Learning a Scalable Algorithm for Improving Betweenness in the Lightning Network

1st Vincent Davis
Computer Science Department
University of Kentucky
Lexington, KY
vmda225@uky.edu

2nd Brent Harrison
Computer Science Department
University of Kentucky
Lexington, KY
bha286@g.uky.edu

***Abstract*—This paper presents a scalable algorithm for solving the Maximum Betweenness Improvement Problem as it appears in the Bitcoin Lightning Network. In this approach, each node is embedded with a feature vector whereby an Advantage Actor-Critic model identifies key nodes in the network that a joining node should open channels with to maximize its own expected routing opportunities. This model is trained using a custom built environment, *lightning-gym*, which can randomly generate small scale-free networks or import snapshots of the Lightning Network. After 100 training episodes on networks with 128 nodes, this A2C agent can recommend channels in the Lightning Network that consistently outperform recommendations from centrality based heuristics and in less time. This approach gives nodes in the network access to a fast, low resource, algorithm to increase their expected routing opportunities.**

***Index Terms*—Bitcoin, Lightning Network, Betweenness Improvement, Reinforcement Learning, Optimization**

I. Introduction

Since its inception in 2008, Bitcoin has reached a magnitude in both use and value that it has become more economical to defer finalization of exchange between individuals. Use cases such as micropayments, subscriptions, and streaming payments have become too expensive to be performed directly on the Bitcoin blockchain. Furthermore, transaction throughput of the Bitcoin network is limited by design. Bitcoin has a fixed blocksize and blocktime which limits the speed of the network to processing on average one 4MB block every 10 minutes or a maximum of seven transactions per second [15]. This design decision keeps Bitcoin's storage requirements accessible, but it also limits Bitcoin's scalablility to a global financial network.

The Lightning Network is a payment protocol built on top of the Bitcoin blockchain meant to answer it's scalability problem while still maintaining a trustless payment system. It is characterized as a peer-to-peer Payment Channel Network (PCN) consisting of nodes and channels, with both capacity and liquidity [16]. Nodes are incentivized by transfer fees to route the payments of others. Payments are source-routed over cheapest paths. Therefore, routing nodes are incentivized to maximize the number of cheapest routes on which they lie. This problem is more formally known as the Maximum Betweenness Improvement Problem (MBI). Opening channels with this goal in mind has been shown to lead to higher expected routing opportunities and expected revenue[8, 11].

However, current approaches to MBI are either intractable or suboptimal. For example, implementations of the exact MBI algorithm [11] take between 30 to 40 minutes per channel on snapshots of the Lightning Network from 2019. The network topology updates as often as every 10 minutes and so the suggestions of an exact algorithm may be obsolete by the time the algorithm completes. The Greedy algorithm for improving betweenness [3] approximates the exact solution within a factor of $1 - \frac{1}{2e}$, but its complexity grows polynomially with the size of the network. There is much time that can be saved by not investigating "low quality" nodes.

Centrality based heuristics, such as LightningNetworkDeamon's *autopilot* which prefers nodes with high betweenness centrality, do not make use of the underlying structure of the network. The effect these algorithms have on improving betweenness has been shown to be superficial as nodes usually place themselves in high competition/low revenue areas[11].

Furthermore, neither exact algorithms nor heuristics make use of previous work. Instead, a reinforcement learning method should be used. By framing the MBI problem on the Lightning Network as a RL problem, an agent can learn the impact that each channel opening has as it relates to the joining node's current position in the network. The insights gained from this work can be exploited to produce better recommendations.

This paper presents a novel reinforcement learning approach to solving the MBI problem as it appears in the Bitcoin Lightning Network. In this approach, each node is embedded with network context using a Graph Convolutional Network. An Advantage Actor-Critic (A2C) agent then identifies key nodes via these embeddings that a joining node should open channels with to maximize its own betweenness centrality. This model is trained using a custom built environment, *lightning-gym*, which can randomly generate small networks or import snapshots of the Lightning Network and simulate channel openings. After 100 training episodes on graphs with 128 nodes, the A2C agent can recommend channels in the Lightning Network that consistently outperform recommendations from centrality based heuristics and other baselines. This approach gives nodes in the network access to a fast, low resource, algorithm to increase their expected routing opportunities.

The major contributions of this study include:

- a scalable reinforcement learning algorithm to the general MBI problem.
- *lightning-gym*, an OpenAI gym environment for the Lightning Network capable of training and comparing multiple attachment strategies on random and real data.
- Monthly snapshots of the Lightning Network from February 2021 to December 2021. This data is available for use in training within *lightning-gym*.

II. Background

A. The Lightning Network

Altogether, the Lightning Network is a graph made up of nodes interconnected by channels containing liquidity balances. Channels are the mechanism by which two parties exchange value over the Lightning Network. They are composed of two types of on-chain transactions: an initial funding transaction, and a commitment transaction. The funding transaction determines the *capacity* of the channel, while the commitment transaction determines the *liquidity* balance. When creating a channel, funds of one or both parties are locked in a 2-of-2 multisignature address on-chain via a funding transaction. Updating the *liquidity* balance of this channel requires a new commitment signed by both parties. Funds are dispersed when the latest signed commitment transaction is broadcast on-chain. The funds are then moved from the channel to on-chain addresses owned by either party according to the balance. Thus, an unlimited number of feeless payments between the parties becomes just two on-chain transactions; one to open the channel, and one to close it [16].

Liquidity balance refers to the division of a channel's capacity into inbound liquidity and outbound liquidity. Inbound and outbound liquidity describe how much funds are available to be received or sent via incident channels, respectively. Sending funds across the Lightning Network reduces the outbound liquidity of the sender's channel and increases the outbound liquidity of the recipient's channel. If a node's channels consists of only outbound liquidity, they are unable to receive funds and vice-versa. Keeping a balanced liquidity allows nodes to route the payments of others.

Routing nodes in the Lightning Network leverage their liquidity to route payments between peers that do not share a direct channel. In exchange for this service, routing nodes will charge an interchange fee. When making a payment, nodes will choose the cheapest path available with respect to these fees. Therefore, for a node to maximize its *expected* routing opportunities, it should open low-fee channels with other nodes such that it lies on as many cheap paths as possible[8]. There is a real-world cost associated with opening and closing channels, and so nodes can only open as many channels as their budget allows. In other words, a routing node wants to open channels such that they maximize the improvement of their betweenness centrality.

B. Maximum Betweenness Improvement Problem

Formally, this problem is known as the Maximum Betweenness Improvement (MBI) problem.

Definition 1 (**Betweenness Centrality**)**.** The betweenness centrality of a vertex is proportional to the total number of shortest paths that pass through that vertex [9], i.e.,

$$\mathbf{bc}(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

The MBI problem is concerned with adding edges between a particular node v, and other nodes in the network such that the betweenness of v is maximized. This paper considers the budget constrained version of this problem.

Definition 2 (**MBI Problem**)**.** Given a graph $G = (V, E)$, a vertex $v \in V$, and a budget $k \in \mathbb{N}$, add k edges incident to node v such that $\mathbf{bc}(v)$ is maximized [3].

Betweenness centrality is a useful graph metric as it indicates how many shortest paths make use of a given node or edge. It can be used to determine key nodes in the flow of information, resources, traffic, etc.

In PCNs like the Bitcoin Lightning Network, flow occurs in the form of payments between individuals across channels.

The cost to send a payment over a channel from node i to a non-adjacent node j, c_{ij}, is determined by its policy. The policy contains information about the base fee, b_c, and the fee rate, r_c. The fee f_c to send a payment amount a over a channel c is $f_c = b_c + r_c * a$. The channel fee policies are replaced in by edge weights in simulation. A fixed size payment of 1M sats (0.001 BTC) is applied to all fee policies and the returned value representing the cost to forward a payment one hop is assigned as the edge weight.

The current implementation of the Lightning Network uses source routing to find the cheapest available path to route a payment. Fee-weighted betweenness centrality indicates how many cheapest paths make use of a given node or channel. Therefore, this study focuses on maximizing fee-weighted betweenness centrality. The only difference being that instead of measuring path distance by hop count, distance is measured using a channel's fee policy and some fixed payment amount.

MBI is an NP-Hard problem with no polynomial time approximation algorithm within $\frac{1}{2e}$ (unless P=NP) [3]. As will be shown later on, the time required to calculate betweenness centrality alone has increased significantly since the inception of this paper.

C. Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique where an agent learns to solve a Markov Decision Process (MDP) by interacting with an environment and observing the outcome. A MDP can be expressed as a tuple of $< S, A, T, R, \gamma >$ where S is the state space and A is the set of actions an agent can take[13]. T describes how states transition to one another when an agent takes an action. R is a function that describes the reward of a state. γ is known as the discount factor which determines how much an agent discounts long-term rewards. The goal of an agent in a MDP is to learn a policy π that specifies the action that should be taken in each state that maximizes expected long-term reward.

Advantage Actor-Critic (A2C) algorithms consist of 3 parts: a policy network, a value network, and an advantage function. The policy network learns the policy π, that dictates which action to take in a given state. The value network predicts the quality of a given state. The advantage function calculates the error between the predicted quality of a state and its actual quality. The advantage is used to teach the value network to predict the quality of a state which in turn teaches the policy network to suggest actions that lead to better states[14].

III. Related Works

The greedy algorithm proposed in [3] iteratively suggests edges that lead to the greatest improvement in betweenness centrality. They are able to save runtime by using a dynamic algorithm to update the betweenness centrality of a node with each new edge. Additional time is saved after the first iteration since the algorithm is able to prune node suggestions from future iterations if the improvement is less than some observed lower bound. While their dynamic algorithm for calculating betweenness is faster, the memory footprint scales polynomially, which makes it unsuitable for large networks.

The most popular Lightning Network protocol implementation, LightningNetworkDaemon (LND), includes an autopilot feature that will automatically open and manage channels on the user's behalf. *autopilot* uses the **Betweenness** heuristic first identified in [17] to suggest with which nodes to open channels. The heuristic suggests nodes with preference to high betweenness centrality based on the intuition is that creating channels with well connected nodes will in turn make the joining node well connected also. Note that this implementation considers shortest paths rather than cheapest paths. The original implementation also calculates betweenness centrality once and selects the top k vertices with respect to betweenness centrality (where k is the budget), whereas the implementation in LND *autopilot*, recalculates betweenness centrality each time it makes a selection.

Others works [3, 8] show that the **Greedy** algorithm is superior to centrality based selection like **Betweenness** in improving betweenness centrality. However, [8] goes further to show that expected revenue improvement is also impacted by the channel attachment strategy. In their setup, they test different algorithms for selecting nodes, and then after making those selections, determine what fees to charge to maximize expected revenue. Other factors kept equal, they showed that using an algorithm that focuses on improving betweenness directly rather than using a centrality based heuristic leads to higher expected revenue improvement. Furthermore, using the **Betweenness** attachment strategy lead to worse expected revenue improvement than **Random** selection. Their explanation is that because of the nature by which the algorithm suggests nodes, there is not much opportunity to change fees to increase expected revenue without decreasing expected routing opportunities [8].

This claim is supported further by the work of [11], which explores the impacts of different attachment strategies on the node's ability to use the network and the network's topology. Of the strategies they evaluated in simulation, **Betweenness** resulted in less routed transactions than **Random** selection. On the other hand, they found that the exact MBI strategy leads to the greatest increase of expected routing opportunities. Unfortunately, the exact MBI algorithm is computationally expensive. Each additional edge in the budget added at least 30 minutes to the runtime of the algorithm.

The problem of maximizing betweenness is being repeatedly solved on slightly changed graphs, yet, neither of the mentioned approaches rely on previous work. Instead, a reinforcement learning model should be used.

Previous research [7] has successfully learned greedy heuristics with low approximation ratios for solving hard problems by using a graph embedding technique, *structure2vec*, and a DeepQNetwork. This type of approach is attractive as the agent can generalize their learning between graphs of different sizes and similar distributions.

Other embedding techniques such as Graph Convolutional Networks (GCN) are useful in solving hard problems [12]. This work explores using a GCN to label whether each node belongs to a solution set, in this case the Maximal Independent Set. Their algorithm uses a trained GCN to produce a probability mapping which guides a depth first search for solutions. The author's approach was able to solve a set of SAT competition problems faster than a state-of-the-art solver, ReduMIS.

GCNs are a powerful method of representation that is both permutation invariant and inductive[18]. Combining this graph representation method with a reinforcement learning model allows the agent to exploit previous work on unseen instances of the problem through relational inductive bias[2]. By using a graph representation method in tandem with a reinforcement learning model, every observation can be turned into a learning opportunity.

IV. Methods

The MBI problem can be framed as a MDP whereby an A2C agent learns which nodes in the network will maximize the betweenness centrality of a joining node. In the process of training, the agent approximates two functions: the state-action function, or policy, and the state-value function.

To facilitate the agent's learning, a representation model is required. This study chose to use a Graph Convolutional Network to embed each node. Given an instance of the network, the GCN embeds each node with information about the node itself and its neighborhood. Successive layers embed information from nodes an increasing distance away. These embeddings will allow the A2C agent to identify key nodes in the network.

A. Markov Decision Process Formulation

Given a graph $G = (V, E)$, a joining node $v \in V$, and a budget $k \in \mathbb{N}$, the state, action, and reward space of the MDP are defined as:

- *Actions*: Any node $u \in V$ which is not a neighbor of the joining node v is a legal action.

- *States*: A state S is a sequence of actions (nodes) on a graph G. A terminal state is reached when the budget k has been exhausted. i.e. when $|S| = k$.
- *Transitions*: Transitions are deterministic. When a node u is selected as an action, the edge (u, v) is added to G.
- *Rewards*: Suppose node u is selected as an action. The reward, $r(S, u)$ at state S is the betweenness improvement of node v after taking action u and transitioning to new state $S' = (S, u)$.

$$r(S, u) = bc_{S'}(v) - bc_S(v)$$

B. lightning-gym

We deployed a node to collect monthly snapshots of the Lightning Network from February 2021 to December 2021. These eleven snapshots contain information about the network's nodes and channels, including fee policies and capacity. These snapshots are used in the custom OpenAI Gym environment, *lightning-gym*. Unlike other Lightning Network simulators, *lightning-gym* uses a common interface defined by OpenAI[5]. This interface is episodic in nature, which allows for the training of reinforcement learning agents.

In training, *lightning-gym* generates a random scale-free directed graph and adds an isolated node. The graph is passed through the GCN to create the initial embedded state. The A2C agent "takes actions" by selecting nodes for the joining node to open channels with based on the embedded state. The environment simulates the channel opening and returns the betweenness improvement of the joining node as the reward and the new embedded graph as the next state. This process is repeated until the A2C agent's budget is exhausted. Afterwards, the agent updates its policy and value networks with regard to the new state-action-reward observations. The cumulative reward of an episode represents the total betweenness improvement the agent was able to collect.

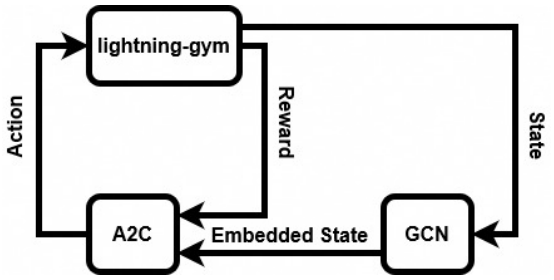

Fig. 1. Training architecture. A2C takes as input a graph embedding from the GCN and then takes an action in the environment, producing a new state-reward pair.

C. Graph Convolutional Network

Features such as a node's location in the network or the policies of its channels can be exploited by a reinforcement learning agent. However, the challenge lies in designing a node representation that captures this information across different networks. In arbitrary graphs, similar nodes should have similar embeddings. Graph Convolutional Networks address this challenge. In GCNs, each node aggregates the feature vectors of the nodes in its neighborhood, passes the aggregation to a linear layer, and finally applies a non-linear activation function [18]. Successive layers capture neighborhood information from nodes an increasing hop distance away.

$$H^{(l)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l-1)}W^{(l)})$$

where

- $H^{(0)} = X$ - the initial node feature tensor
- $\tilde{A} = A + I_N$ - adjacency matrix of G plus self loops
- $\tilde{D} = \sum_j \tilde{A}_{ij}$ - the out-degree of each node along the main diagonal
- W is the learnable parameters
- σ - the nonlinear activation function

The node feature vector X contains information about each node such as its degree centrality and whether or not the node is in the solution S (inclusion). Degree centrality is used as a low-cost indicator of influential nodes. Information about inclusion allows the agent to identify the neighborhood of nodes to which it has already connected. Inclusion is updated after each action.

The A2C agent is trained using random scale-free networks generated by the Barabasi-Albert Model[1]. The training graphs are unweighted. However, testing performance is increased by using the reciprocal of channel costs as coefficients.

$$H^{(l)} = \sigma(\tilde{E} \times H^{(l-1)}W^{(l)})$$

where

- $\tilde{E} = \tilde{D}^{-\frac{1}{2}} \times E \times \tilde{D}^{-\frac{1}{2}}$ - normalized edge weights
- $E = [e_{ij} = \frac{1}{f_{c_{ij}}}]$, the reciprocal of the cost of the channel, if it exists, otherwise $e_{ij} = 0$.

D. Advantage Actor-Critic Model

Actor-critic models separate the roles of evaluating the quality of a state and determining the action to take in given a state. These roles are assigned to the value network and policy network, respectively. This architecture was chosen because it is well suited for problems with large state and action spaces. In the MDP formulation, both the state space and action space scale with the number of nodes in the network.

The A2C agent interprets each node by its embedding from the GCN. To the policy network, the state is represented by the embedded graph $H^{(l)}$. The policy network, P^π, takes as input the embedded network and produces a probability distribution representing each action's likeliness to be the best action in the current state. The policy network learns to increase the probability of selecting nodes that improve the fee weighted betweenness centrality of the joining node.

During training, the agent's policy samples from the probability distribution returned by the policy network, taking $H^{(l)}$ as input. During evaluation, the A2C agent's policy takes the mostly likely action according to the policy distribution.

$$\pi(S) := argmax_{u \in \bar{S}} P^\pi(H^{(l)})$$

To the value network, the state is represented by the aggregation of all of the node embeddings:

$$H_p^{(l)} = \sum_{v \in V} \mu_v$$

The value network is responsible for predicting the value of the current state. In other words, the value network predicts the expected betweenness improvement of the joining node at the end of the episode, given the current partial solution and policy. The value network takes as input a sum pooling of the output layer, $H_p^{(l)}$ and returns a scalar value approximating the quality of the current state.

$$V^{\pi}(H_p^{(l)}) \approx \mathbb{E}\left[\sum_{i=1}^{T} \gamma^{i-1} r_i\right]; \gamma = 0.99$$

During each iteration of an episode, the A2C agent collects the environment observations and outputs of the value and policy networks. The following policy gradient function is used in training the network:

$$\nabla_{\theta} J(\theta) \sim \left(\sum_{t=0}^{T-1} log\pi(a|S_t)\right) A(S_t, a)$$

where $A(S_t, a)$ is the difference between the actual quality of taking an action a in state S at time t and the expected quality:

$$A(S_t, a) = Q(S_t, a) - V^{\pi}(H_p^{(l)})$$

The observations and outputs are collected by the A2C agent to be used as input to the policy gradient function which back-propagates corrections through the value and policy networks.

V. Experiments

Three experiments are conducted to evaluate the A2C agent. Firstly, an experiment comparing performance, second, an experiment comparing scalability, and lastly, an experiment demonstrating the agent's ability to exploit previous work.

A. Baselines

The following baselines are compared against the performance and scalability of the A2C agent:

- **Random** - suggests k nodes sampled uniformly at random.
- **Degree** - suggests k nodes w.r.t. highest degree.
- **Betweenness** - suggests k nodes w.r.t. greatest betweenness centrality.
- k**-Center** - suggests k nodes that the minimize the joining node's *maximum* distance to all other nodes.
- **Greedy** - suggests nodes that result in the greatest betweenness improvement after trying all available actions.
- **Trained Greedy** - like Greedy, except this algorithm suggests the best action out of five sampled from the policy network of the trained A2C agent.

The approximation algorithm from [10] is used for the k-Center algorithm. Results from the **Greedy** algorithm are excluded for graphs with $\geq 1,000$ nodes. The performance of the **Random** algorithm is the average result of 30 trials. The python library, *igraph*, uses Brandes algorithm for calculating betweenness centrality [6, 4]. This is the same algorithm that **Betweenness** uses in *autopilot* for calculating the betweenness centrality of a node. This paper's implementation of **Greedy** is the same as in [8]. If the joining node has less than two channels, **Greedy** will first open up to two new channels with preference to nodes with highest degree.

The goal of the first experiment is to determine whether the A2C agent performs similarly to existing baselines on synthetic networks. To evaluate the performance of the A2C agent, **Experiment 1** compares each algorithm on two different sized synthetic environments with varying budgets. The cumulative reward of each algorithm is compared on two random BA graphs as the budget increases from 1 to 15. The first graph has 128 nodes, which is the same size graph as the agent is trained on. The second graph has 1024 nodes which is eight times larger.

The goal of the second experiment is to investigate the scalability of the A2C agent. To determine the scalability of the A2C agent, **Experiment 2** includes a comparison of the performances of each algorithm on snapshots of the Lightning Network. These performances are analyzed with respect to their runtimes and the size of the Lightning Network. In this real environment scenario, each algorithm is compared on monthly snapshots of the Lightning Network with a fixed budget of 10. The cumulative reward of each algorithm is collected after each episode.

Snapshots of the Lightning Network are pruned beforehand. For a channel to be included in the simulated network, it must be active, have a defined policy, and a minimum capacity. If a channel is not active or its policy is undefined, it cannot be used by the network, and so it is removed. Channels with low capacity are easily made unbalanced with relatively low payments. Therefore, low capacity channels do not add to the connectivity of a node and are removed. The minimum channel capacity is set to 0.01 Bitcoin. If a pair of nodes has more than one channel between them, the higher fee is retained and the capacities are combined.

The goal of the third experiment is to demonstrate the agent's ability to exploit previous work. **Experiment 3** evaluates the training performance of the A2C agent under two different scenarios. *lightning-gym* is designed so that the agent can be trained on a new random instance every episode or repeatedly on a single instance. The former scenario should lead to better performance on the general MBI problem, but the latter scenario is more analogous to repeatedly solving the MBI problem in a specific context, such as the Lightning Network. Repeated training demonstrates whether the A2C agent is able to exploit previous work. The cumulative reward after each episode is plotted as a percentage of the performance of the **Greedy** algorithm. i.e. A score greater than or equal to one means that the agent found a solution as good or better than found by greedy search. The agent is trained for 100 episodes on graphs with 128 nodes. It is a given a budget of 10 channel openings. The table below includes the parameters used for training the value and policy networks.

Training Parameter	Value
GCN Dimensions	1x128x128
Actor Network Dimensions	128x128
Policy Network Dimensions	128x1
Learning Rate	1e-2
Decay Rate	0.999
Activation	ReLu
Optimizer	ADAM

VI. Results

A. Experiment 1 - Performance

Figures 2 and 3 show a comparison of the A2C agent's performance as budget increases on two differently sized synthetic networks. Overall, behavior is similar between the two networks. Both show an uptrend in performance as budget increases, although the trend is milder in Figure 3. Centrality based heuristics show a clear advantage over **Random** and k-**Center**. There is more variation in performance on the network in Figure 2. On the larger network in Figure 3, the performance of the algorithms are similar at low budgets.

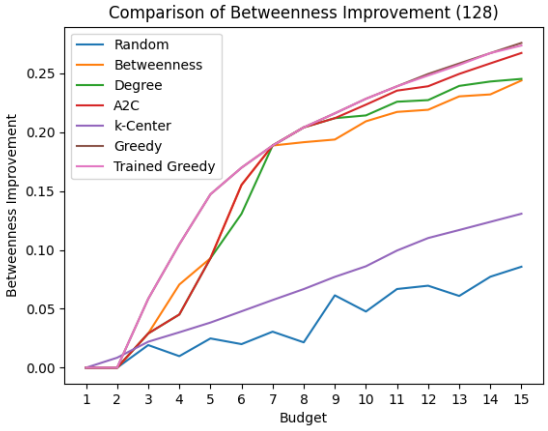

Fig. 2. Performance comparison on a synthetic network with 128 nodes.

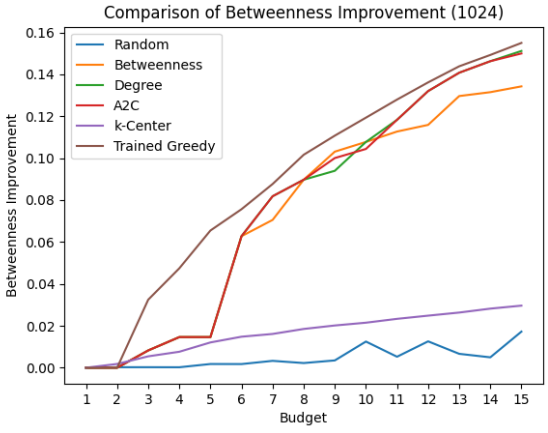

Fig. 3. Performance comparison on a synthetic network with 1024 nodes.

B. Experiment 2 - Scalability

From February '21 to December '21 the number of nodes in the network increased 85% from 10k nodes to 18.5k nodes. In the same time, the number of channels doubled from 20k channels to 40k channels. The size of the network is reduced significantly by the pruning operation. In February, the pruning operation reduced the number of nodes by 75%. However, this reduction decreased over time. In December, the pruning operation reduced the number of nodes by 63%.

The ratio of remaining 'well connected' nodes grew steadily over time relative to the entire network. In February, 75% of the nodes were removed, yet 45% of the edges remained. In the December snapshot, 66% of the nodes were removed and 66% of the edges remained.

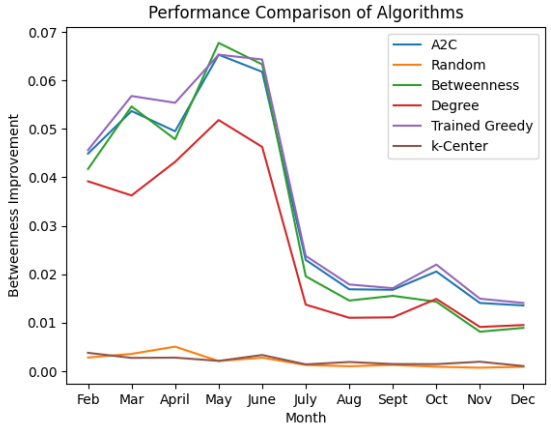

Fig. 4. Performance comparison on Lightning Network snapshots.

TABLE I
CHANGE IN NUMBER OF NODES AND EDGES AFTER PRUNING

Months	Nodes Before	Nodes After	Edges Before	Edges After
Feb	9602	2316	19949	8814
Mar	10516	2676	21063	9818
April	11161	2996	21854	10607
May	11907	3491	24039	12579
June	12770	3876	26550	14528
July	13944	4643	30772	18081
Aug	15469	5316	34600	21277
Sept	16374	5872	37427	23816
Oct	15891	6144	37345	24594
Nov	17572	6409	39672	25298
Dec	18558	6686	40938	26878

Table II compares the cumulative runtime of each algorithm over each month. The cumulative runtime is the total time in seconds each algorithm required to suggest ten channels on the given graph. **Betweenness** and **Trained Greedy** stand out as the two longest running algorithms. The other four algorithms execute faster than the shortest instance of **Betweenness**.

C. Experiment 3 - Learning

Figure 5 shows the A2C agent's performance quickly increasing after 40 episodes. The agent scores at least 80% of

TABLE II
COMPARISON OF RUNTIMES FOR EACH ALGORITHM.

Months	A2C	Random	Betw.	Degree	Trained	kCenter
Feb.	0.52	0.29	2.68	0.30	105.34	0.44
March	0.53	0.31	3.56	0.32	137.05	0.48
April	0.59	0.36	4.35	0.36	169.39	0.50
May	0.69	0.41	6.03	0.44	241.90	0.55
June	0.82	0.49	7.82	0.52	309.40	0.72
July	1.07	0.67	11.86	0.71	473.69	0.92
Aug.	1.28	0.70	15.93	0.74	642.95	1.07
Sept.	1.38	0.87	19.58	0.84	790.56	1.31
Oct.	1.56	0.95	22.13	0.92	855.17	1.25
Nov.	1.52	0.94	23.05	0.90	942.50	1.32
Dec.	1.60	0.97	25.76	0.98	1042.43	1.33

Greedy from there on. The agent's performance converges even sooner when repeatedly trained on a graph. In Figure 6, after 25 episodes, the agent scores at or above 90% of **Greedy**.

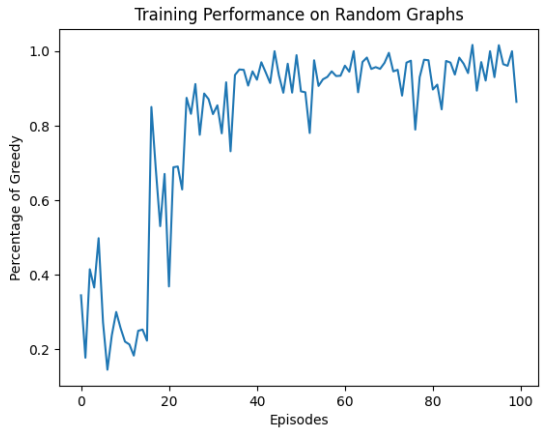

Fig. 5. Performance of agent trained on different random scale free networks with 128 nodes and a budget $k = 10$.

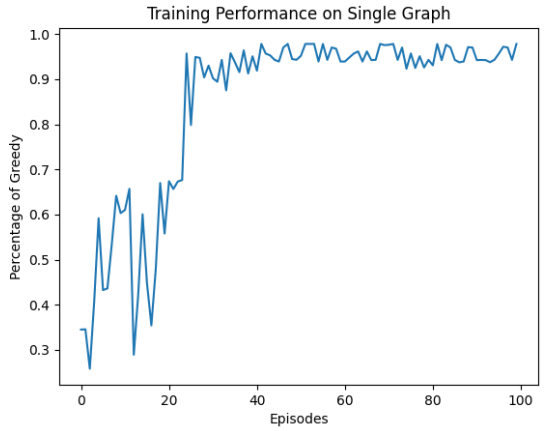

Fig. 6. Performance of agent repeatedly trained on single instance of a random scale free network with 128 nodes and a budget $k = 10$.

VII. DISCUSSION

A. Performance

The A2C agent shows comparable performance in the Figure 2, breaking out from the centrality based heuristics after a budget of 8. The next best performing algorithm is **Degree** followed by **Betweenness**. Surprisingly, k**-Center** is the only strategy whose node has a positive betweenness when given a budget of 2. In Figure 3, **Betweenness** begins to diverge after budget 6, but the A2C and **Degree** algorithm stay relatively equal. The A2C agent performs competitively, even on unseen networks eight times larger than those it was trained on. The results of this experiment indicate that the A2C agent maintains performance even as the size of the graph increases.

The **Trained Greedy** algorithm shows how much improvement can be gained by trading time to explore several recommended actions. This algorithm performs identically to the Greedy algorithm in the smaller network. In the network with 1024 nodes, **Trained Greedy** outperforms every other algorithm.

B. Scalability

In this section, the performance of each algorithm is evaluated with consideration to the runtime required to return their recommendations. This runtime is related to the size of the network and the complexity of the algorithm.

In Table **??**, it can be seen that the size of the network has been increasing consistently. Even though the pruning operation removes over half of the nodes, the number of channels is not as affected. This indicates that removal of these nodes had low impact on the connectedness of the remaining network. The smallest pruned instance of the Lightning Network is 16x larger than the networks the A2C agent was trained on.

Figure 4 shows the performances of each algorithm on monthly snapshots of the network. The **Trained Greedy** algorithm performs best in ten of the eleven months. The **Betweenness** algorithm outperforms both A2C and the **Trained Greedy** algorithm in May.

From February to June, the **Betweenness** and A2C algorithms perform similarly. Of those five months, **Betweenness** outperformed A2C in March, May, and June. From June on, A2C outperforms **Betweenness**. Additional improvement is gained through the **Trained Greedy** algorithm at the expense of time. **Degree** is the next best algorithm after **Betweenness**, but there is a large gap in their performance until October. Meanwhile, k**-Center** has a similar betweenness improvement as picking channels at random. Considering the nature of k-**Center** and the size of the Lightning Network, it may need a higher budget to be influential on the network.

Betweenness requires that the betweenness centrality of the network be calculated at least once. As a result, the minimum runtime of **Betweenness** increased 10 fold this past year from 2.6 seconds in February to 25.7 seconds in December. The **Trained Greedy** algorithm calculates betweenness several times before making a recommendation. Unsurprisingly, its runtime was the greatest among the seven algorithms. However, the **Trained Greedy** algorithm's runtime increased from

105 seconds to 1,042 seconds, a similar factor as **Betweenness**. Unlike the **Greedy** algorithm, which must try all actions before choosing the best one, **Trained Greedy** tries a constant number of actions each iteration and therefore scales at the same rate as **Betweenness**.

In contrast, the total time required for the A2C agent to suggest 10 channels only increased by a factor of 3 from 0.5 seconds in January to 1.6 seconds in December. All algorithms which did not need to calculate betweenness centrality increased by a factor of 3. This rate of growth is the same rate of growth as the number of nodes in the Lightning Network snapshots. This relation suggests that the reinforcement learning algorithm, like **Degree** and **Random**, scales linearly with the size of the network.

C. Training

By measuring performance as a percentage of **Greedy**, it is more clear that the A2C agent learns to generalize between different graphs. Figure 6 shows the performance of the A2C agent when trained repeatedly on a single instance. This use case is more analogous to repeatedly solving for MBI on a slowly growing Lightning Network. The A2C agent quickly approaches **Greedy** performance when trained on a single graph. The results of this experiment show that this reinforcement learning approach is able to exploit previous work done for solving the MBI problem.

VIII. Conclusion

The MBI problem can be formulated as a MDP, thus allowing the application of reinforcement learning techniques such as advantage actor-critic methods. In order to generalize learning, a graph embedding technique is also required. This paper chose to use GCNs as they are able to capture structural information about a node and its neighborhood. A custom developed Lightning Network environment, *lightning-gym*, is used to train an A2C agent on random graphs and evaluate it against six other algorithm for suggesting channels in the Lightning Network. Through experimentation, it is shown that an A2C model can learn a greedy heuristic to suggest channels that increase a joining node's betweenness centrality. After training on small randomly generated graphs for 100 episodes, the agent consistently outperforms the current selection algorithm implemented in the LND *autopilot* as well as four other baselines. Furthermore, this work showed that the runtime of this reinforcement learning algorithm scales linearly with the size of the Lightning Network. Using a trained A2C agent to select nodes in combination with a GCN for node embedding is a fast, low-resource algorithm for improving one's betweenness centrality in the Lightning Network.
Link to project/data:
https://github.com/davisv7/lightning-gym

References

[1] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512.

[2] Peter W Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv preprint arXiv:1806.01261* (2018).

[3] Elisabetta Bergamini et al. "Improving the betweenness centrality of a node by adding links". In: *Journal of Experimental Algorithmics (JEA)* 23 (2018), pp. 1–32.

[4] Ulrik Brandes. "A faster algorithm for betweenness centrality". In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[5] Greg Brockman et al. *OpenAI Gym*. 2016.

[6] Gabor Csardi, Tamas Nepusz, et al. "The igraph software package for complex network research". In: *InterJournal, complex systems* 1695.5 (2006), pp. 1–9.

[7] Hanjun Dai et al. "Learning combinatorial optimization algorithms over graphs". In: *Advances in Neural Information Processing Systems* 2017-Decem (Nips 2017), pp. 6349–6359. ISSN: 10495258.

[8] Oğuzhan Ersoy, Stefanie Roos, and Zekeriya Erkin. "How to profit from payments channels". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2020, pp. 284–303.

[9] Linton C Freeman. *A Set of Measures of Centrality Based on Betweenness*. 1977, pp. 35–41.

[10] Teofilo F. Gonzalez. "Clustering to minimize the maximum intercluster distance". In: *Theoretical Computer Science* 38 (1985), pp. 293–306. ISSN: 0304-3975.

[11] Kimberly Lange, Elias Rohrer, and Florian Tschorsch. "On the impact of attachment strategies for payment channel networks". In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE. 2021, pp. 1–9.

[12] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. "Combinatorial optimization with graph convolutional networks and guided tree search". In: *Advances in neural information processing systems* 31 (2018).

[13] M.L. Littman. "Markov Decision Processes". In: *International Encyclopedia of the Social Behavioral Sciences*. Ed. by Neil J. Smelser and Paul B. Baltes. Oxford: Pergamon, 2001, pp. 9240–9242.

[14] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[15] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: www.bitcoin.org.

[16] Joseph Poon and Thaddeus Dryja. *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments*. 2016.

[17] Rami Puzis, Yuval Elovici, and Shlomi Dolev. *Finding the Most Prominent Group in Complex Networks*. 2007.

[18] Max Welling and Thomas N Kipf. "Semi-supervised classification with graph convolutional networks". In: *J. International Conference on Learning Representations (ICLR 2017)*. 2016.