Context-Aware Object Connection Discovery in Large Graphs

James Cheng #1, Yiping Ke *2, Wilfred Ng #3, Jeffrey Xu Yu *4

#*Hong Kong University of Science and Technology*
1csjames@cse.ust.hk
3wilfred@cse.ust.hk

**Chinese University of Hong Kong*
2ypke@se.cuhk.edu.hk
4yu@se.cuhk.edu.hk

***Abstract*— Given a large graph and a set of objects, the task of *object connection discovery* is to find a subgraph that retains the best connection between the objects. Object connection discovery is useful to many important applications such as discovering the connection between different terrorist groups for counter-terrorism operations. Existing work considers only the connection between individual objects; however, in many real problems, the objects usually have a *context* (e.g., a terrorist belongs to a terrorist group). We identify the context for the nodes in a large graph. We partition the graph into a set of communities based on the concept of *modularity*, where each community becomes naturally the context of the nodes within the community. By considering the context, we also significantly improve the efficiency of object connection discovery, since we break down the big graph into much smaller communities. We first compute the best *intra-community* connection by maximizing the amount of information flow in the answer graph. Then, we extend the connection to the *inter-community* level by utilizing the community hierarchy relation, where the quality of the inter-community connection is also ensured by modularity. Our experiments show that our algorithm is three orders of magnitude faster than the state-of-the-art algorithm, while the quality of the answer graph is comparable.**

I. Introduction

Graph is a powerful modeling tool for representing and understanding objects and their relationships. Many efficient indexes [1], [2], [3] and algorithms for the discovery of correlations [4], [5] and frequent subgraphs [6], [7], [8] have been proposed for graph databases that consist of a set of small graphs. Such graph databases are popular in scientific domains such as chemistry and bio-informatics. However, we also observe that another type of graphs, which are stand-alone large graphs, exists prevalently in a wide spectrum of application domains. Typical examples of such graphs are various social networks, in which we want to find the relationship between people within the network. Such information is very useful in applications such as corporate cooperation and management, law enforcement, national security, counter-terrorism, and many others. Other examples of such graphs include resource environment economics, ecological networks, the Web and many more. However, efficient knowledge discovery techniques for such graphs are still inadequate.

Let $\mathcal{G}$ be a graph in which nodes are uniquely labeled and edges are undirected and weighted. Given a set of objects, a typical task of knowledge discovery is to find the "best" *connection among the objects in $\mathcal{G}$.* We name such a task as *object connection discovery*. We use *objects* and *nodes* interchangeably in this paper and name the set of objects under investigation as *query nodes*.

Object connection discovery is important to a wide spectrum of applications. For example, the relationship between different business people and organizations, and how they do business together, are important for creating new opportunities of corporate cooperations and for designing more effective internal management strategies; the connection between different terrorists and terrorist leaders/groups, and how terrorists plan and conduct terrorist activities together, are useful clues for counter-terrorism and national security operations; how various social networks behave is valuable knowledge for understanding the networks so that appropriate actions may be taken. It can also be applied to other domains such as resource environment economics, gene regulatory networks, viral marketing, the Internet and the WWW, and many more.

However, object connection discovery in a large graph poses significant challenges. First, the graph may contain millions or even billions of nodes and edges; therefore, discovering the best connection is difficult, especially when the query nodes are scattered widely apart from each other. Existing work on finding a connection between a set of query nodes [9], [10], [11] is either not efficient enough or focuses on the query nodes that are relatively close to each other.

Apart from computational complexity, it is also difficult to define the "*bestness*" of a connection. It has been shown in [9] that shortest paths, maximum flow and other graph distance measures are inadequate in capturing the relation between query nodes. Existing work measures the "bestness" by the concepts of *electrical circuits* [9] and *random walks with restart* [10]. However, their studies focus on computing the connection from query nodes to some important nodes, which may neglect the good connection from query nodes to query nodes.

We address the problem of object connection discovery by first observing the semantics of the objects in a large graph. Since a large graph can contain millions of nodes, it is extremely rare that they all belong to the same category or the

same *context*. In many real-life problems, we usually assign objects to a context and we are interested in the relationship between objects in the same context as well as that between contexts. For example, we identify terrorists by their groups and we want to find the internal connection within a group as well as the external connection between the groups.

We propose *context-aware object connection discovery*, which first discovers the context for the nodes in a large graph and then discovers their best connection accordingly. The connection is first computed for nodes within the same context and then extended between the contexts. We illustrate the idea further by the following example.

Example 1: Suppose we want to find the connection between {*Jim Gray*, *Jennifer Widom*, *Michael I. Jordan*, *Geoffrey E. Hinton*} on the DBLP co-authorship graph. Our method first identifies that the four scholars are from two different contexts as shown by the two boxes in Figure 1: *Gray* and *Widom* are from the *database* community, while *Jordan* and *Hinton* are from the *machine learning* community. Then, we compute the best connection between the scholars within each context. After that, we compute the connection between the two contexts with respect to the four scholars. □

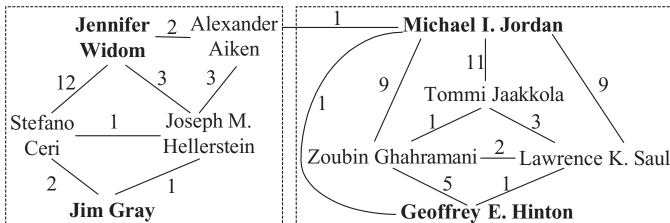


Fig. 1. Context-Aware Connection for {*Jim Gray*, *Jennifer Widom*, *Michael I. Jordan*, *Geoffrey E. Hinton*}

Our first task is to identify the context for the objects. We adopt the concept of *modularity* [12] to cluster the nodes in a large graph into a set of *communities*. Modularity is used in the area of community detection to measure the quality of the communities discovered in networks. Thus, using modularity, we are able to decompose a large graph into a set of high-quality communities, which can be naturally used as the context of the nodes. In addition, we are also able to retain the relationship between different communities by constructing a *community hierarchy tree*.

By considering the context of the objects, our algorithm adopts a *partition-and-conquer* approach. First, we partition a large graph into a set of small communities. Then, we discover the best *intra-community* connection, which is far more efficient since a community is significantly smaller than the original graph. Finally, we extend the connection to the *inter-community* level, which is also efficient by utilizing the community hierarchy tree.

To compute the best intra-community connection, we introduce the concept of *information throughput* within a community. Information throughput is defined based on the concept of *electrical circuits* (or equivalently *random walks*). Given a pair of nodes s and t (*source* and *target*), information throughput of a node i in the community is the total amount of currents flowing through i from s to t. Therefore, the information throughput of node i measures the importance of node i in connecting s and t.

The information throughput of node i indicates the *global importance* of node i in the community with respect to the information flow from s to t. However, an answer graph is a subgraph that displays only partial information of the original graph; thus, we also compute the amount of information flow in a s-to-t path through node i, which is regarded as the *local importance* of the s-to-t path in the answer graph. We propose a *goodness function* by integrating the global importance of the nodes and the local importance of the s-to-t paths in the answer graph. Then, we devise an efficient dynamic programming algorithm to compute the best intra-community connection that maximizes the goodness function.

When some query nodes are in different communities, we need to find the *inter-community connection*. We first utilize the community hierarchy tree to find the connection between the communities to which the query nodes belong. Then, we compute the actual path that connects the query nodes by following the connection between the communities. Since the community hierarchy tree reflects the relationship between the communities as defined by modularity, the quality of the inter-community connection is also ensured. As a result, our algorithm not only achieves high efficiency through the partition-and-conquer approach, but also guarantees the quality of the answer at both the intra- and inter- community levels.

Finally, our experimental results show that we are able to find a set of high-quality communities and our community partition algorithm is efficient. For connection discovery, we compare with the *center-piece subgraph* (*CEPS*) [10]. Our results show that our method obtains comparable high-quality answers as CEPS, but is over three orders of magnitude faster and also consumes significantly less memory.

The rest of the paper is organized as follows. Section II reviews related work. Section III briefs our solution framework. Section IV discusses community partition. Section V discusses intra-community connection. Section VI discusses inter-community connection. Section VII reports the experimental results. Section VIII concludes the paper.

II. Related Work

There are several studies on finding a good connection for some given query nodes in a large graph.

Faloutsos et al. [9] propose to find the *connection subgraph* for two query nodes. They model a graph as an electrical network and use the delivered current from one query node to the other as a goodness measure for the answer graph.

Later, Tong and Faloutsos [10] propose the *center-piece subgraph* (*CEPS*) to deal with multiple query nodes. They define a goodness criterion based on *random walks with restart* (*RWR*). Their algorithm iteratively picks up a most promising destination node based on RWR and then finds the path to connect each query node to this destination node. Consequently,

their answer graph may miss some good connection that does not first direct to a chosen destination node but directly goes from query nodes to query nodes.

Koren et al. [11] also study the problem of finding subgraphs that connect multiple query nodes. However, their objective is different from that of the previous work. They aim to extract the subgraph that retains most of the proximity between query nodes in the original graph. Therefore, their algorithm focuses on query nodes that are relatively close to each other.

Our work is also related to community detection. Among the studies on community detection, there are only a few that handle large networks. Newman [13] proposes an algorithm that requires $\mathcal{O}((|E|+|V|)|V|)$ time for a graph $G=(V,E)$. Around the same period, Wu and Huberman [14] propose a method that adopts the circuit model to discover communities in $\mathcal{O}(|E|+|V|)$ time. However, their algorithm requires to input the number of communities and some seed nodes that are in different communities, which restricts the application of their work. In our work, we improve Newman's algorithm [13] to detect communities in $\Omega(|V|\log|V|)$ and $\mathcal{O}(|V|^2)$ time.

In addition to the above-mentioned work, community detection has also been extensively studied in the context of Web. Web communities [15], [16], [17], [18], [19] are usually characterized by dense bipartite subgraphs based on the assumption that a topically focused community in the Web tends to contain a dense bipartite subgraph. The methods developed for Web communities may not be applicable to our problem since a vertex in the Web graph may not be covered by any extracted Web communities.

III. Solution Framework

In this section, we present the framework of our solution, which consists of the following two phases.

The first phase is *community partition*, which partitions the input graph into *a set of communities*, as well as constructs *the community hierarchy tree*. This phase is done offline. We discuss community partition in Section IV.

The second phase is *object connection discovery*, which is done online whenever a user issues a query. Our algorithm, *PCquery* (*P*artition-and-*C*onquer *query* processing), processes a query, i.e., finds the best connection between the query nodes, in two main steps. The first step computes the *intra-community connection* for query nodes that fall within the same community. Then, the second step of PCquery computes the *inter-community connection* between the communities of the query nodes, with the aid of the community hierarchy tree. The intra- and inter- community connections are finally integrated to give the final answer graph. We discuss in details the intra- and inter- community connection in Sections V and VI, respectively.

IV. Community Partition

We discuss an effective partitioning scheme that not only decomposes a graph into a set of high-quality *communities*, but also retains the relationship between the communities.

A. Modularity

Our partitioning scheme is based on the concept of *modularity*, which measures the quality of the communities discovered in networks. Modularity was proposed by Newman and Girvan [12] for community detection. We apply it to the problem of object connection discovery on a large graph, as an initial step to discover the contexts of the objects. For our purpose, we re-formulate modularity in our problem as follows.

Modularity is a quality function that tests whether a particular community partition of a graph is a good partition. A *community partition*, $\mathcal{P}$, of a graph $\mathcal{G}$ is a partition $\{C_1, C_2, \ldots, C_n\}$ on the set of vertices of $\mathcal{G}$. Each C_i is called *community* i. Given $\mathcal{P}$, the edges in $\mathcal{G}$ are categorized into two types: *intra-community edges* and *inter-community edges*, depending on whether the two incident vertices of an edge are in the same community or not. We denote the set of intra-community edges in C_i as E_{ii} and the set of inter-community edges connecting C_i and C_j $(i \neq j)$ as E_{ij}, respectively.

To define modularity, we need the following notations. The fraction of intra-community edges in C_i is defined as $\rho_{ii} = \frac{|E_{ii}|}{|E|}$. Similarly, the fraction of inter-community edges between C_i and C_j $(i \neq j)$ is defined as $\rho_{ij} = \rho_{ji} = \frac{|E_{ij}|}{2|E|}$ (Note that we have totally $2|E|$ edges here since we consider an edge to be shared by ρ_{ij} and ρ_{ji}). Finally, the fraction of the edges that are incident to at least one vertex in C_i (i.e., both intra- and inter- community edges of C_i) is defined as $\alpha_i = \sum_j \rho_{ij}$.

Now, we give the definition of modularity [12].

Definition 1: (Modularity) The *modularity* of a graph $\mathcal{G}$ with respect to a community partition $\mathcal{P}$ is defined as follows:

$$M(\mathcal{G},\mathcal{P}) = \sum_i (\rho_{ii} - \alpha_i^2). \quad (1)$$

The semantics of modularity can be interpreted as follows. If we pick up edges at random to include them into C_i, the *expected* fraction of intra-community edges in C_i, i.e., the expected value of ρ_{ii}, is α_i^2. In other words, $\sum_i \alpha_i^2$ corresponds to a community partition formed by random chance. Since ρ_{ii} represents the true fraction of intra-community edges in C_i, the subtraction $(\rho_{ii} - \alpha_i^2)$ indicates *the deviation of the community partition $\mathcal{P}$ from a randomized partition*. When all the vertices in $\mathcal{G}$ form a single community or the set of communities is formed by random chance, the value of modularity is 0.

The semantics of modularity indicates that the higher the value of $M(\mathcal{G},\mathcal{P})$, the higher the quality of $\mathcal{P}$ is. Intuitively, $\mathcal{P}$ is a good partition if most of the edges in $\mathcal{G}$ are *intra-community* edges and only a few are *inter-community* edges.

The following example illustrates the concept of modularity.

Example 2: Figure 2 shows a graph $\mathcal{G}$, where $|V|=12$ and $|E|=16$. Consider a community partition $\mathcal{P}=\{C_1, C_2, C_3\}$, where $C_1 = \{v_1, v_2, v_3, v_4\}$, $C_2 = \{v_5, v_6, v_7, v_8, v_9\}$ and $C_3 = \{v_{10}, v_{11}, v_{12}\}$, as shown by the three dotted circles in the figure. We have $|E_{12}| = 1$ since there is only one edge (v_4, v_6) between C_1 and C_2. We also have $|E_{11}| = 5$ since there are five intra-community edges in C_1. By definition, we have $\rho_{11} = \frac{|E_{11}|}{|E|} = \frac{5}{16}$, $\rho_{12} = \frac{|E_{12}|}{2|E|} = \frac{1}{32}$, and $\rho_{13} = \frac{|E_{13}|}{2|E|} = \frac{1}{32}$. We further have $\alpha_1 = \sum_{1\leq j\leq 3} \rho_{1j} = \frac{5}{16} + \frac{1}{32} + \frac{1}{32} = \frac{3}{8}$.

Similarly, we can compute $\rho_{22} = \frac{5}{16}$, $\rho_{33} = \frac{3}{16}$, $\alpha_2 = \frac{3}{8}$, and $\alpha_3 = \frac{1}{4}$. By Equation (1), we calculate the modularity, $M(\mathcal{G}, \mathcal{P}) = \sum_{1 \le i \le 3}(\rho_{ii} - \alpha_i^2) = \frac{15}{32}$. □

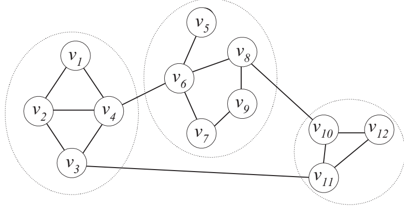


Fig. 2. A Graph, $\mathcal{G}$, and a Community Partition of $\mathcal{G}$

B. Computing the Community Partition

According to the semantics of modularity, the *optimal community partition* is defined as follows:

$$\mathcal{P}_{opt} = argmax_{\ \mathcal{P}}\ M(\mathcal{G}, \mathcal{P}). \tag{2}$$

Equation (2) defines an optimization problem. However, to find the maximal value of modularity, we need to try all possible community partitions. This is clearly intractable since the total number of possible partitions is the $|V|$-th Bell number, which is at least exponential in $|V|$ but $\mathcal{G}$ may contain millions of nodes. We give an approximate solution to this optimization problem by a greedy algorithm.

We adopt the same greedy strategy as in [13], [20] but investigate more properties for efficient computation. The basic idea of the greedy algorithm is as follows. We start from a trivial initial community partition, in which each vertex in $\mathcal{G}$ forms a single community. At each iteration, we first pick up the pair of communities that achieves the greatest increase in modularity and combine them to form a new community. We then compute the new modularity that can be achieved for every pair of communities, assuming that they are to be combined in the next iteration. This process goes on to form larger communities and the approximate optimal community partition is obtained when the modularity decreases for every possible pairs of communities to be combined.

The key operation in the greedy algorithm is to compute the new modularity and retrieve the largest one at each iteration. It has been shown in [20] that the change of modularity can be updated incrementally. We omit the details of the update and refer readers who are interested to [20].

We now discuss how to efficiently retrieve the largest change of modularity, as well as to save update computations. Let $\mathcal{P}_{k-1} = \{C_1, \ldots, C_i, \ldots, C_j, \ldots C_n\}$ be the community partition after the $(k-1)$-th iteration. Let ΔM_k^{ij} be the change of modularity, if C_i and C_j are to be combined at the k-th iteration. To efficiently retrieve the largest ΔM_k^{ij}, we need two max-heaps. First, we use a *local* max-heap for each community C_i to keep ΔM_k^{ij}, for every C_j that has connection with C_i. Then, we further use a *global* max-heap to keep all the local maximum ΔM_k^{ij}, so that we can pick C_i and C_j that have the largest ΔM_k^{ij} to combine at the k-th iteration.

At the k-th iteration, we first use the global max-heap to pick C_i and C_j, which takes $\mathcal{O}(1)$ time. Then, we need $\mathcal{O}(|C_i| + |C_j|)$ time to combine C_i and C_j into C_{new}. We also need $\mathcal{O}(|C_{new}|)$ time to rebuild the local max-heap for C_{new}. Then, for each C_x connecting to C_i or C_j, we need $\mathcal{O}(\log|C_x|)$ time to update the local max-heap of C_x. Thus, in total we need $\mathcal{O}(|C_{new}|\log|C_x|) = \mathcal{O}(|V|\log|V|)$ time to update all C_x. Finally, we update the global max-heap with the local maximums from the max-heaps of C_{new} and of each C_x, which requires $\mathcal{O}(|C_{new}|\log(|\mathcal{P}_k|)) = \mathcal{O}(|V|\log|V|)$ time.

Since initially each vertex in $\mathcal{G}$ forms a single community, at most we have $|V|$ iterations. Therefore, the algorithm requires $\mathcal{O}(|V|^2\log|V|)$ time. However, this time complexity is still too high in practice for large graphs. We further improve the efficiency based on the following two heuristics.

Heuristic 1: Let c_{local} be a *small constant*. Then, the first c_{local} elements in the local max-heap of a community remain relatively stable during the entire process of community partition.

Heuristic 1 states that the first few largest elements in the local max-heap of a community remain to be the first few largest elements most of the time, even the max-heap may be updated frequently during the process of community partition. This is true because a community tends to combine with only a few of its connecting communities based on modularity, rather than to have an even probability of combining with each of its connecting communities.

Based on Heuristic 1, we can limit the size of the local max-heap of a community to c_{local}. In this way, after combining C_i and C_j at the k-th iteration, we can update the local max-heap of each C_x in constant time. In the worst case, when an element that is not one of the first c_{local} largest elements becomes one of the first c_{local} largest elements, we need $\mathcal{O}(\log|C_x|)$ time to first find the element and then insert it into the local max-heap of C_x. However, this worst case rarely happens. According to our experiment, the majority of the updates on the local max-heap of a community, during the entire process of community partition, are operated on the first c_{local} elements of the max-heap.

Heuristic 2: Let c_{global} be a *constant*. Then, the first c_{global} elements in the global max-heap remain relatively stable during the entire process of community partition.

Heuristic 2 states that the first few largest elements in the global max-heap remain to be the first few largest elements most of the time. This is true because the global max-heap keeps the local maximums and, according to Heuristic 1, the local maximums remain to be stable.

By Heuristic 2, we can further reduce the time for each update of the global max-heap from $\mathcal{O}(\log(|\mathcal{P}_k|))$ to constant time.

Now, for each iteration, we still need $\mathcal{O}(|C_i| + |C_j|)$ time to combine C_i and C_j into C_{new}. We need $\mathcal{O}(c_{local}) = \mathcal{O}(1)$ time to rebuild the local max-heap for C_{new}. We need $\mathcal{O}(c_{local}|C_{new}|) = \mathcal{O}(|C_{new}|)$ time to update the local max-heaps of all C_x connecting to C_i or C_j. Finally, we need $\mathcal{O}(c_{global}|C_{new}|) = \mathcal{O}(|C_{new}|)$ time to update the global max-heap. As a result, the total running time for each iteration is now $\mathcal{O}(|C_{new}|)$.

Let $C(\mathcal{P}_k)$ be the new community formed at the k-th iteration (i.e., C_{new} in the above discussion). The overall running time is $\sum_{k=1}^{|V|}\mathcal{O}(|C(\mathcal{P}_k)|)$. In the worst case, at each iteration we combine C with a community that has a single vertex until C contains the set of all vertices in $\mathcal{G}$; thus, $\sum_{k=1}^{|V|}\mathcal{O}(|C(\mathcal{P}_k)|) = (2+3+\cdots+|V|) = \mathcal{O}(|V|^2)$. In the best case (in terms of time complexity), each iteration picks the smallest two communities to combine; then, $\sum_{k=1}^{|V|}\mathcal{O}(|C(\mathcal{P}_k)|) = \mathcal{O}(|V|\log|V|)$. Therefore, we obtain the running time of the algorithm as $\Omega(|V|\log|V|)$ and $\mathcal{O}(|V|^2)$.

Finally, the space requirement of the algorithm is $\mathcal{O}(|V| + |E|)$, since for each community, we only need to keep the communities that connect to it, which is similar to the structure of the adjacency list representation of a graph.

C. Community Hierarchy Tree

In addition to the purpose of breaking a large graph into small communities, the process of community partition by the greedy algorithm also builds a *community hierarchy tree* for computing the inter-community connection to be discussed in Section VI. We show an example of a community hierarchy tree, $\mathcal{H}$, in Figure 3.

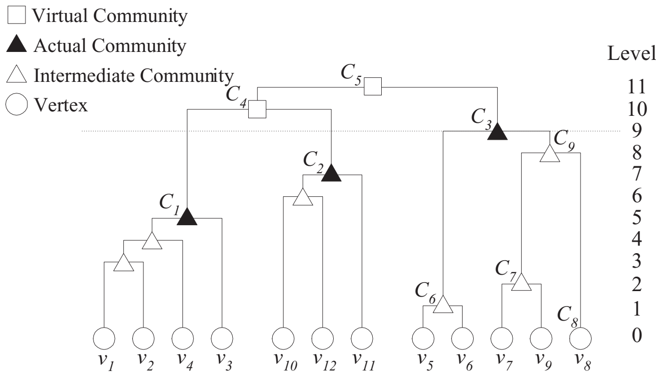


Fig. 3. Community Hierarchy Tree, $\mathcal{H}$, of $\mathcal{G}$

The set of nodes at Level 0 of $\mathcal{H}$ corresponds to the initial community partition in which each vertex of $\mathcal{G}$ forms a community. Then, each level k, $\forall k > 0$, has only one node, v, which corresponds to the community formed by combining the two child nodes of v at the k-th iteration. We categorize the nodes, which represent communities, in $\mathcal{H}$ into three types as follows.

- *Actual communities*, which are the optimal communities approximated by the greedy algorithm.
- *Virtual communities*, which are communities that are ancestors of the actual communities in $\mathcal{H}$.
- *Intermediate communities*, which are communities that are descendants of the actual communities.

In order to create the community hierarchy tree, we run the greedy algorithm until all vertices in $\mathcal{G}$ are combined into a single community. Thus, the set of actual communities are the community partition that is obtained when the modularity reaches its peak (after this point the modularity starts to decrease), while the set of virtual communities is all communities that are formed after the peak.

We discuss how each of the three types of communities is used to facilitate object connection discovery in $\mathcal{G}$. First, the set of actual communities partitions $\mathcal{G}$ into small and high-quality components so that we can compute the connection on each component efficiently.

Second, the set of virtual communities keeps the relationship among the actual communities. Each virtual community C keeps a set of *community edges*. A community edge of C is an edge (C_i, C_j), where both C_i and C_j are actual communities that are descendants of C, such that C_i and C_j are connected in $\mathcal{G}$, i.e., there is at least one edge between C_i and C_j in $\mathcal{G}$. To avoid (C_i, C_j) being kept duplicately at all the common ancestors of C_i and C_j, we keep (C_i, C_j) at an ancestor C if and only if C_i is in the left subtree of C and C_j is in the right subtree of C. The community edges are essential for computing the inter-community connection between the query nodes that are in different actual communities.

Each (C_i, C_j) indicates that there exists at least one edge (v_i, v_j) in $\mathcal{G}$, where $v_i \in C_i$ and $v_j \in C_j$. Since (v_i, v_j) connects C_i and C_j in $\mathcal{G}$, we call (v_i, v_j) a *connecting edge*. We also associate the set of connecting edges with (C_i, C_j).

The following example illustrates the various concepts discussed above.

Example 3: In Figure 3, we have two virtual communities, C_4 and C_5. The set of community edges kept at C_4 is $\{(C_1, C_2)\}$ since there is only one actual community at either side of C_4 as its descendants. The set of connecting edges associated with (C_1, C_2) is $\{(v_3, v_{11})\}$ since C_1 and C_2 are connected by the edge (v_3, v_{11}) in Figure 2. The set of community edges kept at C_5 is $\{(C_1, C_3), (C_2, C_3)\}$. The set of connecting edges associated with (C_1, C_3) is $\{(v_4, v_6)\}$ and that with (C_2, C_3) is $\{(v_{10}, v_8)\}$. □

We do not use intermediate communities in this paper; however, intermediate communities are useful when the user wants to know more about each query node itself in addition to the relationship between query nodes. This is particularly necessary when a query node is in an isolated component, in which case we want to return some useful information to the user rather than the query node itself. For this purpose, we can define *groups* as intermediate communities, since small groups present the most relevant (*local*) information to a query node. The community hierarchy tree provides the flexibility of allowing the user to specify the group size, as well as the fast retrieval of the groups.

V. Intra-Community Connection

After decomposing the large graph $\mathcal{G}$ into a set of much smaller communities, we can now process the query nodes that fall within each community to obtain the context-aware connection. In this section, we first propose the notions of *information throughput* and *information flow* to measure the importance of a node and a path in connecting the query nodes within the community. Then, we integrate the two

concepts to define a goodness function, by which we measure the quality of the intra-community connection. Finally, we propose a dynamic programming algorithm to compute the best connection between the query nodes within a community.

A. Information Throughput and Information Flow

Let G be the induced graph of a community C in $\mathcal{G}$, and $\mathcal{W} = \{w_{ij}\}$ be the adjacent matrix of G. We can model G as an *electrical circuit*, where each edge in G is a resistor whose conductance is the edge weight.

Our circuit model is *source-target* dependent; that is, given a source node s and a target node t, we inject one unit of current into the circuit at s and extract one unit at t. Let V_i^{st} be the voltage of node i in the circuit and I_i^{st} be the current flowing through node i. Since the current going into a node is equal to that going out of the node (by *Kirchhoff's law of current conservation*), I_i^{st} is defined as half of the absolute amount of all the currents flowing through node i. The concept of information throughput is then formally defined as follows.

Definition 2: (Information Throughput) Given a connected graph G, the *information throughput* of a node i with respect to a pair of source-target nodes $\{s, t\}$ is defined as follows:

$$T_i^{st} = I_i^{st} = \begin{cases} \frac{1}{2}\sum_j w_{ij}|V_i^{st} - V_j^{st}|, & \text{if } i \neq s, t \\ 1, & \text{otherwise.} \end{cases} \quad (3)$$

Intuitively, T_i^{st} indicates the importance of node i in connecting s and t.

In order to compute T_i^{st}, we only need to obtain the voltage values by the following equation:

$$(\mathcal{W}^D - \mathcal{W}) \cdot \mathcal{V} = \mathcal{I}^C, \quad (4)$$

where $\mathcal{W}^D$ is a diagonal matrix with entries $w_{ii}^D = \sum_j w_{ij}$, $\mathcal{V} = \{V_i^{st}\}$ is the voltage vector of all nodes, and $\mathcal{I}^C = \{I_i^C\}$ is the cumulative current vector of all nodes ($I_s^C = 1$; $I_t^C = -1$; and $I_i^C = 0$ for $i \neq s, t$).

Information throughput T_i^{st} measures the total amount of information flow from s to t through i in G. Since there may be many paths from s to t passing through i, we also want to know the exact amount of information flow through each path. Let $p^{st} = \langle s = i_1, i_2, \ldots, i_{n-1}, t = i_n\rangle$ be a simple path from s to t. We define the *information flow* of p^{st} as follows:

$$\begin{aligned} & T(p^{st}) = T(\langle s = i_1, i_2, \ldots, i_{n-1}, t = i_n\rangle) \\ = & \prod_{k=1}^{n-1} \frac{w_{i_k i_{k+1}}(V_{i_k}^{st} - V_{i_{k+1}}^{st})}{I_{i_k}^{st}}. \end{aligned} \quad (5)$$

Note that information throughput T_i^{st} is equal to the sum of the information flow of all the paths passing through i. We illustrate the concepts of information throughput and information flow by the following example.

Figure 4 shows a weighted graph, where the weight of each edge is represented by the associated integer. Assume that s and t are two query nodes. The information throughput of each node with respect to s and t is given as the value below each node in Figure 4 and the information flow of each path is given in Table I.

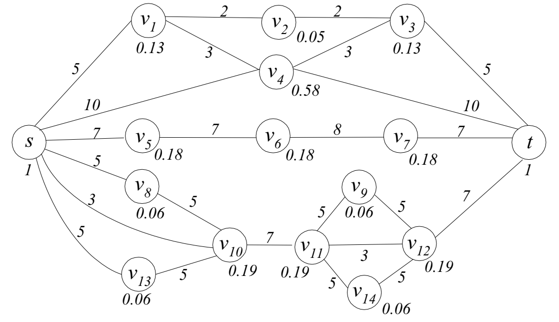


Fig. 4. An Example Graph

TABLE I
INFORMATION FLOW AND VALUE OF PATHS IN FIGURE 4

Path, p^{st}	$T(p^{st})$	$val(p^{st})$
$p_1=\langle s, v_4, t\rangle$	0.46	0.267
$p_2=\langle s, v_5, v_6, v_7, t\rangle$	0.18	0.032
$p_3=\langle s, v_1, v_4, t\rangle$	0.04	0.014
$p_4=\langle s, v_4, v_3, t\rangle$	0.04	0.014
$p_5=\langle s, v_1, v_4, v_3, t\rangle$	0.04	0.011
$p_6=\langle s, v_{10}, v_{11}, v_{12}, t\rangle$	0.03	0.006
$p_7=\langle s, v_1, v_2, v_3, t\rangle$	0.05	0.005
Any one of the remaining 8 paths	0.02	0.003

Example 4: Node v_4 has the highest information throughput since it directly connects s and t with the highest weight 10. Moreover, v_4 also serves as a bridge to connect v_1 and v_3, which forms several paths from s to t. Therefore, v_4 plays a very important role in connecting s and t and it is well indicated by the high value of its information flow.

The nodes v_{10}, v_{11} and v_{12} are also of high information throughput since they form a number of s-to-t paths. However, the number of these paths is large and the paths are relatively long, which weakens the significance of these nodes in connecting s and t. Although this point is not indicated by information throughput of the nodes, it is captured by information flow of the paths. As shown in Table I, all the paths formed by these nodes are of low information flow. □

Given a community C, we only need to compute the corresponding matrix $(\mathcal{W}^D - \mathcal{W})^{-1}$ once. Afterwards, regardless of which query nodes are asked, the information throughput of all nodes and the information flow of all paths with respect to any pair of query nodes can be computed by Equations (3-5).

Computing $(\mathcal{W}^D - \mathcal{W})^{-1}$ takes $\mathcal{O}(|C|^3)$ time for a community C. However, a community C is significantly smaller than the original graph $\mathcal{G}$ and hence computing the matrix inversion on C is much more efficient than on $\mathcal{G}$. Since we only need to compute $(\mathcal{W}^D - \mathcal{W})^{-1}$ once and then use it to answer any queries, it is also reasonable to pre-compute it offline. Afterwards, the time complexity of computing the vectors of voltage and information throughput with respect to any pair of source and target is $\mathcal{O}(|C|^2)$, which can be processed efficiently for online querying since C is not large.

B. The Goodness Function

We now define the goodness function, by which we determine which nodes and paths should be used to connect

the query nodes within a community. Then, we show how to compute the best connection between query nodes that maximizes the goodness function.

Let Q be a set of query nodes in a community C. We compute a connected subgraph $G_Q = (V_Q, E_Q)$ such that $Q \subseteq V_Q \subseteq C$ and G_Q maximizes a goodness function g. We first define g based on information throughput as follows:

$$g(G_Q) = \sum_{s,t \in Q} \sum_{i \in V_Q} T_i^{st}. \tag{6}$$

Equation (6) indicates the total degree of importance of all the nodes in G_Q. However, according to Equation (6), $g(G_Q)$ is maximum when we include all nodes of C into G_Q. To avoid obtaining this trivial answer graph, we set a *budget* b such that our objective is now to maximize $g(G_Q)$ with $|V_Q| \leq b$. The budget is commonly used to control the size of the answer graph for clear visualization [9], [10].

Although the information throughput of a node indicates its importance in connecting s and t, simply summing up the information throughput of the nodes as in Equation (6) has a problem. Given a node i in G_Q, T_i^{st} actually represents the total amount of information flow from s to t through i in C. Thus, T_i^{st} may be high only because there are many paths from s to t going through node i. We illustrate this problem of Equation (6) by Example 5.

Example 5: Consider the graph in Figure 4. Based on Equation (6), if we set the budget $b = 6$ nodes, then the best connection from s to t is shown in Figure 5. The problem is obvious since the path $\langle s, v_{10}, v_{11}, v_{12}, t\rangle$ has very low information flow of only 0.03. The three nodes v_{10}, v_{11} and v_{12} are chosen only because there are many paths passing through them, while all these paths have very low information flow and thus only show weak connection from s to t. □

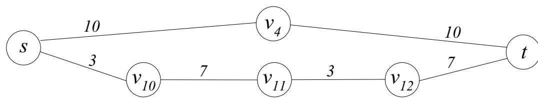


Fig. 5. Answer Graph for Example 5

To address the problem of Equation (6), we attempt to define the goodness function based on an s-to-t path directly instead of on individual nodes as follows:

$$g(G_Q) = \sum_{s,t \in Q} \sum_{p^{st} \in G_Q} T(p^{st}). \tag{7}$$

Equation (7) indicates the total amount of information delivered by all the s-to-t paths in G_Q. However, Equation (6) is not totally meaningless in the sense that a node i having a higher information throughput is indeed more important than another node j with a lower information throughput, because the total amount of information delivered from s to t through node i is indeed higher than that through node j. Thus, considering the information flow of a path alone may also miss some important connection, as illustrated by the following example.

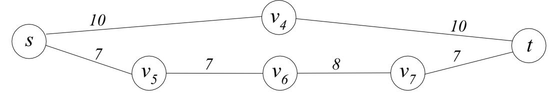


Fig. 6. Answer Graph for Example 6

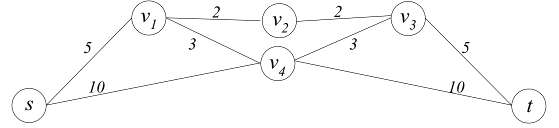


Fig. 7. Answer Graph for Example 7

Example 6: Consider again the graph in Figure 4. Based on Equation (7), the best connection from s to t is shown in Figure 6, which are the two paths that have the highest information flow. However, there are three paths, $\langle s, v_1, v_4, t\rangle$, $\langle s, v_4, v_3, t\rangle$ and $\langle s, v_1, v_4, v_3, t\rangle$, all passing through the most important node v_4 in the s-to-t connection. The overall information flow of these paths, together with the path $\langle s, v_1, v_2, v_3, t\rangle$, is only 0.01 lower than the path $\langle s, v_5, v_6, v_7, t\rangle$. Thus, if we consider both the strong connection of these paths with the important node v_4 and the comparable overall information flow, then Figure 7 gives a better answer graph than Figure 6. More importantly, the combination of the paths contributes to a network of significant connection from s to t, which is far more informative than a single path $\langle s, v_5, v_6, v_7, t\rangle$. □

To address the problems of Equations (6) and (7), we take the merits of both information throughput and information flow by integrating them to define a new goodness function as follows.

$$val(p^{st}) = \begin{cases} T(p^{st}), & \text{if } p^{st} = \langle s, t\rangle \\ T(p^{st}) \frac{\sum_{i \in p^{st}, i \neq s,t} T_i^{st}}{|p^{st}|-2}, & \text{otherwise.} \end{cases}$$

$$g(G_Q) = \sum_{s,t \in Q} \sum_{p^{st} \in G_Q} val(p^{st}). \tag{8}$$

Equation (8) first defines the *value* of a path p^{st}, denoted as $val(p^{st})$, based on which the goodness function of an answer graph is further defined. According to Equation (8), $g(G_Q)$ is high when both the information flow of the s-to-t paths and the average information throughput of the nodes on the paths are high. We can interpret Equation (8) as follows. $T(p^{st})$ in Equation (8) quantifies the *local* information flow in the answer graph, while $(\sum_{i \in p^{st}, i \neq s,t} T_i^{st})/(|p^{st}|-2)$ governs the choice of a node i to be included in the answer graph according to its *global* importance in the community (with respect to s and t).

Example 7: Based on Equation (8), the best connection from s to t is shown in Figure 7. The value of each path p^{st} is shown in Table I. The goodness score of the answer graph in Figure 7 is 0.311, while those of Figure 5 and Figure 6 are 0.273 and 0.299, respectively. □

C. Computing the Optimal Intra-Community Connection

Computing the best connection for query nodes within a community can be defined as an optimization problem as

follows:

$$G_Q^{opt} = argmax\ _{G_Q}\ g(G_Q). \tag{9}$$

To solve this optimization problem, we first attempt to apply a dynamic programming (DP) solution similar to the *0-1 knapsack* problem, by setting the budget b as the total capacity of the knapsack, the set of "p^{st}" paths as the set of items, $|p^{st}|$ as the item weight, and $val(p^{st})$ as the item value.

Let $PathSet$ be the set of "p^{st}" paths and $|PathSet|$ be the number of paths in $PathSet$. The knapsack problem can be solved in $\mathcal{O}(|PathSet|b)$ time. Unlike the knapsack problem, b here is small, since b is defined as the size of a graph that a human user is able to visualize clearly. However, there are still two problems that need to be solved. First, we need to obtain $PathSet$. Second, the nodes on the paths may overlap, while the items in the knapsack problem do not share their weights. We address each of the problems as follows.

1) Path Selection:

Clearly we cannot use the set of all "p^{st}" paths within a community because the number of such paths is too large. We select a set of high-value paths by the following strategy.

We start a *breath-first-search* (*BFS*) from s to collect a set of "p^{st}" paths. The BFS does not visit all nodes but only explores a node j from a node i if $(V_i^{st} - V_j^{st}) > 0$, since we require $val(p^{st}) > 0$. To control the number of paths, for each s-t pair, we stop a new iteration of BFS when the number of distinct nodes on the collected paths is greater than b.

Example 8: Consider the graph in Figure 4. If $b = 6$, the set of s-to-t paths is selected as follows. First, the second iteration of BFS selects $\langle s, v_4, t\rangle$. Then, the third iteration of BFS selects $\langle s, v_1, v_4, t\rangle$ and $\langle s, v_4, v_3, t\rangle$. The fourth iteration of BFS selects $\langle s, v_1, v_2, v_3, t\rangle$, $\langle s, v_1, v_4, v_3, t\rangle$, $\langle s, v_5, v_6, v_7, t\rangle$, and $\langle s, v_{10}, v_{11}, v_{12}, t\rangle$. Now, the number of distinct nodes is greater than b; thus, we stop the BFS path selection and none of the other paths in Figure 4 are selected. □

The BFS strategy has the following three advantages. First, it controls the length of the paths, because we do not prefer long paths due to the limited budget b. Second, the paths collected by BFS tend to have common nodes, thereby giving a higher $val(p^{st})$ within a fixed b. Third, the BFS strategy also collects high-value paths, because $T(p^{st})$ decreases for each node added to the path as reflected by Equation (5).

2) Dynamic Programming:

After we select the set of "p^{st}" paths, $PathSet$, we can construct a $(|PathSet| \times b)$ table for the DP solution. We first sort $PathSet$ in descending order of the values of the paths to facilitate the DP process, since we prefer high-value paths to low-value ones. However, the fact that paths may share common nodes significantly complicates the problem. We address this problem as follows.

Let Tab be the DP table and p_x be the x-th path in $PathSet$. In the 0-1 knapsack setting, $Tab[x, y]$ gives the optimal solution for the sub-problem with x paths and the capacity y, where $Tab[x, y]$ is given as follows.

$$\begin{aligned} Tab[x, y] = MAX\{\, & Tab[x-1, y], \\ & val(p_x) + Tab[x-1, y-|p_x|]\}. \end{aligned} \tag{10}$$

Equation (10), however, has a problem. If some nodes on p_x also appear on some paths processed previously, then it is no longer correct to use $Tab[x-1, y-|p_x|]$ to compute $Tab[x, y]$.

Let $PathSet[1 \cdots (x-1)]$ be the set of paths in $PathSet$ that are processed by DP before we process p_x. We may have z nodes on p_x that are also on some paths in $PathSet[1 \cdots (x-1)]$, where $0 \le z \le |p_x|$. Therefore, we need to check $Tab[x-1, y-(|p_x|-z)]$, for $0 \le z \le |p_x|$, in order to compute $Tab[x, y]$. However, we cannot simply take the maximum $Tab[x-1, y-(|p_x|-z)]$ for some z, since we need to make sure that there are indeed z nodes being repeated on the set of paths from which $Tab[x-1, y-(|p_x|-z)]$ is computed. Let $PathTab[x-1, y-(|p_x|-z)]$ be the set of paths from which $Tab[x-1, y-(|p_x|-z)]$ is computed. We compute $Tab[x, y]$ as follows.

$$\begin{aligned} Tab[x, y] = MAX\{\, & Tab[x-1, y], \\ & val(p_x) + MAX\{\, Tab[x-1, y-(|p_x|-z)]\}\}, \end{aligned} \tag{11}$$

where $0 \le z \le |p_x|$ and there are at least z nodes on p_x that also appear on some paths in $PathTab[x-1, y-(|p_x|-z)]$.

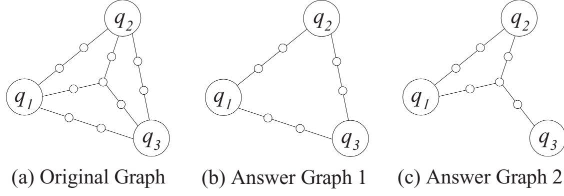
We give the pseudo-code of the DP algorithm in Algorithm 1. The algorithm is self-explanatory except the handling of the common nodes on different paths. In Lines 8-17, we find the number of nodes on p_x that also appear on some paths in $PathTab[x-1, y-(|p_x|-z)]$ as follows.

Algorithm 1 IntraConnection

Input: *PathSet*, b.
Output: G_Q.

```
1.  Sort PathSet in descending order of the path values;
2.  Create Tab with (|PathSet| + 1) rows and (b + 1) columns;
3.  Initialize the first row, Row 0, of Tab to be all '0's;
4.  for each x = 1, ..., |PathSet|, do
5.    for each y = 0, ..., b, do
6.      Tab[x, y] ← Tab[x − 1, y];
7.      PathTab[x, y] ← PathTab[x − 1, y];
8.      Let p_x be the x-th path in PathSet;
9.      for each z = 0, ..., |p_x|, do
10.       z' ← 0;
11.       for each node v on p_x do
12.         if(PathSet(v) ∩ PathTab[x − 1, y − (|p_x| − z)] ≠ ∅)
13.           z' ← z' + 1;
14.         break if z' ≥ z;
15.       if(z' ≥ z and Tab[x, y] < (val(p_x) + Tab[x−1, y−(|p_x|−z)]))
16.         Tab[x, y] ← (val(p_x) + Tab[x − 1, y − (|p_x| − z)]);
17.         PathTab[x, y] ← (PathTab[x−1, y−(|p_x|−z)] ∪ {p_x});
18. Combine the paths in PathTab[|PathSet|, b] to give G_Q;
```

Let $PathSet(v)$ be the set of paths in $PathSet$ that contain a node v. For each v on p_x, we intersect $PathSet(v)$ with $PathTab[x-1, y-(|p_x|-z)]$. If the intersection is not an empty set, then v is repeated on some path in $PathTab[x-1, y-(|p_x|-z)]$. We use z' to keep the total number of nodes being repeated. If there are $z' \ge z$ nodes on p_x that also appear on some paths in $PathTab[x-1, y-(|p_x|-z)]$, we update $Tab[x, y]$ according to Equation (11).



(a) Original Graph (b) Answer Graph 1 (c) Answer Graph 2

Fig. 8. Answer Graph for Example 10

Finally, G_Q is computed by combining the set of path in $PathTab[|PathSet|, b]$. We show how the algorithm works by the following example.

Example 9: Consider the graph in Figure 4 and two query nodes s and t. We set $b = 6$ and after sorted, $PathSet = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ as given in Table I. Table II gives the value of each $Tab[x, y]$ and the set of paths in each $PathTab[x, y]$. To save space, we omit Columns 0-2 and Row 0, which are all 0's.

Recall that $PathTab[x, y] = \{p_i, \ldots, p_j\}$ can be regarded as a subgraph consisting of the set of paths $\{p_i, \ldots, p_j\}$, where the number of distinct nodes in the subgraph is y. Thus, we can construct $PathTab[3, 4] = \{p_1, p_3\}$ from $PathTab[2, 3] = \{p_1\}$, since all the nodes on p_3, except v_1, appear already on p_1. In Row 5, $PathTab[4, 6] = \{p_1, p_2\}$ was replaced by $PathTab[5, 6] = \{p_1, p_3, p_4, p_5\}$, since $Tab[5, 6]$ is now greater than $Tab[4, 6]$. For Row 6, since p_6 consists of three new nodes and hence we need to update $Tab[6, 6]$ from $Tab[5, 3]$. However, $(Tab[5, 3] + 0.006) < Tab[5, 6]$; thus, p_6 is not added and Row 6 remains the same as Row 5. For Row 7, there is only one new node on p_7; thus, we add p_7 to $PathTab[6, 5]$ to give $PathTab[7, 6] = \{p_1, p_3, p_4, p_5, p_7\}$, which is the final answer. □

TABLE II

VALUES OF *Tab* AND *PathTab* IN EXAMPLE 9

	3	4	5	6
1	0.267 $\{p_1\}$	0.267 $\{p_1\}$	0.267 $\{p_1\}$	0.267 $\{p_1\}$
2	0.267 $\{p_1\}$	0.267 $\{p_1\}$	0.267 $\{p_1\}$	0.299 $\{p_1, p_2\}$
3	0.267 $\{p_1\}$	0.281 $\{p_1, p_3\}$	0.281 $\{p_1, p_3\}$	0.299 $\{p_1, p_2\}$
4	0.267 $\{p_1\}$	0.281 $\{p_1, p_3\}$	0.295 $\{p_1, p_3, p_4\}$	0.299 $\{p_1, p_2\}$
5	0.267 $\{p_1\}$	0.281 $\{p_1, p_3\}$	0.306 $\{p_1, p_3, p_4, p_5\}$	0.306 $\{p_1, p_3, p_4, p_5\}$
6	0.267 $\{p_1\}$	0.281 $\{p_1, p_3\}$	0.306 $\{p_1, p_3, p_4, p_5\}$	0.306 $\{p_1, p_3, p_4, p_5\}$
7	0.267 $\{p_1\}$	0.281 $\{p_1, p_3\}$	0.306 $\{p_1, p_3, p_4, p_5\}$	0.311 $\{p_1, p_3, p_4, p_5, p_7\}$

Example 9 shows how to find the answer graph of only two query nodes. The following example further illustrates the case when the number of query nodes is more than two.

Example 10: Figure 8(a) shows a graph with three query nodes. Assume that all paths have the same value. If we set $b = 9$, we can either output Figure 8(b) or Figure 8(c). Figure 8(b) shows strong pair-wise relationship between any pair of query nodes, while Figure 8(c) shows strong relationship among all query nodes and pair-wise connection is included only because the budget still allows. Our algorithm gives Figure 8(c), which reveals another advantage of our method that it values strong relationship among all or the majority of query nodes higher than pair-wise relationship. □

We now analyze the time complexity. To compute each $Tab[x, y]$, we need to check $|p_x|$ number of "$Tab[x-1, y-(|p_x|-z)]$" entries. For each entry, we need to perform z intersections of $PathSet(v)$ and $PathTab[x-1, y-(|p_x|-z)]$. Thus, we need $\mathcal{O}(|p_x|^2|PathSet|)$ time to compute each $Tab[x, y]$. Note that we can terminate the intersection as soon as we find one path in both sets. Thus, in many cases, the intersection terminates earlier.

In total, we need to compute $b|PathSet|$ table entries and hence the total running time is $\mathcal{O}(b|p_x|^2|PathSet|^2)$. In practice, this running time is very small since $|p_x|$, $|PathSet|$ and b are all small.

The space required for $PathSet$ and Tab is $\mathcal{O}(b|PathSet|)$. We also keep $PathTab[x-1, y-(|p_x|-z)]$ with each $Tab[x-1, y-(|p_x|-z)]$, but only for the $(x-1)$-th row and x-th row of Tab. Thus, the extra space needed is at most $(2b|PathSet|)$. In most cases $|PathTab[x-1, y-(|p_x|-z)]|$ is much smaller than $|PathSet|$.

However, there is a potential problem in the DP solution that G_Q may not be connected or may not contain all query nodes, since the DP only picks up paths according to their values. Since we select paths for all s-t pairs, $\forall s, t \in Q$, we simply make G_Q connected with all query nodes by adding the corresponding path(s) with the highest value. Thus, this process may output slightly more than b nodes but should not affect clear visualization.

VI. INTER-COMMUNITY CONNECTION

In Section V, we discuss the intra-community connection between a set of query nodes that are in the same community. In this section, we discuss the *inter-community connection* for a query that contains nodes from different communities.

We give the algorithm for computing the inter-community connection between the query nodes in Algorithm 2.

Algorithm 2 InterConnection

Input: $\mathcal{H}$ and the query $\mathcal{Q}$.
Output: The answer graph $\mathcal{G}_{\mathcal{Q}}$.
1. Let $\mathbb{C}$ be the set of all communities that contain at least one query node;
2. Let A be the common ancestor of all communities in $\mathbb{C}$;
3. Let E_{com} be the set of all community edges of each virtual community on the path from each $C_i \in \mathbb{C}$ to A;
4. Construct a graph, G_{com}, from E_{com};
5. **for each** pair of communities, C_i and C_j, in $\mathbb{C}$ **do**
6. Compute the shortest path between C_i and C_j in G_{com};
7. Let $\mathbb{P}$ be the set of all shortest paths obtained in Steps 5-6;
8. Sort $\mathbb{P}$ in ascending order of the path length;
9. **for each** path $\langle C_i, \ldots, C_j \rangle \in \mathbb{P}$ **do**
10. **if**(C_i and C_j are not yet connected in $\mathcal{G}_{\mathcal{Q}}$)
11. Find the actual path between C_i and C_j, and add it to $\mathcal{G}_{\mathcal{Q}}$;
12. **if**(All query nodes in $\mathcal{Q}$ are connected)
13. Return $\mathcal{G}_{\mathcal{Q}}$;

We describe Algorithm 2 as follows. In Section IV-C, we construct the community hierarchy tree $\mathcal{H}$ that shows the relationship between the communities. For example, if two communities are siblings in $\mathcal{H}$, then they are the closest to each other since they are to be joined as one single

community as measured by modularity. Algorithm 2 utilizes this relationship between two communities C_i and C_j to first find the connection between C_i and C_j at the community level (Lines 2-6); then, we find the actual path in $\mathcal{G}$ that connects the query nodes in C_i to those in C_j (Line 11).

We first discuss how we find the connection between C_i and C_j at the community level. Let $p = \langle C_i, \ldots, C_j \rangle$ be the simple path that connects C_i and C_j in $\mathcal{H}$ (i.e., the path from C_i to A and that from C_j to A in Lines 2-3 of Algorithm 2). Since all nodes on p, except C_i and C_j, are virtual communities, we need to first convert p into a path of actual communities. Recall from Section IV-C that at each virtual community in $\mathcal{H}$, we keep a set of community edges. Let E_{com} be the union of the set of community edges at each virtual community on p. From E_{com}, we can construct a graph, G_{com}. Note that G_{com} must contain at least one path of actual communities from C_i to C_j, because C_i and C_j are reachable to each other in $\mathcal{G}$. Thus, the path that connects C_i and C_j at the community level can be computed as the shortest path from C_i to C_j in G_{com}.

It has been shown in [9] that shortest path is inadequate in capturing the relation between query nodes. We also do not use shortest path to find the relation between query nodes within a community. However, here the context in which we apply shortest path is different, because the distance between two communities in the community hierarchy tree does show how close they are to each other. For this reason, the weight of each community edge in G_{com} is defined as the level at which the community edge is kept in the hierarchy tree, since the level of the hierarchy tree shows the time the communities are combined. The higher the level (the greater the edge weight), the later are two communities combined and so the further away is their relationship. We further illustrate the concept by the following example.

Example 11: Consider the graph in Figure 2 and the hierarchy tree in Figure 3. For clear illustration, let us assume that we obtain the set of actual communities at Level 7 instead of Level 9; that is, the set of actual communities is $\{C_1, C_2, C_6, C_7, C_8\}$. The set of community edges at each virtual community is as follows: we have $\{(C_7, C_8)\}$ at Level 8, $\{(C_6, C_7), (C_6, C_8)\}$ at Level 9, $\{(C_1, C_2)\}$ at Level 10, and $\{(C_1, C_6), (C_2, C_8)\}$ at Level 11. Suppose that the query nodes are in C_1 and C_7. We collect the community edges along the two paths from C_1 and C_7 to their ancestor C_5, and construct G_{com} accordingly as shown in Figure 9.

There are four simple paths connecting C_1 and C_7 but the shortest path is $\langle C_1, C_6, C_7 \rangle$. It can also be seen from Figure 2 that this shortest path can indeed capture the relationship between the two communities. □

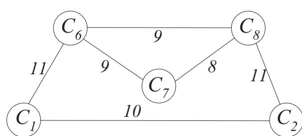


Fig. 9. G_{com} for Example 11

We now discuss how to find the actual path to connect two communities C_i and C_j based on the discovered community path (Line 11 of Algorithm 2). Let $p = \langle C_i, C_x, \ldots, C_y, C_j \rangle$ be the shortest path between two communities C_i and C_j in G_{com}. We utilize p to find the actual path connecting the query nodes in C_i and C_j. A conceptual view of how we apply p is depicted in Figure 10.

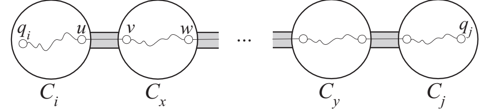


Fig. 10. A Conceptual View of Inter-Community Connection

Recall that each community edge is associated with a set of connecting edges. Thus, we start from C_i and find the highest-value path from a query node to some u, where (u, v) is a connecting edge. We find this highest-value path using the BFS strategy in Section V-C but we do not run dynamic programming since we only need to pick one path. Then, we find the highest-value path from v to w in C_x. This process goes on until we reach C_j, where we connect a query node in C_j with the highest-value path to a connecting edge from C_y. In the same way, we find the actual path from C_j to C_i because the connection is direction-aware. We then select the path with higher overall value to connect the query nodes in C_i and C_j in the answer graph.

The overall connection between C_i and C_j is controlled by both inter- and intra- community concepts. As depicted in Figure 10, the quality of the connection at the community level is ensured by the community hierarchy tree based on modularity. When the connection goes inside a community, we apply the concepts of information throughput of nodes and information flow of paths to compute the highest-value intra-community path.

Finally, in Algorithm 2, we first sort the shortest paths in Line 8 and terminate the process when all query nodes are connected in Lines 12-13. This is a greedy algorithm that computes a connected answer graph by selecting the best inter-community path at each step. Since the inter-community connection is at a much coarser level than the intra-community connection, connecting all query nodes by the greedy strategy suffices, though computing the connection between all communities in $\mathbb{C}$ is also possible and incurs only little overhead.

We now analyze the complexity of the inter-community connection. First, finding the shortest path in G_{com} takes $\mathcal{O}(|E_{com}| \log |V_{com}|)$ time and $(\mathcal{O}(|E_{com}| + |V_{com}|)$ space. Since the sets of community edges kept by the virtual communities are disjoint and we only require the virtual communities on the simple path from C_i to C_j in $\mathcal{H}$, $|E_{com}|$ is small and so is $|V_{com}|$. In total we need to find $(|\mathbb{C}|(|\mathbb{C}|-1)/2)$ shortest paths, but $|\mathbb{C}| \leq |\mathcal{Q}|$ and in general a user does not ask a query with many query nodes.

Computing the actual path from the shortest path $p = \langle C_i, \ldots, C_j \rangle$ takes $\sum_{C_k \in p} \mathcal{O}(|C_k|^2)$, where $\mathcal{O}(|C_k|^2)$ is the time for computing the voltage and information throughput of the nodes in C_k. Note that the time taken for the BFS path

selection is dominated by $\mathcal{O}(|C_k|^2)$. Finally, the space requirement is $\mathcal{O}(MAX\{|C_k|^2\})$. Since the size of a community is small, both the time and space are also small.

VII. Experimental Results

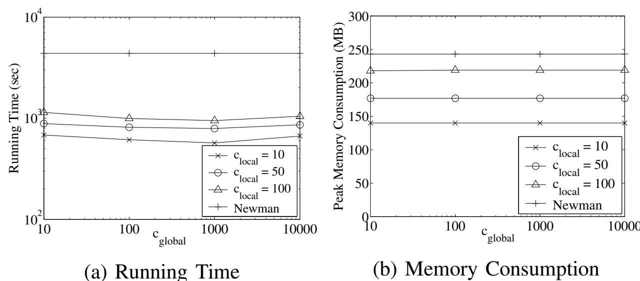
We now evaluate the performance of our algorithm for object connection discovery. We run all experiments on an AMD Opteron 248 with 1GB RAM, running Linux 64-bit.

We use the DBLP co-authorship dataset modeled as a graph. The graph has approximately 316K nodes and 1,834K edges, where a node represents an author and the edge weight is the number of papers co-authored between two authors.

A. Performance of Community Partition

We first evaluate the performance of community partition by our greedy algorithm. We test three settings of c_{local} (10, 50 and 100) and four settings of c_{global} (10, 100, 1000, 10000). We also compare with Newman's algorithm [13].

Figure 11(a) shows that our algorithm is about an order of magnitude faster than Newman's when $c_{local} = 10$. The result also shows that when c_{local} increases, the efficiency decreases, which demonstrates the effectiveness of Heuristic 1.



(a) Running Time (b) Memory Consumption

Fig. 11. Performance of Community Partition

For the effect of Heuristic 2, i.e., c_{global}, the performance is the best when $c_{global} = 1000$. When $c_{local} = 10$, the running time is 682, 613, 570, and 667 seconds for $c_{global} = 10$, 100, 1000, and 10000, respectively. The result can be explained as follows. When c_{global} is too small, many items are not kept in the global max-heap and hence we need to rebuild the heap more often. When c_{global} is too large, there are too many items in the heap and hence the update of the heap takes longer.

Figure 11(b) shows that the peak memory consumption of our algorithm increases when c_{local} increases, since the size of the local max-heaps increases when c_{local} increases. Increasing c_{global}, i.e., the size of the global heap, from 10 to 10000 only increases the memory usage for less than 1 MB since we have only one global heap. However, in all cases, our algorithm consumes considerably less memory than Newman's.

We also record that the value of the modularity of the optimal community partition obtained by the greedy algorithm is 0.71 (note that all the algorithms compute the same partition). According to Newman [13], a modularity value of greater than 0.3 indicates a significant community structure. Therefore, 0.71 is a very high value of modularity and indicates a high-quality community partition.

B. Semantics of Answer Graph: A Case Study

We conduct a case study to compare our answer graph with the *center-piece subgraph* (*CEPS*) [10]. This study aims to first provide a more intuitive view on the answer graphs obtained by our algorithm and CEPS. Then, we perform a more systematic comparison in the following subsection.

We use the query {*Jim Gray*, *Jennifer Widom*, *Michael I. Jordan*, *Geoffrey E. Hinton*}. The four scholars are from two different communities: *Gray* and *Widom* are from the *database* community, while *Jordan* and *Hinton* are from the *machine learning* community. This is clearly captured by our answer graph as shown in Figure 1, which is displayed in Section I.

Figure 12 shows the answer graph of CEPS, which is very similar to our answer graph. The similarity is because both our algorithm and CEPS find nodes that are closely related to the query nodes in order to connect them. Thus, the result shows that both algorithms are able to capture important nodes and paths related to the query nodes. However, our method not only finds a good connection between all query nodes, but also for query nodes that are in the same context, we put more emphasis on their connection than the existing methods. Compare Figure 1 with Figure 12, we clearly see a stronger connection between *Gray* and *Widom* through both *Ceri* and *Hellerstein* in our answer graph than CEPS.

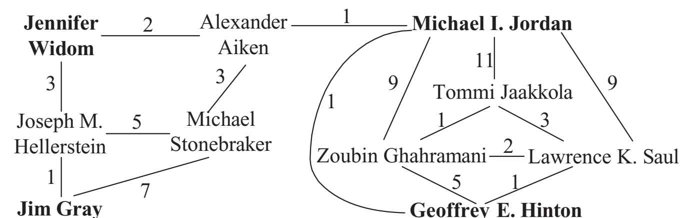


Fig. 12. The Answer Graph of CEPS

Although the quality of the answer graphs is comparable, our algorithm significantly outperforms CEPS: we take only 0.33 seconds to compute our answer graph, while the computation of the CEPS takes 925 seconds. We further compare the two methods using more systematic measures as follows.

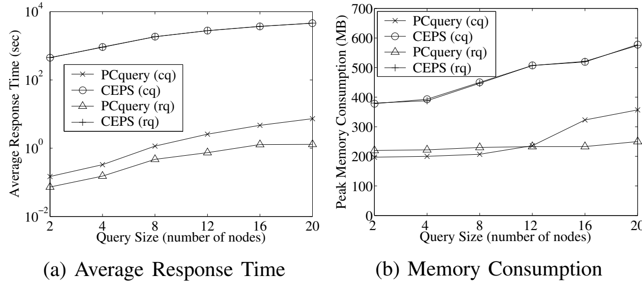
C. Performance of Object Connection Discovery

We compare the performance of our algorithm, PCquery, with CEPS [10]. We set the budget to be twice of the query size. We also verify that the answer graphs obtained by PCquery and CEPS are of roughly the same size. Other settings of CEPS are as its default.

We generate two types of queries: *in-community queries* and *random queries*, which are abbreviated as *cq* and *rq* in the figures. For in-community queries, the nodes in a query are randomly selected from a randomly selected community. For random queries, the nodes in a query are randomly selected from the set of all nodes in the dataset. We generate 100 queries for each type and test the query size from 2 nodes to 20 nodes.

Figure 13(a) reports the average running time of finding the connection for a query. The result shows that PCquery is more

than three orders of magnitude faster than CEPS for both query types. We find that PCquery takes more time to process an in-community query than a random query. This is because the computation of the intra-community connection by dynamic programming is more costly than that of the inter-community connection by tracing the community hierarchy tree.

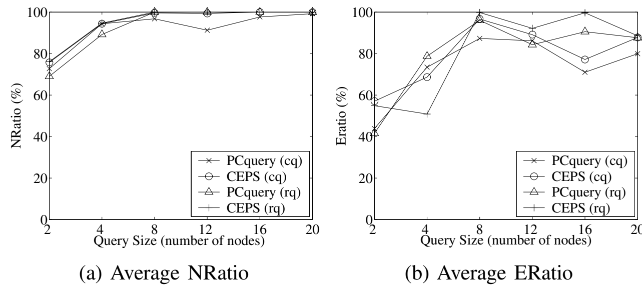


(a) Average Response Time (b) Memory Consumption

Fig. 13. Efficiency of PCquery and CEPS

Figure 13(b) reports the peak memory consumption during the entire running process. The result shows that PCquery also consumes significantly less memory than CEPS in all cases.

In addition to the comparison on efficiency, we also compare the quality of the answer graphs obtained by PCquery and CEPS. For the fairness of comparison, We use the quality metrics proposed in CEPS [10], *NRatio* and *ERatio*, which indicate the percentage of important nodes and edges that are captured by an answer graph, respectively. We report the result in Figure 14.



(a) Average NRatio (b) Average ERatio

Fig. 14. Quality of Answer Graph

Figure 14 shows that both PCquery and CEPS obtain high-quality answer graphs. Both NRatio and ERatio of our answer graphs are comparable to those of CEPS, although on average those of CEPS are slightly better. Considering our algorithm is three orders of magnitude faster and also consumes significantly less memory, we can conclude that our method is both efficient and effective.

VIII. Conclusions

We propose context-aware object connection discovery in a large graph. We adopt a partition-and-conquer approach to achieve both high performance efficiency and high quality results. Our method first partitions a large graph into a set of communities. The concept of community not only naturally defines the context of the nodes, but also significantly improves the efficiency of connection discovery since a community is much smaller than the original graph. We compute the connection between query nodes first at the intra-community level by maximizing the information throughput of the nodes and the information flow of the paths in the answer graph, and then at the inter-community level by retaining the close relation between the communities as defined by modularity. The quality of both the intra- and inter- community connection is thus controlled by the integration of information throughput/flow and modularity. We verify by experiments that our community partition algorithm is efficient and the set of communities obtained has high quality. We also show that our method obtains comparable high-quality answers as the state-of-the-art algorithm, but is more than three orders of magnitude faster and consumes significantly less memory.

Acknowledgement. This work is partially supported by RGC GRF under grant number CUHK419008 and HKUST617808. We thank Mr. Hanghang Tong and Prof. Christos Faloutsos for providing us the source code of CEPS.

References

[1] X. Yan, P. S. Yu, and J. Han, "Graph indexing based on discriminative frequent structure analysis," *ACM TODS*, vol. 30, pp. 960–993, 2005.

[2] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases," in *SIGMOD*, 2007, pp. 857–872.

[3] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: Tree + delta $>=$ graph," in *VLDB*, 2007, pp. 938–949.

[4] Y. Ke, J. Cheng, and W. Ng, "Correlation search in graph databases," in *KDD*, 2007, pp. 390–399.

[5] Y. Ke, J. Cheng, and W. Ng, "Efficient correlation search from graph databases," *To appear in TKDE*, 2008.

[6] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *PKDD*, 2000, pp. 13–23.

[7] X. Yan and J. Han, "Closegraph: mining closed frequent graph patterns," in *KDD*, 2003, pp. 286–295.

[8] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: mining maximal frequent subgraphs from graph databases," in *KDD*, 2004, pp. 581–586.

[9] C. Faloutsos, K. S. McCurley, and A. Tomkins, "Fast discovery of connection subgraphs," in *KDD*, 2004, pp. 118–127.

[10] H. Tong and C. Faloutsos, "Center-piece subgraphs: problem definition and fast solutions," in *KDD*, 2006, pp. 404–413.

[11] Y. Koren, S. C. North, and C. Volinsky, "Measuring and extracting proximity in networks," in *KDD*, 2006, pp. 245–255.

[12] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, p. 066113, 2004.

[13] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Physical Review E*, vol. 69, p. 066133, 2004.

[14] F. Wu and B. A. Huberman, "Finding communities in linear time: a physics approach," *The European Physical Journal B - Condensed Matter and Complex Systems*, vol. 38, no. 2, pp. 331–338, 2004.

[15] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Trawling the web for emerging cyber-communities," *Comput. Netw.*, vol. 31, no. 11-16, pp. 1481–1493, 1999.

[16] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "Extracting large-scale knowledge bases from the web," in *VLDB*, 1999, pp. 639–650.

[17] R. Kumar, U. Mahadevan, and D. Sivakumar, "A graph-theoretic approach to extract storylines from search results," in *KDD*, 2004, pp. 216–225.

[18] D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *VLDB*, 2005, pp. 721–732.

[19] Y. Dourisboure, F. Geraci, and M. Pellegrini, "Extraction and classification of dense communities in the web," in *WWW*, 2007, pp. 461–470.

[20] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, p. 066111, 2004.