CommunityAF: An Example-based Community Search Method via Autoregressive Flow

Jiazun Chen
National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China
chenjiazun@stu.pku.edu.cn

Yikuan Xia
National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China
wfl00014@pku.edu.cn

Jun Gao
National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China
gaojun@pku.edu.cn

ABSTRACT

Example-based community search utilizes hidden patterns of given examples rather than explicit rules, reducing users' burden and enhancing flexibility. However, existing works face challenges such as low scalability, high training cost, and improper termination during the search. Aiming at tackling all these issues, this paper proposes a community search framework named CommunityAF with three well-designed components. The first is a GNN (graph neural network) component that combines community-aware structure features to incrementally learn node embeddings over a large graph for the other two components. The second is an autoregressive flow-based generation component designed for fast training and model stability. The third is a scoring component that evaluates the communities and provides scores for a stable termination. Moreover, to show that CommunityAF has the sufficient expressive power to cover the rules, we demonstrate that the scoring component with node features weighted by degree-related factors is able to mimic the existing structure-based community metrics. We introduce a square ranking loss to guide the training of the scoring component, and further devise a flexible termination strategy based on the inferred score change pattern over a sequence of candidate communities using beam search. We compare CommunityAF with four different categories of community search methods on six real-world datasets. The results illustrate that CommunityAF outperforms these community search methods, and achieves an average 15.3% improvement in effectiveness and 4x to 20x speedups on different datasets relative to the state-of-the-art generative method.

PVLDB Reference Format:
Jiazun Chen, Yikuan Xia, and Jun Gao . CommunityAF: An Example-based Community Search Method via Autoregressive Flow
. PVLDB, 16(10): 2565 - 2577, 2023.
doi:10.14778/3603581.3603595

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/JiazunChen/CommunityAF.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603595

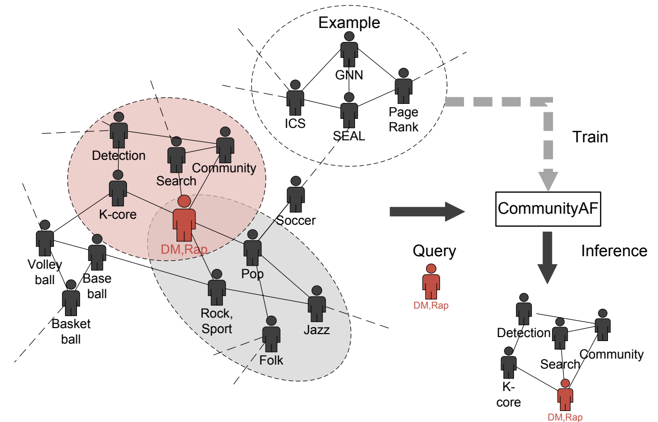

Figure 1: Example-based Community Search.

1 INTRODUCTION

Given a query node (seed) in an underlying graph, community search [11] discovers a subgraph that contains the query node and satisfies requirements in terms of structure and content features. Usually, the nodes in the discovered community share similar content features and cohesive structural relationships. Thus, the discovered community can support various tasks such as recommendation [47], personal background discovery [6], anomaly detection [48], etc. In contrast to community detection [25], which expects to identify all potential communities, community search [11] focuses on subgraphs around the query node, which greatly reduces computational overhead and improves the quality of discovered communities.

Community search faces one fundamental issue of how to express the community, despite significant progress achieved [2, 11, 16, 17, 25, 35]. Numerous communities exist in a graph with varying shapes, and end users search them for various purposes across multiple applications. It is difficult to flexibly find communities through explicit rules such as k-core [35], k-truss [16], k-clique [9], or implicit node scores such as personalized PageRank scores [1, 23]. Nonetheless, when end users find that the searched community cannot meet their specific requirements, they will be confused about whether there exist valid rules and parameters (such as k) in rule-based methods that suit their needs.

An example-based approach is promising to deal with the aforementioned key issue. That is, end users want to find some new communities via a limited number of examples, which guide the

following community search to achieve the goal of *what-you-see-is-what-you-get.* In addition, end users can easily refine the searched community if needed by adjusting the examples. Figure 1 demonstrates an example-based community search from a toy graph. An end user specifies a community example (white circle) whose topic is related to database research. The keywords below the nodes represent their content features. It is observed that there are two communities around the query node, an academic-related community (red circle) as well as a music-related community (gray circle). It is better to provide the end user with an academic community that is more similar to the community example.

Compared with the general community search methods, the example-based methods first need to capture the hidden patterns among the examples. With the advances in deep learning, it is natural and feasible to learn the hidden patterns [2, 19, 48] in terms of structural/content features of given examples, and the learned hidden patterns should be precise as there are a massive number of different candidate communities. At the same time, the example-based methods should support sufficient scalability, as the underlying graph may be large, while the deep learning methods are usually costly. Moreover, we expect the algorithm to be stable and avoid issues like mode collapse [13]. The termination of community generation is another key issue that seriously impacts the quality of discovered communities. The existing fixed stop criteria [12, 19] are obviously not flexible.

There are two types of learning approaches for example-based community search, classified according to the granularity of examples. ICS-GNN and QD-GNN [12, 19] belong to a class of works that handle node-level examples. They require two steps for community generation. First, a binary classification model is built to predict each node's probability of belonging to the community according to the given examples. Second, a rule-based method runs on nodes with probabilities to generate the community, with goals such as maximizing the average probabilities of selected nodes. Despite being efficiently implementable, node-level examples fail to provide patterns that are characteristic of the community. As an example, ICS-GNN has to assume that the community is fixed in size and highly connected in the second phase. These limitations damage the quality of searched communities.

The other class of approaches attempts to capture patterns directly from community-level examples. They usually recast the community search as a generative problem, including a task to produce the candidate community and a task to evaluate it. One of the representative works is SEAL [48]. SEAL learns the node representation using GNN and follows the GAN (Generative Adversarial Network) schema, which contains a generator producing the community node by node based on learned node embedding, and a discriminator to distinguish between the generated and given community examples. By training alternately, its generator is able to generate communities that are difficult for its discriminator to distinguish. In other words, the hidden pattern of the community examples can be captured precisely.

However, the GAN-based approach, such as SEAL, has limitations in the context of community search, as shown in Table 1. Firstly, many works show that GAN-based methods suffer from high training cost, model instability, and even mode collapse [4, 24, 32]. These issues prevent SEAL from fully learning about the example

Table 1: Comparison with example-based community search.

Characteristics	QD-GNN	SEAL	GraphAF	Ours
Community-Level	×	✓	✓	✓
Scalability	×	✓	×	✓
Fast Training	✓	×	✓	✓
Flexible Termination	×	×	×	✓

community patterns and cause it to fall into a single community pattern. Secondly, SEAL adds a virtual node into the graph, and stops generating groups once the virtual node is selected. The unsmooth generation process introduced by virtual nodes also affects the quality of the generated community.

We are interested in whether other kinds of generative models have the potential to meet all requirements, as the generative model can capture community-level patterns, but the GAN-based method has its inherent limitations. We find that autoregressive flows, another kind of generative model, have been studied in various tasks, including audio [29], image [28], and molecular graph generation [33]. The directly related work on graph fields, GraphAF [33], attempts to generate valid molecular graphs from scratch. It follows the autoregressive framework, and trains a series of invertible transformations between the probability density of the molecular graphs and a base distribution, which finally supports molecular graph generation by sampling within the base distribution. As shown in Table 1, GraphAF can be trained in an efficient and stable way. However, GraphAF only produces a small, entirely new graph, and does not consider the large underlying graph. In addition, GraphAF determines the termination of graph generation using chemical bond rules, which are not suitable either in the context of community search.

Inspired by existing works, this paper designs an example-based community search method aiming to meet all these requirements. The basic ideas of our method are as follows. For the issue of **scalability**, we mainly borrow and extend the idea of SEAL, and deliberately employ different strategies in node representation learning and generative model training. We choose an incremental GNN in the large underlying graph for processing node representations with community-aware structure features. The subsequent expensive community generation only handles very limited community-related data. For the issue of **fast training**, we choose the autoregressive flow model to locate communities in a large graph. The autoregressive flow model can be trained in parallel and is easier to converge compared with the GAN-based method [48]. For the issue of **flexible termination**, we plan to introduce a more fine-grained scoring task to measure the currently generated communities. Intuitively, the quality of the generated community should first increase and then decline as nodes are added. Such change patterns can be captured in a sequence of generated communities, and further help to design a flexible and smooth termination strategy.

The contributions can be summarized as follows: (1) This paper proposes a community generation framework named CommunityAF, aiming to tackle the above requirements. Specifically, a GNN component combines community-aware structure features to learn node embeddings over a large graph, which requires low computation resources and supports embedding updates incrementally. In

addition, an autoregressive flow-based generation (AF) component is designed to select the next node to join the current community, which enables fast parallel training and improves model stability compared with other generative models like GAN.

(2) For community evaluation, we design a scoring component over the learned node embedding in a multi-task way, and demonstrate that this component with degree-related factor weighted node features has sufficient expressiveness to mimic the existing structure-based community metrics. In addition, we introduce a square ranking loss to guide the training of the component for a more stable training process. We further devise a flexible termination strategy based on the score change pattern over a sequence of the candidates in beam search for community generation.

(3) We conduct experiments to demonstrate that CommunityAF outperforms existing rule-based and example-based approaches, and improves the training efficiency 4x-20x times compared with the GAN-based method. We also verify that CommunityAF can learn the distribution of different community patterns through visualization.

2 PRELIMINARIES

In this section, we review some preliminary knowledge about autoregressive flows and graph neural networks.

2.1 Autoregressive Flow

Normalizing flows (NFs) [24] are a family of generative models that differ from GANs [13] and variational autoencoders (VAEs) [20] in that they explicitly model the target distribution through a mechanism of invertible probability transformations.

Specifically, a normalizing flow [24] can convert a simple probability distribution $\mathcal{E}$ into a more complex distribution Z by a parameterized invertible deterministic transformation $f_\theta : \mathcal{E} \to Z$. For a real-world data z, its probability density function $p_Z(z)$ can be determined by a density function of $\epsilon \sim p_{\mathcal{E}}$, as well as the determinant of the inverse transformation f_θ^{-1}'s jacobian matrix with the help of the change-of-variable formula as follows:

$$p_Z(z) = p_{\mathcal{E}}\left(f_\theta^{-1}(z)\right)\left|\det \frac{\partial f_\theta^{-1}(z)}{\partial z}\right|. \tag{1}$$

We can maximize the log-likelihood of a given data point z in the training phase using Eq. 1. When it comes to generation, we can sample $\epsilon \sim \mathcal{E}$ and apply $z = f_\theta(\epsilon)$. In order to perform the above calculation efficiently, it is required that the jacobian determinant is easy to compute. Also, NFs use a sequence of transformations to push a basic probability density function to a more complex distribution, analogous to fluid flowing through tubes.

Autoregressive models (ARs) are another type of generative models, such as RNN [27]. The probability distribution of the current time step data is generated based on the previous observations. Additionally, the process of ARs typically requires the calculation of the first d-1 data before computing the d-th data, making the serial calculation process typically inefficient.

ARs exhibit a triangular jacobian matrix for their transformation function f_θ due to the autoregressive process. This property makes ARs as a layer of NFs [24]. The entire ARs can be reparameterized as affine autoregressive flows [28, 30, 33], where the conditional probability of each step can be modeled as follows:

$$p\left(z_d \mid z_{1:d-1}\right) = \mathcal{N}\left(z_d \mid \mu_d, (\alpha_d)^2\right), \tag{2}$$

where $z_{1:d-1}$ is the observation data, $\mu_d = g_\mu\left(z_{1:d-1}; \theta_d\right)$, $\alpha_d = g_\alpha\left(z_{1:d-1}; \theta_d\right)$ represent the mean and deviation of the normal distribution, respectively. In practice, these functions g_μ and g_α can be implemented as neural networks. The affine transformation of autoregressive flows can be written as:

$$\begin{aligned} f_\theta\left(\epsilon_d\right) &= z_d = \mu_d + \alpha_d \odot \epsilon_d; \\ f_\theta^{-1}\left(z_d\right) &= \epsilon_d = \frac{z_d - \mu_d}{\alpha_d}. \end{aligned} \tag{3}$$

In addition, during the training process, autoregressive flows allow g_μ (g_α) to compute $\mu_{1:d}$ ($\alpha_{1:d}$) in parallel [28]. They construct a fully-connected model with d inputs and d outputs and then can use masks to ensure that output i depends only on input 1:i-1, avoiding the inefficient iterative computation.

2.2 Graph Neural Network

Graph neural networks (GNNs) [14, 21, 49] encode node contents and structural relationships into low-dimensional representations, which are optimized by different training signals. Over the years, researchers have introduced ideas such as convolution [21] and self-attention [37] for designing the architectures of GNNs and proposed a variety of GNNs that differ in the methods of transforming nodes' features and aggregating features from neighbors. Most GNNs are composed of multiple layers, which can be expressed as:

$$H^i = Upd\left(H^{i-1}, Agg\left(A, H^{i-1}, \Theta_1^i\right), \Theta_0^i\right), \tag{4}$$

where A is the adjacent matrix, $\Theta_{1/0}^i$ are the layer-specific learned parameters, Agg collects neighbor messages based on different weights, and Upd is responsible for transforming nodes' current embeddings. The first layer's hidden feature H^0 is initialized by $H^0 = Upd(X, \Theta_0^0)$ based on the node content features X. Various loss functions provide the training signals for the models, depending on task scenarios. The models' parameters are optimized using strategies such as gradient descent to minimize the loss.

3 METHODOLOGY

In this section, we first present the framework of CommunityAF, then introduce its three components, and finally describe its efficient training algorithm.

3.1 Framework

Let $G = (V, E, X)$ denote an undirected graph, where V, E, X are the set of nodes, edges, and the feature matrix, respectively. $x(u) \in \mathbb{R}^d$ is the feature vector of node u. We directly represent a community $G_c = (C, E_c, X_c)$ using the set of nodes $C = \{q, u_1, \ldots, u_T\}$ in the subgraph for brevity. We use ∂C to denote the neighbors (boundaries) of the community, that is, the set of nodes that are connected to the community nodes but not part of the community.

CommunityAF includes three well-designed components to achieve the goals of the example-based community search, as shown in Figure 2. The community, initialized by the query node, is generated node-by-node. In each step, the incremental GNN component computes partially changed embeddings by fusing the structural

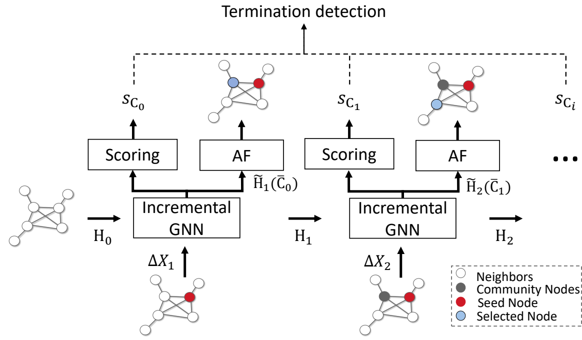

Figure 2: Interactions among 3 components of CommunityAF in community generation progress.

(including community-related features) and content features of the nodes, and updates the previous ones. The updated node embeddings about the current community are then fed into the AF component and the scoring component. The AF component aims to maximize the likelihood of the community examples, and is responsible for selecting the next-to-be-joined nodes during generation. The scoring component learns the community quality based on ranking and also comprehensively determines whether to terminate community generation based on quality changes.

3.2 Incremental GNN Component

The purpose of the incremental GNN component is to extract the node and community features needed for community generation and scoring tasks. Additionally, there is a need for scalability to support large-scale graph data. We take the similar GNN component as that in SEAL [48], because the challenges faced here are the same. Furthermore, we enhance features that are important to express the character of the communities and analyze the complexity of the incremental GNN component.

When performing the community generation task, incorporating information related to the query node as well as the nodes within the current community is crucial. For instance, if a node is present in the vicinity of the query node or all of its neighboring nodes are considered part of the community, it is probable that the node also belongs to the community. Thus, similar to SEAL [48], CommunityAF enhances the node features by indicating whether a node is a query node or belongs to the current community. Moreover, for each node, we introduce a feature whose aggregated result in GNN equals the node degree and another feature about the distance of nodes from the query node to estimate the community scale [42]. All these enhanced features are named community-aware structure features, which can be written as the following equation at step t:

$$\widehat{X}_t = [X, 1, e_{C_{t-1}}, e_{\{q\}}, id_{\{q\}}]. \tag{5}$$

Here, 1 is the newly-introduced all-1 vector to help calculate the node degree during aggregation. We define $e_{C_{t-1}}$ to denote whether a node belongs to the current community C_{t-1}. The binary vector $e_{\{q\}}$ indicate whether a node is the query q. $id_{\{q\}}$ is a vector that captures the node's distance to the query node q, where $id_{\{q\}}(u)$ equals the inverse of the shortest distance between the node u and the query q. We let 1 as the minimal shortest distance to prevent division by 0.

Community-aware features come at a cost, despite that they can be used to better reflect community structures and help community generation. For example, when a node is added into the community, the vectors $e_{C_{t-1}}$ should be correspondingly adjusted. The precise node embedding learning actually requires the GNN model to be run t times when there are t nodes in the community. The operation is obviously expensive, which seriously impacts scalability.

At the same time, node feature changes are very localized. When a node joins the community, only a small number of node embeddings will change. Therefore, efficient incremental computation is possible. We borrow iGPN from SEAL to provide a lightweight incremental update on the node embedding. iGPN inherits the idea of APPNP [22], which makes the aggregation phase completely parameter-free and linear by using fixed parameters as weights to aggregate neighbor messages. The aggregation process of the i-th layer iGPN can be written as follows:

$$H_t^i = (1-\beta)\widehat{A}H_t^{(i-1)} + \beta H_t^{(0)}, \tag{6}$$

where $\widehat{A}$ is the symmetrically normalized adjacent matrix, β is the damping factor, which can be set to 0.85, and $H_t^{(0)} = \widehat{X}_t$. Since $iGPN(.)$ performs a propagation of features $\widehat{X}_t$ along with the graph structural A, an incremental update rule, similar to SEAL [48], can be defined as:

$$\begin{aligned} H_0 &= iGPN(\widehat{X}_0) = iGPN([X, 1, 0, 0, 0]), \\ \Delta H_1 &= iGPN(\Delta\widehat{X}_1) = iGPN([O, 0, e_{\{q\}}, e_{\{q\}}, id_{\{q\}}]), \\ \Delta H_t &= iGPN(\Delta\widehat{X}_t) = iGPN([O, 0, e_{\{u_{t-1}\}}, 0, 0])(t > 1), \end{aligned} \tag{7}$$

where O and 0 represent matrices or vectors with all zeros. iGPN decomposes the node embeddings at step t by $H_t = H_{t-1} + \Delta H_t$ to avoid a full propagation for each generated node.

The incremental node embeddings are fed into *MLP* (Multilayer Perceptron) to maintain the flexibility of the final representations, as the previous steps are parameter-free. As only nodes in the community with its neighbors $\bar{C}_{t-1}$ are involved at step t, we perform MLP computation over the embedding of these nodes to obtain the final embeddings:

$$\widetilde{H}_t\left(\bar{C}_{t-1}\right) = \mathrm{MLP_g}(H_t(\bar{C}_{t-1})). \tag{8}$$

We analyze the complexity of plain and incremental GNNs. Let $G_c = (C, E_c, F_c)$ be a community example in an underlying graph $G = (V, E, X)$, e be the number of epochs in training, and the input as well as the output feature matrix dimensions be both d for calculation convenience. The time complexity for forward propagation of the plain GNN layer requires $O\left(e|C|\left(|E|d + |V|d^2\right)\right)$, where $|E|d$ and $|V|d^2$ are the complexity of neighbor aggregation and parameter transformation, respectively. The plain GNN requires the same time for backward propagation and an additional space complexity of $O(|V|d)$ to store the parameters and gradients. In contrast, the complexity of the incremental GNN is $O(|E|d + e|C||E|) + O(e|C|\bar{C}|d^2)$, in which the former part is the parameter-free incremental GNN layer, and the latter part is the subsequent MLP layer computation. We can see that the incremental GNN takes much less cost than the plain GNN because H_0

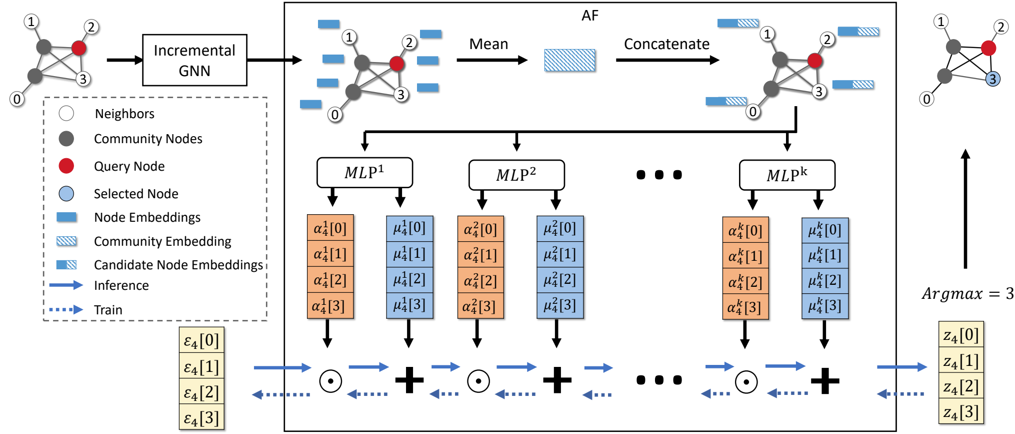

Figure 3: One step of community generation process in the autoregressive flow-based generation component.

needs to be computed only once and the aggregation process is parameter-free.

3.3 Autoregressive Flow-based Generation Component

The AF component is responsible for learning the conditional distribution of the example community, which supports the node selection in the community generation. The entire process is autoregressive, indicating that the data for the t-th step is generated based on the previous t-1 steps. Thus, there are two primary factors to consider, how to represent the data prior to step t, and how to select the next nodes. Figure 3 illustrates one generation step, in which the upper figure describes the representation of previous data, and the lower part shows the next node selection using the invertible transformation learned by a flow-based method.

The data representation prior to step t actually considers the currently generated community. With the embeddings of nodes in the community, we produce the community embedding by pooling the node embeddings:

$$\widetilde{h}_{C_{t-1}} = Mean(\widetilde{H}_t(C_{t-1})). \tag{9}$$

Mean readout function is selected as the pooling function in CommunityAF over *Sum* function [48], as we believe that the common features instead of the number of nodes are important for the representation of a community. Two pooling functions will be experimentally studied later.

We use the neighboring nodes of the current community ∂C_{t-1} as the candidate node set, in order to ensure connectivity. For each candidate node $u \in \partial C_{t-1}$, we concatenate its own embedding with the community embedding to get the final embedding of that node. Then we stack the final embedding of all candidate nodes as $\widehat{H}_{\partial C_{t-1}} \in \mathbb{R}^{d_{t-1} \times 2d_0}$:

$$\begin{aligned} \widehat{h}_t(u) &= Concatenate(\widetilde{h}_t(u), \widetilde{h}_{C_{t-1}}), \\ \widehat{H}_{\partial C_{t-1}} &= Stack([\widehat{h}_t(u^1), \ldots, \widehat{h}_t(u^{d_{t-1}})]), \end{aligned} \tag{10}$$

where $\partial C_{t-1} = \{u^1, \ldots, u^{d_{t-1}}\}$, and d_{t-1} is the number of neighbors of the current community.

We follow the flow-based model in selecting the next node by estimating the probability density function of candidate nodes. That is, for each candidate node, we can learn its probability density function in the form of a Gaussian distribution, from which the next nodes can be sampled. The conditional probability density function is defined as follows:

$$P(u|C_{t-1}) = \mathcal{N}(\mu_t, (\alpha_t)^2), \tag{11}$$

where $u \in \partial C_{t-1}$ is a candidate node from the current community neighbors, $\mu_t = g_\mu(\widehat{H}_{\partial C_{t-1}})$ and $\alpha_t = g_\alpha(\widehat{H}_{\partial C_{t-1}})$ are the mean and deviation of the standard normal distribution learned by g_μ and g_α using the current candidate data.

We further enhance the capability of invertible transformations using flow. Specifically, the transformation process is decomposed into k different modules, where each module uses *MLP* to compute the mean and variance, as illustrated in Figure 3. The training stage is the inverse of the generation stage, which we discuss in Section 3.5. The i-th ($1 \le i \le k$) transformation module is calculated as follows:

$$\begin{aligned} \mu_t^i / \alpha_t^i &= MLP^i_{\mu/\alpha}(\widehat{H}_{\partial C_{t-1}}), \\ z_t^i &= \alpha_t^i \odot z_t^{i-1} + \mu_t^i, \end{aligned} \tag{12}$$

where $z_t^i \in \mathbb{R}^{d_{t-1}}$ is a d_{t-1}-dimensional vector, and $z_t^0 = \epsilon_t$, which is randomly sampled from the standard normal distribution, and $\odot$ is the element-wise multiplication. In practice, the community node u_t is generated by taking the *argmax* of the generated vector z_t^k, i.e., $u_t = argmax_u(z_t^k)$.

3.4 Scoring Component

Flexible termination is crucial to the quality of community generation, as mentioned before. The existing termination strategies, such as the fixed stopping criteria [12, 19] and the virtual node policy [48], have limitations. For example, the fixed stopping criteria obviously cannot suit the cases when the community examples are arbitrary. The virtual node is introduced by considering the current community embedding and its neighbor embedding, and it serves

as the termination indicator when the virtual node is selected. However, the virtual nodes cannot leverage the trends of the community and may lead to early termination due to randomness.

CommunityAF plans to introduce a scoring component to produce scores for each generated community based on the similarity of the community examples to overcome the aforementioned limitations. Besides being a more fine-grained assessment of generated sequences, the scoring component can be used to capture the quality trend of the community. When a community evolves with node addition, the quality score rises before all the correct nodes are added and falls afterward. The quality changing pattern can be learned from the community examples, and then help to determine the termination in the community generation. In such a way, a more stable and smooth community generation can be supported. We also note that classic scoring functions based on subgraph structural properties such as density [42], conductance[1, 15] are extensively utilized to evaluate community quality. However, these predefined rule-based scores lack flexibility in handling different examples.

In the following, we first show how the scoring component is designed and then show that it can mimic classical scoring functions such as conductance if needed. Finally, we design a flexible termination strategy based on the score generated.

Score Computation. The scoring component evaluates the quality of the generated community, which is represented by the community embedding $\widetilde{h}_{\bar{C}_{t-1}}$ in the previous subsection. In addition, we introduce another self-attention based community embedding as follows, which not only considers different weights of nodes in aggregation, but also incorporates neighbor nodes outside the community. These two community embeddings are concatenated as the input to the scoring component:

$$\begin{aligned} a_t(u) &= \frac{\exp(\theta_a^T \widetilde{h}_t(u))}{\sum_{v \in \bar{C}_{t-1}} \exp\left(\theta_a^T \widetilde{h}_t(v)\right)}, \\ \ddot{h}_{\bar{C}_{t-1}} &= \sum_{u \in \bar{C}_{t-1}} a_t(u) \cdot \widetilde{h}_t(u), \\ h_{s_{\bar{C}_{t-1}}} &= stack([\ddot{h}_{\bar{C}_{t-1}}, \widetilde{h}_{C_{t-1}}]). \end{aligned} \tag{13}$$

For community C_{t-1}, its score can be written as follows:

$$s_{C_{t-1}} = [1 + \exp(MLP_s(h_{s_{\bar{C}_{t-1}}}))]^{-1}, \tag{14}$$

where $s_{C_{t-1}} \in [0, 1]$, and a larger $s_{C_{t-1}}$ means a higher quality of C_{t-1}.

Analysis of the Scoring Component. We argue that the scoring component can naturally combine the structural and content features to learn the patterns inside the examples, making it more expressive and flexible than existing rule-based metrics. In order to illustrate the power of the scoring component, we discuss the relationships between the scoring component and the rule-based structural properties such as conductance, which is the most commonly used rule-based metric [1, 8].

The previous work proves that the GNN with the labeling trick features can predict the density and cut ratio of subgraphs [40]. In fact, the community-aware structure features can also be regarded as an enhanced labeling trick. We consider the factor of node degree to weight community-aware features $h_u^{(0)}$ in Eq. 15. Similarly, we prove that there exists a score component that can precisely predict conductance using $h_u^{(0)}$. We show that $h_u^{(0)}$ is transformed using the incremental GNN into another form $\widetilde{h}_u$, which can be used to compute conductance. These computations can be fit by the neural network used by a score component.

$$h_u^{(0)} = \sqrt{|N(u)|} \begin{bmatrix} I(u \in C) \\ 1 \end{bmatrix}. \tag{15}$$

Theorem 1. *Given any graph G with its node features weighed using Eq. 15, there exists a scoring component that can precisely predict the conductance of any community C in G.*

Proof Sketch. According to the incremental GNN computation rule in Eq. 6, the node representation of $h_{v \in C}^{(1)}$ can be computed as:

$$\begin{aligned} h_{v \in C}^{(1)} &= (1-\beta) \sum_{u \in N(v)} \left(\frac{1}{\sqrt{|N(u)|}\sqrt{|N(v)|}} h_u^{(0)} \right) + \beta h_v^{(0)} \\ &= \begin{bmatrix} \frac{1-\beta}{\sqrt{|N(v)|}} |N(v) \cap C| + \beta \sqrt{|N(v)|} \\ \sqrt{|N(v)|} \end{bmatrix}. \end{aligned} \tag{16}$$

Subsequently, the intermediate node embedding is fed into the MLP_g in Eq. 8, which is used to fit the following equation:

$$\begin{aligned} \widetilde{h}_{v \in C} &= MLP_g(h_{v \in C}^{(1)}) \\ &= \begin{bmatrix} (h_{v \in C}^{(1)}[0] \cdot h_{v \in C}^{(1)}[1] - \beta h_{v \in C}^{(1)}[1]^2)/(1-\beta) \\ h_{v \in C}^{(1)}[1]^2 \end{bmatrix} \\ &= \begin{bmatrix} |N(v) \cap C| \\ |N(v)| \end{bmatrix}. \end{aligned} \tag{17}$$

Next, we obtained the average number of inner edges and the average number of node degrees of the community by mean pooling the node embeddings. The conductance can be precisely predicted by a scoring component that fits the following equation:

$$\begin{aligned} \widetilde{h}_C &= Mean(\widetilde{H}(C)) = \frac{\sum_{v \in C} \widetilde{h}_v}{|C|}, \\ cond_C &= 1 - 2 \sum_{v \in C} |N(v) \cap C| / \sum_{v \in C} |N(v)| \\ &= 1 - 2\widetilde{h}_C[0]/\widetilde{h}_C[1] \end{aligned} \tag{18}$$

■.

We should note that the proof is to illustrate the expressiveness of the scoring component, with the restrictions on the node features and computation steps. In practice, considering both the features of the query node and the graph content can make the scoring component more flexible. In our experimental test, we find that the node features with degree weighting do not always work well. It may be due to the fact that conductance is not a suitable measurement for some datasets. Thus, we uses the node features in Eq. 5 by default.

Usage of Scores in Termination. The scoring component enables CommunityAF to leverage the score changing of the community to determine the termination, which results in a stable and smooth generation. That is, given a community C_T, we maintain a m_s-size sliding window for an evolving community, and represent the scores of its generation process as $(s_{C_{T-1}}, \ldots, s_{C_{T-m_s}})$. The generation can be terminated when the condition $s_{C_T} < \min(s_{C_{T-1}}, \ldots, s_{C_{T-m_s}})$ is satisfied, and the community with the highest score $s_{argmax(S)}$

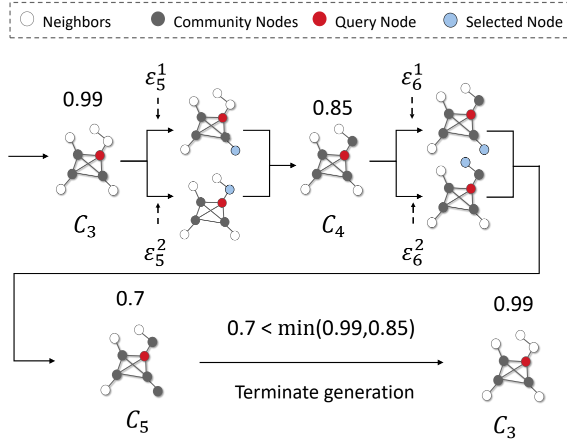

Figure 4: Example of score and its usage in stop generation.

is selected as the final community. By contrast, the virtual node strategy terminates generation once the virtual node is selected.

We can further expand the sliding window if we generate multiple candidate communities in each step. In fact, the next node is selected by sampling from the probability density distribution in CommunityAF. We can then sample m_e nodes $\{\epsilon^1, \ldots, \epsilon^{m_e}\}$, and each sampled node results in an independent community snapshot, which is similar to beam search in text generation [41]. We then generate scores for all candidate communities, and greedily select the community with the highest score as C_t for the next step, repeating until the generation stops.

We illustrate how our scoring component works in Figure 4 with $m_s = m_e = 2$. The current community C_3 has a score of 0.99, and we generate C_4^1 and C_4^2 by sampling ϵ_4^1 and ϵ_4^2 from a Gaussian distribution. Next, we select the community with the highest score as C_4. We repeat the procedure to generate C_5, and we notice that $s_{C_5} < min(s_{C_4}, s_{C_3})$. Therefore, we decide to stop generating this community, and the community with the highest score C_3 is selected as the final generated community.

3.5 Efficient Training Process

In this subsection, we first discuss how to parallel preprocess data for fast training. Then, we present the loss functions for the AF component and scoring component. Finally, we give the overall training algorithm and analyze its time complexity.

Parallel Preprocessing for Training. Inspired by the fact that autoregressive flows can compute the hidden representation $\widetilde{H}_{1:t}$ during the parallel training [33], we plan to support efficient training in CommunityAF. To accomplish this, we need to produce a node sequence to determine the next node to be selected. CommunityAF generates the node sequence for a given community by using breadth-first search, which is widely used in graph generation work [31, 33, 45]. Specifically, for an ordered community $C = \{u_0, \ldots, u_T\}$, we takes u_0 as the query node and obtains snapshots of the generation process, as $\{C_0 = \{u_0\}, \ldots, C_T = \{u_0, ..., u_T\}\}$.

With the node sequence, CommunityAF can learn the node embeddings incrementally. It is difficult to compute hidden representations for large graphs on a single GPU, even though the incremental

Algorithm 1: Efficient Training of CommunityAF.

Input: Graph $G = (V, E, X)$, Train set C_{train}, Batch size m_b.
1 Initial Parameters Θ of CommunityAF;
2 **while** Θ *is not converged* **do**
3 **for** $b = 1, \ldots, m_b$ **do**
4 Sample a community C from train set C_{train} with size $T + 1$;
5 Reorder C according to BFS as $\{u_0, u_1, \ldots, u_T\}$;
6 Select the current community $C_{list} = \{C_0, C_1, \ldots, C_T\}$ and the next generated node $Z = \{z_1, \ldots, z_T\}$ according to the BFS order;
7 Calculate the node embeddings $H = \{H_1, \ldots, H_T, H_{T+1}\}$ using Eq. 7;
8 **for** $t = 1, \ldots, T$ **do**
9 Calculate $\widehat{H}_{\partial C_{t-1}}$ using Eq. 8, 9, 10;
10 **for** $i = k, \ldots, 1$ **do**
11 $\mu_t^i = MLP_\mu^i(\widehat{H}_{\partial C_{t-1}})$, $\alpha_t^i = MLP_\alpha^i(\widehat{H}_{\partial C_{t-1}})$;
12 Calculate ϵ_t, $Loss_g^t$ according to Eq. 19, Eq. 21;
13 $Loss_g = \sum_{t=1,\ldots,T}(Loss_g^t)$;
14 Sampling m_{neg} negative nodes from ∂C_T as $C_{neg} = \{C_{neg}^1, \ldots, C_{neg}^{m_{neg}},\}$ and calculate the node embeddings using Eq. 7 ;
15 Initialize $R_p = \{(C_0, C_1), \ldots, (C_{T-1}, C_T)\}$;
16 $R = R_p + \{(C_{neg}^1, C_T), \ldots, (C_{neg}^{m_{neg}}, C_T)\}$;
17 Calculate the score for each community in $C_{list} \cup C_{neg}$ according to Eq. 8, 9, 13, 14;
18 Calculate $Loss_s$ according to Eq.23;
19 $Loss_m^b = Loss_g + \lambda Loss_s$;
20 Back propagate $Loss_m = \sum_b Loss_m^b$ and update Θ;

GNN model has reduced parameter and gradient memory usage. Thus, we decouple the underlying graph data from the subsequent training. Since the order of the nodes in the example community is given, we can efficiently process incremental embeddings $\Delta H_{1:t}$ for each snapshot in batches on CPU. Next, for each snapshot with the underlying graph embedding H, we can store the community embedding $H(C)$ and the candidate node embedding $H(\partial C)$ to the disk to avoid repeated computations during subsequent training. In order to facilitate subsequent generation in batches on GPU, we fix the boundary size of the community $\bar{C}$ to 200. CommunityAF takes a sampling strategy when the number of candidate neighbors exceeds this size limit, or uses zero vector padding otherwise.

Loss Functions for AF and Scoring Components. For the generation task, our goal is to maximize the log-likelihood of the training data. In CommunityAF, we set the conditional generation vector from training data as a one-hot vector z_t^k, which indicates the selected candidate node in this snapshot. The jacobian matrix of the inverse process $f^{-1} : Z \rightarrow \mathcal{E}$ is a triangular(diagonal) matrix, and its determinant can be calculated very efficiently:

$$\left| \det \frac{\partial f^{-1}(z_t^k)}{\partial z_t^k} \right| = \left| \prod_{i=1}^{k} \frac{1}{Mul(\alpha_t^i)} \right|. \tag{19}$$

Also, the inverse deduction of $\epsilon_t(z_t^0)$ is straightforward:

$$z_t^{i-1} = \left(z_t^i - \mu_t^i\right) \odot \frac{1}{\alpha_t^i}(1 \le i \le k). \tag{20}$$

So the optimization goal of a certain community generation snapshot is:

$$Loss_g = -\log(\text{Mul}(p_{\mathcal{E}}(\epsilon_t))) - \sum_{i=1}^{k} \log(\left|\text{Mul}(\frac{1}{\alpha_t^i})\right|), \tag{21}$$

where $\text{Mul}(v)$ represents the multiplication of all elements in a vector v, and $p_{\mathcal{E}}$ is the probability density function of the standard normal distribution. In the training phase, we compute α and μ according to the candidates' embeddings, calculate ϵ_t in the inverse direction, and finally calculate the required log-likelihood.

The scoring component is used to evaluate the generated community. Instead of modeling the absolute score labeled by end users or some rules, CommunityAF attempts to model the relative scores, which not only detect the quality change in community evolution but also require no extra labeling. That is, we use $R_p = \{(C_t, C_{t+1})\}$ to indicate the ranking pairs during the generation process of an example community C, where C_t is the previous snapshot of C_{t+1} in the generation, and the community score $s_{C_{t+1}}$ at step $t+1$ should be higher than the community score s_{C_t} at step t. In addition, we randomly add nodes in the neighborhood of C_T to form m_{neg} negative examples of community C_{neg}^i to introduce negative samples to help the component learn community scores better. The pairs for each negative example are added into the training pairs $R = R_p + \{(C_{neg}^i, C_T)\}$. Note that the embedding of these negative communities can also be computed in the preprocessing stage.

The plain loss function measures the relative score for given two communities, where $0 \le m \le 1$ is the margin to allow a tolerance of the error in the ranking:

$$Loss = \sum_{(C_i, C_{i+1}) \in R} \max(0, s_{C_i} - s_{C_{i+1}} + m). \tag{22}$$

However, the plain loss function may not work properly in some community ranking scenarios. For example, suppose that we expect $s_{C_{i+1}} - s_{C_i} > m$ as well as $s_{C_i} - s_{C_{i-1}} > m$. The sum operation results in $s_{C_{i+1}} - s_{C_{i-1}} > 2m$, leaving the score of community s_{C_i} untrained. One solution is to use listwise ranking loss [5], but it requires artificially setting the ground truth community score for each stage in advance, while the community is flexible.

We further propose a square pairwise ranking loss to overcome the limitation of the plain loss function. Due to the square operation, we can see that a positive term in a pair cannot be canceled by the corresponding negative term in another par, making the scores of all communities fully learned:

$$Loss_s = \sum_{(C_i, C_{i+1}) \in R} \max\left(0, ((s_{C_i} - s_{C_{i+1}} + 1)^2 - (1-m)^2)\right). \tag{23}$$

Overall Algorithm and Complexity Analysis. The overall training of CommunityAF is shown in Algorithm 1. The training is conducted in multiple epochs until the model is fully trained. In each epoch, the patterns in m_b example communities are learned. That is, for a sampled example community, we generate one node sequence in line 5, and prepare the training data, including community embedding and candidate node embedding, in lines 6–7. Then, we compute the loss in each generation step to produce the generative loss in the AF component. The ranking loss in the scoring component is computed in line 18. The overall loss function in line 19 takes the form of $Loss_m = Loss_g + \lambda Loss_s$, where λ is a hyperparameter to adjust the weights of two losses.

We analyze the time complexity of Algorithm 1. Let $G_c = (C, E_c, F_c)$ be a community example in an underlying graph $G = (V, E, X)$, e be the number of epochs in training. The complexity of the incremental computation in lines 5-7 and 14 is $O(e|C||E|)$. The preprocessing can reduce the complexity of this part to $O(|E|)$, because it can compute the incremental part in parallel, and avoid repeated computations. The rest of the calculations are only related to the size of the community and its neighbors, independent of the size of the underlying graph.

Table 2: Statistics of datasets.

	$\|V\|$	$\|E\|$	$\|C\|$	$Max(C)$	$Mean(C)$
Facebook	3K	72K	130	72	15.6
Amazon	13K	33K	4517	30	9.3
DBLP	114K	466K	4559	16	8.4
Twitter	87K	1M	2838	34	10.9
Youtube	216K	1M	2865	25	7.7
LiveJournal	316K	5M	4510	54	17.6

4 EXPERIMENTS

In this section, we first introduce our experimental setup and comparison methods. Then, we validate the effectiveness and efficiency of CommunityAF, and perform the ablation experiment and hyperparameter study to demonstrate the rationality of the CommunityAF design. Finally, we design an experiment to demonstrate the ability of CommunityAF to learn the distribution of different community patterns, and validate it through visualization. Due to the page limitation, the hyperparameter settings and experiment environment can be found in our code repository.

4.1 Experiment Setup

Metrics. We choose the Bi-matching $F1$ ($\mathbb{BI}$) metric [2, 48] commonly used in community-related fields to evaluate method performance. Specifically, for a dataset with n real communities $\{C^i\}$, we split it into a train set, a validation set, and a test set. For each community C in the validation set and test set, we uniformly sample one node q from it as a query node to generate the community. Finally, we obtain m generated communities $\{\widehat{C}^i\}$. The evaluation metrics can be calculated as:

$$\frac{1}{2}(\frac{1}{m}\sum_i \max_j F1(\widehat{C}^{(i)}, C^{(j)}) + \frac{1}{n}\sum_j \max_i F1(\widehat{C}^{(i)}, C^{(j)})), \tag{24}$$

where the forward matching $F1$ ($\mathbb{F} = \frac{1}{m}\sum_i \max_j F1(\widehat{C}^{(i)}, C^{(j)})$) indicates that how the generated community is alike one of the existing ground-truth communities, while the backward matching $F1$ ($\mathbb{B} = \frac{1}{n}\sum_j \max_i F1(\widehat{C}^{(i)}, C^{(j)})$) shows that how the ground-truth community is alike one of the generated communities.

Datasets. We use 6 real-life graphs in our experiments, which are available from the snap project [43]. Among them, Facebook and

Twitter have both content features and ground-truth communities, while the rest of the datasets have only ground-truth communities. For non-attributed graphs, we assume that each node has the same attributes, i.e. $x(u) = 0$. The statistics of the datasets are shown in Table2. For each dataset, we select 450 communities (28 from Facebook) as the train set, 50 communities (2 from Facebook) as the validation set, and the rest as the test set. The example-based models will be trained on the train set and tuned for the hyperparameters on the validation set. Eventually, all methods will be evaluated on the test set.

Preprocessing. For each dataset, we follow the same preprocessing strategies as the previous work [48] to facilitate training and fair comparison. The data preprocessing strategies include the removal of outliers, the removal of irrelevant nodes that are not community nodes or neighbors, and feature reduction, which are presented in detail in our code repository. Additionally, we perform a study on the datasets that contain these irrelevant nodes.

4.2 Competitors

To the best of our knowledge, example-based community search is a relatively new field with only a few related works. Therefore, we further select some representative general community search work for comparison.

Local modularity-based methods. Mod-m [8] defines local modularity to measure community quality based on the proportion of boundary edges in the community, and stops according to a threshold. Mod-r [26] proposes a new metric for the local modularity of communities considering subgraphs.

Random walk-based methods. HK (Heat Kernel) [23] finds communities using random-walk with restart probabilities to sort the nodes and utilizes conductance to determine the termination of community generation. LOSP [15] introduces a local spectral subspace and seeks out a sparse indicator vector in the subspace to identify the community that contains the given seed. LLSA [34] attempts to first sample communities using HK algorithm, and then optimizes these communities using Lanczos method. MRW [3] relies on multiple walkers to capture the local community structure and allows walkers with similar visit histories to reinforce each other.

Structural cohesiveness-based methods. CTC [18] tries to find connected k-truss subgraphs that contain the query nodes with the smallest diameter. SCS [44] aims to locate a community whose minimum node degree is the largest among all candidates.

Example-based methods. We select SEAL, a representative search method using community-level examples, as our major competitor. SEAL can additionally train a seed selector extension to the community detection problem, but in our context, the seed is given. In addition, we extend three search methods using node-level examples into our context, ICS-GNN [12], ICS$^+$-GNN [7], QD-GNN [19] using node-level examples. Originally, ICS-GNN [12] and ICS$^+$-GNN [7] learn a classifier to determine whether a node belongs to a community. Similar to QD-GNN, their classifier can be trained on nodes in all trained communities, which can capture the patterns among multiple communities to some extent. Then all these methods search for the communities using heuristic rules. In detail, ICS$^+$-GNN additionally introduces an unsupervised clustering task in the training phase that enhances node embeddings' capability of capturing some community-level features. QD-GNN introduces local query dependency structures and global graph embeddings, which enable the node embedding to obtain sufficient information about the query node and the underlying graph. Considering that QD-GNN focuses on community search on small graphs, we add the process of subgraph sampling, which is similar to that in ICS-GNN.

Variations of CommunityAF. We use a limited train set, e.g., 5 communities for training, in order to validate whether CommunityAF works well with limited examples, which called CommunityAF-F. CommunityAF-A means that we use *Sum* for the pooling function in Eq. 9 to get the community representation. CommunityAF-V represents using the virtual node strategy instead of our scoring component. Specifically, we add the input embedding of the scoring component $h_{s_{\hat{C}_{t-1}}}$ as the virtual node embedding to the candidate node embeddings, and stop the generation when the virtual node is selected. CommunityAF-C stops community generation with conductance. CommunityAF-D uses a degree-related factor to weight the specific community-aware features in Eq.15 to facilitate precisely predicting community structural properties. CommunityAF-P uses the plain ranking loss for training in Eq. 22.

4.3 Comparison with Competitors

In this subsection, we compare the effectiveness and efficiency of CommunityAF with its competitors on different datasets.

Effectiveness on preprocessed datasets. Table 3 shows that CommunityAF is superior to the majority of methods on all datasets, and significantly outperforms SEAL on five datasets. The rule-based methods perform differently on various datasets. For example, the local modularity-based methods work well on Facebook and Amazon, the random walk-based methods only do well on Amazon, and the structural cohesiveness-based methods are adept at the DBLP, Amazon, Twitter, and LiveJournal datasets.

Compared with the node-level example-based methods, CommunityAF achieves better performance. These two-stage methods can learn patterns on community nodes, but are difficult to capture at the community-level patterns, and they stop community generation using fixed criteria, which impacts the quality of the generated community seriously.

CommunityAF outperforms SEAL, the major competitor, on most datasets, with 15.3% average improvement in performance. We also note that SEAL achieves slightly better results than CommunityAF on Amazon. We guess that the community patterns on Amazon are relatively simple and homogeneous, with clear structural characteristics, because almost all approaches perform well on it. The backward $F1$ of CommunityAF is lower than that of SEAL, probably because CommunityAF is overfitted on this dataset.

CommunityAF has higher scores in the $\mathbb{F}$ and lower scores in the $\mathbb{B}$, and we can note that similar results can be found in other example-based methods. It is due to the difference in patterns between the training and testing communities. Usually, it is relatively easy to find a test community C that shares a similar pattern to communities in the train set, and then the generated community is similar to C, resulting in a higher $\mathbb{F}$. However, there may exist a

Table 3: Results of experiments with competitors. The best result is boldfaced.

	Facebook			DBLP			Amazon			Twitter			Youtube			LiveJournal		
	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$	$\mathbb{F}$	$\mathbb{B}$	$\mathbb{BI}$
Mod-m	0.511	0.365	0.438	0.630	0.573	0.602	0.816	0.807	0.811	0.354	0.260	0.307	0.320	0.271	0.296	0.696	0.651	0.673
Mod-r	0.528	0.374	0.451	0.648	0.587	0.618	0.845	0.835	0.840	0.363	0.270	0.316	0.353	0.299	0.326	0.724	0.668	0.696
HK	0.387	0.196	0.292	0.489	0.430	0.459	0.837	0.779	0.808	0.303	0.189	0.246	0.284	0.201	0.243	0.623	0.544	0.584
LLSA	0.508	0.346	0.427	0.570	0.517	0.544	0.677	0.662	0.670	0.360	0.288	0.324	0.270	0.232	0.251	0.675	0.624	0.650
LOSP	0.496	0.391	0.444	0.639	0.586	0.613	0.757	0.748	0.753	0.402	0.342	0.372	0.373	0.331	0.352	0.634	0.599	0.616
MRW	0.459	0.239	0.349	0.509	0.450	0.479	0.876	0.845	0.860	0.310	0.193	0.251	0.443	0.395	0.419	0.683	0.594	0.639
CTC	0.451	0.353	0.402	0.682	0.712	0.697	0.795	0.784	0.790	0.404	0.328	0.366	0.420	**0.422**	0.421	0.701	0.680	0.691
SCS	0.473	0.333	0.403	0.686	**0.715**	0.700	0.869	0.856	0.862	0.428	0.310	0.369	0.350	0.313	0.331	0.716	0.622	0.669
ICS-GNN	0.430	0.288	0.359	0.606	0.566	0.586	0.813	0.799	0.806	0.400	0.292	0.346	0.390	0.343	0.367	0.578	0.532	0.555
ICS^{+}-GNN	0.475	0.286	0.381	0.704	0.687	0.696	0.834	0.819	0.826	0.397	0.343	0.370	0.455	0.396	0.426	0.587	0.548	0.568
QD-GNN	0.385	0.317	0.351	0.715	0.675	0.701	0.877	0.893	0.885	0.334	0.287	0.311	0.433	0.378	0.405	0.689	0.654	0.672
SEAL	0.414	0.327	0.370	0.694	0.568	0.631	**0.916**	**0.920**	**0.918**	0.359	0.260	0.310	0.431	0.304	0.368	0.692	0.661	0.677
CommunityAF	**0.539**	**0.395**	**0.467**	**0.749**	0.687	**0.718**	0.914	0.905	0.910	**0.440**	**0.345**	**0.393**	**0.470**	0.406	**0.438**	**0.742**	**0.702**	**0.722**

community in the test set with distinct patterns that are difficult to learn in the train set, and a lower $\mathbb{B}$ occurs.

Table 4: Bi-matching F1 results of the example-based methods varying raw datasets.

Datasets	SEAL	QD-GNN	ICS^{+}-GNN	CommunityAF
Facebook	0.355	0.345	0.369	**0.461**
DBLP	0.629	0.659	0.651	**0.688**
Amazon	**0.827**	0.782	0.743	0.793
Twitter	0.364	0.371	0.382	**0.390**
Youtube	0.346	0.369	0.365	**0.381**
LiveJournal	0.632	0.599	0.446	**0.684**

Effectiveness on raw datasets. Table 4 reports the results of example-based community search methods on raw data. The raw data, compared with the fully preprocessed data, keep all irrelevant nodes in the graph. These irrelevant nodes are not community nodes or community neighbors, which may be noise in the community generation. The experiments show that all methods are slightly degraded, but CommunityAF still performs the best overall.

Table 5: Comparison of training costs (seconds) varying datasets.

	SEAL		CommunityAF	
	pretraining	trainning	preprocessing	trainning
Facebook	30	2507	**3**	**121**
DBLP	412	2565	**42**	**663**
Amazon	254	2266	**3**	**322**
Twitter	290	2248	**11**	**212**
Youtube	1851	14431	**1446**	**548**
LiveJournal	2654	31844	**1805**	**1836**

Efficiency. A major overhead of deep learning is the training time. We test the time costs in seconds of two generative methods, CommunityAF and SEAL, on six varying-sized graphs, as shown in Table 5. From this table, we can see that CommunityAF only accounts for less than 4% of the SEAL's training time in the best case. Even if we consider the one-time preprocessing and pretraining time, CommunityAF still achieves 4x to 20x speedups on different datasets. As previously analyzed, low training time in CommunityAF comes from parallel training and easy convergence in the autoregressive flow-based method. Here, we do not report the time used by the node-level example-based methods, as they do not capture the community-level pattern and therefore are faster naturally.

Table 6: Ablation experiment with Bi-matching F1.

	Facebook	DBLP	Amazon	Twitter	Youtube
CommunityAF-A	0.372	0.496	0.888	0.314	0.362
CommunityAF-V	0.417	0.662	0.808	0.382	0.321
CommunityAF-C	0.456	0.626	0.860	0.382	0.393
CommunityAF-F	0.438	0.610	0.863	0.366	0.411
CommunityAF-D	**0.467**	**0.722**	0.876	0.383	0.422
CommunityAF-P	0.463	0.710	0.909	0.382	0.422
CommunityAF	**0.467**	0.718	**0.910**	**0.393**	**0.438**

4.4 Ablation Experiment

We test the results of different variants of CommunityAF in Table 6. The results of CommunityAF-A using *Sum* as its pooling function verify our previous expectations that the averaged community features are more suitable to determine the next node in the community. Moreover, the performances of CommunityAF-V with the virtual node strategy and CommunityAF-C with conductance indicate that the scoring component is more robust and capable of adapting to different datasets. The result of CommunityAF-F shows that providing examples is not a large burden for users, since only a small number of examples can outperform most rule-based methods. CommunityAF-D uses a degree-related weighting to precisely calculate community properties, achieving optimal results on two datasets. We think it is possible that the conductance or similar subgraph structural properties of the metrics are not applicable to other datasets. The results of CommunityAF-P with the plain ranking loss verify that the square ranking loss is more stable.

4.5 Hyperparameters Study

Study of the AF Component. We study the effect of the number of transformation module k in the AF component on two representative datasets, as shown in Figure 5, respectively. Obviously, the number of layers k in the autoregressive flow-based generation

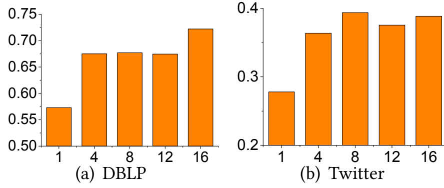

Figure 5: Bi-matching F1 varying k.

component is related to the complexity of the distribution of different community patterns. The increase of k can enlarge the model capability, and initially improve its performance. However, when k is too large, the performance degrades due to issues like overfitting.

Study of the Scoring Component. We study the effects of m_s and m_e in the scoring component in Figure 6, where m_s is the length of the sliding window, and m_e is the number of samples in each step. The community quality increases when m_s increases, indicating that a large sliding window can avoid the previous improper early stop. However, when m_s is too high, the generation is difficult to stop, and the size of the generated communities increases, which has a negative impact. Figures 6(b) and 6(d) show that the increase of m_e has only led to a slight improvement in the quality of the final community, indicating that CommunityAF has a stable performance in generation progress. In contrast to m_s, a higher m_e makes the component generation more conservative and tends to generate smaller communities.

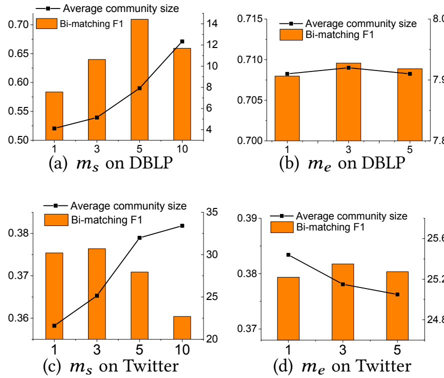

Figure 6: Bi-matching F1 and size of the generated communities under different parameters.

Study of Multi-task Training Tricks. We experiment with different training tricks in CommunityAF, such as considering the loss of generation and scoring tasks together or learning the generation task first and then learning together. Intuitively, the latter will yield more stable node embeddings because the generation tasks have more explicit signals. Specifically, the number of the horizontal axis of the coordinates, e.g., “20” in Figure 7, represents that we first train 20 epochs of the generation task, and then train two tasks together in the remaining epochs. As expected, the results show that training the generation task first improves performance on most datasets, except for Facebook.

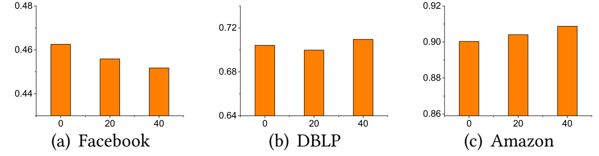

Figure 7: Bi-matching F1 varying different training tricks.

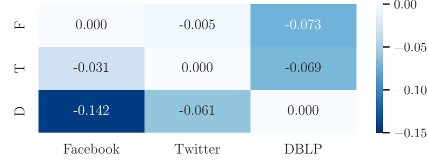

Figure 8: Bi-matching F1 descent results of the models tested under different datasets.

4.6 Community Hidden Patterns Study

In this subsection, we attempt to verify our previous claim that CommunityAF is able to capture the hidden patterns of given examples. In detail, we learn a model from a source dataset (e.g., Facebook), and apply the model to another target dataset (e.g., DBLP). We argue that the learned model can extract the relevant patterns if the communities discovered using the model in the target dataset are still similar to ground-truth communities in the source rather than ground-truth communities in the target dataset.

We choose Facebook, Twitter as representatives of social network datasets, and select DBLP as a representative community dataset for academic networks. As those features vary greatly across datasets, models are trained without using node attribute features. The learned models on different datasets are named as F (Facebook), T (Twitter), and D (DBLP) for short. Next, we apply these models to the other two datasets and record the changes of bi-matching $F1$ in Figure 8, where the darker color indicates the worse performance. It can be noted that the performances of models trained on the social network dataset drop slightly on similar social networks, but have more severe degradation on the academic network dataset. A similar result is observed for the model trained on the academic network dataset. This indicates that different types (social and academic) of graphs have different community patterns, and CommunityAF is able to learn community patterns similar to those in the train set.

Further, we choose 4 representative ground-truth communities from Twitter and DBLP, select a query node for each community (in red color), search communities using F, T, and D models, and visualize both ground-truth and searched communities. It is obvious that academic communities are more tightly structured, while social network communities are more tolerant of stray nodes and prefer to have a chain-like structure. These patterns can be learned by CommunityAF. For example, the D model intends to produce a community with a large minimal node degree in Twitter, while the T model guides to generating a community with densely connected nodes as well as nodes with degree 1.

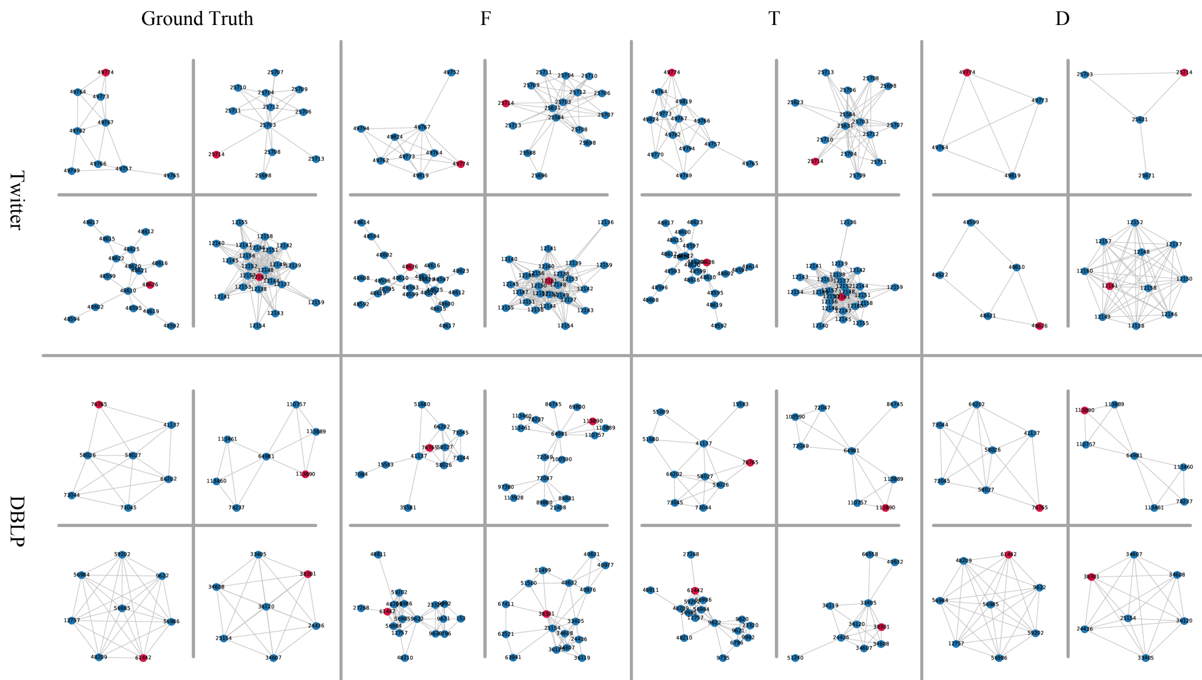

Figure 9: Visualization of community results. The labels of the nodes correspond to their indexes.

5 RELATED WORK

We review the following related works, except those methods mentioned as competitors in the experiment part.

Community Search. Community search [11], also known as local community detection, or seed extension problem, aims at finding a subgraph that contains a given query node (seed). It's the consensus of most work [11] that communities are subgraphs with content similarity and structural cohesion. In addition to the methods already mentioned in the experiment section, there are some methods that try to use attributes for queries. LocATC [17] proposes the concept of attribute community search (ACS), which requires users to input additional keywords (attributes) to locate communities where the corresponding keywords appear as frequently as possible. AQD-GNN [19] is based on QD-GNN, which models nodes and attributes as bipartite graphs and proposes the feature fusion operator to solve the ACS problem. Because we don't need to give attributes for querying in our scenario, only QD-GNN is chosen as the comparison method.

Subgraph Pattern Matching. Subgraph pattern matching (subgraph isomorphism) [10] aims to find subgraphs that are isomorphic to a given query graph. Subgraph isomorphism is an NP-complete problem, so existing methods tend to relax the structural requirements [36], design heuristic rules, utilize graph cache technology [39] or choose distributed computing methods [46] to support matching in a large graph. When relaxing strict structural constraints and considering the content features among multiple community examples, the existing subgraph pattern matching methods can be extended to handle the problem in this paper.

Normalizing Flows. Normalizing Flows [24] have made significant progress and have been successfully applied to a variety of tasks, including density estimation, variational inference, and image generation. Among existing works, IAF [30] first uses an autoregressive model as a form of normalizing flow and proposes an easily computable invertible variation with a triangular jacobian matrix. MAF [28] builds on IAF, which employs a network with masks to support fast parallel training. Normalizing flow has also received attention in the field of database research. For example, FACE [38] achieves good progress in the cardinality estimator task by using normalizing flow to learn the joint probability distribution of relational data.

6 CONCLUSION

This paper presents a new framework named CommunityAF, designed to handle the example-based community search problem with three well-designed key components in a multi-task way. CommunityAF utilizes an incremental GNN component for learning node embeddings in a large underlying graph to meet scalability, an autoregressive flow-based generation component for fast parallel training, and a scoring component over the learned node embeddings for flexible termination. We use a square ranking loss during training to ensure stability and introduce a flexible way to end the community generation process based on the score changes observed during beam search. Experimental results demonstrate that CommunityAF outperforms existing approaches and can learn various community patterns.

ACKNOWLEDGMENTS

This work was partially supported by NSFC under Grant No. 62272008 and 61832001.

REFERENCES

[1] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 475–486. https://doi.org/10.1109/FOCS.2006.44

[2] A. Bakshi, S. Parthasarathy, and K. Srinivasan. 2018. Semi-Supervised Community Detection Using Structure and Size. In *2018 IEEE International Conference on Data Mining (ICDM)*.

[3] Yuchen Bian, Yaowei Yan, Wei Cheng, Wei Wang, Dongsheng Luo, and Xiang Zhang. 2018. On Multi-query Local Community Detection. In *2018 IEEE International Conference on Data Mining (ICDM)*, Vol. NaN.

[4] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Józefowicz, and Samy Bengio. 2016. Generating Sentences from a Continuous Space. In *CoNLL*. ACL, 10–21.

[5] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. 129–136.

[6] Tanmoy Chakraborty, Sikhar Patranabis, Pawan Goyal, and Animesh Mukherjee. 2015. On the Formation of Circles in Co-authorship Networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams (Eds.). ACM, 109–118. https://doi.org/10.1145/2783258.2783292

[7] Jiazun Chen, Jun Gao, and Bin Cui. 2023. ICS-GNN$^+$: lightweight interactive community search via graph neural network. *VLDB J.* 32, 2 (2023), 447–467.

[8] Aaron Clauset. 2005. Finding local community structure in networks. *Physical Review E* 72, 2 (2005), 026132.

[9] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *SIGMOD*. 277–288.

[10] W. Fan, J. Li, M. Shuai, T. Nan, and Y. Wu. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *Proceedings of the VLDB Endowment* 3, 1 (2010), 264–275.

[11] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDB J.* 29, 1 (2020), 353–392.

[12] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: Lightweight Interactive Community Search via Graph Neural Network. *Proc. VLDB Endow.* 14, 6 (2021), 1006–1018. https://doi.org/10.14778/3447689.3447704

[13] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *NIPS*. 2672–2680.

[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*. 1024–1034.

[15] Kun He, Yiwei Sun, David Bindel, John Hopcroft, and Yixuan Li. 2015. Detecting Overlapping Communities from Local Spectral Subspaces. In *2015 IEEE International Conference on Data Mining*, Vol. NaN.

[16] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[17] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *Proc. VLDB Endow.* 10, 9 (2017), 949–960.

[18] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (2015), 276–287.

[19] Yuli Jiang, Yu Rong, Hong Cheng, Xin Huang, Kangfei Zhao, and Junzhou Huang. 2022. Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed. *Proc. VLDB Endow.* 15, 6 (2022), 1243–1255.

[20] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *ICLR*.

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[22] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *ICLR (Poster)*. OpenReview.net.

[23] Kyle Kloster and David F. Gleich. 2014. Heat kernel based community detection. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, Vol. NaN.

[24] I. Kobyzev, S. Prince, and M. Brubaker. 2020. Normalizing Flows: An Introduction and Review of Current Methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PP, 99 (2020), 1–1.

[25] Fanzhen Liu, Shan Xue, Jia Wu, Chuan Zhou, Wenbin Hu, Cécile Paris, Surya Nepal, Jian Yang, and Philip S. Yu. 2020. Deep Learning for Community Detection: Progress, Challenges and Opportunities. In *IJCAI*. 4981–4987.

[26] Feng Luo, James Wang, and Eric Promislow. 2006. Exploring Local Community Structures in Large Networks. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI'06)*. 233–239.

[27] Larry R Medsker and LC Jain. 2001. Recurrent neural networks. *Design and Applications* 5 (2001), 64–67.

[28] George Papamakarios, Iain Murray, and Theo Pavlakou. 2017. Masked Autoregressive Flow for Density Estimation. In *NIPS*. 2338–2347.

[29] Kainan Peng, Wei Ping, Zhao Song, and Kexin Zhao. 2020. Non-Autoregressive Neural Text-to-Speech. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Hal Daumé III and Aarti Singh (Eds.), Vol. 119. PMLR, 7586–7598. https://proceedings.mlr.press/v119/peng20a.html

[30] Diederik P.Kingma, Tim Salimans, and Max Welling. 2016. Improving Variational Inference with Inverse Autoregressive Flow. *CoRR* abs/1606.04934 (2016).

[31] Mariya Popova, Mykhailo Shvets, Junier Oliva, and Olexandr Isayev. 2019. MolecularRNN: Generating realistic molecular graphs with optimized properties. *CoRR* abs/1905.13372 (2019).

[32] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved Techniques for Training GANs. In *NIPS*. 2226–2234.

[33] C. Shi, M. Xu, Z. Zhu, W. Zhang, and J. Tang. 2020. GraphAF: a Flow-based Autoregressive Model for Molecular Graph Generation. In *ICLR*.

[34] Pan Shi, Kun He, David Bindel, and John E. Hopcroft. 2017. Local Lanczos Spectral Approximation for Community Detection. In *ECML/PKDD (1) (Lecture Notes in Computer Science)*, Vol. 10534. Springer, 651–667.

[35] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*. 939–948.

[36] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[37] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[38] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84. https://doi.org/10.14778/3485450.3485458

[39] Jing Wang, Zichen Liu, Shuai Ma, Nikos Ntarmos, and Peter Triantafillou. 2018. GC: A Graph Caching System for Subgraph/Supergraph Queries. *Proc. VLDB Endow.* 11, 12 (2018), 2022–2025.

[40] Xiyuan Wang and Muhan Zhang. 2022. GLASS: GNN with Labeling Tricks for Subgraph Representation Learning. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=XLxhEjKNbXj

[41] Sam Wiseman and Alexander M. Rush. 2016. Sequence-to-Sequence Learning as Beam-Search Optimization. In *EMNLP*. The Association for Computational Linguistics, 1296–1306.

[42] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *Proc. VLDB Endow.* 8, 7 (feb 2015), 798–809. https://doi.org/10.14778/2752939.2752948

[43] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. arXiv:1205.6233 [cs.SI]

[44] Kai Yao and Lijun Chang. 2021. Efficient Size-Bounded Community Search over Large Networks. *Proc. VLDB Endow.* 14, 8 (2021), 1441–1453.

[45] Jiaxuan You, Rex Ying, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In *ICML (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 5694–5703.

[46] Ye Yuan, Delong Ma, Zhenyu Wen, Zhiwei Zhang, and Guoren Wang. 2021. Subgraph matching over graph federation. *Proceedings of the VLDB Endowment* 15, 3 (2021), 437–450.

[47] Jiawei Zhang, Philip S. Yu, and Yuanhua Lv. 2017. Enterprise Employee Training via Project Team Formation. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.). ACM, 3–12. https://doi.org/10.1145/3018661.3018682

[48] Yao Zhang, Yun Xiong, Yun Ye, Tengfei Liu, and Philip S. Yu. 2020. SEAL: Learning Heuristics for Community Detection with Generative Adversarial Networks. In *KDD '20*.

[49] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. *CoRR* (2018).