SEAL: Learning Heuristics for Community Detection with Generative Adversarial Networks

Yao Zhang
yaozhang18@fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
China

Yun Xiong
yunx@fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
Shanghai Institute for Advanced Communication and Data Science
China

Yun Ye
Tengfei Liu
Weiqiang Wang
yeyun.yy@antfin.com
aaron.ltf@antfin.com
weiqiang.wwq@antfin.com
Ant Financial Services Group
China

Yangyong Zhu
yyzhu@fudan.edu.cn
Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University
Shanghai Institute for Advanced Communication and Data Science
China

Philip S. Yu
psyu@cs.uic.edu
University of Illinois at Chicago
United States

ABSTRACT

Community detection is an important task with many applications. However, there is no universal definition of communities, and a variety of algorithms have been proposed based on different assumptions. In this paper, we instead study the semi-supervised community detection problem where we are given several communities in a network as training data and aim to discover more communities. This setting makes it possible to learn concepts of communities from data without any prior knowledge. We propose the **S**eed **E**xpansion with generative **A**dversarial **L**earning (SEAL), a framework for learning heuristics for community detection. SEAL contains a generative adversarial network, where the discriminator predicts whether a community is real or fake, and the generator generates communities that cheat the discriminator by implicitly fitting characteristics of real ones. The generator is a graph neural network specialized in sequential decision processes and gets trained by policy gradient. Moreover, a locator is proposed to avoid well-known free-rider effects by forming a dual learning task with the generator. Last but not least, a seed selector is utilized to provide promising seeds to the generator. We evaluate SEAL on 5 real-world networks and prove its effectiveness.

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403154

CCS CONCEPTS

• **Computing methodologies** → **Reinforcement learning**; Semi-supervised learning settings; • **Mathematics of computing** → *Combinatorial optimization.*

KEYWORDS

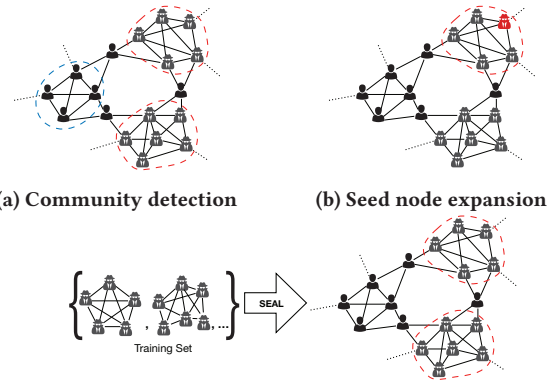
generative adversarial networks, reinforcement learning, community detection, graph combinatorial optimization

ACM Reference Format:
Yao Zhang, Yun Xiong, Yun Ye, Tengfei Liu, Weiqiang Wang, Yangyong Zhu, and Philip S. Yu. 2020. SEAL: Learning Heuristics for Community Detection with Generative Adversarial Networks. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403154

1 INTRODUCTION

Networks (graphs) provide a powerful framework for describing relationships among entities. To better understand a network, we often focus on subgraphs whose members are closely related, i.e., communities [37]. The community detection problems aim to identify communities from large networks.

Sometimes detecting all communities from a large network is expensive and unnecessary. For example, we would like to detect groups of fraudulent traders in an online trading network [22] as shown in Figure 1(a). Trading networks are typically overwhelmed by communities of normal accounts. An exhaustive detection would result in many communities irrelevant to the fraud detection task. The seed expansion [3] may be more useful in this scenario, which is also referred to as local community detection [34] or community search [10]. Given a query node, the goal of seed expansion is to find a local community that contains the query node. As shown in Figure 1(b), given the seed node, i.e., a risky account, we could detect the fraudulent group the node belongs to without touching the entire



(a) Community detection (b) Seed node expansion

(c) Semi-supervised community detection.

Figure 1: Comparison of related tasks. This is a subgraph of a trading network. (a) Community detection methods may detect one normal community (blue circle) and two fraudulent communities (red circles). (b) Given a seed node, i.e., a fraudulent account (red node), seed expansion methods only detect the community at the top-right corner, but neglect another one. (c) Semi-supervised community detection methods discover all fraudulent communities and filter out the normal one if gotten trained on fraudulent communities.

network. A limitation of seed expansion is that its coverage rate would be low since it is impractical to provide seed nodes for every anomalous communities.

Unfortunately, for both community detection and seed expansion problems, there is no universal definition of communities and a variety of assumptions were proposed [2, 10, 36, 37]. When we take attributes into consideration, the situation worsens [14, 20, 32, 38]. Trails are inevitable before we could select an appropriate "definition" when faced with a new dataset.

Recently, Bakshi et al. proposed Bespoke, a semi-supervised approach to community detection [4]. The authors observed that communities in a network are similar to each other and can be clustered into 3 to 5 community patterns. Given several training communities, Bespoke could conclude patterns from them without relying on any prior knowledge about communities. Then Bespoke computes matching scores between nodes and patterns, and nodes with highest scores are recognized as seed nodes. For each seed node, its 1-ego net, i.e., its neighbors along with itself, is treated as a community and returned. However, we can notice that Bespoke still makes an assumption at inference phase that communities are 1-ego nets. Moreover, extraction of community patterns only considers internal connectivities of communities. It also cannot deal with attributes. These limit its capacity.

In this paper, we study community detection problems in the similar semi-supervised setting as Bespoke. But we impose no restrictions on structures of output communities. We also want to capture characteristics of training communities more comprehensively, e.g., with the help of attributes. As shown in Figure 1(c), a model is expected to take a set of training communities, e.g., fraudulent communities, as inputs, and detect communities that are similar to training ones. Training communities can be labeled by experts. It is also possible to use semi-supervised community detection as a supplement to seed expansion where communities found by seed expansion are treated as training data, and we aim to cover more communities without seeds. Despite the significance of the semi-supervised community detection problem, it is challenging due to the following reasons:

• **Diverse definitions:** There are many aspects that definitions of communities are based on, such as internal/external connectivity and attributes similarity [37]. It is necessary but challenging to capture characteristics of communities from as many aspects as possible since we do not know what kind of networks we may encounter.

• **Flexibility:** Most existing methods make assumptions on structures of communities, e.g., k-clique, k-truss, and k-core [10, 26]. Only communities meeting the structure constraints are considered. How to design a model that is flexible enough to detect communities with arbitrary structures is a challenge.

• **Free-rider effects:** A well known issue for community detection is free-rider effects [34] where communities are over-estimated. This is also an unwanted effect we need to avoid.

To tackle above issues we resort to heuristic learning [16] that learns heuristic rules from training data and trials. We propose SEAL (**S**eed **E**xpansion with generative **A**dversarial **L**earning) composed of 4 components as shown in Figure 2. The generator and the discriminator compose a generative adversarial network [11]. Since it is hard to make an universal definition of communities, we use a discriminator to learn it from data. Considering the superior expressive power of graph neural networks (GNNs) [18, 33, 35], we use a GNN as the discriminator that predicts whether a community is real or produced by the generator. Given a seed node, the generator generates a community that cheats the discriminator by implicitly fitting characteristics of real communities. At the equilibrium, the discriminator can no longer distinguish between real and generated communities, and the generator produces high-quality communities that are similar to training ones. By learning what communities are from data, we solve the issue of diverse definitions.

To improve flexibility of community detection, we model the community generation by a novel sequential decision process [16, 19, 39] with rewards from the discriminator. In this case, the generator is able to generate communities with arbitrary structures. We incrementally construct a community by picking nodes one after another, i.e., seed expansion, using "heuristic rules". The heuristic rules are provided by a GNN and trained with reinforcement learning [30]. Specifically, we propose an innovative mechanism, the **G**raph **P**ointer **N**etwork with **i**ncremental updates (iGPN) as the generator, which is a highly efficient GNN specialized for sequential decision problems on graphs.

To deal with free-rider effects, we propose a novel locator mechanism that locates the seed nodes of generated communities. The task of the locator is analogous to detecting influential nodes in communities. We notice the novel dual structure between the influential nodes detection and the seed expansion: the former locates the seed of a community while the latter generates a community from a seed. Thus we form a dual learning task [13] between the generator and the locator (green rectangle in Figure 2). The locator provides regularization signals to the generator: if a generated community involves irrelevant nodes, i.e., the free-rider effect, the

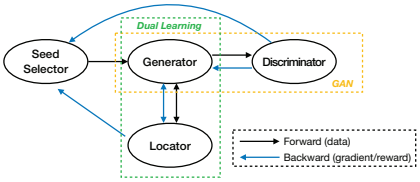


Figure 2: Relationships among 4 components of SEAL.

Table 1: Important Notations

Symbol	Definition
$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X})$	graph
$\mathbf{A}$	symmetrically normalized adjacent matrix
$C = \{u_0, \ldots, u_T\}$	community
$\mathcal{N}(u)$	neighbors of node u
$\mathcal{G}_C = (C, \mathcal{E}_C)$	sub-graph w.r.t C
$C_t = \{u_0, \ldots, u_t\}$	partial community at step t
$\partial C = \bigcup_{u \in C} \mathcal{N}(u) \backslash C$	outer boundary of a community
$\bar{C} = C \cup \partial C$	community with its boundary
$\mathbb{G}, \mathbb{D}, \mathbb{L}$	generator, discriminator and locator

locator would find it hard to locate the seed node and returns a low reward; on the contrary, the seed of a good community could be easily identified and the generator thus receives a high reward. On the other hand, the training target of the locator is to pick a node from which the generator could re-construct the given community.

Recall that inputs of the generator are seed nodes. A bad seed won't result in a good community, e.g., an isolated node in the worst case. So we also propose a seed node selector, which can effectively select promising nodes. The seed selector is basically a semi-supervised node regression model that predicts how good a node would be if it was used as the seed. By choosing the top-k nodes with the highest scores, we finally detect top-k communities from the network in a semi-supervised manner.

The contribution of this paper includes: (1) We propose SEAL, which learns heuristics for community detection from training data with generative adversarial networks. (2) We model the community generation as a sequential decision process and develop a specialized GNN, i.e., iGPN. (3) To deal with the free-rider effect we also propose a locator. The locator and the generator form a closed loop and get iterative improvements. We believe this is the first attempt that adopts dual learning into community detection. (4) A node selection method is proposed for providing promising seeds to the generator. (5) Experiments on 5 real-world datasets show the effectiveness of our proposal.

2 RELATED WORK

•Community Detection: Popular community detection methods can be classified into 3 groups. (1) Optimization based methods [5, 28] design community scoring functions like modularity and n-cut. Optimizing these functions reveals underlying communities. (2) Matrix factorization methods [20, 32] try to decompose adjacency matrices (and attribute matrices) to learn latent factors for communities. (3) Generative models [1, 36, 38] fit graphs and then infer communities. Note that these three categories have overlaps. For example, the baseline method for overlapping community detection, BigClam [36], and its variant on attributed graphs, CESNA [38], are based on matrix factorization and generative models.

Recently, with the development of graph representation learning, some deep learning based community detection methods have been proposed. For example, ComE [7] designed a framework that jointly optimizes community embedding, community detection and node embedding problems. CommunityGAN [15] extended the generative model of BigClam from edge level to motif level.

•Seed Expansion: Seed expansion tasks, also known as local community detection or community search problems aim at detecting a local community given the query (seed) node. A number of approaches [2, 37] define community scoring functions like conductance and cut-ratio. Another direction [10, 29] is to impose structure constraints, e.g., k-core, k-clique, k-truss etc., on output communities. Heuristics are often used to construct a local community which optimizes scoring functions and meets constraints. A comprehensive survey can be found in [10].

•Semi-supervised Community Detection: Semi-supervised community detection methods mainly refer to tasks with must/must-not link constraints [9, 40]. In [4], the authors proposed Bespoke, a method that uses size and structural information of communities from a training set to find communities in the rest of the network. They regarded their task as a "semi-supervised" community detection problem. It differs from previous work since it uses several complete communities as a training set instead of must/must-not link constraints. We follow this terminology in the paper. Another related work is the supervised community detection [8]. It is more like a permutation-invariant node classification problem than community detection and cannot be applied on large data since permutation enumeration is required.

•Graph Combinatorial Optimization with Deep (Reinforcement) Learning: Seed expansion can be formulated as a graph combinatorial optimization (GCO) problem, where we need to choose a subset of nodes to optimize the objective function, e.g., conductance, subject to some constraints e.g., k-core. Most GCO problems are NP-hard, and we heavily relied on heuristic algorithms to solve them [29]. Recently, the success of deep reinforcement learning (DRL) in video games [25] inspired researchers. S2V-DQN [16] is the first method focused on GCO with graph neural networks and deep Q-learning. S2V-DQN turns a GCO problem into a sequential decision problem where the solution is incrementally constructed. In [23], Graph Pointer Networks (GPNs) were proposed to solve TSP problems. However, a common issue of these methods is that the times of full forward/backward propagation through GNNs are proportional to sizes of solutions, which brings computational overhead and limits the input size. In [24], the authors proposed a two-stage method. It uses supervised learning to obtain node embeddings and then trains a Q-learning on node embeddings. As other supervised methods for GCO [21, 27], it requires labeled data.

3 METHODOLOGY

We describe our proposed method SEAL for learning heuristics for community detection. Important notations are summarized in Table 1. Specifically, boldfaced lowercase letters like $\mathbf{x}$ denote column

Algorithm 1 SEAL

Require: graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X})$, training set $\mathcal{D}$
1: Pre-train $\mathbb{G}$ on $\mathcal{D}$ by optimizing Equation 8
2: Pre-train $\mathbb{G}$ on $\mathcal{D}$ by optimizing Equation 7
3: **for** epoch = 1, ... **do**
4: **for** D_step = 1, ... **do**
5: Sample batch $\mathcal{B}_{\text{real}} \subset \mathcal{D}$ uniformly
6: Sample seeds $\mathcal{S} \subset \mathcal{V}$ uniformly
7: Generate communities $\mathcal{B}_{\text{fake}} = \{\mathbb{G}(s)|s \in \mathcal{S}\}$
8: Train $\mathbb{D}$ on $\mathcal{B} = \mathcal{B}_{\text{real}} \cup \mathcal{B}_{\text{fake}}$
9: **end for**
10: **for** L_step = 1, ... **do**
11: Sample batch $\mathcal{B}$ as line 5-8
12: Train $\mathbb{L}$ on $\mathcal{B}$ with policy gradient in Equation 9
13: **end for**
14: **for** G_step = 1, ... **do**
15: Sample batch $\mathcal{B}_{\text{real}} \subset \mathcal{D}$ uniformly
16: Teacher-force $\mathbb{G}$ on $\mathcal{B}_{\text{real}}$ by optimizing Equation 7
17: Sample seeds $\mathcal{S} \subset \mathcal{V}$ uniformly
18: Train $\mathbb{G}$ on $\mathcal{S}$ with policy gradient in Equation 6
19: **end for**
20: **end for**
21: Sample seeds $\mathcal{S} \subset \mathcal{V}$ uniformly
22: Fit $\{y(s) = r(\mathbb{G}(s))|s \in \mathcal{S}\}$ by semi-supervised node regression
23: Predict $\{\hat{y}(s)|s \in \mathcal{V}\}$ and select top-k seeds $\mathcal{S}_{\text{top-}k}$
24: **return** $\{\mathbb{G}(s)|s \in \mathcal{S}_{\text{top-}k}\}$

vectors, while uppercase letters like $\mathbf{X}$ denote matrices by stacking the corresponding vectors.

As shown in Figure 2, SEAL consists of 4 components. The generator takes seed nodes as inputs and generate communities. The discriminator is used to distinguish real communities from generated ones. Meanwhile, to eliminate free-rider effects, we introduce a locator that would guide the generator to generate compact and seed-aware communities. The generator and the locator form a closed loop corresponding to a dual learning task. Last but not least, a seed selector is utilized to provide promising seed nodes to the generator. In Section 3.1~3.4 we detail each component. The overall algorithm is shown in Algorithm 1.

3.1 Discriminator

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{X})$ be an undirected graph. $\mathbf{X} \in \Re^{|\mathcal{V}| \times d_0}$ is the feature matrix with $\mathbf{x}(u) \in \Re^{d_0}$ being the feature vector of node u. If the graph is non-attributed, we simply assume $\mathbf{x}(u) = 1,\ \forall u$. A community $\mathcal{G}_C$ is a connected subgraph of $\mathcal{G}$. To make expressions less cumbersome, we directly use the set of nodes C in the subgraph to denote a community.

The goal of the discriminator $\mathbb{D}$ is to determine whether a community $C \subset \mathcal{V}$ is real or generated, which naturally corresponds to a graph classification problem. We use a graph neural network (GNN) as the discriminator for its expressive power.

Note that if we only take a community C and its induced subgraph as inputs, how the community interacts with remaining nodes would be unclear. No matter how powerful the GNN is, only the internal connectivity of the community is modeled. In most cases, a community is not only determined by its internal structure, but also by connections between its members and external nodes in the network [37]. Inspired by popular community scoring functions [37], we would like to model both internal and external connectivities. So, we instead use $\bar{C} = C \cup \partial C$, where $\partial C = \{u \notin C | \exists v \in C \text{ s.t. } (u, v) \in \mathcal{E}\}$ is the outer boundary of C.

A L-layer GNN usually follows a layer-wise propagation schema [17, 18, 35]. The initial representations should denote whether a node is within the community or at the boundary:

$$\mathbf{z}^{(0)}(u) = \mathbf{W}_0 \mathbf{x}(u) + \mathbb{1}\{u \in C\} \cdot \mathbf{w}_1 + \mathbb{1}\{u \in \partial C\} \cdot \mathbf{w}_2,$$

where $\mathbf{W}_0 \in \Re^{d \times d_0}, \mathbf{w}_1, \mathbf{w}_2 \in \Re^d$ are parameters. $\mathbb{1}\{\text{condition}\}$ is the indicator function that returns 1 if the condition is true, otherwise 0. The representation of u at the l-th layer is obtained by aggregating representations of its neighbors. In this paper, we adopt the Graph Isomorphism Networks (GINs) [35]. Formally, the l-th layer of a GIN can be written as

$$\mathbf{m}^{(l)}(u) = \sigma\left(\sum\nolimits_{v \in \mathcal{N}_{\bar{C}}(u)} \mathbf{z}^{(l-1)}(v)\right), \quad \text{(AGGREGATE)}$$

$$\mathbf{z}^{(l)}(u) = \mathbf{W}_l \left(\mathbf{z}^{(l-1)}(u) + \mathbf{m}^{(l)}(u)\right), \quad \text{(COMBINE)}$$

where $\mathbf{m}^{(\cdot)}(u), \mathbf{z}^{(\cdot)}(u)$ are intermediate representation vectors of node u, and $\mathbf{W}_l \in \Re^{d \times d}$ is the weight matrix at the l-th layer. $\sigma(\cdot)$ is the activation function. $\mathcal{N}_{\bar{C}}(u)$ is the set of nodes adjacent to u within the given augmented community. The final representation of node u is given by the concatenation of representations at each layer: $\mathbf{z}(u) = [\mathbf{z}^{(0)}(u), \mathbf{z}^{(1)}(u), \ldots, \mathbf{z}^{(L)}(u)]$. To get a single representation of the entire subgraph, we also need a readout function to aggregate representations of all nodes: $\mathbf{z}(\bar{C}) = \sum_{u \in \bar{C}} \mathbf{z}(u)$. Finally, we finish our discriminator by using a sigmoid function to get the probability of C being real $\mathbb{D}(C; \mathcal{W}) = \Pr(C \text{ is real}) = \left[1 + \exp(\mathbf{w}_3^T \mathbf{z}(\bar{C}))\right]^{-1}$, where $\mathcal{W} = \{\mathbf{W_0}, \ldots, \mathbf{W_L}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{w_3}\}$ is the set of parameters.

We adopt GINs because of their choice of aggregator and readout functions. The SUM functions could capture the full multiset [35]. This is of importance for classifying a community since the number of nodes is helpful. For example, in a non-attributed graph, a one-node community would have the exactly same representation as a 3-clique community if we use the MEAN or MAX function. But clearly, a single node is not a good community.

3.2 Generator

Given a seed node u_0, the generator $\mathbb{G}$ tries to generate a local community C containing the seed: $\mathbb{G}(u_0) = C = \{u_0, u_1, \ldots, u_T\}$.

Inspired by seed expansion methods [2, 37], we propose a greedy community generation policy that incrementally constructs a solution. Different from previous methods where heuristics are handcrafted, we directly model seed expansion as a sequential decision process and learn heuristics from data.

Specifically, at the t-th step, given the current partial solution $C_{t-1} = \{u_0, u_1, \ldots, u_{t-1}\}$, we would pick a node u_t from the boundary ∂C_{t-1} and grow the community $C_t = C_{t-1} \cup \{u_t\}$. Following the terminology in the field of reinforcement learning, we use a

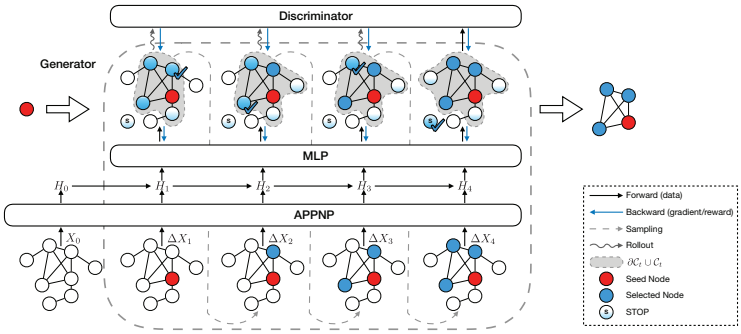


Figure 3: The iGPN and an example generation process.

policy $\mathbb{G}(a_t|u_0, C_{t-1}; Q)$[1] parameterized by Q to model the expansion. Here, $a_t \in A_t$ is the next action, and (u_0, C_{t-1}) is the current state. Note that the action space $A_t = \partial C_{t-1} \cup \{\text{STOP}\}$ varies over t, since we only consider nodes that are directly adjacent to nodes in the partial solution. Moreover, a special STOP action is added to notify the generator to stop the generation process. The state contains two terms. The first one u_0 is to denote which node this community centers at, while the second term C_{t-1} represents the current partial solution. For simplicity, we suppress the seed node u_0 in the remaining part of this paper and directly use $\mathbb{G}(a_t|C_{t-1})$.

3.2.1 Design of the Agent. Now, let us discuss how to implement the policy $\mathbb{G}(a_t|C_{t-1})$.

We augment node initial features at step t as follows:

$$\mathbf{h}_t^{(0)}(u) = \mathbf{Q}\mathbf{x}(u) + \mathbb{1}\{u = u_0\} \cdot \mathbf{q}_1 + \mathbb{1}\{u \in C_{t-1}\} \cdot \mathbf{q}_2, \quad (1)$$

where $\mathbf{Q} \in \Re^{d \times d_0}, \mathbf{q}_1, \mathbf{q}_2 \in \Re^d$ are parameters. In this way, we inject the knowledge about the partial solution as well as the seed node. The latter ensures that the generator is able to produce a community around the seed u_0.

Similarly, node features are processed by a GNN:

$$\tilde{\mathbf{H}}_t = \text{SOME\_GNN}\left(\mathbf{H}_t^{(0)}\right), \quad (2)$$

where $\tilde{\mathbf{H}}_t$ and $\mathbf{H}_t^{(\cdot)}$ are stacked node representations. When decide whether to stop, the model should take a global view. Thus, we aggregate the information on nodes with self-attention as the representation of STOP action:

$$\begin{aligned} \alpha_t(u) &= \frac{\exp(\mathbf{q}_3^T \tilde{\mathbf{h}}_t(u))}{\sum_{v \in \bar{C}_{t-1}} \exp(\mathbf{q}_3^T \tilde{\mathbf{h}}_t(v))}, \\ \tilde{\mathbf{h}}_t(\text{STOP}) &= \sum\nolimits_{u \in \bar{C}_{t-1}} \alpha_t(u) \cdot \tilde{\mathbf{h}}_t(u). \end{aligned} \quad (3)$$

Given the action representation $\tilde{\mathbf{h}}_t(a_t)$, we would like to add a new node to the partial solution or stop the generation process with the probability

$$\mathbb{G}(a_t|C_{t-1}) \propto \begin{cases} \exp\left(\mathbf{q}_4^T \tilde{\mathbf{h}}_t(a_t)\right) & \text{if } a_t \in \partial C_{t-1} \text{ or } a_t = \text{STOP}, \\ 0 & \text{otherwise}, \end{cases} \quad (4)$$

where $\mathbf{q}_4$ is a parameter.

[1]We use $\mathbb{G}(u_0)$ to denote a generated/sampled community, and $\mathbb{G}(a_t|u_0, C_{t-1})$ to denote a probability distribution over actions.

3.2.2 iGPN. Though we could use any GNN in Equation 2, at each time-step, a full forward/backward pass through the GNN is needed, which clearly brings computational bottleneck. So we propose the iGPN, **G**raph **P**ointer **N**etwork with **i**ncremental updates, specialized for sequential decision problems on graphs. iGPNs achieve more than 30x speedup (see Appendix) over conventional GNNs [12, 35] from two improvements. From the forward view, iGPNs can avoid full forward passes by doing incremental update at each step via decomposing node features in Equation 1. This avoids full forward passes. From the backward view, the neighbors aggregation step of iGPNs involves no parameters and thus is free of back-propagation.

The core building-block is parameter-free GNNs, e.g., SGC [33] and APPNP [18]. [33] observed that the superior performance of GNNs is mainly from the local aggregation rather than the non-linearity, and [18] separated the neural network from the information propagation on graphs. We define $\mathbf{X}_t = [\mathbf{X}, \mathbf{e}_{\{u_0\}}, \mathbf{e}_{C_{t-1}}]$, where $\mathbf{e}_{\mathcal{S}}$ is a binary vector with $\mathbf{e}_{\mathcal{S}}(u) = 1$ if $u \in \mathcal{S}$ otherwise 0. Then the APPNP takes the form:

$$\mathbf{H}_t^{(L)} = \text{APPNP}(f(\mathbf{X_t})) \Rightarrow \begin{cases} \mathbf{H}_t^{(0)} = f(\mathbf{X}_t) = \tilde{\mathbf{Q}}\mathbf{X}_t, \\ \mathbf{H}_t^{(l+1)} = (1-\beta)\mathbf{A}\mathbf{H}_t^{(l)} + \beta\mathbf{H}_t^{(0)}, \end{cases}$$

where $\mathbf{A}$ is the symmetrically normalized adjacent matrix. f could be any neural network, but we simply let f be a linear function with $\tilde{\mathbf{Q}} = [\mathbf{Q}, \mathbf{q}_1, \mathbf{q}_2]$ defined in Equation 1. If we have $\beta = 0$, the APPNP degenerates to a SGC. Here we use the default value $\beta = 0.85$.

Now we have $\mathbf{H}_t^{(L)} = \text{APPNP}(f(\mathbf{X}_t)) = \tilde{\mathbf{Q}} \cdot \text{APPNP}(\mathbf{X}_t)$ since $\text{APPNP}(\cdot)$ is a polynomial function. More importantly, we can have the following update rules:

$$\begin{aligned} \mathbf{X}_0 &= [\mathbf{X}, \mathbf{0}, \mathbf{0}], & \mathbf{H}_0 &= \text{APPNP}(\mathbf{X}_0), \\ \Delta\mathbf{X}_1 &= [\mathbf{O}, \mathbf{e}_{\{u_0\}}, \mathbf{e}_{\{u_0\}}], & & \\ \Delta\mathbf{X}_t &= [\mathbf{O}, \mathbf{0}, \mathbf{e}_{\{u_{t-1}\}}]\ (t > 1), & \Delta\mathbf{H}_t &= \text{APPNP}(\Delta\mathbf{X}_t) \\ \mathbf{H}_t &= \mathbf{H}_{t-1} + \Delta\mathbf{H}_t = \text{APPNP}(\mathbf{X}_t), & & \end{aligned} \quad (5)$$

where $\mathbf{O}$ and $\mathbf{0}$ represent matrices or vectors with all zeros. We decompose node features at step t by $\mathbf{X}_t = \mathbf{X}_{t-1} + \Delta\mathbf{X}_t$. $\Delta\mathbf{X}_t$ is a highly sparse matrix with only 1 or 2 non-zero elements. Since $\mathbf{A}$ is also sparse, the computation of $\Delta\mathbf{H}_t$ is very efficient. In this way, we obtain $\mathbf{H}_t$ at step t with an incremental update where little computation is needed. Moreover, $\mathbf{H}_0$ only needs to be calculated once and can be reused across multiple runs.

To enhance the model capability, we use a MLP to get the final representations. Recall that the policy $\mathbb{G}$ only involves nodes in $\bar{C}_{t-1}$, so we reduce the computation by only forwarding corresponding rows in $\mathbf{H}_t$ to the MLP:

$$\tilde{\mathbf{H}}_t(\bar{C}_{t-1}) = \text{MLP}(\mathbf{H}_t(\bar{C}_{t-1}); \mathcal{Q}'),$$

where $\mathcal{Q}'$ is the set of parameters in the MLP. Then we use Equations 3 and 4 to finalize the iGPN. The overall parameters in the generator is $\mathcal{Q} = \mathcal{Q}' \cup \{\mathbf{q}_3, \mathbf{q}_4\}$, where $\tilde{\mathbf{Q}} = [\mathbf{Q}, \mathbf{q}_1, \mathbf{q}_2]$ is absorbed in the MLP. We separate the MLP from neighbors aggregation step, i.e., Equation 5, so no gradients will pass through aggregation operation as illustrated in Figure 3.

3.2.3 Optimizing the Generator with policy gradient. Since sampling is involved, we train the generator via policy gradient methods. We define the reward for a completed community as $r(C) = -\log(1 - \mathbb{D}(C))$. There are no intermediate rewards. The objective of the generator is to maximize the expected reward given the seed, whose policy gradient [30] with respect to $\mathcal{Q}'$ is

$$\begin{aligned}\nabla J_{\mathbb{G}}(\mathcal{Q}'|u_0) &= \nabla \mathbb{E}_{C|u_0\sim\mathbb{G}}[r(C)] \\ &= \mathbb{E}_{u_1,\ldots,u_T|u_0\sim\mathbb{G}}\left[\sum_{t=1}^{T} \nabla \log \mathbb{G}(u_t|C_{t-1}) \cdot Q(C_{t-1}, u_t)\right],\end{aligned} \tag{6}$$

where $Q(C_{t-1}, u_t) = \mathbb{E}_{u_{t+1},\ldots,u_T|C_t\sim\mathbb{G}}[r(C)]$ is the state-action value function. We use Q's estimation to approximate the policy gradient. Specifically, we use the Monte-Carlo estimation as in SeqGAN [39]

$$\hat{Q}(C_{t-1}, u_t) = \begin{cases} \frac{1}{M}\sum_{i=1}^{M} r(C^{(i)}) & \text{if } t < T, \\ r(C_{t-1} \cup \{u_t\}) & \text{if } t = T, \end{cases}$$

where $C^{(i)}$ $(i = 1, \ldots, M)$ are rollouts sampled from the policy $\mathbb{G}$ given the partial solution $C_{t-1} \cup \{u_t\}$. It is well-known that we could use an action-independent baseline to reduce the variance. But we found it did not help due to the fluctuating rewards from the discriminator. This is also the reason why we didn't use Q-learning as used in S2V-DQN [16].

From Figure 3 we can see that at all but the last step, we sample several rollouts, i.e., complete communities, given the partial solution and get the estimated state-action values from the discriminator. In this way, we have tackled the sparse reward problem and distributed reward signals at all steps.

3.2.4 Pre-training and Teacher Forcing. Training procedures of GANs and reinforcement learning are notoriously brittle. It is common to use pre-training and teacher forcing with Maximum Log-Likelihood Estimation (MLE) [19, 39]. The pre-training procedure could provide a good initial agent, while the teacher forcing use an extra supervised learning step along with each reinforcement learning step to prevent the agent from deteriorating in some training batches and getting stuck.

The generator produces an unordered set but in an ordered manner. To compute the log-likelihood of a community, we need to maximize over all possible orderings [31]:

$$L(C) = \max_{\pi} \sum_{i=1}^{T} \log \mathbb{G}(u_{\pi(i)}|\{u_0, u_{\pi(1)}, \ldots, u_{\pi(i-1)}\}),$$

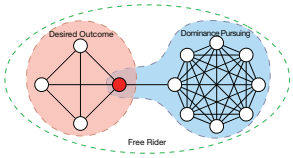
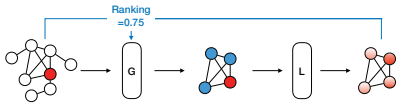
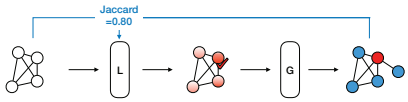


Figure 4: The free rider effect and the dominance pursuing effect.



(a) The generator receives rewards from the locator.



(b) The locator receives rewards from the generator.

Figure 5: The dual learning between the generator and the locator

where π is any permutation that meet the condition

$$u_{\pi(i)} \in \partial\{u_0, u_{\pi(1)}, \ldots, u_{\pi(i-1)}\},$$

i.e., a valid expansion. Considering there are $T!$ orderings in worst cases, we approximate it in a bootstrapped way with set2set [31]:

$$L^{\text{set}}(C) = \sum_{i=1}^{T} \log \mathbb{G}(u_{\pi^*(i)}|\{u_0, u_{\pi^*(1)}, \ldots, u_{\pi^*(i-1)}\}), \tag{7}$$

where $\pi^*(i) = \arg\max \mathbb{G}(\cdot|\{u_0, u_{\pi^*(1)}, \ldots, u_{\pi^*(i-1)}\})$, i.e., we decide training targets along with training procedure. We denote optimizing Equation 7 as the set-wise MLE. As pointed out by [31], if we naively optimize Equation 7, the model would pick a random ordering and gets stuck on it. To explore the space of orderings, a list-wise MLE is also necessary before training with set-wise MLE. Specifically, we sample a permutation π' in advance, and optimize the log-likelihood with respect to π'

$$L^{\text{list}}(C) = \sum_{i=1}^{T} \log \mathbb{G}(u_{\pi'(i)}|\{u_0, u_{\pi'(1)}, \ldots, u_{\pi'(i-1)}\}). \tag{8}$$

The set-wise MLE is used during pre-training and teacher forcing phase, e.g., line 2 and line 16 in Algorithm 1. The list-wise MLE is only used at the very beginning.

3.3 Locator

In this section, we propose anthor key component of SEAL, the locator. The locator needs to locate the seed node of a given community. If the seed of a generated community could be easily identified, the generator would receive an extra reward for generating a compact and seed-centered community. This saves the generator from annoying free-rider effects [34]. The task of the locator is analogous to detecting the most influential node in a community. We utilize the novel dual structure between influential nodes detection and

seed expansion by forming a closed loop between the generator and the locator as shown in Figure 2.

Let us understand free-rider effects with a small example shown in Figure 4. There are two communities in this example, one with 4 nodes forming a 4-clique and one forming an 8-clique. Two communities are connected via a pair of nodes. When we use the red node in the 4-clique as the seed, the desired outcome of the generator should be the 4-clique. However, we notice that sometimes the union of two communities is generated. This is called the free-rider effect [34]. A more undesirable situation is that the generator entirely neglects the 4-clique but tries to include the 8-clique. We call this phenomenon the dominance pursuing effect that the generator pursues a far but heavier community as it may bring a higher reward from the discriminator. We resort to a key observation that the seed node should be the most influential node within the generated community, i.e., we could easily predict which node is the seed given a community. So we utilize the locator to locate the seed node given a community. The locator and the generator are dual: given a community, a locator should produce a good seed node from which the raw community can be re-constructed by the generator; given a seed node, a generator should generate a community whose seed can be easily detected by the locator.

The locator has the same structure as the discriminator without the graph-level readout function. We skip redundant steps and assume we have obtained the node representation $\mathbf{z}(u)$. The probability of u being seed is given by a softmax classifier $\mathbb{L}(u|C;\mathcal{P}) = \frac{\exp \mathbf{p}^T \mathbf{z}(u)}{\sum_v \exp \mathbf{p}^T \mathbf{z}(v)}$, where $\mathcal{P}$ is the set of parameters in the locator. We augment the reward with the ranking of seed u_0:

$$r(C) = -\log(1-\mathbb{D}(C)) + \lambda_l \cdot \text{ranking}(u_0|C),$$

where $\text{ranking}(u|C) = \frac{1}{|C|}\sum_{v\in C} \mathbb{1}\{\mathbb{L}(u|C) \geq \mathbb{L}(v|C)\}$. We use the ranking instead of $\mathbb{L}(u_0|C)$ or $-\log(1-\mathbb{L}(u_0|C))$ as there may be multiple good seeds. For example, in Figure 4, the left-most three nodes are equally good and should have the highest reward, i.e., ranking=1. As show in Figure 5(a), the generator recives rewards from the locator.

The locator needs to produce a good seed node from which the raw community can be recovered by the generator. The locator recevies rewards from the generator as shown in Figure 5(b) and gets trained with policy gradient:

$$\begin{aligned}\nabla J_{\mathbb{L}}(\mathcal{P}|C) &= \nabla \mathbb{E}_{s|C\sim\mathbb{L},\hat{C}|s\sim\mathbb{G}}\left[\text{Jaccard}(C,\hat{C})\right] \\ &\approx \nabla \log \mathbb{L}(s|C)\cdot \text{Jaccard}(C,\hat{C}),\end{aligned} \tag{9}$$

where we use one sample $s|C \sim \mathbb{L}$ and $\hat{C}|s \sim \mathbb{G}$ to approximate the policy gradient. $\text{Jaccard}(C,\hat{C}) = \frac{|C\cap\hat{C}|}{|C\cup\hat{C}|}$ is the Jaccard similarity.

On the other hand, we would like to assume that a generated community should be around the seed node. So we can introduce the radius penalty and augment the reward function:

$$r(C) = -\log(1-\mathbb{D}(C)) + \lambda_l \cdot \text{ranking}(u_0|C) - \lambda_r \cdot \max_u D(u,u_0), \tag{10}$$

where $D(u,u_0)$ is the length of the shortest path from the seed node u_0 to u, and the $\lambda_r \geq 0$ is the hyper-parameter. We call $\max_{u\in C} D(u,u_0)$ the radius of the community C. The 4-clique in Figure 4 may be preferable now due to the smaller radius.

3.4 Seed Selection

To generate high-quality communities, it is crucial to provide promising seeds to the generator. A brute force approach is using $y(u) = \mathbb{E}_{C|u\sim\mathbb{G}}[r(C)] \approx r(\mathbb{G}(u))$ as node scores and select top-k nodes as well as communities. However, it would be expensive to sample communities and compute scores for every node. We instead sample a small number of nodes and compute their scores as the training set. Then we estimate scores of remaining nodes with semi-supervised node regression with GNNs. Specifically, we augment node features with the local degree profile [6] as follows:

$$\begin{aligned}\mathbf{x}'(u) = [&\mathbf{x}(u), degree(u), \max(DN(u)), \min(DN(u)),\\ &\text{mean}(DN(u)), \text{std}(DN(u)), 1],\end{aligned}$$

where $DN(u) = \{degree(v)|v \in \mathcal{N}(u)\}$. This step is very useful for non-attributed graphs. We process features using APPNP as in the generator, and train a MLP on the training set to get the prediction $\hat{y} = \text{MLP}(\text{APPNP}(\mathbf{X}'))$. Then, we select k nodes with highest $\hat{y}$ as seeds. As shown in Algorithm 1, we only use the node selector at the inference phase.

4 EXPERIMENTS

4.1 Evaluation Metrics

The most used evaluation metrics for community detection are the bi-matching F1 and Jaccard scores [4, 15, 36]. Given M generated communities $\{\hat{C}^{(i)}\}$ and N ground truth communities $\{C^{(i)}\}$, we compute scores in the following way:

$$\frac{1}{2}\left(\frac{1}{M}\sum\nolimits_i \max\nolimits_j \delta(\hat{C}^{(i)}, C^{(j)}) + \frac{1}{N}\sum\nolimits_j \max\nolimits_i \delta(\hat{C}^{(i)}, C^{(j)})\right), \tag{11}$$

where δ can be F1 or Jaccard function. However, it is not an appropriate metric for our setting. Our method, as well as Bespoke, could produce arbitrary number of communities. For example, if we enumerate all possible subsets of $\mathcal{V}$, we would obtain a score of at least 0.5 since the second term in Equation 11 is independent on M. Thus in this paper we only use the first term as evaluation metrics, i.e., $\frac{1}{M}\sum_i \max_j \delta(\hat{C}^{(i)}, C^{(j)})$. We also report results with Equation 11 as metrics in Appendix for reference.

4.2 Datasets and Comparing Methods

We use 5 real world networks containing overlapping communities from the SNAP[2], among which Facebook and Twitter are attributed networks. We pre-process datasets and show their statistics in Appendix. In each network, we use 450 communities (28 in Facebook) as the training set, 50 communities (2 in Facebook) as the validation set, and the rest as the test set.

We compare the following methods in the experiments: (1) BigClam [36] and (2) its assisted version BigClam-A, (3) CESNA [38] and (4) its assisted version CESNA-A, (5) ComE [7], (6) CommunityGAN [15], (7) Bespoke [4], (8) our proposal SEAL and (9) SEAL without attributes (SEAL w/o attr. for short). Methods (3), (4), (8) and (9) can deal with attributes. (7)-(9) are semi-supervised methods requiring training communities. We limit numbers of their outputs to 5000 communities (200 in Facebook). BigClam provides a way to choose the appropriate numbers of communities, which are used

[2] http://snap.stanford.edu/data/

Table 2: Experimental Results. The best method is boldfaced. The best baseline method is underlined.

	F1					Jaccard				
	DBLP	Amazon	Youtube	Facebook	Twitter	DBLP	Amazon	Youtube	Facebook	Twitter
BigClam	0.3466	0.5616	0.1461	0.3205	0.2654	0.2459	0.4436	0.0866	0.2088	0.1642
BigClam-A	0.3465	0.5852	0.1511	0.3232	0.2672	0.2463	0.4691	0.0903	0.2073	0.1655
CESNA	-	-	-	0.3237	0.2667	-	-	-	0.2300	0.1654
CESNA-A	-	-	-	0.3238	0.2689	-	-	-	0.2306	0.1668
ComE	0.2175	0.5207	0.0908	0.2782	0.1566	0.1348	0.4089	0.0713	0.1454	0.0887
CommunityGAN	-	0.5512	-	0.3209	-	-	0.4291	-	0.2087	-
Bespoke	0.5452	0.6044	0.2236	0.3225	0.2696	0.4993	0.4932	0.1503	0.2333	0.1674
SEAL	**0.6733**	**0.8872**	**0.2467**	**0.3289**	**0.2734**	**0.5733**	**0.8105**	**0.1633**	**0.2366**	**0.1702**
SEAL w/o attr.				0.3255	0.2730				0.2347	0.1699

by all unsupervised methods (1)-(6). Moreover, for methods (1)-(6), we filter detected communities who have more than 50% overlaps with communities in training/validation sets.

The data pre-processing steps and comparing methods are detailed in Appendix.

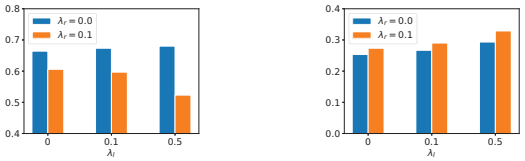
4.3 Experimental Results

We show the experimental results in Table 2. Clearly, our proposal outperforms all baseline methods, especially on DBLP and Amazon datasets. Bespoke also performs well, which proves the necessity of semi-supervised approaches to community detection. On the smallest dataset, Facebook, all methods have close results. ComE shows the worst performance across all datasets. This may result from clustering steps in the algorithm, which fails when faced with thousands of communities. CommunityGAN runs out of memory on DBLP, Youtube and Twitter since it maintains two dense copies of community membership matrices.

SEAL does not have a substantial advantage over SEAL w/o attr. on Facebook and Twitter. By comparing results of BigClam and CESNA we can conclude that attributes in these dataset only bring minor benefits. On the other hand, we find that attributes make the discriminator prone to overfitting. For example, there are 27201 attributes in Twitter, while only 450 communities are in the training set. The discriminator could easily remember these training communities by exploiting node attributes. Nonetheless, even SEAL w/o attr. outperforms all baseline methods. This prove the effectiveness of our framework.

4.4 Parameter Study

We study the importances of the two hyper-parameters involved in the reward function, i.e., the coefficient of the locator λ_l, and the coefficient of the radius penalty λ_r. In Figure 6, we show F1 scores on DBLP and Facebook. We can notice that the radius penalty on DBLP harms the performance, while on Facebook an opposite result is obtained. This means the low-radius assumption is not suitable for DBLP. We get benefits from the locator with $\lambda_l > 0$ in most situations, except DBLP with $\lambda_r = 0.1$ where the model may have been distorted by the strong radius penalty.



(a) DBLP **(b) Facebook**

Figure 6: Parameter Study

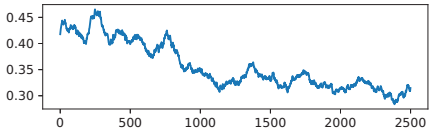


Figure 7: F1 scores of top-2500 communities. Consecutive 100 scores are averaged for a clear view.

4.5 Effectiveness of the Seed Selector

In most applications, we select top-k communities as outputs, e.g., the most suspicious groups of accounts in a trading network. So we need to test if SEAL can rank the communities generated. This can be reduced to testing if the seed selector could sort nodes according to their "potential".

We use a seed selector trained on Twitter to obtain top-2500 communities, and we compute F1 scores of each community without averaging, i.e., $\max_j \text{F1}(\hat{C}^{(i)}, C^{(j)})$ for $i = 1, \ldots, 2500$. We plot the scores in Figure 7. As i increases, the F1 score of $\hat{C}^{(i)}$ would generally decrease. Though curves on other datasets are not as clear as Twitter in Figure 7, we fit linear regression models on them and find that they all have negative slopes. This means that the node selector can successfully sort nodes such that a node with higher ranking is more likely to generate a good community. As a result, SEAL is able to rank communities during generation.

4.6 Targeted Community Detection

Recall that we motivate this work with an example trading network. We claim if only fraudulent communities are given, a semi-supervised model would detect fraudulent communities while leave normal ones alone. We support this claim by experiments.

We stack DBLP and Amazon networks by randomly adding 10,000 cross-network links to form a larger network with 127,273 nodes and 9,076 communities. Thus this network contains two types of communities. Now we use 450 communities from DBLP and train SEAL on this stacked network. As in previous experiments, we get 5000 communities from SEAL. By examining seeds of generated communities, we find that 4,525 (90.50%) communities are from DBLP network with F1 score of 0.6239 (averaged over multiple runs). This means that SEAL concentrated on DBLP-styled communities and filtered out communities in Amazon. If we train SEAL with 450 communities from Amazon, we can observe a similar result that 3623 (72.46%) communities are from Amazon with averaged F1 score of 0.8091. This shows that SEAL is able to target a particular group of communities. This also indirectly proves the effectiveness of the seed selector, which can find seeds of interested communities.

5 CONCLUSION

In this paper, we propose SEAL, which learns heuristics for community detection. Experiments on 5 real-world datasets prove the effectiveness of our proposal.

As for future work, we will explore data augmentation techniques for graph classification to prevent the discriminator from overfitting. Besides, the iGPN we proposed is a general GCO agent. We only use it to solve the seed expansion problem. We will consider other GCO problems to test if it is still effective and efficient. Since the locator aims at detecting seed nodes, it is also a potential work to use a trained locator to detect influential nodes in networks.

ACKNOWLEDGMENTS

This work is funded in part by Ant Financial through the Ant Financial Science Funds for Security Research, the National Natural Science Foundation of China Projects No. U1636207, No. U1936213, and NSF under grants III-1526499, III-1763325, III-1909323, and CNS-1930941.

REFERENCES

[1] Edoardo M Airoldi, David M Blei, Stephen E Fienberg, and Eric P Xing. 2008. Mixed membership stochastic blockmodels. *JMLR* 9, Sep (2008), 1981–2014.
[2] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local graph partitioning using pagerank vectors. In *focs*. 475–486.
[3] Reid Andersen and Kevin J Lang. 2006. Communities from seed sets. In *WWW*. 223–232.
[4] Arjun Bakshi, Srinivasan Parthasarathy, and Kannan Srinivasan. 2018. Semi-Supervised Community Detection Using Structure and Size. In *ICDM*. IEEE, 869–874.
[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *JSTAT* 2008, 10 (2008), P10008.
[6] Chen Cai and Yusu Wang. 2018. A simple yet effective baseline for non-attributed graph classification. In *ICLR Workshop*.
[7] Sandro Cavallari, Vincent W Zheng, Hongyun Cai, Kevin Chen-Chuan Chang, and Erik Cambria. 2017. Learning community embedding with community detection and node embedding on graphs. In *CIKM*. 377–386.
[8] Zhengdao Chen, Xiang Li, and Joan Bruna. 2017. Supervised community detection with line graph neural networks. In *ICLR*.
[9] Eric Eaton and Rachael Mansbach. 2012. A spin-glass model for semi-supervised community detection. In *AAAI*.
[10] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2019. A survey of community search over big graphs. *The VLDB Journal* (2019), 1–40.
[11] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *NeurIPS*. 2672–2680.
[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*. 1024–1034.
[13] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. In *NeurIPS*. 820–828.
[14] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *PVLDB* 10, 9 (2017), 949–960.
[15] Yuting Jia, Qinqin Zhang, Weinan Zhang, and Xinbing Wang. 2019. Communitygan: Community detection with generative adversarial nets. In *WWW*. 784–794.
[16] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *NeurIPS*. 6348–6358.
[17] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.
[18] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *ICLR*.
[19] Jiwei Li, Will Monroe, Tianlin Shi, Sébastien Jean, Alan Ritter, and Dan Jurafsky. 2017. Adversarial Learning for Neural Dialogue Generation. In *EMNLP*. 2157–2169.
[20] Ye Li, Chaofeng Sha, Xin Huang, and Yanchun Zhang. 2018. Community detection in attributed graphs: An embedding approach. In *AAAI*.
[21] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*. 539–548.
[22] Jun Ma, Danqing Zhang, Yun Wang, Yan Zhang, and Alexey Pozdnoukhov. 2018. GraphRAD: A Graph-based Risky Account Detection System. In *MLG*.
[23] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. 2019. Combinatorial Optimization by Graph Pointer Networks and Hierarchical Reinforcement Learning. *arXiv preprint arXiv:1911.04936* (2019).
[24] Akash Mittal, Anuj Dhawan, Sourav Medya, Sayan Ranu, and Ambuj Singh. 2020. Learning heuristics over large graphs via deep reinforcement learning. In *AAAI*.
[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
[26] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814–818.
[27] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. 2019. Learning to solve NP-complete problems: A graph neural network for decision TSP. In *AAAI*. 4731–4738.
[28] Jianbo Shi and Jitendra Malik. 2000. Normalized cuts and image segmentation. *PAMI* 22, 8 (2000), 888–905.
[29] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *KDD*. 939–948.
[30] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*. 1057–1063.
[31] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order matters: Sequence to sequence for sets. In *ICLR*.
[32] Xiao Wang, Di Jin, Xiaochun Cao, Liang Yang, and Weixiong Zhang. 2016. Semantic community identification in large attribute networks. In *AAAI*.
[33] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying Graph Convolutional Networks. In *ICML*. 6861–6871.
[34] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust local community detection: on free rider effect and its elimination. *PVLDB* 8, 7 (2015), 798–809.
[35] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks?. In *ICLR*.
[36] Jaewon Yang and Jure Leskovec. 2013. Overlapping community detection at scale: a nonnegative matrix factorization approach. In *WSDM*. 587–596.
[37] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Kais* 42, 1 (2015), 181–213.
[38] Jaewon Yang, Julian McAuley, and Jure Leskovec. 2013. Community detection in networks with node attributes. In *ICDM*. 1151–1156.
[39] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI*.
[40] Zhong-Yuan Zhang. 2013. Community structure detection in complex networks with partial background information. *EPL* 101, 4 (2013), 48005.

A REPRODUCIBILITY

The codes of SEAL and pre-processing scripts are available at https://github.com/yzhang1918/kdd2020seal. Processed datasets with train/validation/test split are also included.

We implement SEAL in PyTorch[3] and DGL[4]. Experiments were conducted on a single GeForce RTX 2070 with 8G memory.

A.1 Data Pre-processing

Table 3: Statistics of datasets. The first 4 columns denote the numbers of nodes, edges, communities, and attributes respectively. C_{Max} denotes the largest community size while C_{Avg} denotes the average community size.

	# N	# E	# C	# A	C_{Max}	C_{Avg}
DBLP	114,095	466,761	4,559	-	16	8.4
Amazon	13,178	33,767	4,517	-	30	9.3
Youtube	216,544	1,393,206	2,865	-	25	7.7
Facebook	3,622	72,964	130	317	72	15.6
Twitter	87,760	1,293,985	2,838	27,201	34	10.9

We do the following pre-processing:

(1) We neglect communities whose sizes are beyond the 90-th percentile. For example, in the DBLP, the largest community contains 7556 nodes, while the 90-th percentile is only 16. By doing so, we exclude outliers.

(2) For each dataset, we extract a subgraph that contains only the nodes in communities and their outer boundaries. This is because datasets are not fully labeled, which means that even though an algorithm produces a high-quality community, it would be assigned a low score. For example, in the Amazon dataset, there are 334,863 nodes in the raw network, but only 16,716 nodes are covered in the ground-truth communities.

A.2 Comparing methods

- BigClam [36]: This is a strong baseline for overlapping community detection. Following [4], we also consider an assisted version of BigClam which makes use of training communities. We denote it by BigClam-A.
- CESNA [38]: This is an extension of BigClam on attributed graph. It has an assisted version denoted by CESNA-A.
- ComE [7]: This is a embedding based method jointly solves node embedding, community embedding and community detection.
- CommunityGAN [15]: This is a recently proposed community detection method based on GAN. It extends the BigClam from edge-level to motif-level modeling.
- Bespoke [4]: This is a semi-supervised community detection method based on structure and size information.

[3]https://pytorch.org/
[4]https://www.dgl.ai/

Executable files for BigClam and CESNA are from SNAP[5]. Codes for ComE[6], CommunityGAN[7] and Bespoke[8] are provided by the authors. All experiments are repeated 5 times and the scores of the best 3 runs are averaged and reported.

ComE needs to compute covariance matrices for every community, which are too big to fit in memory. So we let covariance matrices be diagonal matrices to reduce the memory usage. CommunityGAN also suffers from high memory usage. CommunityGAN extends BigClam but the community membership matrix is dense in CommunityGAN due to implementation. So we reduce its memory usage by replacing its Adam optimizer with a SGD optimizer.

A.3 Implementation Details

In this section, we emphasize some details of the SEAL implementation. Hyper-parameters are summarized in Table 4.

Usage of Validation Sets: Validation sets are used for parameter searching and early-stopping. For each community C in the validation set, we uniformly sample one node s from it and then sample a community $\hat{C}|s \sim \mathbb{G}$. Jaccard$(C, \hat{C})$ across all validation communities are averaged as the indicator for parameter searching and early-stop. So test sets are not exposed until the very end.

Seed Selector: The seed selector is not involved during training and validation phases. The seed selector is trained with at most 500 epochs, where extra 20% samples are used for early-stopping. For SEAL and Bespoke, nodes in training/validation communities are not considered as seeds.

Feature Pre-processing: Note that in Table 4 the MLP in the seed selector has input size 64, which doesn't match attributes size in Facebook and Twitter. This is because we use PCA for dimensionality reduction to save GPU memory usage. We could view this as an extra untrainable linear layer before the MLP.

Episode Length: During training, we stop generating a community if it exceeds the maximum size of communities (in Table 3). This is useful at the beginning of the training when the model hasn't learned when to stop and may traverse the entire network.

B EXPERIMENTS

B.1 Supplementary Results to Table 2

In Section 4.1 we explained why bi-matching metrics are not suitable since Bespoke and SEAL could cheat by generating as many communities as possible. Here, we show supplementary results with bi-matching metrics in Table 5 for reference. Our proposal still achieves the best performance.

B.2 Efficiency of iGPNs

In this section, we show how efficient iGPNs are. We compare iGPNs with two GNNs: GIN [35], a representative of full neighbors aggregation methods and GraphSAGE [12], a representative of methods based on node sampling and aggregation. GIN and GraphSAGE are 3-layered and pluged into Equation 2. We use subgraphs as inputs of GIN to reduce memory usage. The number of samples per layer

[5]http://snap.stanford.edu/snap/index.html
[6]https://github.com/andompesta/ComE
[7]https://github.com/SamJia/CommunityGAN
[8]https://github.com/abaxi/bespoke-icdm18

Table 4: Hyper-parameters in SEAL

Hyper-parameter	Value
Batch size	32
Number of epochs for list-wise MLE pre-training	10
Number of epochs for set-wise MLE pre-training	25
Number of epochs	20
Number of steps of the inner $\mathbb{G}, \mathbb{D}, \mathbb{L}$ loops	5
Number of rollouts	5
Hidden dimension	64
MLP in the generator	64-64-64
MLP in the seed selector	64-64-64-1
Batch-norm	Before every linear layer
Residual connection	Before every linear layer
Dropout	Before every linear layer (except layers in $\mathbb{G}$)
Number of layers of GINs	3
Number of layers of APPNP	3
Activation function	Swish
Learning rate	1e-2
Optimizer	Adam
Number of seeds for training seed selector	500(Facebook), 5000(Others)
Number of epochs for training seed selector	500
Dropout rate (for attributes projecting layers)	Searched from $\{0, 0.2, 0.5, 0.8, 1.0\}$
Dropout rate (for other layers)	Searched from $\{0, 0.2, 0.5\}$
λ_r	Searched from $\{0, 0.1\}$
λ_l	Searched from $\{0, 0.1, 0.5\}$

Table 5: Experimental results with bi-matching metrics. The best method is boldfaced. The best baseline method is underlined.

	F1					Jaccard				
	DBLP	Amazon	Youtube	Facebook	Twitter	DBLP	Amazon	Youtube	Facebook	Twitter
BigClam	0.4041	0.5379	0.1264	0.3292	0.2433	0.2890	0.4527	0.0755	0.2347	0.1557
BigClam-A	0.4040	0.5226	0.1231	0.3280	0.2462	0.2896	0.4355	0.0728	0.2374	0.1579
CESNA	-	-	-	0.3374	0.2446	-	-	-	0.2473	0.1558
CESNA-A	-	-	-	0.3402	0.2473	-	-	-	0.2486	0.1587
ComE	0.2524	0.4823	0.0851	0.2792	0.1589	0.1573	0.3838	0.0542	0.1847	0.0899
CommunityGAN	-	0.5109	-	0.3205	-	-	0.4171	-	0.2104	-
Bespoke	0.3513	0.5550	0.2137	0.2820	0.2471	0.3170	0.4737	0.1472	0.2144	0.1651
SEAL	**0.5206**	**0.8035**	**0.2321**	**0.3402**	0.2491	**0.4939**	**0.7614**	**0.1480**	**0.2491**	**0.1683**
SEAL w/o attr.				0.3385	**0.2503**				0.2391	0.1646

is set to 4 for GraphSAGE. All three methods have the same number of parameters: 12,736 (BatchNorm excluded).

Averaged time required (in seconds) for generating communities with 20 nodes on DBLP and their peak GPU memory usage are shown below:

	iGPN	GIN	GraphSAGE
Forward Propagation	0.06 ± 0.01	2.33 ± 1.33	4.05 ± 2.23
Backward Propagation	0.01 ± 0.00	0.06 ± 0.02	0.03 ± 0.01
Peak Memory Usage	719 MB	5172 MB	741 MB

Note that gradients only pass through the MLP part in iGPNs. iGPN has a significant time advantage over GIN and GraphSAGE and has the lowest memory usage. This proves the efficiency of iGPNs. The reason why GIN is faster than GraphSAGE is that GraphSAGE needs to sample neighbors at every step, which slows down the entire process.