Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed

Yuli Jiang*[1], Yu Rong†[2], Hong Cheng*[3], Xin Huang‡[4], Kangfei Zhao†[5], Junzhou Huang†[6] *
*The Chinese University of Hong Kong, †Tencent AI Lab, ‡Hong Kong Baptist University, China
{[1]yljiang,[3]hcheng,[5]kfzhao}@se.cuhk.edu.hk, [2]yu.rong@hotmail.com, [4]xinhuang@comp.hkbu.edu.hk, [5]zkf1105@gmail.com, [6]jzhuang@uta.edu

ABSTRACT

Given one or more query vertices, Community Search (CS) aims to find densely intra-connected and loosely inter-connected structures containing query vertices. Attributed Community Search (ACS), a related problem, is more challenging since it finds communities with both cohesive structures and homogeneous vertex attributes. However, most methods for the CS task rely on inflexible pre-defined structures and studies for ACS treat each attribute independently. Moreover, the most popular ACS strategies decompose ACS into two separate sub-problems, i.e., the CS task and subsequent attribute filtering task. However, in real-world graphs, the community structure and the vertex attributes are closely correlated to each other. This correlation is vital for the ACS problem. In this vein, we argue that the separation strategy cannot fully capture the correlation between structure and attributes simultaneously and it would compromise the final performance.

In this paper, we propose Graph Neural Network (GNN) models for both CS and ACS problems, i.e., Query Driven-GNN (QD-GNN) and Attributed Query Driven-GNN (AQD-GNN). In QD-GNN, we combine the local query-dependent structure and global graph embedding. In order to extend QD-GNN to handle attributes, we model vertex attributes as a bipartite graph and capture the relation between attributes by constructing GNNs on this bipartite graph. With a Feature Fusion operator, AQD-GNN processes the structure and attribute simultaneously and predicts communities according to each attributed query. Experiments on real-world graphs with ground-truth communities demonstrate that the proposed models outperform existing CS and ACS algorithms in terms of both efficiency and effectiveness. More recently, an interactive setting for CS is proposed that allows users to adjust the predicted communities. We further verify our approaches under the interactive setting and extend to the attributed context. Our method achieves 2.37% and 6.29% improvements in F1-score than the state-of-the-art model without attributes and with attributes respectively.

PVLDB Reference Format:
Yuli Jiang, Yu Rong, Hong Cheng, Xin Huang, Kangfei Zhao, Junzhou Huang. Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed. PVLDB, 15(6): 1243 - 1255, 2022.

doi:10.14778/3514061.3514070

*This work is done during Yuli's internship at Tencent AI Lab. Yu Rong and Hong Cheng are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514070

1 INTRODUCTION

Graph is an essential data structure to represent entities and their relationships, e.g., social networks, protein-protein interaction networks, web graphs, and knowledge graphs, to name a few. Community, a subgraph of densely intra-connected and loosely inter-connected structure, naturally exists as a functional module in real-world graphs. Community Search (CS) [7, 11, 19, 21, 25, 37] is a vital application in graph analytics. Concretely, given any query vertices, CS aims to find a vertex set with cohesive structure according to the query, i.e., query-dependent communities. Attributed Community Search (ACS), a related but more challenging problem, has attracted a lot of attention recently [10, 11, 22, 23, 25]. Given any query vertex and attribute set, ACS aims at finding query-dependent communities with homogeneous attributes, which means the community members share similar attributes with the query attributes.

For the CS and the ACS problems, existing studies suffer from two serious limitations, that is, ***structure inflexibility*** and ***attribute irrelevance***. Structure inflexibility refers to the problem that most community search models are based on a pre-defined subgraph pattern, such as k-core [8, 10, 37], k-truss [1, 21, 22], k-clique [7, 44], and k-edge connected component (ECC) [6, 19]. The pre-defined subgraph pattern imposes a very rigid requirement on the topological structure of communities, which may not perfectly hold in real-world communities. Attribute irrelevance means existing models treat each attribute independently [10, 22]. However, in the real graphs, vertex attributes are not independent of each other. Ignoring such implicit relations would harm the quality of queried communities.

Figure 1 depicts a toy example illustrating the limitation of existing algorithms. The faculty hierarchy is a tree-like structure from the faculty dean, department chairman to the professors in each department. Using existing methods based on pre-defined subgraph patterns, we can only find a 1-core community of vertex 6 in H_1 and a 2-truss community in H_2, which are the entire graph. These k-core [37] and k-truss [21] patterns cannot discover the tree-like department communities owing to the structure inflexibility. For attributed community search, when querying the community of vertex 6 and attribute "ML", current methods [10, 22] find the community H_3 since they ignore the implicit relations between "ML", "DL" and "CV". Thus, existing studies suffer from these two inadequacies on structure and attribute respectively.

Moreover, for the ACS problem, existing studies [10, 22] usually adopt a two-stage strategy which first finds the candidate community by considering the topological structure only, and then performs a filtering on the candidate community by considering the attribute similarity. The two-stage strategy *treats the structure*

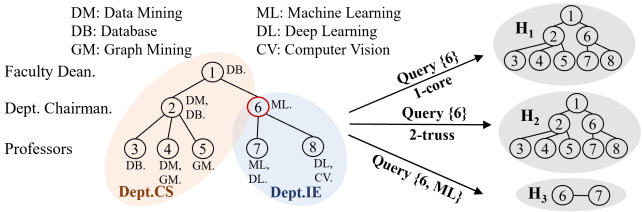


Figure 1: An attributed graph depicting a faculty hierarchy with two departments: Dept.CS and Dept.IE. Attributes represent research topics. For vertex 6, there is a ground-truth Dept.IE community (shown in blue) as vertices $6-8$ are close and work on similar topics. On the right, in response to queries on each arrow, there are three result communities found by existing algorithms, which are quite different from the ground-truth community.

cohesiveness and attribute homogeneity separately. But there is usually a correlation between structure and attribute, for instance, in protein-protein interaction networks, proteins with similar functions (i.e., attributes) are more likely to interact with each other [39]. Independently dealing with the structure and attribute would harm the quality of queried communities.

Inspired by the success of Graph Neural Network (GNN) [27] on combining attribute and structure in many graph problems, Gao *et al.* [14] proposed a GNN-based framework, ICS-GNN, to solve the community search problem in an interactive fashion (i.e., users can adjust predicted communities during the query process). Specifically, it enhances the non-attributed queries by the GNN model [27] which exploits the information from the existing vertex attributes in graphs. However, for every query, ICS-GNN re-trains the whole model. This re-training process is time-consuming and hinders its applications in real-world scenarios, especially for the online query case. On the other hand, even though ICS-GNN makes use of the attributes to enhance the community search performance, its model architecture cannot accept the query attributes as input. Therefore, ICS-GNN cannot be extended to support interactive attributed community search easily.

To address the above limitations, in this paper, we propose GNN-based models for both CS and ACS problems. For the CS problem, to address the structure inflexibility issue, we design a two-branch model: Query Driven-GNN (QD-GNN) to encode the information from both the query and graph. Concretely, QD-GNN contains two encoders, *Query Encoder* and *Graph Encoder*. *Query Encoder* encodes the structural information from query vertices and focuses on modeling the local topology around the queries. *Graph Encoder* combines the global structure and attributes to learn the query-independent node embeddings. As a learning-based model, QD-GNN can search communities without imposing any restriction on the community structure. Furthermore, we design an additional *Attribute Encoder* to extend QD-GNN to support the attributed community search. *Attribute Encoder* exploits a node-attribute bipartite graph to model the attribute relations and can encode more meaningful information from the attribute space. To process structure and attribute simultaneously, we employ a *Feature Fusion* component to fuse the information from different encoders and make the final output. Furthermore, we design a new query framework which detaches the model training from the online query stage. Therefore, our framework does not need the time-consuming re-training phase for online query applications.

To summarize, we make the following contributions.

- We propose a Query Driven-GNN model (QD-GNN) for community search, which combines the local query-dependent structure and global node embeddings. Given any query, QD-GNN only needs a model inference step and avoids the time-consuming re-training.
- To the best of our knowledge, this is the first work that proposes a GNN model for the attributed community search problem, called Attributed Query Driven-GNN (AQD-GNN). Our novel learning framework extends GNN into ACS through a node-attribute bipartite graph, and learns the community information from both the local structure and similar attributes of queries.
- We conduct extensive experiments on real-world data sets with ground-truth communities for performance evaluation. Experiments demonstrate that our model significantly outperforms state-of-the-art methods in terms of community quality with only 4.31 milliseconds average response time.
- We apply AQD-GNN to the interactive community search problem and extend it into the attributed context. Experiments show that our models can improve the performance of ICS-GNN [14] in both non-attributed and attributed manner with 2.37% and 6.29% improvements in F1-score respectively.

Roadmap. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 gives some preliminaries. Section 4 presents the common framework of the proposed models. Section 5 introduces the QD-GNN model for community search problem, and Section 6 describes the AQD-GNN model for attributed community search. We present the experimental results in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

Our study is closely related to community search (CS) and graph neural network (GNN).

Community Search. The problem of CS [37] is to find densely connected communities containing the query vertices. A comprehensive survey of CS models and existing approaches can be found in [11, 25]. Various community models have been proposed based on different cohesive graph patterns, including k-core [8, 37], k-truss [1, 21, 24], k-clique [7, 44], and k-edge connected component (ECC) [6, 19]. These pre-defined cohesive metrics are inflexible and can be too loose (e.g., k-core) or too tight (e.g., k-clique) to capture the topology structure of communities. If the real-world communities do not follow any of the above graph patterns, these models would fail to discover the true communities. A learning-based model ICS-GNN [14] has recently been proposed for interactive community search. ICS-GNN first finds a candidate subgraph starting from query vertices, then learns the node embeddings through applying GNN model on subgraph, and finally employs a BFS based algorithm to select the k-sized community with maximum GNN scores. ICS-GNN does not support attributed community search as the query only involves vertices but no attributes. It also needs to re-train the entire model for each query, which is costly for this online query problem.

For attributed community search, ACQ [10] and ATC [22] have been proposed, which aim to discover communities that contain query vertices and have similar attributes to the query attributes. ACQ is based on k-core and finds communities with the maximum number of common query attributes shared by community members. ATC finds k-truss communities with the maximum pre-defined attribute score. Both adopt a two-stage process. They first impose a pre-defined structural constraint to find candidate communities, then optimize functions of attribute score to select the most related communities. However, the attribute score functions ignore the similarities between attributes, and these two-stage methods fail to capture the correlation between structure and attribute. In this paper, we propose QD-GNN, which considers the cohesive structure and homogeneous attributes in an integrated way.

Graph Neural Network. Inspired by the huge success of neural networks in natural language processing and computer vision, many graph analytic problems have been solved via graph neural networks [27], such as node classification[4, 18, 35], graph classification [20, 29], drug discovery [31, 34, 43], adversarial attacks [2, 3, 5, 47] and graph algorithmic tasks [45, 46]. To build good models, the advanced techniques of pooling [13, 28, 32] and attention [12, 28, 40] have been developed. However, most learning models are designed for specific tasks based on graph embedding [30, 42] or end-to-end solutions [15, 36]. Existing GNN models cannot extend to attributed community search straightforwardly. To the best of our knowledge, we are the first to propose a GNN-based model for attributed community search and extend ICS-GNN to the attributed context as well.

3 PRELIMINARIES

In this section, we first introduce the notations and define the problems of CS and ACS formally, and then describe a general GNN as the foundation of our proposed models.

3.1 Definitions

Let $G(\mathcal{V}, \mathcal{E})$ be a graph with a set $\mathcal{V}$ of vertices and a set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ of edges. Let $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ be the number of vertices and edges respectively. We denote $\mathcal{N}(\mathrm{v}) = \{\mathrm{u} \mid (\mathrm{u}, \mathrm{v}) \in \mathcal{E}\}$ as the neighborhood set of vertex v. Moreover, let $\mathcal{N}^+(\mathrm{v}) = \{\mathrm{v}\} \cup \mathcal{N}(\mathrm{v})$ be the vertex set containing v's neighbors and v itself.

Community Search (CS). For a graph $G(\mathcal{V}, \mathcal{E})$, given a vertex query set $\mathcal{V}_q \subseteq \mathcal{V}$, the problem of Community Search (CS) is to find the query-dependent community $C_q \subseteq \mathcal{V}$. Vertices in community C_q need to be densely intra-connected, i.e., having cohesive structure.

Let $G(\mathcal{V}, \mathcal{E}, \mathcal{F})$ be an attributed graph where $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ is the set of vertex attributes and $\mathcal{F}_i$ is the attribute set of vertex v_i. Define $\hat{\mathcal{F}}$ as the union of all the vertex attribute sets, i.e., $\hat{\mathcal{F}} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_n$. Let d be the number of unique attributes $d = |\hat{\mathcal{F}}|$. The attribute set of each vertex, e.g., $\mathcal{F}_i$, is encoded to a d-dimensional vector $\boldsymbol{f}_i$. For a keyword attribute f_k, if vertex v_i has this keyword, i.e., $\mathrm{f}_k \in \mathcal{F}_i$, then $\boldsymbol{f}_{ik} = 1$; otherwise, $\boldsymbol{f}_{ik} = 0$. For a numerical attribute f_j, $\boldsymbol{f}_{ij}$ is the value of vertex v_i on this attribute. Then the set of vertex attributes $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ is encoded to an attribute matrix $\boldsymbol{F} = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n]^T \in \mathbb{R}^{n \times d}$.

Attributed Community Search (ACS). For an attributed graph $G(\mathcal{V}, \mathcal{E}, \mathcal{F})$, given a query $\langle \mathcal{V}_q, \mathcal{F}_q \rangle$ where $\mathcal{V}_q \subseteq \mathcal{V}$ is a set of query vertices, and $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ is a set of query attributes, the problem of Attributed Community Search (ACS) is to find the query-dependent community $C_q \subseteq \mathcal{V}$. Vertices in community C_q need to be both structure cohesive and attribute homogeneous, i.e., vertices in a community are densely intra-connected in structure and attributes of these vertices are similar.

In this paper, we formulate the above two problems as a binary classification task. Given a query $q = \langle \mathcal{V}_q \rangle$ or $q = \langle \mathcal{V}_q, \mathcal{F}_q \rangle$, we classify the graph vertices into two classes (belonging to a community C_q of query q or not). We use the one-hot vector $\boldsymbol{c}_q \in \{0, 1\}^n$ to represent the output community C_q by a model $\mathcal{M}$. If the output value $\boldsymbol{c}_{q_k} = 1$, vertex v_k belongs to the result community C_q predicted by $\mathcal{M}$.

3.2 A General GNN Model

We introduce a general framework of Graph Neural Network (GNN) as the cornerstone of our models.

A GNN layer is known as a message passing procedure from neighborhoods. After the linear transformation of neighbors' hidden features, there are many alternative techniques within one layer, e.g., batch normalization technique [26]. We list one of the possible intra-layer processes in the layer-wise propagation function as:

$$\boldsymbol{h}_v^{(l+1)} = \mathrm{Dr}\left\{\phi\left(\mathrm{BN}[\mathrm{AGG}(\boldsymbol{h}_u^{(l)}\boldsymbol{W}^{(l+1)} + \boldsymbol{b}^{(l+1)}, \mathrm{u} \in \mathcal{N}^+(\mathrm{v}))]\right)\right\}, \tag{1}$$

where $\boldsymbol{h}_v^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the learned new features of vertex v in the $(l+1)$-th layer, $\boldsymbol{h}_u^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the hidden features of vertex u from the l-th layer, and the input feature $\boldsymbol{h}_v^{(0)} \in \mathbb{R}^d$ is the normalized form of attribute vector $\boldsymbol{f}_v$. $\boldsymbol{W}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ and $\boldsymbol{b}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ are trainable weights. $\mathrm{AGG}(\cdot)$ is an aggregation function such as SUM, MAX, or MIN. $\mathrm{BN}(\cdot)$ is batch normalization [26] that reduces internal covariate shift. $\phi(\cdot)$ is the non-linear activation function, such as $\mathrm{ReLU}(\cdot)$. Last, $\mathrm{Dr}(\cdot)$ is the dropout method [38] to dilute the data and reduce the overfitting in neural networks.

For example, one of the most classical GNN models, Vanilla Graph Convolutional Network (Vanilla GCN) [27], is defined as:

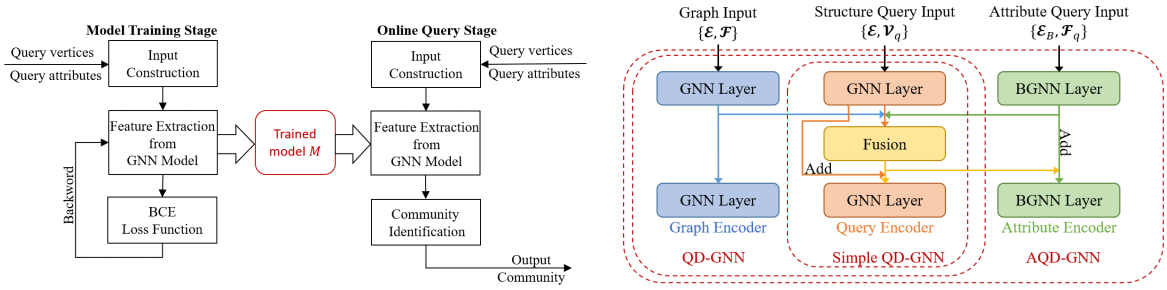
$$\boldsymbol{h}_v^{(l+1)} = \mathrm{Dr}\left\{\mathrm{ReLU}\left(\mathrm{SUM}(\{\frac{\boldsymbol{h}_u^{(l)}}{\sqrt{d'_{\mathrm{u}} d'_{\mathrm{v}}}}\boldsymbol{W}^{(l+1)} : \mathrm{u} \in \mathcal{N}^+(\mathrm{v})\})\right)\right\}, \tag{2}$$

which applies SUM as the aggregation operation, and $\mathrm{ReLU}(\cdot)$ as the activation function $\phi(\cdot)$ with the dropout method. In this GNN model, batch normalization is not adopted and Laplacian smoothing is employed where $d'_{\mathrm{u}} = d_{\mathrm{u}} + 1$ and d_{u} is the degree of vertex u.

In the following, we will focus on the way of aggregation in our proposed GNN models. The dropout, activation function, batch normalization and the trainable bias $\boldsymbol{b}$ described above are adopted in our models, and will be omitted in our following presentation.

4 THE QUERY FRAMEWORK

Before describing the detailed design of the proposed models, we introduce the common framework of our models for both CS and ACS problems. As Figure 2a shows, the proposed models consist of two main stages: *the model training stage* and *the online query stage*. Firstly, we train the embedding model $\mathcal{M}$ offline with the loss function in the model training stage as shown in Figure 2a (left). After that, in the online query stage, whenever the query comes, we apply the model from the training stage to predict the



(a) Framework of our proposed models. **(b) Intermediate design of model $\mathcal{M}$ with four components.**

Figure 2: The architecture of proposed models.

community without re-training, as Figure 2a (right) presents. This framework is highly flexible. In the following, we first introduce how to construct the inputs from the graph and queries in both stages. Then, we describe the two main stages respectively.

4.1 Input Construction

Since the GNN model $\mathcal{M}$ needs vectorized inputs, we introduce the vectorization scheme for the vertex set and attribute set.

Construct query vertices. We encode each query vertex set $\mathcal{V}_q \subseteq \mathcal{V}$ to a one-hot vector $\boldsymbol{v}_q \in \{0,1\}^n$. For a query $\mathcal{V}_q$, if vertex $v_i \subseteq \mathcal{V}_q$, $\boldsymbol{v}_{q_i} = 1$; otherwise, $\boldsymbol{v}_{q_i} = 0$. For example, when querying the community of vertex v_6 in Figure 1, the encoded vector is $\boldsymbol{v}_q = [0,0,0,0,0,1,0,0]^T$.

Construct query attributes. Similar to query vertices, we encode each query attribute set $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ to a one-hot vector $\boldsymbol{f}_q \in \{0,1\}^d$, where $d = |\hat{\mathcal{F}}|$ is the number of unique attributes.

The encoded query vertex set and query attribute set are then submitted to our proposed GNN models as input features.

4.2 Model Training Stage

In the model training stage, with a set of training queries as input, we iteratively train the embedding model $\mathcal{M}$ offline through the Binary Cross Entropy (BCE) loss function and obtain a trained model for the online query stage.

Given a set of training queries $Q_{\text{train}} = \{q_1, q_2, ...\}$ and corresponding ground-truth communities $C_{\text{train}} = \{C_{GT1}, C_{GT2}, ...\}$, we train a GNN model $\mathcal{M}$ to minimize the loss function to fit the training data. Given a validation query set Q_{val} and corresponding ground-truth communities C_{val}, we select the parameters of model $\mathcal{M}$ and threshold $\gamma \in [0,1]$ which achieve the best performance in the validation set. The queries in Q_{train} and Q_{val} can be attributed $q = \{\mathcal{V}_q, \mathcal{F}_q\}$ for ACS or non-attributed $q = \{\mathcal{V}_q\}$ for CS.

First, we construct all query inputs as one-hot vectors. Then we repeatedly input queries into the model $\mathcal{M}$, i.e., Simple QD-GNN, QD-GNN or AQD-GNN, which will be introduced in Section 5 and Section 6. With the model $\mathcal{M}$'s output $\boldsymbol{h}_q$ for each query q in an iteration, we compute BCE loss function and gradients of the model parameters. The gradients are propagated backward to update $\mathcal{M}$ at the end of this iteration. With the updated parameters, $\mathcal{M}$ moves to the next iteration, outputs $\boldsymbol{h}_q$, calculates loss and back propagates gradients until convergence. The loss function of the three proposed models is the same and we describe it formally in the following.

Loss Function. We formulate community search as a binary classification problem. Assume that $\boldsymbol{h}_q \in \mathbb{R}^n$ is the output of $\mathcal{M}$ for query q after the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, where $\boldsymbol{h}_{q_v} \in [0,1]$ represents the output for vertex v. $\boldsymbol{y}_q \in \{0,1\}^n$ represents the ground-truth vector for query q. $\boldsymbol{y}_{q_v} = 1$ if and only if vertex $v \in C_{GTq}$; otherwise, $\boldsymbol{y}_{q_v} = 0$. Then we utilize Binary Cross Entropy (BCE) function as the loss function to minimize the BCE between the model output $\boldsymbol{h}_q$ and the ground-truth label $\boldsymbol{y}_q$ for q. The optimization loss function can be formulated as:

$$\min \mathcal{L} = \sum_{q \in Q_{\text{train}}} \frac{1}{n} \sum_{i=1}^{n} -(\boldsymbol{y}_{q_i} \log(\boldsymbol{h}_{q_i}) + (1 - \boldsymbol{y}_{q_i}) \log(1 - \boldsymbol{h}_{q_i})). \tag{3}$$

Algorithm 1 Constrained BFS for Community Identification

Input: Graph: $G = (\mathcal{V}, \mathcal{E})$, a query vertex set: $\mathcal{V}_q$, a model output vector: $\boldsymbol{h}_q$, a threshold: γ.
Output: a vertex set of community : C_q.

1: Initialize set $Q = \mathcal{V}_q$, $C_q = \mathcal{V}_q$
2: **while** Q is not empty **do**
3: select a vertex v from Q
4: **for** u $\in \mathcal{N}$(v) and $\boldsymbol{h}_{q_u} \geq \gamma$ **do**
5: $Q \leftarrow Q \cup \{u\}$
6: $C_q \leftarrow C_q \cup \{u\}$
7: **return** C_q;

4.3 Online Query Stage

In the online query stage, we utilize the well-trained model $\mathcal{M}$ and threshold γ from the model training stage to process the online query q and produce the community C_q without re-training. We first construct query inputs as one-hot vectors. Then the constructed vectors are fed into model $\mathcal{M}$, which only runs once and outputs the vector $\boldsymbol{h}_q$. To ensure the connectivity between query vertices and community members, we employ a constrained Breadth-First Search (BFS) starting from the query vertices in Algorithm 1. When visiting vertex u, if $\boldsymbol{h}_{q_u} \geq \gamma$ (line 4), we add vertex u to the output community C_q (line 6).

Please note that the connectivity of the output community also depends on the user-specified query vertices. If the induced subgraph of the query vertices is connected, then our models are guaranteed to find a connected community. If the induced subgraph of the query vertices is not connected, our models may still find a connected community through some bridging vertices. But there is possibility that the discovered community is not connected as one component, especially when the query vertices are distant or disconnected in the graph. In this case, our models can still find some connected components, each of which contains part of the query vertices, as the answer community.

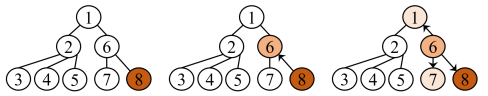


Figure 3: Query propagation paths in Query Encoder.

5 QD-GNN MODEL FOR CS

In this section, we introduce the construction of the embedding model $\mathcal{M}$ in the proposed framework for community search. We first propose a Simple task-oriented Query Driven-Graph Neural Network (Simple QD-GNN) and then design useful functional encoders to improve it as QD-GNN model. As Figure 2a shows, with query vectors as input, the Simple QD-GNN or QD-GNN model $\mathcal{M}$ outputs $\boldsymbol{h}_q$ into the BCE loss function during the training process. In the online query stage, the model output $\boldsymbol{h}_q$ is translated into community members as described in Section 4.

5.1 Simple QD-GNN

The Simple QD-GNN model is designed based on the general GNN introduced in Section 3.2 and uses query vector $\boldsymbol{v}_q$ as the input features of the model. This model input enables query-centered structural propagation, i.e., propagating from the query vertices to its neighborhood, to better capture the local query structure information.

We name this query driven propagation as *Graph Encoder*. In order to fully make use of vertex features in each layer, Query Encoder is designed to equip with a self feature modeling [12]. The inter-layer propagation function for vertex v is formally defined as:

$$\boldsymbol{h}_{Q_v}^{(l+1)} = \boldsymbol{h}_{Q_v}^{(l)} \boldsymbol{W}_{Q_{\text{self}}}^{(l+1)} + \text{SUM}(\{\boldsymbol{h}_{Q_u}^{(l)} \boldsymbol{W}_{Q}^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}), \quad (4)$$

where the first component emphasizes the self features (hidden features of the vertex v) with learnable weight parameter matrices $W_{Q_{\text{self}}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$. The second component is similar to Eq. (1) with a subscript Q, and chooses SUM as the aggregation function as Vanilla GCN [27] does. Similarly, $\boldsymbol{W}_{Q}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the trainable weight matrix , $\boldsymbol{h}_{Q_v}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the learned new features of vertex v in the $(l+1)$-th layer of Query Encoder. Different from Eq. (1), the input feature of the first layer $\boldsymbol{h}_{Q_v}^{(0)}$ is the one-hot query vector $\boldsymbol{v}_{q_v}$.

Example. *We follow the example in Figure 1 and show the propagation paths in Figure 3. For query $\mathcal{V}_q = \{\text{v}_8\}$ highlighted in Figure 3a, the query vector $\boldsymbol{v}_q$ is $[0, 0, 0, 0, 0, 0, 0, 1]^T$. According to Eq.* (4)*, in the first layer, the query information propagates to the neighbor of* v_8*, i.e.,* v_6 *as depicted in Figure 3b. Then, the 2-hop neighbors of the query vertex,* v_1 *and* v_7*, acquire the knowledge from* v_6 *in the second layer as depicted in Figure 3c.*

5.2 QD-GNN

Recent studies [9, 14, 41] have found that attributes on graph vertices can be leveraged for structural learning problems, for example, link prediction [9] and community search [14]. Inspired by their findings, we design an improved QD-GNN model based on Simple QD-GNN and Vanilla GCN [27]. Similar to ICS-GNN [14], QD-GNN combines the network structure and vertex attributes to solve the community search problem.

5.2.1 Overview. Figure 2b presents the architecture overview of QD-GNN model, which consists of two convolution branches (*Graph Encoder* and *Query Encoder*) and a *Feature Fusion* operator. *Graph Encoder* provides the query-independent information with both graph structure and vertex attributes as input, i.e., the edge set $\mathcal{E}$ and vertex attribute set $\mathcal{F}$. *Query Encoder* (the same as that in Simple QD-GNN) provides the interface for query vertices and learns the query-specific local topology features. It takes the input of graph structure and query vertices, i.e., the edge set $\mathcal{E}$ and query vertices $\mathcal{V}_q$. The *Feature Fusion* operator combines the above encoder embedding results and obtains the final query-specific output vectors. This fusion makes use of both global graph knowledge and local query information which can achieve a good balance, and finally obtains the model output $\boldsymbol{h}_q$ for each query q.

5.2.2 Graph Encoder. Graph Encoder focuses on global graph structure and vertex attributes, both of which are independent of queries. We apply the layer-wise forward propagation of the general GNN to construct Graph Encoder, which has been introduced in Section 3.2. Similar to Simple QD-GNN, the forward layer of Graph Encoder is defined with a self feature modeling [12] as:

$$\boldsymbol{h}_{G_v}^{(l+1)} = \boldsymbol{h}_{G_v}^{(l)} \boldsymbol{W}_{G_{\text{self}}}^{(l+1)} + \text{SUM}(\{\boldsymbol{h}_{G_u}^{(l)} \boldsymbol{W}_{G}^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}), \quad (5)$$

where the notations are the same as Eq. (1) with a subscript G, and $W_{G_{\text{self}}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ are the weight parameter matrices. The input feature of vertex v in the first layer $\boldsymbol{h}_{G_v}^{(0)} \in \mathbb{R}^{d}$ is the normalized attribute vector $\boldsymbol{f}_v$ encoded in Section 3.1. Graph Encoder propagates the attribute information through graph structure and learns query-independent knowledge.

Example. *We follow the example in Figure 1 to illustrate how Graph Encoder works. For vertex* v_8*, in the first layer, its attributes ("DL" and "CV") are propagated to its neighbor, vertex* v_6*, with a learnable weight. At the same time, the attribute of vertex* v_6 *("ML") is also propagated to vertex* v_8*. In the next layer, those attributes are propagated to their neighbors respectively as well. By this propagation, the attributes of vertices* v_6*,* v_7 *and* v_8 *become more similar. This information is used by the Feature Fusion operator to identify the community members more accurately.*

5.2.3 Query Encoder. Query Encoder is the same as that of Simple QD-GNN and provides an interface for query vertices and obtains the local structure knowledge. Inputs of the Query Encoder are based on the graph topology (graph edges $\mathcal{E}$) and structural query (query vertices $\mathcal{V}_q$). The inter-layer propagation function of Query Encoder is the same as that in Eq. (4).

5.2.4 Feature Fusion. The Feature Fusion operator combines output features learned by the above two encoders, and balances the global and local information to get the final output of QD-GNN. The inputs of Feature Fusion are based on the output of the two encoders, i.e., $\boldsymbol{h}_G$ and $\boldsymbol{h}_Q$. It fuses them and transmits the fusion result to Query Encoder as shown in Figure 2b.

Based on the output of the two encoders, the forward layer of Feature Fusion is formulated as:

$$\boldsymbol{h}_{FF_v}^{(l+1)} = \text{AGG}(\boldsymbol{h}_{G_v}^{(l+1)}, \boldsymbol{h}_{Q_v}^{(l+1)}), \quad (6)$$

where $\boldsymbol{h}_{FF_v}^{(l+1)}$ is the output of Feature Fusion for vertex v and also the final output of the entire QD-GNN model in the $(l+1)$-th layer, $\text{AGG}(\cdot)$ is the aggregation function (e.g., Concatenation, SUM, etc.),

Algorithm 2 The k-Layer QD-GNN Propagation

Input: Graph: $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$,
a set of queries: $\mathcal{Q} = \{\mathcal{V}_{q_1}, \mathcal{V}_{q_2}, \dots\}$,
QD-GNN model: $\mathcal{M} = \{\boldsymbol{h}_Q, \boldsymbol{h}_G, \boldsymbol{h}_{FF}\}$.
Output: a set of output vectors: $\mathcal{H} = \{\boldsymbol{h}_{q_1}, \boldsymbol{h}_{q_2}, \dots\}$.

1: Construct attribute matrix $\boldsymbol{F}$ for $\mathcal{F}$
2: $\mathcal{H} \leftarrow \varnothing$
3: **for** each $\mathcal{V}_q \in \mathcal{Q}$ **do**
4: Construct one-hot vector $\boldsymbol{v}_q$ for $\mathcal{V}_q$, initialize $\boldsymbol{h}_Q^{(0)}$ with $\boldsymbol{v}_q$
5: Initialize $\boldsymbol{h}_G^{(0)}$ with $\boldsymbol{F}$
6: $\boldsymbol{h}_Q^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_Q^{(0)}, \mathcal{E})$ in Eq. (4)
7: $\boldsymbol{h}_G^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(0)}, \mathcal{E})$ in Eq. (5)
8: $\boldsymbol{h}_{FF}^{(1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(1)}, \boldsymbol{h}_Q^{(1)})$ in Eq. (6)
9: $l \leftarrow 1$
10: **while**$(l < k)$ **do**
11: $\boldsymbol{h}_Q^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E})$ in Eq. (8)
12: $\boldsymbol{h}_G^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(l)}, \mathcal{E})$ in Eq. (5)
13: $\boldsymbol{h}_{FF}^{(l+1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(l+1)}, \boldsymbol{h}_Q^{(l+1)})$ in Eq. (6)
14: $l \leftarrow l + 1$
15: $\mathcal{H} \leftarrow \mathcal{H} \cup \boldsymbol{h}_{FF}^{(k)}$
16: **return** $\mathcal{H}$;

and $\boldsymbol{h}_{G_v}^{(l+1)}$, $\boldsymbol{h}_{Q_v}^{(l+1)}$ are the outputs of each encoder for vertex v in the $(l+1)$-th layer respectively.

For Graph Encoder, we do not use the fusion result and just use the output of Graph Encoder itself in the l-th layer $\boldsymbol{h}_G^{(l)}$ as the input of the $(l+1)$-th layer. Thus, we keep Graph Encoder independent of query information in the intermediate layer. This query-independent features provide stable "prior" knowledge about the graph and supply additional information for community search problem, which makes QD-GNN a stronger model.

For Query Encoder, we replace the feature propagation between neighbors as the fusion features in the intermediate layers. This fusion operation transmits the vertex attributes and global structure features into Query Encoder and delivers these features around query vertices. We define $\hat{\boldsymbol{h}}_Q$ as the input feature of each layer which can be formally written as:

$$\hat{\boldsymbol{h}}_{Q_i}^{(l)} = \begin{cases} \boldsymbol{v}_{q_i}, & \text{if } l = 0; \\ \boldsymbol{h}_{FF_i}^{(l)}, & \text{otherwise.} \end{cases} \tag{7}$$

The propagation function of Query Encoder can be rewritten as:

$$\boldsymbol{h}_{Q_v}^{(l+1)} = \boldsymbol{h}_{Q_v}^{(l)} \boldsymbol{W}_{Q_{\text{self}}}^{(l+1)} + \text{SUM}(\{\hat{\boldsymbol{h}}_{Q_u}^{(l)} \boldsymbol{W}_Q^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}). \tag{8}$$

5.2.5 Algorithm. The QD-GNN model for the community search problem is presented in Algorithm 2. For easy description, we simplify the propagation function in each encoder as $\text{Propg}(\boldsymbol{h}, \mathcal{E})$, which means propagating feature $\boldsymbol{h}$ through edges in $\mathcal{E}$. At the beginning, we construct the feature matrix for the graph (line 1) and set the output as empty (line 2). For each query, we also construct the query vector and initialize Query Encoder and Graph Encoder (line 4-5). In the first layer (line 6-9), Query Encoder and Graph Encoder propagate their input features through the graph edges (line 6-7), and Feature Fusion fuses the output of them (line 8). In the intermediate layers (line 10-14), Query Encoder utilizes the fused feature from Feature Fusion (line 11), while Graph Encoder takes its own output $\boldsymbol{h}_G$ as the input feature to remain independent of the query (line 12). The final output of QD-GNN is the fused feature $\boldsymbol{h}_{FF}$ and we add it into the output set $\mathcal{H}$ (line 15).

6 AQD-GNN MODEL FOR ACS

In this section, we extend QD-GNN by incorporating the query attributes and propose the GNN model for attributed community search, named Attributed Query Driven-Graph Neural Network (AQD-GNN). We first identify the challenges of attributed community search when using GNN models. Then, we describe the components of AQD-GNN one by one in detail.

6.1 Challenges

Different from community search [1, 8, 14, 21, 37], the attributed community search task [10, 22] needs to integrate the query attributes into models. However, the meaning of query attributes F_q and the dimension of query attributes vector f_q are different from those of query vertices. It is not feasible to input query attribute information as we handle query vertices in Section 5.

The ICS-GNN model [14] utilizes the query vertex information as the labels of vertices, and aligns the output embedding and the labels through a loss function. Since previous studies always focus on tasks at the level of vertices and edges, such as node classification and link prediction, but not at the attribute level for attributed queries, the design of their loss functions also centers on the vertices. The BCE loss function in Eq. (3) is an example, which focuses on the class of each vertex. Therefore, ICS-GNN cannot incorporate the query attributes in the loss function directly and thus is not able to extend to the ACS problem.

The similar phenomenon can be observed from the QD-GNN model, which considers query vertices as the input features and propagates the query information via edges to find local structures surrounding the query vertices. But the query attributes cannot be easily incorporated as model input due to the different dimensionality. Even if we have a mechanism to take query attributes as input features, this attribute information can only propagate to adjacent vertices via graph topology by QD-GNN, but cannot reach vertices having similar attributes to the query attributes, as ACS aims to do.

The above discussions reveal that *incorporating query attributes into the learning model* and *identifying the vertices with similar attributes automatically* are two key issues to be addressed in applying GNN models into the ACS problem. In AQD-GNN, we design a bipartite graph to represent the relations between vertices and attributes. Leveraging this bipartite graph, AQD-GNN can accept an input of query attributes and translate this query attribute knowledge into vertex knowledge. Finally, AQD-GNN can find the vertices which have similar attributes with the query attributes.

6.2 Overview

AQD-GNN takes an attributed graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ and a group of attributed queries $q = \langle \boldsymbol{v}_q, f_q \rangle$ vectorized as inputs, and predicts the community vector $\boldsymbol{h}_q$ as outputs for each query q. Figure 2b illustrates the inter-layer design of AQD-GNN, which consists of a Feature Fusion operator and three GNN components: Graph Encoder, Query Encoder and Attribute Encoder. Note that Graph Encoder and Query Encoder are the same as those of QD-GNN in Section 5.2. Attribute Encoder is a new component specifically designed for ACS. Accordingly, Feature Fusion needs to be revised due to the new Attribute Encoder. In the following, we describe Attribute Encoder and the revised Feature Fusion operator.

Attribute Encoder. Attribute Encoder serves as the interface of query attributes and provides attribute information related to

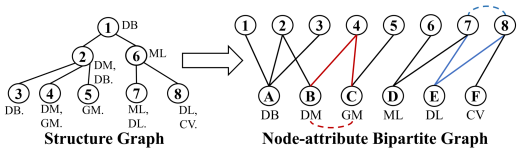


Figure 4: An example of node-attribute bipartite graph.

queries. It views each attribute as an individual vertex and models vertex attributes as a bipartite graph between vertex set $\mathcal{V}$ and attribute set $\hat{\mathcal{F}}$. With this bipartite graph, query attributes can be inputted into the attribute side directly and propagated between the vertex side and attribute side. Through this propagation, Attribute Encoder learns query-specific attribute node embeddings and identifies the vertices with attributes similar to queries.

Feature Fusion. The Feature Fusion component combines all the above embeddings and obtains the final query-specific output of the ACS problem. It takes the outputs of the three encoders as inputs, mixes global graph features and local query features, fuses structure and attribute information, and balances them to get an accurate community. Note that the final output of the entire model is the fused result in the last layer.

In the following sections, we will illustrate the detailed working mechanism of Attribute Encoder and Feature Fusion.

6.3 Attribute Encoder

The Attribute Encoder provides the interface for query attributes $\mathcal{F}_q$ and produces the vertex embeddings based on the related attributes of queries. Attribute Encoder aims to figure out the underlying relationship among different attributes and find the related attribute of queries. In addition, as analyzed in Section 6.1, Attribute Encoder needs to represent such attribute information in the form of vertices since the final output community is represented by a set of vertices.

To achieve the above goals, we model a bipartite graph called node-attribute bipartite graph $BG(\mathcal{V}, \hat{\mathcal{F}}, \mathcal{E}_B)$. For clarity, we call the vertices in the structure graph as nodes here. This bipartite graph is formed by two vertex sets: graph nodes $\mathcal{V}$ and graph attributes $\hat{\mathcal{F}}$. An edge between node v_i and attribute f_j is added to the edge set $\mathcal{E}_B$, if and only if node v_i has attribute f_j, i.e., $f_j \in \mathcal{F}_i$.

Example. *Figure 4 illustrates the node-attribute bipartite graph for the example in Figure 1. Based on the structure graph with node attributes on the left, we construct a node-attribute bipartite graph shown in Figure 4 (right), where the node set $\mathcal{V} = \{1, \ldots, 8\}$ is on the top and the attribute set $\hat{\mathcal{F}} = \{A, \ldots, F\}$ is at the bottom. Since node 4 has two attributes "DM" and "GM" in the structure graph, node 4 is adjacent to attribute B ("DM") and attribute C ("GM") as connected by red lines in Figure 4 (right).*

We apply Bipartite Graph Neural Network (BGNN) [17] on the constructed bipartite graph. BGNN consists of propagations in two directions between two vertex sets. In our node-attribute bipartite graph, the propagations are from the attribute side to the node side (denoted as A→N), and also from the node side to the attribute side (denoted as N→A).

Propagation A→N. We encode each query attribute set $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ to a one-hot vector $\boldsymbol{f}_q \in \{0,1\}^d$ as described in Section 4.1, where $d = |\hat{\mathcal{F}}|$. Benefiting from the node-attribute bipartite graph, we are able to take the query attribute vector $\boldsymbol{f}_q$ as input features in the attribute side, and propagate this attribute information from the attribute side to the node side.

Example. *When the attribute query is $\mathcal{F}_q = \{$"DL"$\}$, the one-hot vector is $\boldsymbol{f}_q = [0,0,0,0,1,0]^T$ according to the order of attribute vertices A to F in Figure 4. The query attribute information of "DL" will propagate to node 7 and node 8, the neighbors of "DL" vertex, through the blue edges in the bipartite graph of Figure 4.*

This propagation from the attribute side to the node side (A→N) collects attribute features for each node and translates the attribute features to node features. The layer-wise propagation function of A→N in BGNN is formally defined as:

$$\boldsymbol{h}_{N_u}^{(l+1)} = \text{SUM}(\{\boldsymbol{h}_{A_\text{f}}^{(l)} \boldsymbol{W}_{A\to N}^{(l+1)}, \text{f} \in \mathcal{N}_B(\text{u})\}), \tag{9}$$

where node $u \in \mathcal{V}$, attribute $\text{f} \in \hat{\mathcal{F}}$, and $\mathcal{N}_B(\text{u})$ is the neighbor set of node u in the bipartite graph. $\boldsymbol{h}_{N_u}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the hidden feature of node u in the $(l+1)$-th layer, $\boldsymbol{h}_{A_\text{f}}^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the input feature of attribute f in the l-th layer, and $\boldsymbol{W}_{A\to N}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is a learnable parameter matrix in propagation from the attribute side to the node side. The input feature of attribute f in the first layer is equal to the value of attribute f in the one-hot query attribute vector, i.e., $\boldsymbol{h}_{A_\text{f}}^{(0)} = \boldsymbol{f}_{q_\text{f}}$.

Propagation N→A. After the propagation from the attribute side to the node side in the $(l+1)$-th layer, the learned features also need to be transmitted back to form an iterative propagation in the bipartite graph. Here, we also emphasize the attribute in the last layer and add a self feature modeling [12]. Similarly, the layer-wise propagation function from the node side to the attribute side (N→A) in BGNN is defined as:

$$\boldsymbol{h}_{A_\text{f}}^{(l+1)} = \boldsymbol{h}_{A_\text{f}}^{(l)} \boldsymbol{W}_{\text{self}}^{(l+1)} + \text{SUM}(\{\boldsymbol{h}_{N_u}^{(l+1)} \boldsymbol{W}_{N\to A}^{(l+1)} : \text{u} \in \mathcal{N}_B(f)\}), \tag{10}$$

where the notations are the same as Eq. (9). $\boldsymbol{h}_{N_u}^{(l+1)}$ is the input features of node u in propagation N→A, which is learned in Eq. (9). $\mathcal{N}_B(\text{f})$ is the neighbor set of attribute f in the bipartite graph. $\boldsymbol{W}_{N\to A}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is a learnable parameter matrix in the propagation from the node side to the attribute side, and $\boldsymbol{W}_{\text{self}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the self feature parameter matrix in the $(l+1)$-th layer.

With these two propagations, Attribute Encoder can employ the query attribute as input features and transmit this attribute information through the node-attribute bipartite graph. Propagation A→N transforms the attribute features $\boldsymbol{h}_A$ into node features $\boldsymbol{h}_N$. Propagation N→A translates the node features $\boldsymbol{h}_N$ back to attribute features $\boldsymbol{h}_A$ and provides the input of propagation A→N in the next layer. With these bidirectional propagations, the features can spread in the bipartite graph and BGNN can be superimposed to multiple layers. Note that the node features $\boldsymbol{h}_N$ are the output of Attribute Encoder to Feature Fusion, since the community search problem focuses on the node and other encoders also provide node embeddings rather than attribute embeddings.

6.4 Feature Fusion

The Feature Fusion operator combines the output features of the three encoders, balances the global graph and local query knowledge, and mixes the structure and attribute information to obtain the final output of the AQD-GNN model.

The forward layer of Feature Fusion is formulated as:

$$\boldsymbol{h}_{FF_v}^{(l+1)} = \text{AGG}(\boldsymbol{h}_{G_v}^{(l+1)}, \boldsymbol{h}_{Q_v}^{(l+1)}, \boldsymbol{h}_{N_v}^{(l+1)}), \tag{11}$$

where $\boldsymbol{h}_{FF_v}^{(l+1)}$ is the fused feature of node v and also the final output of the AQD-GNN model in the $(l+1)$-th layer, $\text{AGG}(\cdot)$ is the aggregation function (e.g., Concatenation, SUM, etc.), and $\boldsymbol{h}_{G_v}^{(l+1)}$, $\boldsymbol{h}_{Q_v}^{(l+1)}$, $\boldsymbol{h}_{N_v}^{(l+1)}$ are the outputs of the three encoders in the $(l+1)$-th layer respectively. Note that $\boldsymbol{h}_{N_v}^{(l+1)}$ is the hidden features of the node side in Attribute Encoder.

In Eq. (11), we aggregate the three encoders to fuse all types of node embeddings. In order to consider the correlation between structure and attribute and process these two types of information simultaneously, we replace the input node features in the intermediate layers with the fused feature $\boldsymbol{h}_{FF}$ in Query Encoder and Attribute Encoder as shown in Figure 2b. Graph Encoder just uses the output of itself in the l-th layer $\boldsymbol{h}_G^{(l)}$ as the input of the $(l+1)$-th layer to capture the global query-independent node embeddings, as Feature Fusion does in QD-GNN. For Query Encoder, this fusion operation transmits the query-specific attribute features and global graph features into Query Encoder and delivers these features between vertices. Similar to Feature Fusion in QD-GNN, we employ $\hat{\boldsymbol{h}}_Q$ in Eq. (7) as the input features for Query Encoder, and rewrite the propagation function in Eq. (8). For Attribute Encoder, Feature Fusion enriches the features passed on the bipartite graph with local query structure and global graph features. Similar to Query Encoder, we replace the input node features in Eq. (10) with fused features when propagating from the node side to the attribute side. We define $\hat{\boldsymbol{h}}_N$ as the input node features:

$$\hat{\boldsymbol{h}}_{N_u}^{(l)} = \boldsymbol{h}_{FF_u}^{(l)}. \tag{12}$$

In this way, the structure features and attribute features learned by AQD-GNN can influence each other and these two encoders are correlated. Thus AQD-GNN is able to learn local structure and related attribute information of queries simultaneously. AQD-GNN provides an end-to-end attributed community search model, which takes queries as input and produces community vectors as answers.

6.5 Algorithm

Algorithm 3 describes the k-layer propagation of AQD-GNN. AQD-GNN first constructs the attribute matrix from $\mathcal{F}$, and builds an empty output set $\mathcal{H}$ (line 1-2). For each query, AQD-GNN constructs the one-hot vectors for both query vertex set and query attribute set, and initializes three encoders with them (line 3-6). In the first layer (line 7-11), Query Encoder propagates query vertices in the structure graph (line 7), Graph Encoder propagates vertex attributes in the graph (line 8), and Attribute Encoder propagates the query attributes from the attribute side to the node side in the bipartite graph (line 9). Feature Fusion fuses the output features of the three encoders (line 10). In the intermediate layers (line 12-18), Query Encoder propagates the fused features in graph (line 13), Graph Encoder still propagates the query-independent features from itself $\boldsymbol{h}_G$ (line 14), and Attribute Encoder utilizes the fused features $\boldsymbol{h}_{FF}$ as node side features and transmits node features back to attribute features (line 15). Then, Attribute Encoder is able to acquire the node hidden features in the next layer through propagating attribute features to the node side in the bipartite graph (line 16). Feature Fusion fuses the three encoders (line 17). The final output of AQD-GNN is the fused result in the last layer, which is added into the output set $\mathcal{H}$ (line 19).

Algorithm 3 The k-Layer AQD-GNN Propagation

Input: Graph: $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$,
a set of attributed queries: $\mathcal{Q} = \{\{\mathcal{V}_{q_1}, \mathcal{F}_{q_1}\}, \{\mathcal{V}_{q_2}, \mathcal{F}_{q_2}\}, \ldots, \}$,
AQD-GNN model: $\mathcal{M} = \{\boldsymbol{h}_Q, \boldsymbol{h}_G, \{\boldsymbol{h}_N, \boldsymbol{h}_A\}, \boldsymbol{h}_{FF}\}$.
Output: a set of output vectors: $\mathcal{H} = \{\boldsymbol{h}_{q_1}, \boldsymbol{h}_{q_2} \ldots, \}$.

1: Construct attribute matrix $\boldsymbol{F}$ for $\mathcal{F}$
2: $\mathcal{H} \leftarrow \emptyset$
3: **for** each $\{\mathcal{V}_q, \mathcal{F}_q\} \in \mathcal{Q}$ **do**
4: Construct one-hot vector $\boldsymbol{v}_q$ for $\mathcal{V}_q$, initialize $\boldsymbol{h}_Q^{(0)}$ with $\boldsymbol{v}_q$
5: Initialize $\boldsymbol{h}_G^{(0)}$ with $\boldsymbol{F}$
6: Construct one-hot vector $\boldsymbol{f}_q$ for $\mathcal{F}_q$, Initialize $\boldsymbol{h}_A^{(0)}$ with $\boldsymbol{f}_q$
7: $\boldsymbol{h}_Q^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_Q^{(0)}, \mathcal{E})$ in Eq. (4)
8: $\boldsymbol{h}_G^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(0)}, \mathcal{E})$ in Eq. (5)
9: $\boldsymbol{h}_N^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_A^{(0)}, \mathcal{E}_B)$ in Eq. (9)
10: $\boldsymbol{h}_{FF}^{(1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(1)}, \boldsymbol{h}_Q^{(1)}, \boldsymbol{h}_N^{(1)})$ in Eq. (11)
11: $l \leftarrow 1$
12: **while**$(l < k)$ **do**
13: $\boldsymbol{h}_Q^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E})$ in Eq. (8)
14: $\boldsymbol{h}_G^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(l)}, \mathcal{E})$ in Eq. (5)
15: $\boldsymbol{h}_A^{(l)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E}_B)$ in Eq. (10)
16: $\boldsymbol{h}_N^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_A^{(l)}, \mathcal{E}_B)$ in Eq. (9)
17: $\boldsymbol{h}_{FF}^{(l+1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(l+1)}, \boldsymbol{h}_Q^{(l+1)}, \boldsymbol{h}_N^{(l+1)})$ in Eq. (11)
18: $l \leftarrow l + 1$
19: $\mathcal{H} \leftarrow \mathcal{H} \cup \boldsymbol{h}_{FF}^{(k)}$
20: **return** $\mathcal{H}$;

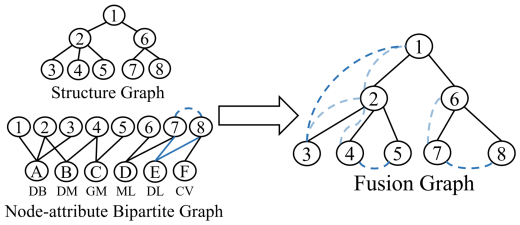


Figure 5: An example of fusion graph.

As described in Section 4, in the training stage, the output set $\mathcal{H}$ is used in the loss function to optimize the model learning. In the online query stage, the output is translated to the predicted communities through the Community Identification process as described below.

6.6 Community Identification

For the attributed community search problem, we need to find vertices having both dense structure and similar attributes to the query. Thus on top of the online query stage described in Section 4.3 which ensures connectivity with the query vertices, we also enhance the connectivity between graph vertices sharing identical attributes by a fusion graph $G_F = \{\mathcal{V}, \mathcal{E}_F\}$ which combines the information of structure graph $G = \{\mathcal{V}, \mathcal{E}\}$ and bipartite graph $G_B = \{\mathcal{V}, \hat{\mathcal{F}}, \mathcal{E}_B\}$.

To build the fusion graph, we link vertices with the same attributes in the structure graph. The connectivity in the fusion graph represents both the structure connectivity and attribute similarity. Then the fusion graph G_F is fed to Algorithm 1 for a constrained BFS with the model output $\boldsymbol{h}_q$ for community identification.

Example. *Figure 5 shows the fusion graph for our running example. We add a dashed blue edge between two vertices in the structure*

graph if they have the same attribute, e.g., vertices 7 and 8 are connected by a dashed blue edge because they both have attribute "DL".

6.7 Complexity Analysis

In order to analyze the time complexity of QD-GNN and AQD-GNN, we first present the complexity of general GNN in Eq.(1). This GNN aggregates neighbors' features for every vertex with the cost of $\sum_{i=\mathrm{u}}^{n} d_{\mathrm{u}}$, where d_{u} is the degree of vertex u and n is number of vertices. Thus the complexity of general GNN is $O(|\mathcal{E}|)$.

For QD-GNN, Query Encoder and Graph Encoder have the same time complexity of $O(|\mathcal{E}|)$ as general GNN. For AQD-GNN, the time cost of Attribute Encoder is also dependent on the sum of vertices' degree in the bipartite graph with the complexity of $O(|\mathcal{E}_B|)$. The aggregation operation in Feature Fusion, e.g., MAX, Concatenation, etc., is implemented in parallel and the complexity is just $O(1)$. Suppose QD-GNN or AQD-GNN is a k-layer model with t iterations, where $k = 3$ and $t = 300$ are typical settings. The complexity of QD-GNN is $O(k \times t \times |\mathcal{E}|)$ in the model training stage and $O(k \times |\mathcal{E}|)$ in the online query stage. Similarly, the complexity of AQD-GNN is $O(k \times t \times (|\mathcal{E}| + |\mathcal{E}_B|))$ in the model training stage and $O(k \times (|\mathcal{E}| + |\mathcal{E}_B|))$ in the online query stage.

7 EXPERIMENTS

In this section, we present our experimental studies to validate the performance of our framework with the three proposed models in different scenarios. We first introduce the setup of our experiment in Section 7.1. Then we evaluate the performance in both attributed and non-attributed community search problem in Section 7.2. To further verify the effectiveness of our models, we compare our models with the interactive community search model ICS-GNN in Section 7.3. Moreover, we evaluate the performance of our model for ACS on large graphs in Section 7.4. Finally, we conduct the ablation study in Section 7.5 to demonstrate the effectiveness of Feature Fusion, the sensitivity test of the parameter γ, and the data split ratio.

7.1 Experimental Setup

7.1.1 Data Sets. To thoroughly evaluate the performance of our framework, we conduct experimental studies on 15 attributed graphs. Table 1 reports the dataset statistics. The first six networks, Cornell, Texas, Washington (Washt), Wisconsin (Wiscs), Cora and Citeseer, are publication citation networks. Each attribute describes the absence/presence of one word in a publication. All these graphs can be found at LINQS website[1]. Reddit [16] is an online discussion website where each vertex is a post and an edge links two posts if they have comments from the same user. Facebook [33] is a social network where vertices are users and edges are friend relationships. It contains 8 ego-networks with different attributes as shown in Table 1. We consider each ego-network as an independent data set. All data sets contain ground-truth communities.

7.1.2 Baseline Models. We compare our models with five state-of-the-art approaches, including two non-attributed community search algorithms: CTC [24] and k-ECC [6], two attributed community search algorithms: ACQ [10] and ATC [22], and a GNN-based interactive community search model ICS-GNN [14].

[1]https://linqs.soe.ucsc.edu/data

Table 1: Dataset Statistics. $|\mathcal{V}|$ and $|\mathcal{E}|$ are the number of vertices and edges. $|\hat{\mathcal{F}}|$ is the number of distinct attributes, K is the number of communities and AS is the average size of communities. Here, M= 10^6.

Data set		$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\hat{\mathcal{F}}\rvert$	$\lvert\mathcal{E}_B\rvert$	K	AS
Citation Networks	Cornell	195	283	1703	18496	5	39
	Texas	187	280	1703	15437	5	37.4
	Washt	230	366	1703	19953	5	46
	Wiscs	265	459	1703	25479	5	53
	Cora	2708	5278	1433	49216	7	386.86
	Citeseer	3312	4536	3703	105165	6	552
Social Networks	Reddit	232965	114**M**	602	140**M**	50	4659.3
	FB-0	348	2852	224	3348	24	13.54
	FB-107	1046	27783	576	11827	9	55.67
	FB-1684	793	14810	319	6131	17	45.71
	FB-1912	756	30772	480	8066	46	23.15
	FB-3437	548	5347	262	4263	32	6
	FB-348	228	3416	161	2398	14	40.5
	FB-414	160	1843	105	1566	7	25.43
	FB-686	171	1824	63	999	14	34.64

7.1.3 Query Setting. For each data set, we generate $n_q = 350$ pairs of input query set $Q = \{< \mathcal{V}_q, \mathcal{F}_q >\}_{q=1}^{n_q}$ and the corresponding ground-truth community $\mathcal{Y}_q$. To generate the query vertex set $\mathcal{V}_q$, vertex sets containing 1-3 vertices are randomly selected from the ground-truth community. To generate the query attribute set $\mathcal{F}_q$, we design three different types as described below for fair comparison with different existing CS and ACS methods. The query vertex set and corresponding ground-truth communities are shared across the three types of input queries.

- **Empty attribute query (EmA)**. To compare with methods for non-attributed community search, we set the attribute query set empty ($\mathcal{F}_q = \emptyset$) and generate the EmA set $Q_{\mathrm{EmA}} = \{< \mathcal{V}_q, \emptyset >\}$.
- **Attribute from community (AFC)**. As suggested by [10, 22], to construct the query attribute set ($\mathcal{F}_q = \mathcal{F}_q^{\mathrm{c}}$), we use 5 most common attributes in ground-truth communities. Therefore, we have $Q_{\mathrm{AFC}} = \{< \mathcal{V}_q, \mathcal{F}_q^{\mathrm{c}} >\}$. AFC is used to validate the contribution of the attributes in the community search.
- **Attribute from node (AFN)**. We simulate real queries provided by users and select 5 most common attributes from attributes of query vertices as the query attribute set, i.e, $\mathcal{F}_q = \mathcal{F}_q^{\mathrm{n}}$. In other words, $\mathcal{F}_q^{\mathrm{n}}$ may be unrelated to the ground-truth communities. We construct the AFN set $Q_{\mathrm{AFN}} = \{< \mathcal{V}_q, \mathcal{F}_q^{\mathrm{n}} >\}$. Obviously, AFN is a more challenging setting and closer to the real scenarios.

7.1.4 Data Splitting. For each data set, we split 350 query-community pairs into training data, validation data and test data with the ratio of 150:100:100 by default. We use training data to train our models, validation data to select the best weights during the training process, and test data to measure the performance of all methods. In the ablation study, we vary the data splitting ratio to evaluate its influence on the performance.

7.1.5 Evaluation Metrics. Let $D_{\mathrm{test}} = \{Q, \hat{C}, \mathcal{Y}\}$ be the test data set, where Q is the query set, $\hat{C}$ is the predicted community set by a method and $\mathcal{Y}$ is the ground-truth community set. To measure the quality of communities found by different methods, we employ F1-score to evaluate the quality of the predicted set $\hat{C}$. F1-score is defined as:

$$F1(\hat{C}, \mathcal{Y}) = \frac{2 \cdot pre(\hat{C}, \mathcal{Y}) \cdot rec(\hat{C}, \mathcal{Y})}{pre(\hat{C}, \mathcal{Y}) + rec(\hat{C}, \mathcal{Y})}$$

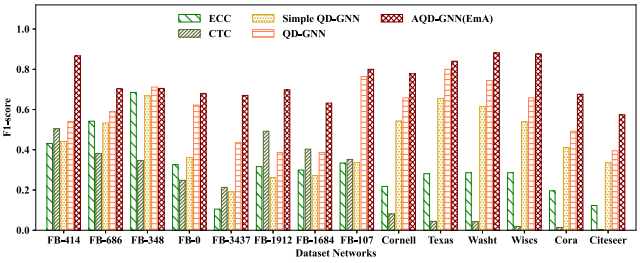


Figure 6: Non-attributed community search performance comparison.

where $pre(\hat{C}, \mathcal{Y})$ is the precision of predicted community set $\hat{C}$ on the ground-truth community set $\mathcal{Y}$, $rec(\hat{C}, \mathcal{Y})$ is the recall of the predicted communities:

$$pre(\hat{C}, \mathcal{Y}) = \frac{\sum_{c_q \in \hat{C}} \boldsymbol{c}_q \& \boldsymbol{y}_q}{\sum_{c_q \in \hat{C}} \sum_{i=0}^{n} \boldsymbol{c}_{qi}}, rec(\hat{C}, \mathcal{Y}) = \frac{\sum_{c_q \in \hat{C}} \boldsymbol{c}_q \& \boldsymbol{y}_q}{\sum_{\boldsymbol{y}_q \in \mathcal{Y}} \sum_{i=0}^{n} \boldsymbol{y}_{qi}}.$$

Here, $\boldsymbol{c}_q \in \{0,1\}^{n \times 1}$ and $y_q \in \{0,1\}^{n \times 1}$ are the predicted and ground-truth community vectors for query q respectively.

7.1.6 Implementation Details. In our models, we build three layers with 128 neurons in the hidden layer. We train 300 iterations with a learning rate of 0.001. In the Feature Fusion component, we choose concatenate as the aggregation function in Eq. (6) and Eq. (11). In each layer except the output layer, we employ ReLU as activation function, batch normalization with batch size 4 and dropout rate 0.5 [38] for each branch.

7.2 Community Search Performance

We present comprehensive experiments to validate the query performance of the three proposed models under two settings: non-attributed community search, and attributed community search.

7.2.1 Non-attributed community search. In order to compare to non-attributed community search algorithms, we generate the multi-vertex queries set without query attributes $\mathcal{Q}_{\text{EmA}}$, and compare our three models Simple QD-GNN, QD-GNN, AQD-GNN with CTC and k-ECC. Figure 6 shows the F1-score. We can observe that:

- CTC performs reasonably well in Facebook ego networks but poorly in citation networks, since CTC searches communities using the k-truss subgraph pattern, which may fit the dense social networks well, but does not fit the sparser citation networks.
- By capturing the local query structure only, Simple QD-GNN can outperform ECC and CTC in citation networks and achieve comparable performance in Facebook ego networks. It demonstrates the learning-based models can apply to different types of networks and discover communities with different structural properties.
- QD-GNN can substantially outperform Simple QD-GNN by improving the F1-score by 0.14 on average. It validates the effectiveness of the query-independent graph features learned from Graph Encoder.
- We also apply AQD-GNN to non-attributed community search, where we set the query attribute set to empty, $\mathcal{F}_q = \emptyset$. Interestingly, AQD-GNN can achieve the best performance in almost all data sets in Figure 6. This is owing to the Feature Fusion operator and Attribute Encoder design in AQD-GNN. Specifically, Feature Fusion can transmit graph information and query vertices information to Attribute Encoder before the second layer. Then Attribute Encoder can utilize the information from the second layer and learn hidden relations between attributes.

7.2.2 Attributed community search. We compare AQD-GNN with two attributed community search algorithms: ACQ and ATC. ACQ can only handle one query vertex while ATC and our model AQD-GNN can handle multiple query vertices. Thus we compare ACQ and AQD-GNN for one-vertex queries in Figure 7a, and compare ATC and AQD-GNN for multi-vertex queries in Figure 7b. We can observe that:

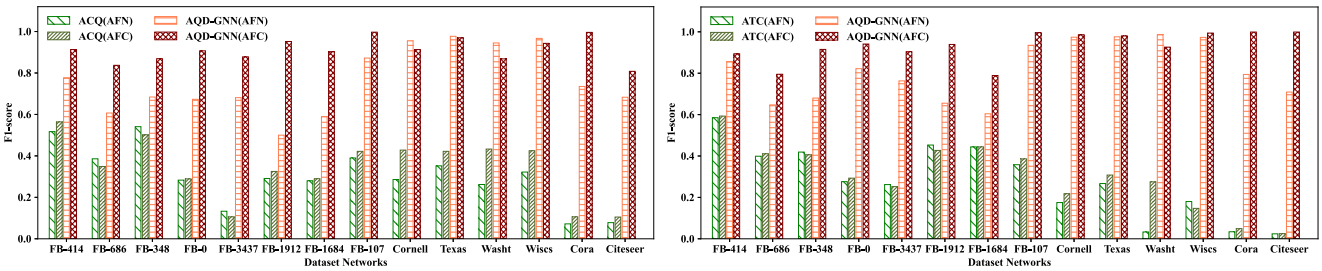
- For Cora and Citeseer with large ground-truth communities (hundreds of vertices in a community), the performances of ATC and ACQ are quite poor (around 0.1 in F1-score). It is because their pre-defined community patterns (i.e., k-core and k-truss) are too strict to find large communities in the real-world graphs.
- AQD-GNN consistently performs the best on all data sets. As a data-driven approach, AQD-GNN is capable of learning communities with varied sizes and shapes. It performs stably on all graphs benefiting from learning adaptive weight matrices for different data sets.
- Compared to AFC, all methods suffer from performance degradation under the AFN setting in most data sets, since AFC is a more favorable setting where the query attribute set is directly extracted from the most common attributes of the ground-truth. For example, in the Washington data set in Figure 7b, ATC achieves 0.275 F1-score under AFC, but only 0.033 under AFN.
- Under the more realistic but challenging AFN setting, we can observe AQD-GNN achieves a significant performance improvement over the baselines, with 0.46 and 0.53 improvements on F1-score for one-vertex queries and multi-vertex queries respectively. This is because AQD-GNN exploits the node-attribute bipartite graph to find similar attributes, while the baselines simply require vertices in a community have identical attributes with query attributes.

7.2.3 Query Efficiency. We evaluate the query efficiency of AQD-GNN in the test set. Table 2 shows the average query time (in milliseconds) of 100 test queries by AQD-GNN and baselines. The last column reports the average query time among all data sets.

Overall, the query time AQD-GNN is much faster than that of all baselines except ACQ. ACQ is a simple baseline which only allows one query vertex and considers vertices' degrees and common attributes. Its query performance in terms of F1-score is very poor as shown in Figure 7. It is worth noting that AQD-GNN achieves a stable query time of around 5 milliseconds on all data sets, while the query time of CTC, ECC and ATC increases significantly when the graph is large. In particular, CTC takes almost 5, 000 milliseconds for a query on FB-1912, while AQD-GNN only costs 4.96 milliseconds. This experiment shows that AQD-GNN is more suitable for online search in real-world applications.

7.3 Interactive Community Search

ICS-GNN [14] is a recent GNN-based model for interactive community search. Given a query, ICS-GNN returns an answer community. If the user is not satisfied with the answer, he/she can give a feedback (e.g., adding some additional vertices), and then ICS-GNN will respond with a revised answer. This interaction continues until



(a) Compared with methods supporting one-vertex queries. **(b) Compared with methods supporting multi-vertex queries.**

Figure 7: Attributed community search performance compared with other approaches.

Table 2: Average query time (in milliseconds) of different community search methods.

Methods		FB-414	FB-686	FB-348	FB-0	FB-3437	FB-1912	FB-1684	FB-107	Cornell	Texas	Washt	Wiscs	Cora	Citeseer	Average
Non-Attributed	CTC	34.78	41.41	120.92	49.25	131.67	4903.38	604.67	2498.82	0.45	0.40	0.42	0.63	1.96	1.28	599.00
	ECC	3.52	2.29	5.20	2.57	6.60	154.23	26.85	93.62	0.24	0.28	0.23	0.33	2.67	1.76	21.50
Attributed	ACQ	<0.01	<0.01	1.45	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.10
	ATC	4.40	5.10	7.70	5.90	10.30	43.60	22.70	40.10	7.90	14.02	2.11	18.60	11.49	3.68	9.80
	AQD-GNN	3.31	3.32	3.41	3.63	4.32	4.96	4.56	5.46	4.15	4.10	4.16	4.41	5.54	5.32	4.31

Table 3: F1-score (in %) and time cost (in seconds) of interactive community search methods on different networks.

Method	FB-414		FB-686		FB-348		FB-0		FB-3437		FB-1912		FB-1684		FB-107		Cora		Citeseer		Reddit		Average	
	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time
ICS-GNN	56.63	0.14	43.53	0.15	33.88	0.26	24.94	0.26	26.49	0.51	20.23	2.36	20.76	1.28	36.46	2.40	30.52	0.14	30.29	0.14	19.41	1.84	31.19	0.86
QD-GNN	**62.02**	0.14	**44.28**	0.14	34.74	0.26	31.07	0.27	27.38	0.52	21.10	2.45	24.79	1.31	38.39	2.49	32.56	0.12	31.53	0.12	21.29	1.68	$33.56_{+2.37}$	$0.87_{+0.01}$
AQD (AFN)	61.25	0.14	43.05	0.14	**35.80**	0.25	31.27	0.26	29.03	0.50	20.44	2.29	36.51	1.23	41.07	2.41	31.81	0.14	**33.09**	0.12	20.71	1.85	$34.91_{+3.72}$	$0.85_{-0.01}$
AQD (AFC)	57.34	0.14	38.87	0.14	35.35	0.25	**35.63**	0.26	**30.22**	0.50	**37.52**	2.29	**37.91**	1.23	**49.67**	2.41	**33.19**	0.14	31.77	0.12	**24.86**	1.85	$\mathbf{37.48}_{+6.29}$	$0.85_{-0.01}$

the user is satisfied. In each interaction, ICS-GNN first finds a candidate subgraph, learns the vertex embedding through a Vanilla GCN model [27] and finally employs a BFS based algorithm to select k-sized community with the maximum GNN scores. Note that ICS-GNN does not use any training queries with ground-truth communities to train the model; for each user query, it re-trains the GNN model to obtain the vertices' embeddings only from the knowledge of the given query. ICS-GNN only supports non-attributed community search.

In this experiment, we replace the Vanilla GCN model [27] in the ICS-GNN [14] with our community search models QD-GNN and AQD-GNN to compare the performance of interactive community search problem.

7.3.1 Performance in Effectiveness. We first use QD-GNN to replace the GNN model in ICS-GNN framework for non-attributed community search. As shown in Table 3, QD-GNN outperforms the original ICS-GNN in all data sets with 2.37% improvement in F1-score. We also use AQD-GNN to replace the GNN model in ICS-GNN so it supports interactive attributed community search. As shown in Table 3, AQD-GNN further improves the F1-score of the original ICS-GNN for all data sets by 3.72% (AFN) and 6.29% (AFC) on average. This experiment proves that our QD-GNN and AQD-GNN models are more effective than Vanilla GCN in the ICS-GNN framework.

7.3.2 Performance in Efficiency. We report the average time of community search per interaction by ICS-GNN, QD-GNN and AQD-GNN in Table 3. The running time of the three models are very close. Without increasing the time cost, we improve the performance of ICS-GNN and extend it to support attributed interactive community search problem.

Table 4: The performances of ACS methods on large data sets.

Methods	Reddit			Enlarged_Reddit		
	Index/Train Time	Query Time	F1-score	Index/Train Time	Query Time	F1-score
ACQ	**42.4 s**	32.2 ms	0.53	**852.7 s**	5726.6 ms*	0.38
ATC #	-	-	-	-	-	-
AQD-GNN	4993.6 s	**6.7 ms**	**0.91**	3898.5 s	**5.3 ms**	**0.91**

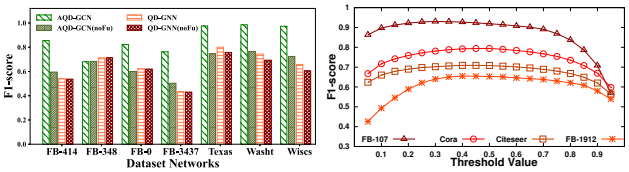
* 25 out of 100 queries are out of memory when processing. The query time are the average of the rest 75 queries.
\# ATC did not finish building its index in 7 days on both two data sets.

7.4 ACS on Large Graphs

In this experiment, we evaluate the performance of our model for ACS on large graphs. We design a subgraph training mechanism to train our models on large graphs. We first select neighbors of query vertices as the candidate subgraph for each query. According to the number of neighbors, we select 1 or 2-hop neighbors in the fusion graph described in Section 6.6. Then we train our model on these small subgraphs and predict communities.

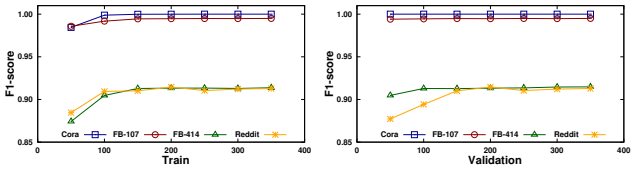
We compare AQD-GNN with ACQ and ATC for attributed community search on Reddit and an enlarged version of Reddit, denoted as Enlarged_Reddit. To enlarge Reddit and preserve the ground-truth communities at the same time, we add some new vertices for edges within a community. A new vertex is linked to the two ends of an edge, and the attributes of the new vertex are the average attribute values of the two ends. The Enlarged_Reddit has 3.12M vertices and 126M edges.

Table 4 reports the index/training time, query time and F1-score of the discovered communities. ACQ takes only 42.4 seconds and 852.7 seconds to build index on Reddit and Enlarge_Reddit. But in terms of the query time, it costs 32.2 milliseconds and 5726.6 milliseconds, while AQD-GNN only costs 6.7 milliseconds and 5.3



(a) F1-score w/wo Feature Fusion. (b) F1-score by varying γ.

Figure 8: Ablation studies for Feature Fusion and γ.



(a) Vary training set size. (b) Vary validation set size.

Figure 9: Ablation study for data split ratio.

milliseconds respectively. It is worth noting that ACQ runs out of memory for 25 out of 100 queries on a 300GB memory server. This is because ACQ finds a k-core community with the largest k containing the query vertices. The k-core community can be quite large, for example, for a query vertex, ACQ first finds a 2-core community with more than 800 thousand candidate vertices. The average F1-score of ACQ is much lower than that of our method in both data sets. ATC did not finish building its index in 7 days and we treat it as timed out. From this experiment, we can see that AQD-GNN achieves a good balance between training time and query time in large graphs, and its F1-score is the best.

7.5 Ablation Study

In this section, we report the ablation studies of our models, including the effectiveness of Feature Fusion, the sensitivity test of the parameter γ, and the data split ratio in the attributed community search task.

7.5.1 Ablation Study for Feature Fusion. In our model, we use the aggregation result $\boldsymbol{h}_{FF}$ in Eq. (6) in QD-GNN and Eq. (11) in AQD-GNN to fuse all information, and assign fused features to Query Encoder and Attribute Encoder in Eq. (7) and Eq. (12). To verify the effectiveness of Feature Fusion, we compare the original AQD-GNN and QD-GNN models with AQD-GNN-noFu and QD-GNN-noFu where the encoders do not aggregate in the hidden layer. They only aggregate after the last layer to output the final results.

The comparison results are shown in Figure 8a. For the non-attributed community search problem, the effect of Feature Fusion is more significant in citation networks than that in Facebook ego networks. In QD-GNN, Feature Fusion aims to mix the global graph information and local query knowledge. The Facebook ego networks themselves are local graphs. Therefore, Feature Fusion in QD-GNN can only improve the model slightly on Facebook ego networks. For ACS problem, AQD-GNN outperforms AQD-GNN-noFu substantially in both Facebook ego networks and citation networks. This is because Feature Fusion in AQD-GNN not only fuses the global graph feature, local query structure and similar attribute information at the same time, but also processes query vertices and query attributes simultaneously through the updating of Query Encoder and Attribute Encoder by fused features. This fusion and updating operations significantly improve the results.

7.5.2 Ablation Study for the threshold γ. When translating the model output vector h_q from $\mathbb{R}^n$ to the community vertex set $\hat{C}_q$ in Section 4.3, we use a threshold $\gamma \in [0, 1]$ in the constrained BFS in Algorithm 1: if $h_{qi} \geq \gamma$, then vertex $v_i \in \hat{C}$, otherwise $v_i \notin \hat{C}$. In above experiments, we choose γ which achieves the best performance in the validation set. To analyze the impact of the threshold, we vary γ from 0.05 to 0.95 and report the F1-score in Figure 8b. When γ is between 0.3 and 0.7, there is very little fluctuation in the performance. Therefore, AQD-GNN is not very sensitive to the selecting of this threshold γ.

7.5.3 Ablation Study for the data split ratio. In all the experiments above, we fix the training/validation/test size ratio as 150:100:100. To test the sensitiveness of data split ratio, we vary the training set size from 50 to 350, and fix both the validation and test set size as 100. The results are plotted in Figure 9a. We also vary the validation set size and fix the training set size as 150 and the test set size as 100. The results are plotted in Figure 9b.

When the training set size increases from 50 to 100, the F1-score of all data sets has a notable increase, but when the training set size further increases from 100 to 350, the F1-score remains quite stable. When varying the validation set size, for Cora and FB-107 the F1-score remains stable; for FB-414 and Reddit, the F1-score increases when the validation set size increases from 50 to 100, and then remains stable afterwards.

This experiment shows that when the training/validation set is very small, increasing the size can improve the performance; but when the size is above 100, the performance remains stable.

8 CONCLUSIONS

In this paper, we propose the QD-GNN and AQD-GNN for non-attributed community search and attributed community search respectively. In QD-GNN, we first propose a query-driven component to acquire queries directly and avoid the re-training process in the existing GNN-based community search model ICS-GNN. Then we combine the local query-dependent structure and global query-independent vertex embedding. For attributed community search, we model vertex attributes as a bipartite graph and further propose the AQD-GNN model. To the best of our knowledge, AQD-GNN is the first GNN model for attributed community search. Moreover, we apply QD-GNN and AQD-GNN in the framework of ICS-GNN for interactive attributed community search. Experiments demonstrate that the proposed models outperform previous approaches significantly. The proposed models are trained through historical queries (training queries), then applied for online query. In the future, more research on training query selection can be carried out to train the model with limited training queries for large graphs. In addition, as time goes by, more historical queries can be collected and the model can be updated with them as training queries to improve its performance. The model update mechanism is worth further study.

ACKNOWLEDGMENTS

The work was supported by grants from NSFC Grant No. U1936205, the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205618], Tencent AI Lab RhinoBird Focused Research Program GF202101, and CUHK Direct Grant No. 4055159. Additional funding was provided by the HK RGC Grant Nos. 22200320 and 12200021.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *PVLDB* 10, 11 (2017), 1298–1309.

[2] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *ICML*. 695–704.

[3] Heng Chang, Yu Rong, Tingyang Xu, Yatao Bian, Shiji Zhou, Xin Wang, Junzhou Huang, and Wenwu Zhu. 2021. Not All Low-Pass Filters are Robust in Graph Convolutional Networks. *NeurIPS* 34 (2021).

[4] Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Somayeh Sojoudi, Junzhou Huang, and Wenwu Zhu. 2021. Spectral graph attention network with fast eigen-approximation. In *CIKM*. 2905–2909.

[5] Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Honglei Zhang, Peng Cui, Wenwu Zhu, and Junzhou Huang. 2020. A restricted black-box adversarial framework towards attacking graph embedding models. In *AAAI*, Vol. 34. 3389–3396.

[6] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based optimal algorithms for computing Steiner components with maximum connectivity. In *SIGMOD*. 459–474.

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

[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.

[9] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. 2020. GraphZoom: A multi-level spectral approach for accurate and scalable graph embedding. *ICLR* (2020).

[10] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[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. *VLDBJ* 29, 1 (2020), 353–392.

[12] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In *NeurIPS*. 6530–6539.

[13] Hongyang Gao and Shuiwang Ji. 2019. Graph U-Nets. In *ICML*. 2083–2092.

[14] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: lightweight interactive community search via graph neural network. *PVLDB* 14, 6 (2021), 1006–1018.

[15] Arushi Goel, Keng Teck Ma, and Cheston Tan. 2019. An End-to-End Network for Generating Social Relationship Graphs. In *CVPR*. 11186–11195.

[16] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*. 1025–1035.

[17] Chaoyang He, Tian Xie, Yu Rong, Wenbing Huang, Junzhou Huang, Xiang Ren, and Cyrus Shahabi. 2019. Cascade-bgnn: Toward efficient self-supervised representation learning on large-scale bipartite graphs. *arXiv preprint arXiv:1906.11994* (2019).

[18] Chaoyang He, Tian Xie, Yu Rong, Wenbing Huang, Yanfang Li, Junzhou Huang, Xiang Ren, and Cyrus Shahabi. [n.d.]. Bipartite graph neural networks for efficient node representation learning. *arXiv preprint arXiv:1906.11994* ([n. d.]).

[19] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying minimal steiner maximum-connected subgraphs in large graphs. In *CIKM*. 1241–1250.

[20] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *NeurIPS*. 4558–4567.

[21] 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.

[22] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *PVLDB* 10, 9 (2017), 949–960.

[23] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2017. Community search over big graphs: Models, algorithms, and opportunities. In *ICDE*. 1451–1454.

[24] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate closest community search in networks. *PVLDB* 9, 4 (2015), 276–287.

[25] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*. PMLR, 448–456.

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

[28] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. 2019. Self-Attention Graph Pooling. In *ICML*. 3734–3743.

[29] Jia Li, Yu Rong, Hong Cheng, Helen Meng, Wen-bing Huang, and Junzhou Huang. 2019. Semi-Supervised Graph Classification: A Hierarchical Graph Perspective. In *WWW*. 972–982.

[30] Ye Li, Chaofeng Sha, Xin Huang, and Yanchun Zhang. 2018. Community Detection in Attributed Graphs: An Embedding Approach. In *AAAI*. 338–345.

[31] Hehuan Ma, Yatao Bian, Yu Rong, Wenbing Huang, Tingyang Xu, Weiyang Xie, Geyan Ye, and Junzhou Huang. 2022. Cross-Dependent Graph Neural Networks for Molecular Property Prediction. *Bioinformatics* (2022).

[32] Yao Ma, Suhang Wang, Charu C. Aggarwal, and Jiliang Tang. 2019. Graph Convolutional Networks with EigenPooling. In *KDD*. 723–731.

[33] Julian J. McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NeurIPS*. 548–556.

[34] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. 2020. Self-supervised graph transformer on large-scale molecular data. *NeurIPS* 33 (2020), 12559–12571.

[35] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *ICLR*.

[36] Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, and Bowen Zhou. 2019. End-to-End Structure-Aware Convolutional Networks for Knowledge Base Completion. In *AAAI*. 3060–3067.

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

[38] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR* 15, 1 (2014), 1929–1958.

[39] Damian Szklarczyk, Andrea Franceschini, Stefan Wyder, Kristoffer Forslund, Davide Heller, Jaime Huerta-Cepas, Milan Simonovic, Alexander Roth, Alberto Santos, Kalliopi P Tsafou, et al. 2015. STRING v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic acids research* 43, D1 (2015), D447–D452.

[40] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. KGAT: Knowledge Graph Attention Network for Recommendation. In *KDD*. 950–958.

[41] Xiao Wang, Meiqi Zhu, Deyu Bo, Peng Cui, Chuan Shi, and Jian Pei. 2020. AM-GCN: Adaptive Multi-channel Graph Convolutional Networks. In *SIGKDD*. 1243–1253.

[42] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. 2019. A Vectorized Relational Graph Convolutional Network for Multi-Relational Network Alignment. In *IJCAI*. 4135–4141.

[43] Junchi Yu, Tingyang Xu, Yu Rong, Yatao Bian, Junzhou Huang, and Ran He. 2021. Graph Information Bottleneck for Subgraph Recognition. In *ICLR*.

[44] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *TKDE* 30, 5 (2017), 922–935.

[45] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *SIGMOD*. 2142–2155.

[46] Kangfei Zhao, Zhiwei Zhang, Yu Rong, Jeffrey Xu Yu, and Junzhou Huang. 2021. Finding critical users in social communities via graph convolutions. *TKDE* (2021).

[47] Dingyuan Zhu, Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2019. Robust Graph Convolutional Networks Against Adversarial Attacks. In *KDD*. 1399–1407.