFS-based, Distributed Bellman-Ford, and DL-based routing algorithms. We have the stats depicted using different plots for various permutations and combinations of the inputs.

General Inputs

We proceed with the experiments and the evaluations with a set of general inputs. In any comparison or plot analysis, if there is nothing mentioned about the inputs, assume that this set of inputs is taken.

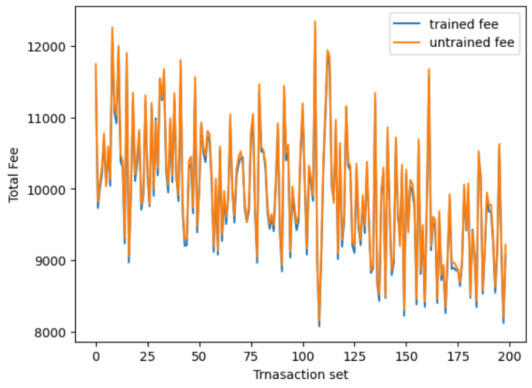
Input	**Values**
Number of Nodes	200
Average Degree	15
Average Capacity	5000
Base Fee Range	2 to 5
Proportional Fee Range	20 to 100
Imbalance Fee Range	20 to 100
Number of Transactions	20000
Amount Range	1 to 100

5.1 Average Fee

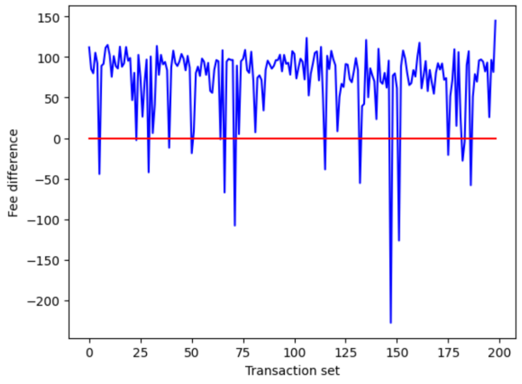
Fee as training progresses

The plot in Fig. 5.1 (a) shows how the total fee varies as the training progresses. By transaction set on Y-axis, we mean the transaction set no. which is fed to the graph. In the untrained model, the prior probabilities of each transaction on each node are equal. So, we see a slight decrease in the total fee as the model trains more

The plot in Fig. 5.2 (b) represents the fee difference(untrained fee - trained fee) over each transaction set. As we can see, in this plot there is a visible difference between the performances of the two models, the fee in the case of the learned model is less as compared to the untrained one as expected.



(a) Total Fee versus transaction set plot



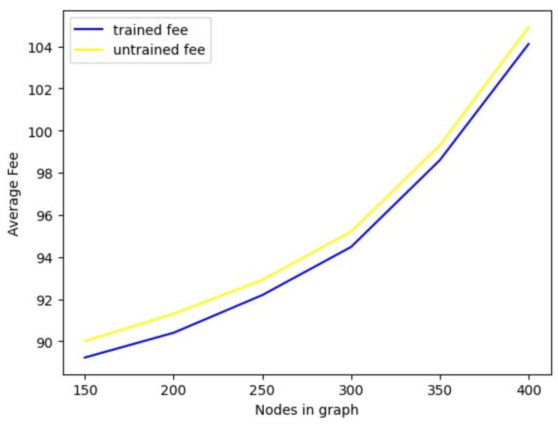
(b) Fee difference versus transaction set plot

Fig. 5.1: Improvement in the fees as new transaction sets arrive.

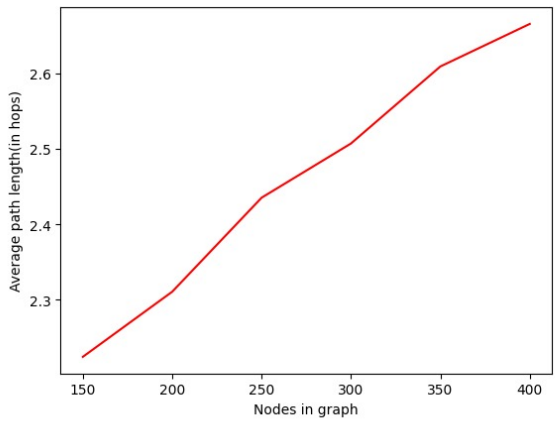
Number of nodes

Now we see how the average fee responds to the variation in the number of nodes in Fig. 5.2. As explained earlier, the fee taken is calculated as the sum of the base, proportional and imbalance fee. The transactions are generated randomly in the simulator, i.e. the sender and receiver are chosen randomly. As the number of nodes in the graph increases, the graph becomes more sparse, and the distance between any two random points in the graph increases as the plot in Fig 5.2 (b) reflects. Since the average path length increases for each transaction, therefore the total fee also increases.

Also, the fee for the untrained model is more as compared to the trained one and we get better results in fee improvement when the number of nodes is higher as we can see the gap is slightly more in less number of nodes. Thus, our model works better with a lesser number of landmark nodes.



(a) Average Fee versus Number of Nodes plot



(b) Average path Length versus Number of Nodes plot

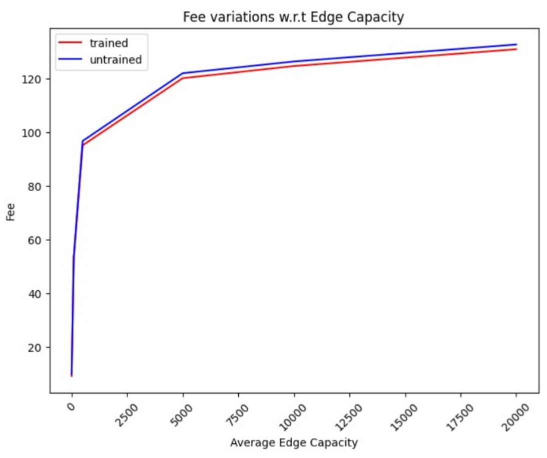
Fig. 5.2: Comparison of average fees and average path length with the variation in number of nodes.

Edge Capacity

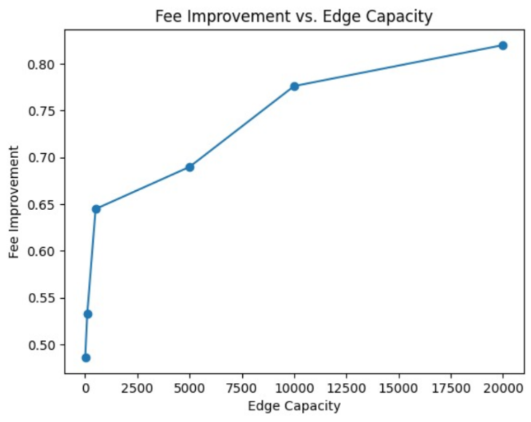
The plot in Fig. 5.3(a) shows how the average fee is increasing as we increase the channel capacities of the network. Again we can see the results are better in the case of our model. The average fee increases with the channel capacity mainly due to two factors-

1. The transactions with higher amounts also pass successfully, which have a high fee as the proportional fee gets higher in that case.
2. Secondly, the imbalance fee also increases as we increase the channel capacity values.

The plot in Fig. 5.3(b) shows our model works better when the channel capacities are higher, i.e. when more transactions get successful, the reason behind this may be that more transactions are getting ready to be forwarded, leading to more need of the priority-aware scheduling.



(a) Average Edge Capacity versus Fee plot

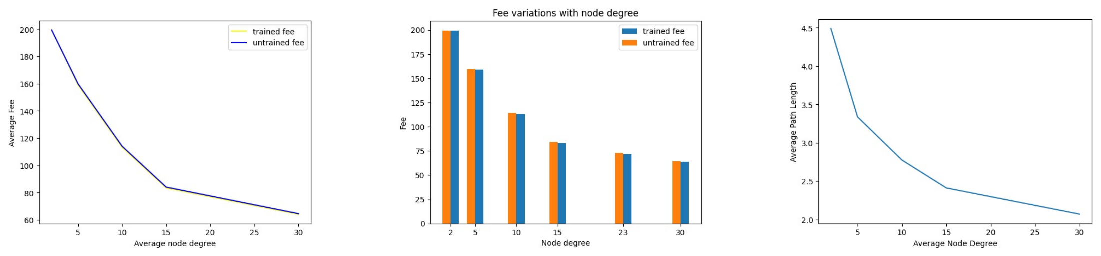


(b) Fee improvement in percentage versus average edge capacity plot

Fig. 5.3: Comparison of fee on increasing the channel capacity

Node Degree

The plots in Fig. 5.4. show that having a more dense network helps us in many ways. As the graph becomes more connected, the average distance between any two randomly chosen nodes decreases. The path length becomes less and the fee lowers as shown in the plots. Here, the path-finding algorithm alone is solving the problems as now they have too many path options to go through. This model does not improve the performance too much here as we can see in the plot of Fig 5.4(b).



(a) Average Fee versus Average Node Degree plot

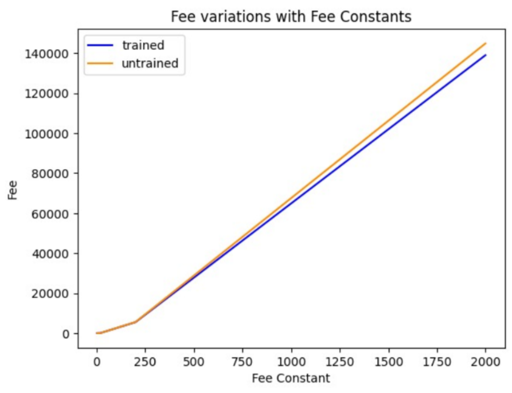
(b) Average Fee versus Average Node degree plot

(c) Average path length versus Average Node Degree plot

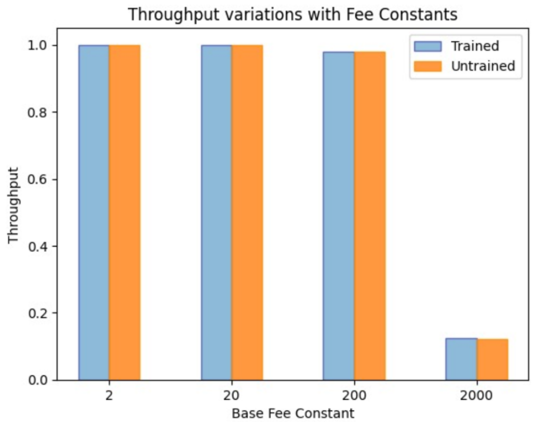
Fig. 5.4: Variations in fee and average path length on changing the node degree.

Fee Constants

The plot in Fig. 5.5(a) shows how the total fee increases with the increase in Fee constant values, this is straight as the calculation of fee is directly getting affected by fee constants. The throughput decreases with the increase in fee constants because the amount sent by the sender increases with the increase in fee taken by the intermediate nodes. This will lead to more usage of channel balances as the amount is increased. Thus, the channels will be exhausted more frequently, ultimately, causing a decrement in throughput.



(a) Fee versus Fee Constant plot plot



(b) Throughput versus Fee Constant plot

Fig. 5.5: Variations in fee and throughput on changing the fee Constants

5.2 Throughput

Now, we see how throughput gets affected after all these simulations. The plot in Fig. 5.6 shows the throughput for trained vs untrained. As we can see, both lines overlap i.e. the throughput is not changing by much. The values of the both trained and untrained model are listed below.

Throughput trained: 0.9972776989383355
Throughput untrained: 0.9972776989383355

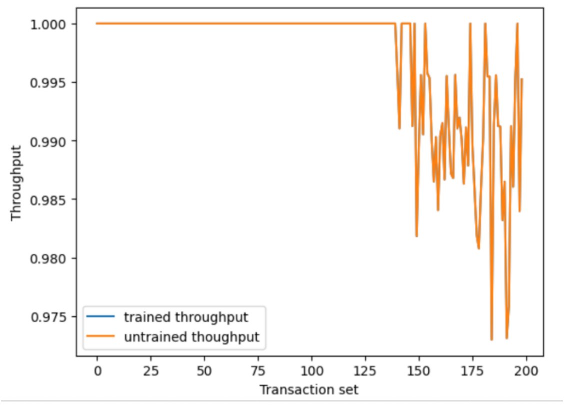
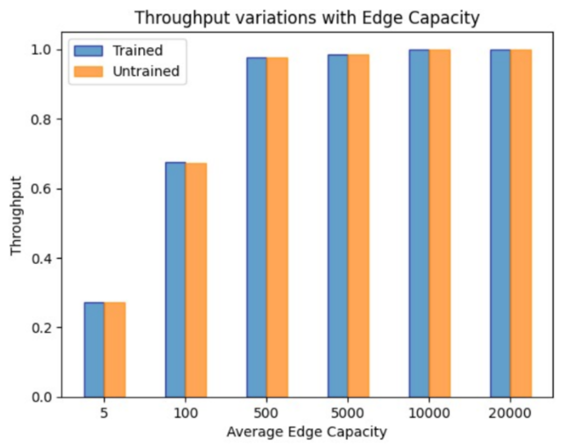


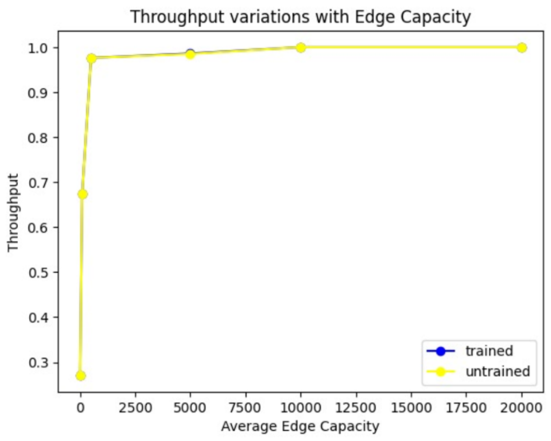
Fig. 5.6: Throughput as the model training progresses.

The plots in Fig. 5.7 show the throughput trends for trained and untrained models as we increase the average channel capacity. As we can see, throughput for trained as well as untrained model increases with edge capacity as the number of transactions failing due to lack of balances decrease with an increase in the channel capacity. The throughput reaches a high value when the channel capacity is very high compared to the amount passed. In this case, all the transactions get completed and the throughput is max in this case.

If the edge capacity is very low, very few transactions with lesser amounts pass, therefore the throughput is less. The throughput shoots exponentially after increasing the channel capacity to a certain amount because the amount passed in each transaction is in the range of 1-100 only, therefore after increasing the edge capacity to a certain amount, almost all of the transactions get completed.



(a) Throughput versus Average edge capacity plot

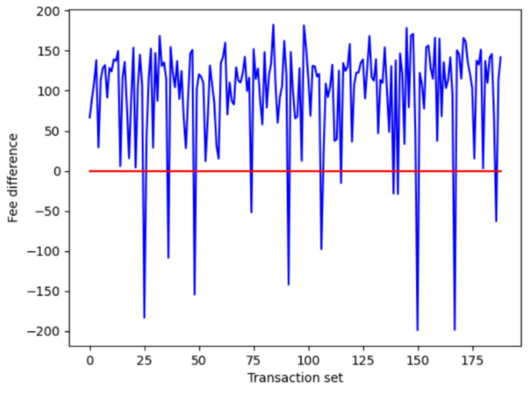


(b) Throughput versus Average edge capacity plot

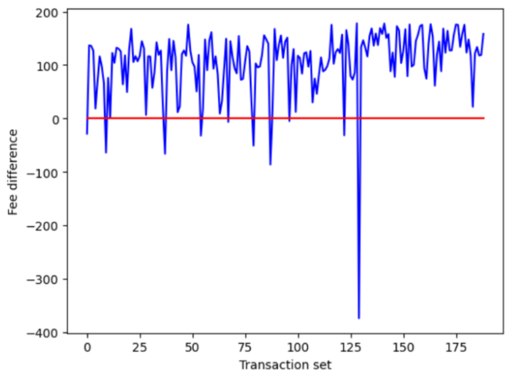
Fig. 5.7: Throughput of the network on increasing edge capacity.

5.3 Comparison of non-learning algorithms

In this last section of the evaluation, we compare the two path-finding algorithms we used in our simulator, the DFS based and the Bellman Ford-based routing algorithm. Note that the comparison done below is based on this context: The path provided by which algorithm when fed to the model gives better results.



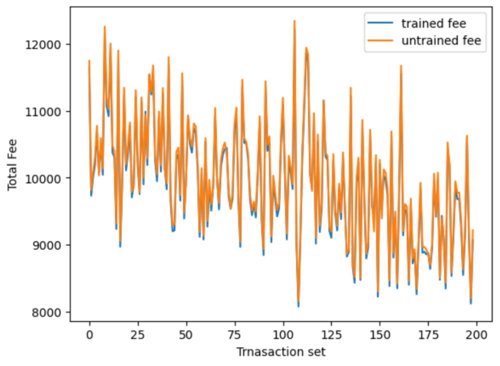
(a) Fee difference versus transaction set - DFS



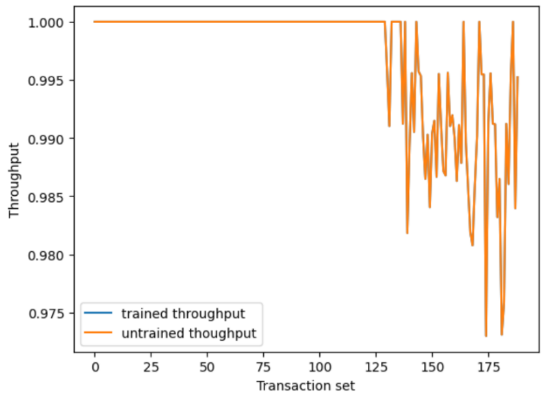
(b) Fee Difference versus Transaction set - Bellman-Ford plot

Fig. 5.8: Average fee for both algorithms

The average fee is always better irrespective of the algorithm used to find the path when the model uses the paths provided by them. However, the fee is higher in the case of DFS as the path given by this algorithm is not optimized and the learning algorithm does not change the path followed by the transaction, while Bellman-Ford finds the path based on which path has a lesser fee.



(a) Throughput versus transaction set - DFS



(b) Throughput versus Transaction set - Bellman-Ford plot

Fig. 5.9: Throughput for both algorithms

From the plots in Fig. 5.9, we can see the throughput values are almost similar but DFS has a slight edge over Bellman-Ford. The reason is that for each transaction, the path is calculated beforehand using the same graph. But when after the execution of transactions, some of the channel balances get exhausted. Now, the Bellman-Ford algorithm returns the shortest path. While finding the shortest path in PCNs, the balances around nodes with higher degrees are utilized more, therefore, their balances may exhaust earlier when comparing this to pathfinding in DFS based algorithm as it returns a random path. Therefore the throughput in the case of DFS is slightly better than that of Bellman-Ford.

Chapter 6

Conclusions and Future Work

In this report, we proposed a Priority Aware Transaction Scheduling done by our DL model on the pre-calculated paths found by simple routing algorithms. After obtaining the results in Chapter 6, we saw that the model tries to minimize the total fee of the network and it does up to some extent.
We compare our results with other routing algorithms where the transactions get executed without the help of any model like Distributed Bellman-Ford or DFS. The results show that around 0.7 to 1.0 percent improvements have been observed in forwarding fees per transaction. Although, this is not a significant decrement, but is a decrement indeed which can help users as the throughput of the network does not lower too much to minimize the average fee.

The average results by our DL-based algorithm suggest three things, those are as follows.

1. We are not always going to get outstanding results when we use a learning-based model, because sometimes, like in this case, the model does supervised learning and gives output as priorities but we do not have the ideal priorities of any (Graph, Payment Requests, Paths) combination. Thus, learning for the model becomes very difficult as these calculated priorities need to go into another function that calculates the total fee. Now, ideally, the total fee should tend to be 0, thus the model can learn from here. But this composition of two functions makes learning very difficult.

2. Some Another model also can be used which does not require pre-calculated paths it can find some path for every payment request which can improve other parameters. In that case, the process will be dynamic and maybe the model does better.

3. Since each node is a user and the whole system is decentralized, it will be a good option where each node acts as an agent and use Reinforcement Learning which guides the learning agent to maximize its reward by performing actions as per the status of the dynamic environment i.e. the use of Multi-Agent Reinforcement Learning.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] V. Buterin, "A next-generation smart contract and decentralized application platform," 2014.

[3] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," 2014.

[4] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[5] V. Sivaraman, X. Wang, and P. Viswanath, "High throughput cryptocurrency routing in payment channel networks," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, pp. 777–796, 2020.

[6] P. Gupta and P. R. Kumar, "Landmark-based routing in dynamic networks," in *Proc. 16th Annual Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, pp. 1546–1553, 1999.

[7] W. Li, J. Wu, Y. Zhang, and H. Xie, "Mpcn: A new routing protocol for mobile ad hoc networks with route prediction," *EURASIP Journal on Wireless Communications and Networking*, vol. 2006, no. 1, pp. 1–10, 2006.

[8] P. Wang, H. Xu, X. Jin, and T. Wang, "Flash: Efficient dynamic routing for offchain networks," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, (New York, NY, USA), pp. 370–381, Association for Computing Machinery, 2019.

[9] G. Malavolta, P. Moreno-Sanchez, A. Kate, and M. Maffei, "Silentwhispers: Enforcing security and privacy in decentralized credit networks," *IACR Cryptology ePrint Archive*, vol. 2016, pp. 1054–1071, 2016.

[10] C. Decker and R. Russell, "Flare: An approach to routing in lightning network," in *Proceedings of the 2nd Workshop on Bitcoin Research*, pp. 49–61, ACM, 2015.

[11] R. Zhang and Y. Chen, "Speedy murmurs: Efficient payment routing for blockchain networks," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 98–108, IEEE, 2019.

[12] E. Rohrer, J.-F. Laß, and F. Tschorsch, "Towards a concurrent and distributed route selection for payment channel networks," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pp. 411–419, Springer, 2017.

[13] Y. Zhang, D. Yang, and G. Xue, "Cheapay: An optimal algorithm for fee minimization in blockchain-based payment channel networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 2, pp. 245–258, 2017.

[14] Y. Zhang, J. Li, G. Xue, and D. Yang, "Robustpay: Securing off-chain payment hubs against malicious bitcoin-based attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 4, pp. 640–653, 2018.

[15] Y.-C. Chiang, C.-C. Wang, and H.-Y. Lin, "The ant routing algorithm for the lightning network," in *2019 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1917–1925, IEEE, 2019.

[16] X. Lu, Y. Zhang, J. Liu, and D. Yang, "Learning-based off-chain transaction scheduling in prioritized payment channel networks," *IEEE Transactions on Services Computing*, 2021.

[17] J. C. D. M. Marco Conoscenti, Antonio Vetrò, "Cloth: A lightning network simulator," 2021.

[18] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[19] B. Visuals, "Bitcoin visuals." `https://bitcoinvisuals.com/`, accessed 2023-04-21.

[20] A. Clauset, C. R. Shalizi, and M. E. Newman, "Power-law distributions in empirical data," *SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[21] M. J. Hirsch and M. Habibi, "A polynomial-time algorithm for finding a havel-hakimi sequence and a degree sequence realizing it," *Journal of Graph Theory*, vol. 83, no. 1, pp. 83–92, 2016.

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research*, vol. 9, no. Jun, pp. 249–256, 2010.