Efficient Team Formation in Social Networks based on Constrained Pattern Graph

Yue Kou[1], Derong Shen[1], Quinn Snell[2], Dong Li[1], Tiezheng Nie[1], Ge Yu[1], Shuai Ma[3]
[1]*Northeastern Universtiy,* China [2]*Brigham Young University*, United States [3]*Beihang University,* China
{kouyue, shenderong, nietiezheng, yuge}@cse.neu.edu.cn lidongmason@163.com snell@cs.byu.edu mashuai@buaa.edu.cn

***Abstract*—Finding a team that is both competent in performing the task and compatible in working together has been extensively studied. However, most methods for team formation tend to rely on a set of skills only. In order to solve this problem, we present an efficient team formation method based on Constrained Pattern Graph (called CPG). Unlike traditional methods, our method takes into account both structure constraints and communication constraints on team members, which can better meet the requirements of users. First, a CPG preprocessing method is proposed to normalize a CPG and represent it as a CoreCPG in order to establish the basis for efficient matching. Second, a Communication Cost Index (called CCI) is constructed to speed up the matching between a CPG and its corresponding social network. Third, a CCI-based node matching algorithm is proposed to minimize the total number of intermediate results. Moreover, a set of incremental maintenance strategies for the changes of social networks are proposed. We conduct experimental studies based on two real-world social networks. The experiments demonstrate the effectiveness and the efficiency of our proposed method in comparison with traditional methods.**

Keywords—team formation, social networks, Constrained Pattern Graph, Communication Cost Index

I. INTRODUCTION

Team formation in social networks is essential for an organization or institute's viability. However, most existing works tend to rely on a set of skills and only considers the size of team's diameter, the size of the minimum spanning tree (MST) [1] or the total communication cost among members [2-6]. Usually, people with different skills in a team have different degrees of communication. For example, in a team, project managers usually communicate more with software developers than with test engineers, while software developers always communicate more with test engineers than with secretaries. The communication relationship among different members forms structure constraints. Also the degree of their communication shows communication constraints. Neither structure constraints nor communication constraints can be characterized via a simple skill set. Therefore, pattern graphs are used by some methods to represent users' requirements. These solutions are mainly based on the idea of subgraph isomorphism [7-20]. Although some subgraph isomorphism supports matching for constraint pattern graph (e.g. [10, 13, 17, 19]) and can be applied to team formation, it aims to return the entire matching subgraphs and incurs a NP-complete problem [21]. However, in some applications, such as group collaborative learning and expert recommendation, people are interested in finding nodes, rather than the entire subgraphs. The uniqueness of the team formation problem leaves much room for improvements. For example, we can reduce the size of indexes and focus on smaller candidate set by only considering the member nodes. Let us consider the following motivating scenarios.

Example 1.1: Fig. 1(a) depicts a social network which includes a set of individuals (v_1~$v_{2m+2n+6}$). Each node represents a person with one (or multiple) label(s) such as Project Manager (PM), Software Engineer (SE), Database Developer (DD), Test Engineer (TE), Data Analyst (DA), Secretary (S) and so on. Here, we use different colors to denote different skills. Each edge indicates a collaboration relationship between two persons. The weight of an edge reflects the distance between the two ends. The more frequently they communicate with each other, the closer they are, and the smaller the edge weight. Suppose a project needs 6 types of people, namely, PM, DD, SE, TE, DA and S. As shown in Fig. 1(b), we can represent the task requirement as a Constrained Pattern Graph (called CPG), where each edge in the CPG is labeled with a constant k indicating the communication cost of the two nodes (in this paper it represents the upper threshold of the shortest path length) connected by the edge in social networks should be no larger than k. For example, a PM needs to be associated with a SE within the distance of 2. We will map the CPG to the social network to find a team that satisfies the task requirement.

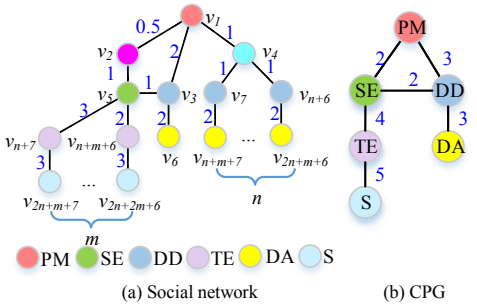


Fig. 1. CPG based team formation

Suppose the matching order of nodes in the CPG is (PM, SE, TE, S, DD, DA), and in the social network there are m partial mappings consisting of (PM, SE, TE, S) which have to be combined with n+1 partial mappings consisting of (DD, DA). So we have to consider m×(n+1) kinds of combination in all, most of which are false positive and redundant. However, if we use the matching order (PM, SE, DD, TE, S, DA), we can know early the n partial mappings consisting of (DD, DA) are not satisfied because the communication cost between DD and SE for them is larger than 2. They can be eliminated before mapping with TE, S or DA. At this time, it only leads to n+1+m partial mappings.

In order to improve the efficiency of matching process, some approaches are proposed to generate a matching order of nodes. But they are mainly used for subgraph isomorphism, which aims to find the entire matching subgraphs in the data graph. In addition, most subgraph isomorphism methods perform edge-to-edge mappings, which are somewhat too strict to identify teams in real-world social networks. Some extensions of subgraph isomorphism are studied to extend mappings from edge-to-edge to edge-to-path, but there are no communication constraints on the paths. In our approach, we

Dong Li is the corresponding author.

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00082

only need to check the communication cost between nodes, without needing to check whether there is an edge between them. Also, communication constraints in a CPG are useful for us to filter out more irrelevant nodes as early as possible, which can further reduce the size of intermediate results.

Example 1.2: Suppose we have already constructed a team according to the requirements in Example 1.1. Let's consider two situations. First, a new label 'TE' is added to the node v_x in the social network. It means the person v_x acquire a new skill. Then how is the team built as before? Second, suppose one of members becomes unavailable. Then how are adjustments made to the current team?

In this example, if we repeat the whole procedure of team formation, it will consume lots of time. In fact, the change only affects partial matches. For the first situation, the node v_x will become a new candidate due to its new label. We only need to check partial matches affected by it. These matches constitute an affected region. Only the communication relationships in the affected region rather than that in the whole social network need to be rechecked. For the second situation, deletions of nodes might only increase the communication cost of some pairs of nodes. Compared to the previous candidates, no new candidates will be chosen as a result. We just need to choose the new result from the previous candidates rather than from the whole network.

With incremental maintenance, the cost of team formation can be reduced greatly. Although some incremental team formulation methods (e.g. [10, 15, 18, 22-25]) have been proposed, they mainly focus on the maintenance for the entire matching subgraphs (e.g. [24, 25]) or the ranking of result data (e.g. [18]). Most methods only support the incremental maintenance for the changes of network structure rather than the changes of nodes' labels.

However, team formulation in social networks based on Constrained Pattern Graph is a highly challenging problem. The major challenges are as follows: First, how to implement the efficient matching between a CPG and social networks? Traditional pattern graph matching is an NP-complete problem. The input pattern graph for subgraph isomorphism cannot fully reflect the constraints in users' requirements. Their edge-to-edge mappings are too restricting for team formation. Therefore, we need to propose a new efficient matching approach for team formation. Second, how to improve the efficiency of maintenance for the result of team formation? Current techniques only support the incremental maintenance for the changes of network structure. Thus we need to consider more types of changes and the corresponding incremental maintenance strategies.

In this paper, we present an efficient team formation method based on CPG. Unlike traditional methods, we consider both structure constraints and communication constraints on team members, which can better meet the requirements. We make the practial contributions:

(1) A CPG preprocessing method is proposed to normalize a CPG and represent it as a CoreCPG in order to establish the basis for efficient matching.

(2) An auxiliary data structure (called Communication Cost Index, CCI) is designed to speed up the matching between the CPG and social networks. First, we construct a CCI in a top-down way starting from the core node in the CPG. Then, we refine the constructed CCI in a bottom-up fashion to reduce the storing space of the CCI.

(3) Based on the CoreCPG and the constructed CCI, a node matching algorithm is proposed to determine the matching order of nodes in the CPG. The cost model for matching is defined and three heuristic matching rules are proposed to minimize the total number of intermediate results.

(4) The incremental maintenance strategies are proposed for the changes of social networks including the insertion or deletion of nodes' labels, insertion or deletion of edges, insertion or deletion of nodes, increase or decrease of edges' weights.

(5) The extensive experimental studies based upon two real-world social networks are conducted. The experiments demonstrate the effectiveness and the efficiency of our proposed method.

The rest of this paper is organized as follows. Section II reviews the related work. Section III formulates the problem and gives an overview of our solution. Section IV presents the CPG preprocessing method. Section V and Section VI propose the CCI construction algorithm and the CCI-based node matching algorithm respectively. Section VII proposes the incremental maintenance strategies. Section VIII shows the experimental results and Section IX makes conclusion.

II. RELATED WORK

Some communication cost based team formation methods are proposed to find a group of members such that not only their skills can cover a set of required skills, but also the size of team's diameter (or MST) [1] or the total communication cost among members [2-6] is minimum. For example, in [2], a community detection algorithm is proposed to alternatively find a densely-connected subgraph based on a set of given nodes. In [3] and [4], some team formation algorithms are proposed to not only minimize the communication cost among team members, but also minimize the cost between team members and team leaders. In [5], the influence maximization is combined with team formation to facilitate the organization of social events. In [6], an interactive team formation system is designed to model teams as hierarchical structure which can reflect the ubiquitous nature of teams.

Subgraph isomorphism based team formation is to find all the matches of the nodes in social networks according to a given pattern graph. Because the problem of subgraph isomorphism is NP-complete, some approaches are proposed to reduce the cost of matching. For example, in [7] and [8], infrequent-labels first strategy and path enumeration are proposed respectively to generate a matching order of nodes. In [9], query decomposition is proposed to minimize the total number of intermediate results. In [10], a BGS (Bounded Graph Simulation) based approximate solution is proposed to find inexact matches between the pattern graph and social networks. In [11], a system called ExpFinder is presented to find experts in social networks based on graph pattern matching. In [12], an indexing algorithm that selects features directly from data graphs is presented. In [13], QGPs (quantified graph patterns) are proposed and parallel scalable algorithms for quantified matching are discussed. In [14-16], extensions are studied to extend mappings from edge-to-edge to edge-to-path. In [17], exact edge matching is replaced with the notion of path matching constrained by a path length. In [18-20], some methods for finding top-K teams are proposed.

In [18], a graph pattern matching approach for top-K team formation is proposed by incorporating both structure constraints and capacity bounds. In [19], a multi-constrained top-K graph pattern matching method is proposed. In [20], a two-level-based framework is designed to recommend top-K teams for spatial crowdsourcing tasks.

In order to further improve the efficiency of team formation, some incremental maintenance strategies [10, 15, 18, 22-25] are proposed. In [10] and [22], approximate algorithms based on bounded graph simulation are designed for incrementally finding matches when data graphs are updated. In [15], an index is built to incrementally record the shortest path length range between different label types, and the affected parts will be identified when pattern graphs or data graphs are updated. In [23], an incremental pattern mining method is proposed, which uses different pruning strategies as well as an index structure to enable fast access to matching and fast updates.

The differences between our work and existing work are as follows: First, most subgraph isomorphism performs edge-to-edge mappings. Although some extensions of subgraph isomorphism support edge-to-path matching, they focus on either the overall communication cost or the aggregated contribution of a group. In this paper, we focus on more specific query constraints such as each pair's communication cost and each member's skills. Second, some subgraph isomorphism supports matching for constraint pattern graph (e.g. [10, 13, 17, 19]), but they aim to find the entire matching subgraphs in the data graph. Although most of them can be applied to team formation by delivering the entire matching subgraphs in data graph, they incur a high time complexity. Different from them, our goal is to find team members, but not the subgraphs containing the members. Third, most incremental graph pattern matching methods mainly focus on the incremental maintenance for the result of subgraph isomorphism (e.g. [24, 25]), or the ranking of result data (e.g. [18]). Although some methods (e.g. [15]) focus on the incremental maintenance for the matching nodes, they are only suitable for the changes of network structure, rather than the changes of labels. We consider more types of changes and propose the corresponding incremental maintenance strategies for the matching nodes.

III. Solution Overview

Suppose there are n individuals in a social network which is modeled as a data graph G.

Definition 3.1 (Data Graph): The data graph is a weighted graph $G=(V_D, E_D, L_D, W_D)$, where V_D is the vertex set of size n to denote individuals, $E_D \subseteq V_D \times V_D$ is the edge set to denote the communication relationships among individuals, L_D is a function such that $L_D(v)$ is a set of labels to denote the skills owned by each node $v \in V_D$, and W_D is a function defined on E_D such that $W_D(v, v')$ is the weight between v and v' ($(v, v') \in E_D$).

The weight of an edge in G reflects the distance between the two ends. It is relevant to the degree of communication between them. The more frequently they communicate with each other, the smaller the edge weight is. The length of a path in G equals to the sum of the weights of all the edges on the path. We build the shortest path length matrix C to record the shortest path length between each pair of nodes in G. For each node pair (v_i, v_j), we use the weights to quantify their communication cost $C(v_i, v_j)$ which is defined as the length of the shortest path between v_i and v_j.

Definition 3.2 (Constrained Pattern Graph, CPG): A CPG is a query graph $Q=(V_Q, E_Q, L_Q, C_Q)$, where V_Q and E_Q are the node set and the edge set respectively, L_Q is a function defined on V_Q such that $L_Q(u)$ is the label of the node u ($u \in V_Q$), and C_Q is a function defined on E_Q such that $C_Q(u, u')$ is the upper threshold of the communication cost between u and u' ($(u, u') \in E_Q$).

For example, Fig. 1(b) depicts a CPG. We use V_Q, E_Q and L_Q in the CPG to describe structure constraints in user's requirements and use C_Q to express communication constraints among team members. The nodes in the CPG have the same labels as the nodes in G. But the edges in the CPG and in G are different. The former reflects either direct relationships or indirect relationships (e.g. the node labeled with PM and the node labeled with SE are not requested to be connected directly in G). Each edge in the CPG has a weight as the bounded communication cost. But edges in G only reflect direct relationships, that is, two nodes connected by an edge in G can communicate each other directly.

One challenge is that by recommending only one team to the requester, it might fail to recruit the team. Thus we consider top-K team formation.

Definition 3.3 (Top-K Team Formation based on CPG): Given a data graph G and a CPG (suppose $|V_Q|=m$), the problem of top-K team formation is to find K teams each of which has m nodes in G (denoted as V_m) such that:

(1) Nodes in V_m should satisfy the structure constraints in the CPG. There are m kinds of label mappings between V_Q and V_m. For each mapping (u, v) in each label ($u \in V_Q$, $v \in V_m$), $L_Q(u) \subseteq L_D(v)$. For each edge (u, u') in E_Q, there exists a path from v to v' in G such that (u, v) and (u', v') are two mappings.

(2) Nodes in V_m should satisfy communication constraints in the CPG. For each edge (u, u') in E_Q, suppose (u, v) and (u', v') are two mappings, the communication cost (i.e. the shortest path length) between v and v' in G should not be larger than $C_Q(u, u')$. Also the total communication cost of all edges constructing the final team should be top-K minimum.

To efficiently find the top-K teams, we have to address two key issues: (1) how to develop an effective index to speed up matching, and (2) how to devise a heuristic matching order in the computation.

Example 3.1: In Fig. 1, both the node set $\{v_1, v_3, v_5, v_6, v_{n+7}, v_{2n+m+7}\}$ and the node set $\{v_1, v_3, v_5, v_6, v_{n+m+6}, v_{2n+2m+6}\}$ constitute a group of candidate nodes respectively. Their total communication cost is 12.5 and 11.5 respectively. So the second group is more dominant than the former to be the final result.

Definition 3.4 (Incremental Maintenance): Given a data graph G, a list updates ΔG to G, the current index and the current query result, the problem of incremental maintenance is to determine the affected regions in the index, and then update the index and the result, but not to re-perform the whole procedure of team formation.

Frequently used notations in this paper are summarized in Table I. Our solution is shown in Fig. 2. If the whole CPG is taken as the matching object, the matching cost is relatively high. Therefore, we adopt the idea of divide and conquer, first

divide the CPG into several parts, and then match them one by one to reduce the matching cost. It mainly includes four parts: 1) CPG preprocessing (see Section IV). The CPG is normalized and represented as a CoreCPG, which consists of one core part with a core node and several noncore parts, in order to establish the basis for efficient matching. 2) CCI construction (see Section V). In order to speed up node matching, an auxiliary index called CCI is constructed starting from the core node in the CPG. 3) CCI-based node matching (see Section VI). Based on the CoreCPG and the constructed CCI, the matching order of nodes in the CPG is determined to minimize the total number of intermediate results. The candidate with the top-K minimum total communication cost will be selected to form a team. 4) Incremental maintenance (see Section VII). According to the updates ΔG of the data graph and the current CCI, the query result is maintained incrementally.

TABLE I. NOTATIONS

Notation	Meaning
G and Q	Data graph and CPG
$L_D(v)$ and $L_Q(u)$	Labels of node v in G and label of node u in Q
$C(v, v')$	Communication cost between v and v' in G
$C_Q(u, u')$	Communication constraint between u and u' in Q
Q_T	Minimum spanning tree of Q
Q_C and u_{core}	Core part in Q and the core node in Q_C
Q_N	Noncore part in Q
$M(u)$	A set of nodes in G matching with u
NS and ES	Node set and edge set of an index node in CCI

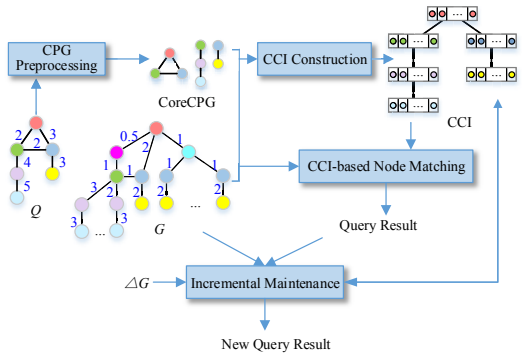


Fig. 2. Solution overview

IV. CPG PREPROCESSING

We propose a CPG preprocessing method that normalizes the CPG and reprensents it as a CoreCPG in order to establish the basis for efficient matching.

A. CPG Normalization

A CPG is often given by users. Inevitably there exists some data redundancy. It is necessary to normalize the CPG before matching. According to the triangle inequality theorem, any side of a triangle is always shorter than the sum of the other two sides. The communication cost (i.e. the length of the shortest path) among nodes in G also follows this theorem. For any three nodes v_i, v_j and v_k in G, the triangle inequality theorem states that $C(v_i, v_j)$ should not be larger than the sum of $C(v_i, v_k)$ and $C(v_k, v_j)$. The triangle inequality theorem assures the following theorem.

Theorem 4.1: For any triangle (suppose u_i, u_j and u_k are three nodes constituting the triangle) in a CPG, if $C_Q(u_i, u_j)$ is larger than the sum of $C_Q(u_i, u_k)$ and $C_Q(u_k, u_j)$, then (u_i, u_j) is a redundant edge.

Proof: Suppose (u_i, v_i), (u_j, v_j) and (u_k, v_k) are three mappings between Q and G. If $C(v_i, v_k) \le C_Q(u_i, u_k)$ and $C(v_k, v_j) \le C_Q(u_k, u_j)$ are satisfied respectively, due to $C(v_i, v_j) \le C(v_i, v_k) + C(v_k, v_j)$, then $C(v_i, v_j) \le C_Q(u_i, u_k) + C_Q(u_k, u_j)$ must be satisfied. Also, because $C_Q(u_i, u_j)$ is larger than $C_Q(u_i, u_k) + C_Q(u_k, u_j)$, $C(v_i, v_j)$ must be no larger than $C_Q(u_i, u_j)$. That is, if the constraints $C_Q(u_i, u_k)$ and $C_Q(u_k, u_j)$ are satisfied, the constraint $C_Q(u_i, u_j)$ must be satisfied too. Therefore, the edge (u_i, u_j) in the CPG is redundant.

The goal of CPG normalization is to remove all the redundant edges from the CPG. For simplicity, in the remainder of this paper, we use the term "CPG" to refer to the "normalized CPG".

B. CoreCPG Representation

Definition 4.1 (CoreCPG): A CoreCPG consists of one core part Q_C with a core node u_{core} and a set of noncore parts $QNset$. Here the core part $Q_C=(V_C, E_C, L_Q, C_Q)$ is the minimal connected subgraph in Q, which contains all non-tree edges (edges in Q but not appearing in Q's MST (denoted as $Q_T(V_T, E_T)$)). Each noncore part $Q_N=(V_N, E_N, L_Q, C_Q)$ in $QNset$ is the subgraph of Q consisting of all other edges not in Q_C. The node u in V_C with fewer matching nodes in G (having the same label as u, denoted as $M(u)$) and closer communication with other nodes in V_C is selected as u_{core}.

Since the cost for matching non-tree edges will increase sharply with the increase of intermediate results (see Section VI), Q_C that contains all such edges should be matched first, and then $QNset$. For u_{core}, fewer matching nodes means less intermediate results being generated, while closer communication means lower traverse cost. So u_{core} will be matched preferentially over the other nodes in Q_C. Therefore we need to represent the normalized CPG as a CoreCPG. Our method is similar to [9], but we use different condition to select the core node. The communication constraints are considered. The pseudocode is shown in Algorithm 1.

Algorithm 1: CoreCPG representation

Input: a normalized Q
Output: Q_C, u_{core}, $QNset$

```
1  Q_T ← generateMST(Q);
2  Q_C ← determineCore(Q, Q_T);
3  for each node v_i in V_C do
4    v_i.com ← 0;
5    for each edge (v_i, v_j) in E_C do
6      v_i.com ← v_i.com + C_Q(v_i, v_j);
7    v_i.coreDegree ← |M(v_i)| × v_i.com;
8  u_core ← argmin_{u_i ∈ V_C} u_i.coreDegree;
9  QNset ← determineNoncore(Q, Q_C);
10 return Q_C, u_core and QNset;
```

Step 1: We first generate Q's MST, i.e. $Q_T(V_T, E_T)$, where V_T and E_T have the same definitions as V_Q and E_Q respectively (Line 1). E_T is the subset of E_Q.

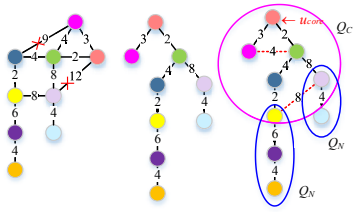
Step 2: According to Q_T, determine the core part Q_C (Line 2). We apply the method in [9] to determine Q_C.

Step 3: Determine the core node u_{core} in V_C (Line 3-8). We use communication constraints to select the core node in V_C. We quantify u's core degree as $|M(u)| \times \Sigma_i C_Q(u, u_i)$ (u, $u_i \in V_C$, $(u, u_i) \in E_C$) and select the node with the minimum value as the core node. When there are multiple nodes with

equal core degree, the node with the smallest identifier will be selected as the core node.

Step 4: Determine the set of noncore parts *QNset* in Q (Line 9). Note that there are some nodes that appear in both the core part and the noncore part (i.e. $V_C \cap V_N$). They act as the connection nodes between the two parts.

Example 4.1: As shown in Fig. 3(a), there are two edges removed from the CPG after normalization. Then the minimum spanning tree Q_T is generated (shown in Fig. 3(b)). The minimal connected subgraph of Q that contains all the non-tree edges is determined as Q_C (shown in Fig. 3(c)). It contains all the non-tree edges in Q (denoted as red dashed lines). Also the core node is selected from Q_C. Suppose for each node u in Q_C, the number of matching nodes (i.e. $|M(u)|$) is shown in Fig. 3(c), we can select the core node having the lowest core degree. Finally, each noncore part Q_N is determined. The CoreCPG consists of Q_C with the core node u_{core} and two Q_Ns in this example.



u	$\lvert M(u)\rvert$	$\Sigma_i C_Q(u, u_i)$	CoreDegree
	2	5	10
	3	7	21
	4	18	72
	8	6	48
	6	16	96
	7	20	140

(a)Normalization (b)Q_T generation (c) CoreCPG representation

Fig. 3. An example for CPG preprocessing

In fact, both CPG normalization and MST generation can remove the redundant edges from CPG. Besides the redundant edges, the non-tree edges can also be removed from MST. If without CPG normalization, we cannot tell whether a removed edge is a redundant edge or a non-tree edge. Then all the removed edges will be viewed as non-tree edges, resulting in higher cost of the subsequent node matching. So both of the steps are necessary.

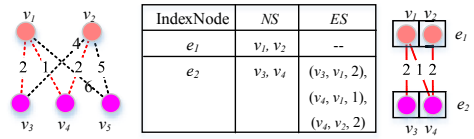
V. Communication Cost Index

Starting from the core node, we will construct a Communication Cost Index (CCI). It is used to compactly encode all possible embeddings of Q in G. Each embedding is a group of nodes satisfying the constraints in the CPG and is a candidate to form a team.

A. CCI Data Structure

Given a CPG and a data graph, a CCI is a tree with the same structure as Q_T. Each index node in the CCI carries the same label as the corresponding node in Q_T. Also the parent-child relationships among nodes are the same in Q_T and the CCI. The data structure of the CCI is as follows. 1) Each index node e in the CCI corresponds to a node u in Q_T. It is a matching node set of u, which includes u's matching nodes $M(u)$ (or denoted as $M(e)$ due to the equivalence between e and u) in G. 2) The relationships between two adjacent index nodes in the CCI (e.g. e_i and e_j) are many-to-many. An index edge between v ($v \in M(e_i)$) and v' ($v' \in M(e_j)$) is built if and only if $C(v, v')$ is not larger than $C_Q(e_i, e_j)$. For each index node, we use a **node set** *NS* and an **edge set** *ES* to store its matching nodes and the communication relationships with its parent node, respectively.

Example 5.1: Let's take the first two nodes in the table of Fig. 3(c) as an example (denoted as u_1 and u_2 here). Suppose that their matching nodes are $M(u_1)=\{v_1, v_2\}$ and $M(u_2)=\{v_3, v_4, v_5\}$, and the relevant fragment of G is shown in Fig. 4(a). Because the relationships among the matching nodes may be indirect, we use dashed lines with the corresponding communication cost to represent the relationships. There are six paths between nodes in $M(u_1)$ and nodes in $M(u_2)$. The corresponding index nodes in the CCI are shown in Fig. 4(b). There are two index nodes (denoted as e_1 and e_2) corresponding to u_1 and u_2 respectively. Only three paths in Fig. 4(a) are left which satisfying communication constraints in the CPG (i.e. no larger than 3). So the node sets of e_1 and e_2 are $\{v_1, v_2\}$ and $\{v_3, v_4\}$, respectively. And there are three index edges between them which are stored in the edge set of e_2, i.e. $\{(v_3, v_1, 2), (v_4, v_1, 1), (v_4, v_2, 2)\}$.



(a) Fragment of G (b) CCI

Fig. 4. An example for index nodes in a CCI

Remark. There are also some existing auxiliary data structures (e.g. $\text{Turbo}_{\text{ISO}}$ [8], CPI [9] and S-Index [17]) to speed up subgraph matching. However, the CCI is inherently different. First, for most indexes, the entire matching subgraphs in the data graph need to be found and stored. Each index edge represents a direct relationship. However, CCI-based node matching aims to find the matching nodes satisfying the constraints in the CPG. Each index edge represents a path. Only the two ending nodes on each path need to be stored. Second, although some indexes support edge-to-path mappings and path matching constrained by a path length, they do not use communication constraints to refine, resulting in large storage space and traversal cost. By checking communication constraints, we can prune unpromising nodes during CCI construction. As a result, the CCI is better suited for team formation with less storage space.

B. CCI Construction

The CCI is constructed to compactly encode all possible embeddings of Q in G and to speed up the matching between them. Similar to a parent-child relationship in Q_T, there is also a parent-child relationship between two adjacent nodes in the CCI. It is constructed in a top-down fashion. Given the core node u_{core}, Q_T, G and the shortest path length matrix C, the algorithm for CCI construction is shown in Algorithm 2.

Algorithm 2: CCI Construction

Input: u_{core}, Q_T, G, C
Output: *CCI*
1 *root.NS*←getMatchingNodes($L_Q(u_{core})$, G);
2 *root.ES*←∅;
3 *CCI*← *root*;
4 *CCI*← extendCCI(*root*, u_{core}, Q_T, G, C, *CCI*);
5 **return** *CCI*;

Step 1: Generate the root node of the CCI. Intuitively, the node with stronger pruning power should be the root of the CCI, so that it can be traversed first. As discussed in Section IV.B, the core node u_{core} has both fewer matching nodes and smaller communication cost with other nodes. Its pruning power is strongest, so it should be selected as the root of the

CCI. Its *NS* and *ES* are the set of matching nodes in G (sharing the common label with u_{core}) and ∅, respectively.

Step 2: Call Algorithm 3 recursively to generate the rest part of the CCI. Given the current CCI, a parent index node e_p in it (corresponding to the node u_p in Q_T), G and the shortest path length matrix C, Algorithm 3 is to generate the descendants of e_p. The children U_c of u_p in Q_T are checked at first. If U_c is null, e_p will not be extended because u_p is a leaf node. Otherwise, there will be $|U_c|$ child nodes of e_p being generated. For each node in U_c, it corresponds to an index node e_c which will be extended as a child node of e_p. The extension process is as follows (Line 4-16).

Algorithm 3: extendCCI

Input: the current *CCI*, a parent index node e_p, u_p, Q_T, G, C
Output: the extended *CCI*

1 $U_c \leftarrow$ getChildren(u_p, Q_T);
2 **if** U_c is not null **then**
3 **for** each node u_c in U_c **do**
4 $e_c.NS \leftarrow \emptyset$;
5 $e_c.ES \leftarrow \emptyset$;
6 $M(e_c) \leftarrow$getMatchingNodes($L_Q(u_c)$, G);
7 **for** each node v_i in $M(e_c)$ **do**
8 **for** each node v_j in $e_p.NS$ **do**
9 **if** $C(v_i, v_j) \le C_Q(u_c, u_p)$ **then**
10 $e_c.NS \leftarrow e_c.NS \cup \{v_i\}$;
11 $e_c.ES \leftarrow e_c.ES \cup (v_i, v_j, C(v_i, v_j))$;
12 $e_p.childList$.add(e_c);
13 $CCI \leftarrow$ updateCCI(CCI, e_p, e_c);
14 $u_p \leftarrow u_c$;
15 $e_p \leftarrow e_c$;
16 $CCI \leftarrow$ extendCCI(e_p, u_p, Q_T, G, C, CCI);
17 **return** *CCI*;

First, initialize *NS* and *ES* of e_c. Second, get the matching nodes of e_c (i.e. $M(e_c)$) in G according to the label of u_c. Third, by comparing the communication cost between nodes in $M(e_c)$ and nodes in e_p's *NS* with the communication constraints in the CPG, refine $M(e_c)$. Only the nodes satisfying the communication constraints continue to be stored in e_c's *NS*. Correspondingly, the index edges between e_c and e_p will be stored in e_c's *ES*. Fourth, extend e_c to the current CCI and call Algorithm 3 recursively with e_c as the new parent index node. Finally, generate the descendants of e_c.

To reduce the storing space of the CCI, for each two index nodes with the parent-child relationship in the CCI (parent is denoted as e_p and child is denoted as e_c), we define two rules, R_1: $\forall v_i(v_i \in e_c.NS) \rightarrow \exists v_j(v_j \in e_p.NS \wedge C(v_i, v_j) \le C_Q(u_c, u_p))$ and R_2: $\forall v_j(v_j \in e_p.NS) \rightarrow \exists v_i(v_i \in e_c.NS \wedge (v_i, v_j, C(v_i, v_j)) \in e_c.ES)$.

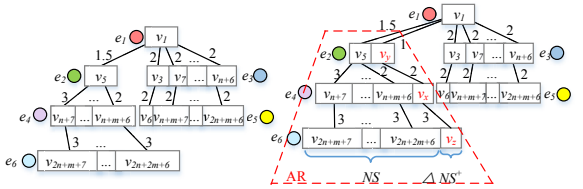
R_1 reveals that, for any node in e_c's *NS*, at least there exists one node in e_p's *NS* where the communication cost between them can satisfy the requirement. R_2 is used to guarantee that, for any node in e_p's *NS*, at least there exists one node in e_c's *NS* where the communication cost between them can satisfy the requirement, i.e. the edge $(v_i, v_j, C(v_i, v_j))$ has been stored in e_c's *ES*. In a word, each node in both e_c's *NS* and e_p's *NS* is involved in the parent-child relationship.

In Algorithm 3, when extending an index node e_c, its ancestors will construct e_c's *NS* and *ES*, which can guarantee R_1. But R_2 might not be satisfied. It is possible that some nodes exist in e_p's *NS* not satisfying the communication cost requirement with e_c. So we propose Algorithm 4 to further refine the CCI in a bottom-up fashion. Each index node next to the leaves (i.e. at the (max_level-1)-th level) is considered as e_p and is refined at first (Line 1-2). Each node in e_p's *NS* is checked whether it is involved in the parent-child relationship with a node in the leaves' (i.e. e_c's) *NS*s (Line 3-7). We remove from e_p's *NS* such nodes that cannot satisfy R_2 (Line 8-9). e_p's *ES* is updated (Line 10). Then, follow this procedure and process the upper level in the CCI iteratively.

Algorithm 4: refineCCI

Input: the current *CCI*
Output: the refined *CCI* based on parent-child relationship

1 **for** each level l from CCI.max_level-1 to 1 **do**
2 **for** each index node e_p at level l **do**
3 **for** each node v_j in $e_p.NS$ **do**
4 v_j.status←false;
5 **for** each index node e_c in $e_p.childList$ **do**
6 **for** each node v_i in $e_c.NS$ **do**
7 **if** $(v_i, v_j, C(v_i, v_j)) \in e_c.ES$ **then** v_j.status←true;
8 **if** v_j.status==false **then**
9 $e_p.NS \leftarrow e_p.NS$ - $\{v_j\}$;
10 $e_p.ES \leftarrow$ updateEdge(e_p, $e_p.NS$);
11 **return** *CCI*;



(a) An example of CCI (b) Incremental maintenance for CCI

Fig. 5. An example of CCI construction and maintenance

Example 5.2: Consider the data graph and the CPG in Fig. 1. The constructed CCI is as Fig. 5(a). There are six index nodes, each of which stores a group of candidate nodes in G. Only the nodes and the edges satisfying communication constraints in the CPG can be retained in the CCI.

C. Complexity Analysis

The average worst-case size of index nodes: Suppose that there are l_D different labels in G. Then the average size of nodes in G with each label is $|V_D|/l_D$. We only need to store the nodes and the edges satisfying the input constraints into the CCI. Suppose that the average probability for an edge in G (denoted as (v_i, v_j)) satisfying the corresponding communication constraint in the CPG is λ. Then the average probability for v_i (or v_j) included by an index node is $\sqrt{\lambda}$. Thus the average worst-case size of index nodes is $(|V_Q||V_D|\sqrt{\lambda})/l_D$.

The average worst-case size of index edges: For each pair of parent-child nodes in Q, there are an average of $|V_D|^2/l_D^2$ paths in G to meet the requirement about labels. Then there are an average of $|V_D|^2\lambda/l_D^2$ index edges to meet both the label requirement and communication constraints. Therefore, the average worst-case size of index edges is $(|V_Q||V_D|^2\lambda)/l_D^2$.

Time complexity of CCI construction: Let H denote the product of the number of leaf nodes in Q_T and the height of Q_T. Then the top-down checking and the bottom-up refining takes at most $O(H|V_D|^2/l_D^2)$.

VI. CCI-BASED NODE MATCHING

During CCI construction, we only consider the parent-child relationships in Q_T. However, communication constraints reflected by the non-tree edges is not included in these relationships. We need to make matching further. In this section, we propose three heuristic matching rules which are applied to our CCI-based node matching algorithm.

A. Core First Matching Rule

A cost model is proposed in [7] to estimate the matching cost based on a matching order of nodes (u_1, …, u_n) (as in (1)). Here B_i is the total number of embeddings in the CCI derived from (u_1, …, u_i) (suppose that each u_i corresponds to the index node e_i in the CCI), d_i^j is the number of candidates induced by the j-th embedding of u_{i-1}, and r_i is the number of non-tree edges between u_i and nodes before u_i in the matching order.

$$T = B_1 + \sum_{i=2}^{n} \sum_{j=1}^{B_{i-1}} d_i^j (r_i + 1) \quad (1)$$

Example 6.1: Let's consider the constructed CCI in Example 5.2. We compare the matching cost under two matching orders, (e_1, e_2, e_4, e_6, e_3, e_5) and (e_1, e_2, e_3, e_4, e_6, e_5). For presentation simplicity, we assume that the values of d_i^j for all the embeddings of e_{i-1} are the same.

Node	B_{i-1}	d_i^j (j=1~B_{i-1})	r_i
e_1	1	1	0
e_2	1	1	0
e_4	1	m	0
e_6	m	1	0
e_3	m	n+1	1
e_5	m	1	0

(a) Matching order: (e_1,e_2,e_4,e_6,e_3,e_5)

Node	B_{i-1}	d_i^j (j=1~B_{i-1})	r_i
e_1	1	1	0
e_2	1	1	0
e_3	1	n+1	1
e_4	m	1	0
e_6	m	1	0
e_5	m	1	0

(b) Matching order: (e_1,e_2,e_3,e_4,e_6,e_5)

Fig. 6. Comparison of the matching cost

In Fig. 6(a), the number of embeddings (i.e. search breadth) for e_1, e_2 and e_4 is 1. The number of candidates derived from e_4's parent (i.e. e_2) is m, so the search breadth of e_4 (i.e. B_4) is m. That is, there are m embeddings generated now. For each embedding, it derives one candidate in e_6. Therefore, by matching (e_1, e_2, e_4, e_6), there are m candidate paths (i.e. embeddings) generated to be combined with post-order mappings. When matching e_3, there are n+1 candidates in its *NS*, each of which needs to be combined with the candidate paths formed before. So the number of comparison is $m\times(n+1)$ for combination. The number of non-tree edges between e_3 and the nodes before it in the matching order is 1. Thus the candidates in e_3 need to be checked whether they can satisfy the communication constraints with the candidates in e_2. The number of comparison is $m\times(n+1)$ for checking too. So the cost of matching e_3 is $2\times m\times(n+1)$. The rest can be done in the same manner. Finally, the total cost of matching regarding the matching order (e_1, e_2, e_4, e_6, e_3, e_5) is $2\times m\times (n+1)+3\times m+2$. In Fig. 6(b), e_3 is matched earlier than e_4 and e_6. Before matching e_3, the search breadth is only 1. So the number of comparison for matching e_3 is only $2\times(n+1)$, the sum of the number of comparison (n+1) for combination and the number of comparison (n+1) for checking. The total cost regarding the matching order (e_1, e_2, e_3, e_4, e_6, e_5) is $2\times(n+1)+3\times m+2$, which is less than the former.

Remark. From the above example, we can see that the search breadth grows quickly and dominates in the matching cost. So we should try to deal with non-tree edges before the search breadth becomes wider. The combination between candidates via non-tree edges should be done as early as possible. The core part is the minimal connected subgraph of Q that contains all the non-tree edges. Therefore, we propose the core first matching rule: the nodes in Q_C should be matched before the nodes in Q_N. For example, in Fig. 5(a) the index nodes e_1, e_2 and e_3 corresponds to the nodes in Q_C of Q. Therefore, e_1, e_2 and e_3 should be matched before e_4, e_5 and e_6.

B. Pruning Power First Matching Rule

Next, we further determine the matching order of nodes in the core part. There are a set of root-to-leaf paths (l_1, …, l_k) sharing the root node in the core part. Our goal is to compute an efficient order of these path, then to obtain the matching order of query nodes.

Suppose that there are two index nodes (e_m and e_n) in the CCI. There is a non-tree edge between their corresponding query nodes in Q. We define the satisfaction rate of e_m (denoted as $\mu(e_m)$, as in (2)) as the proportion of nodes in e_m's *NS* satisfying communication constraints with nodes in e_n's *NS*. For other index nodes not connected by non-tree edges, their satisfaction rates are always 1.

$$\mu(e_m) = \frac{|\{v_i | v_i \in e_m.NS \wedge \exists v_j (v_j \in e_n.NS \wedge C(v_i, v_j) \le C_q(e_m, e_n))\}|}{|e_m.NS|} \quad (2)$$

To estimate the cost based on a matching order of paths, we redefine the cost model in [7]. In fact, the matching order of query nodes can be obtained from the order of paths. We assume that the matching order of query nodes is (u_1, …, u_n) and the position of the last node of l_i in the matching order is l_i. Given a path-based order (l_1, …, l_k) and a CCI, the total cost of a backtracking algorithm for matching can be calculated as (3). Here B_{li} is the search breadth of l_i, and r_{li} is the number of non-tree edges between l_i and paths before l_i in the matching order. μ_{i-1} is the product of the satisfaction rates in the CCI derived from (e_1, …, e_{i-1}). Suppose that e_{i-1} is an index nodes connected by a non-tree edge. Then the probability that each node in e_{i-1}'s *NS* preserved in the CCI is μ_{i-1}, so the search breadth of e_{i-1} is $B_{i-1}\mu_{i-1}$. It has an effect on the number of comparisons during matching the next node (see the first line in (3)). We can also assume that in the next index node the number of candidates under each embedding is reduced uniformly (see the second line in (3)). Further we can get the third line in (3), which will be used to determine the matching order of nodes in the core part. For each path l_i, the matching cost is relevant to: B_{li}, the number of non-tree edges connected to l_i, and the product of the satisfaction rates derived from these non-tree edges (i.e. μ_{li}). Then we define pruning power for each path (as in (4)).

$$\begin{aligned} T' &= B_1 + \sum_{i=2}^{n} \sum_{j=1}^{B_{i-1}\mu_{i-1}} d_i^j (r_i + 1) \\ &\approx B_1 + \sum_{i=2}^{n} \mu_{i-1} \sum_{j=1}^{B_{i-1}} d_i^j (r_i + 1) \\ &\approx B_{l_1} + \sum_{i=2}^{k} \mu_{l_i} B_{l_i} (r_{l_i} + 1) \end{aligned} \quad (3)$$

$$P(l_i) = \frac{r_{l_i}}{B_{l_i} \times \mu_{l_i}} \quad (4)$$

Remark. Pruning power is used to measure the number of intermediate results derived from a root-to-leaf path in the CCI. As discussed before, if the combination between candidates via non-tree edges is done early, more

intermediate results can be pruned early. Thus large r_{li} will raise the value of $P(l_i)$. On the contrary, if the search breadth of l_i is wider and the satisfaction rates are higher, more candidates will be generated as intermediate results. The large B_{li} and μ_{li} will lower the value of $P(l_i)$. It is a hard problem to minimize T', so we propose the pruning power first matching rule: given a set of root-to-leaf paths (l_i, …, l_k) sharing the root node in the core part of the CCI, the path with strongest pruning power should be matched first.

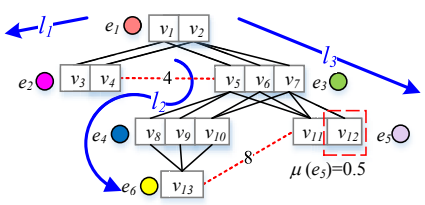


Fig. 7. Comparison of pruning power

Example 6.2: Suppose that a CCI is depicted as Fig. 7, corresponding to Q_C in Example 4.1 (here the communication cost for the parent-child relationships is omitted). We assume that $\mu(e_5)$=0.5 (suppose that $C(v_{12}, v_{13})$ is larger than $C_Q(e_5, e_6)$), and the values of μ of other index nodes are 1. There are three root-to-leaf paths (l_1, l_2, l_3). Let's take l_2 as an example. There are four embeddings ($v_1v_5v_8v_{13}$, $v_1v_5v_9v_{13}$, $v_2v_6v_{10}v_{13}$ and $v_2v_7v_{10}v_{13}$) along it, so its search breadth is 4. In this way, we can know B_{l1} =2, B_{l2} =4, and B_{l3} =4 respectively. Their pruning power is $P(l_1)$=1/2, $P(l_2)$=2/4, and $P(l_3)$=2/(4×0.5) respectively. So the nodes on l_3 should be matched at first.

C. Search Breadth First Matching Rule

After matching the nodes in the core part, we begin to match the nodes in each noncore part Q_N. Note that there are no non-tree edges in Q_N, we only use search breadth to measure the number of intermediate results induced by Q_N.

We propose the search breadth first matching rule: given a set of Q_N, the noncore part with less search breadth should be matched first. As for each Q_N, the path with less search breadth in Q_N's root-to-leaf paths should be matched first.

So we first estimate the size of search breadth for each Q_N, and then sort these Q_N s in an ascending order regarding their search breadth. Then the root-to-leaf paths in each Q_N are ordered according to their search breadth too. In this way, we obtain the matching order of nodes in the noncore parts. The estimation of search breadth is similar to the matching process within the core part, so we omit the details here.

D. CCI-based node matching algorithm

We propose a CCI-based node matching algorithm (Algorithm 5). Given a CCI, Q_C, a *QNset* and the shortest path length matrix C, it is to determine the matching order of index nodes in the CCI. According to the core first matching rule, we first match the nodes in Q_C, and then in *QNset*.

Step 1: According to the pruning power first matching rule, generate the matching order of nodes in Q_C. First, we use the set L to store all root-to-leaf paths in Q_C and select the first path from it (Line 1-4). The path having the maximum pruning power in the CCI will be selected. Correspondingly, the nodes along the path will be added to the matching queue as the first matched index nodes. Second, we choose the next path from L (Line 5-11). The next path is chosen based on the current queue. Each path l_i in L shares a prefix with the queue. To reduce computation, we only consider the different part of l_i from the current queue, i.e. starting from the last shared node to the leaf (denoted as l_i'). Iteratively, we substitute l_i' for l_i and choose the path having the maximum pruning power from L. Finally, we get the matching order of nodes in Q_C.

Step 2: According to the search breadth first matching rule, generate the matching order of nodes in *QNset* (Line 12-16). First, for each Q_N in *QNset*, the size of its search breadth will be estimated based on the CCI. The noncore parts in *QNset* will be sorted by their search breadth. Those having less search breadth should be matched first. Second, for each Q_N (denoted as QN_j), we sort all root-to-leaf paths according to their search breadth in the CCI. Similarly, the paths having smaller search breadth should be matched preferentially. Finally, the matching order of index nodes in *QNset* can be generated according to the order of paths.

Algorithm 5: CCI-based node matching

Input: *CCI*, Q_C, *QNset*, C
Output: the matching order of index nodes
1 L←all root-to-leaf paths in Q_C on *CCI*;
2 l^*←argmax$_{l\in L}$ $P(l)$;
3 Add nodes of l^* to *queue*;
4 L← L-{l^*};
5 **while** $L\neq\varnothing$ **do**
6 **for** each path l_i in L **do**
7 l_i' ←getDifferentPath(l_i, *queue*);
8 l_i ← l_i';
9 l^*←argmax$_{l\in L}$ $P(l)$;
10 Add nodes of l^* to *queue*;
11 L← L-{l^*};
12 *QNList*←sortQNByBreadth(*QNset*, *CCI*);
13 **for** each QN_j in *QNList* **do**
14 LN_j←all root-to-leaf paths in QN_j;
15 *pathList*←sortPathByBreadth(LN_j, *CCI*);
16 Add nodes to *queue* according to *pathList*;
17 **return** *queue*;

Let $|ES|$ denote the average size of ES regarding to each index node in the CCI. Then generating the matching order of nodes in Q_C takes time $O(|ES|\Sigma_{l\in L}\ len(l))$. And generating the matching order of nodes in *QNset* takes time $O(|ES|\Sigma_{QNj\in QNset}\Sigma_{l\in LNj}\ len(l))$. Here $len(l)$ means the number of index edges in the path l. The total time complexity is $O(|ES|\Sigma_{l\in L}\ len(l)+|ES|\Sigma_{QNj\in QNset}\Sigma_{l\in LNj}\ len(l))$ which is at most $O(H|ES|)$. Here H has the same value as that of H in Section V.C. Compared with G, the size of a CCI is generally small. So the time is linear to $|ES|$.

After determining the matching order of index nodes, we will enumerate all the embeddings of Q in G, i.e. all possible combinations of team members following the constraints in the CPG. We match each index node according to the matching order. As for the index nodes not connected by non-tree edges, they are used to generate the embeddings only. As for the index nodes connected by non-tree edges, the nodes in their *NS* should be checked whether they can satisfy the constraints along the non-tree edges. If not, the nodes should be removed from the CCI. Finally, the embeddings with the top-K minimum total communication cost will be selected to form each team. The nodes included by each embedding will act as team members.

VII. INCREMENTAL MAINTENANCE

In this section, we present incremental maintenance strategies for the changes of G, including the insertion or deletion of nodes' labels, insertion or deletion of edges, insertion or deletion of nodes, increase or decrease of edges' weights, denoted by ΔG_L^+, ΔG_L^-, ΔG_E^+, ΔG_E^-, ΔG_N^+, ΔG_N^-, ΔG_W^+ and ΔG_W^- respectively.

(1) For each label $a \in \Delta G_L^+$ (suppose that v_x is the changed node in G), which might just generate some new embeddings, but not affect the communication cost among the current nodes in the CCI, we consider the following situations.

If a is not one of labels requested in the CPG, then both the CCI and the query result remain unchanged. Otherwise, an affected region (AR) in the CCI will be identified, which initially includes the index node e_a (with label a), e_a's parent and e_a's children. The communication cost between v_x and each node in AR will be checked:

1) If one of the communication cost cannot satisfy the request in the CPG, both the CCI and the query result remain unchanged because v_x is not possible to be a candidate.

2) Otherwise, both the CCI and the query result should be updated. First, index nodes in the current AR are updated. The node v_x is added to e_a's NS, which might result in the insertion of more candidates to e_a's parent and e_a's children. We denote the insertion part and the new NS as ΔNS^+ and NS' respectively ($NS'=NS \cup \Delta NS^+$). Second, the initial AR is extended. Along the path from e_a's parent to the root, the last index node with nonempty ΔNS^+ is chosen as the root node of AR. The root node with its descendants in the CCI together constitutes the new AR. Let h denote the length of the path from e_a's parent to the root. The average size of nodes in G with each label is $|V_D|/l_D$. Then extending AR takes time $O(h|\Delta NS^+||V_D|/l_D)$. Third, for each index node in AR, its ΔNS^+ is checked whether the nodes in it satisfy the communication constraints and the rules defined in Subsection V.B. If so, the nodes are kept in ΔNS^+. Otherwise, they are removed from ΔNS^+. Let H_{AR} denote the product of the number of leaf index nodes in AR and the height of AR. Then the top-down checking and the bottom-up refining takes time at most $O(H_{AR}|\Delta NS^+||V_D|/l_D)$. Finally, the query result is updated. If AR contains the index nodes connected with non-tree edges, node matching needs to be recomputed based on the updated CCI. Otherwise, all the embeddings via v_x in AR are enumerated and their total communication cost is calculated. If an embedding via v_x is more dominant, the current query result will be replaced by it. Otherwise, the query result will remain unchanged. Let ΔES denote the changed part of ES. Then updating the query result takes time at most $O(H_{AR}|\Delta ES|)$. Therefore, the total time complexity is $O(H_{AR}(|\Delta NS^+||V_D|/l_D+|\Delta ES|))$.

Example 7.1: Fig. 5(a) gives an example of a CCI. When a new label 'TE' is added to the node v_x (shown in Fig. 5(b)), e_2, e_4 and e_6 constitute the initial AR. Because the communication cost between v_x and each node in AR can satisfy the request in the CPG, the CCI needs to be updated. Along the path from e_2 to e_1, the last index node with nonempty ΔNS^+ is e_2. So AR remains unchanged. Since e_2 connects with e_3 via a non-tree edge, the nodes in e_2's ΔNS^+ should be matched with e_3. Thus node matching needs to be recomputed based on the updated CCI.

(2) For each label $a \in \Delta G_L^-$ (suppose that v_x is the changed node), which might just make some embeddings in the CCI ineffective, but won't generate any new embeddings and won't affect the current communication cost, we only need to prune the current result. We consider the following situations.

If a is not one of labels requested in the CPG, then both the CCI and the query result remain unchanged. Otherwise, the CCI should be updated. First, the initial AR is identified by performing the same procedure as ΔG_L^+. The node v_x is removed from e_a's NS. Also the embeddings via v_x are removed from AR. The deletion of v_x might result in the deletion of more candidates from the CCI, because they might no longer meet the rules defined in Subsection V.B. We denote the deletion part and the new NS as ΔNS^- and NS' respectively ($NS'=NS-\Delta NS^-$). Second, the initial AR is extended. The process is similar to ΔG_L^+. Along the path from e_a's parent to the root, the last index node with nonempty ΔNS^- is chosen as the root node of the new AR. For each NS, we just need to refine it, not to extend it. Thus extending AR takes time $O(h|NS|)$. Third, for each index node in AR, its NS' is checked and refined with the time complexity $O(H_{AR}|NS|)$. Finally, the query result is updated. If AR contains index nodes connected with non-tree edges, node matching needs to be recomputed. Otherwise, the current query result is checked. If it doesn't include any deleted nodes, it will remain unchanged. Otherwise, we will choose one from the current embeddings with the minimum total communication cost as the new query result. The total time complexity is $O(H_{AR}(|NS|+|\Delta ES|))$.

(3) For each edge $(v, v') \in \Delta G_E^+$ (assume that $W_D(v, v')$ is not larger than the current $C(v, v')$), the communication cost of any pairs of nodes in G keeps unchanged or decrease, because it just adds more candidates to the shortest path between each pair of nodes. The current embedddings in the CCI can still satisfy the constraints. Besides, some new embeddings might meet the constraints and be more dominant. So the shortest path length matrix C, the CCI and the current query result should be updated respectively. As for the update for C, we adopt the method proposed in [26] to incrementally get the new shortest path length matrix. As for the update for the CCI, we consider the whole CCI as AR and extend each index node by adding ΔNS^+ to its NS. The subsequent process is similar to ΔG_L^+. The total time complexity is $O(H_{AR}|V_D|^2/l_D^2+|\Delta ES|)$. Adding an edge between two nodes is equivalent to reducing the distance between them, i.e., reducing the edge's weight. Therefore, the process of ΔG_W^- is the same as the above strategy for ΔG_E^+.

(4) For each edge $(v, v') \in \Delta G_E^-$, the communication cost of any pair of nodes in G remains unchanged or increases. The node pair not meeting the constraints before still cannot satisfy the constraints now. Therefore, except for the embeddings in the CCI, no new embeddings will appear as a result. We just need to update the CCI and choose an embedding from it as the new query result. In particular, we will check the edges in each index node's ES. If the communication cost no longer meets the constraints requested in the CPG, the edges should be removed, which might result in the deletion of the two ending nodes further. We start updating the CCI from its root and refine each index node by deleting ΔNS^- from its NS. The subsequent process is similar to ΔG_L^-. The total time complexity is $O(H_{AR}(|NS|+|\Delta ES|))$. Also the process of ΔG_W^+ is the same as the strategy for ΔG_E^-.

(5) For each node $v \in \Delta G_N^+$, if v has no links with other nodes in G, then v is an isolated node and might not be the matching node. So it does not affect the CCI and the current result. If v leads to one or several new edges in G, the process of the insertion of these new edges is the same as the above mentioned strategy for ΔG_E^+.

(6) For each node $v \in \Delta G_N^-$, the edges with an end point of v will also be deleted. The process of these deleted edges is the same as the above mentioned strategy for ΔG_E^-.

The above maintenance strategies focus on maintaining the CCI, which mainly includes two types of operations: ΔNS^+ (e.g. ΔG_L^+, ΔG_E^+, ΔG_N^+ and ΔG_W^-) and ΔNS^- (e.g. ΔG_L^-, ΔG_E^-, ΔG_N^- and ΔG_W^+). For each index node, let NS, NS' and NS'' denote the node set before updating, the node set updated by our strategies and the correct node set based on the updated G respectively.

Theorem 7.1: Taking ΔG_L^+, ΔG_E^+, ΔG_N^+ or ΔG_W^- as input, our incremental maintenance strategies can maintain the CCI correctly.

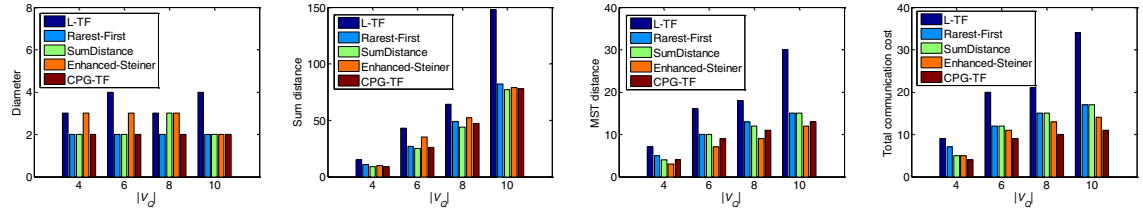
Proof: Suppose $NS' \neq NS''$, then there is at least one node v in the updated G such that (1) $v \in NS' \wedge v \notin NS''$ or (2) $v \in NS'' \wedge v \notin NS'$. If (1) is true, since $NS' \supseteq NS$, then $v \notin NS$. Since $NS' = NS \cup \Delta NS^+$, then $v \in \Delta NS^+$ which contradicts $v \notin NS''$. If (2) is true, since $NS' \supseteq NS$, then $v \notin NS$. Since $v \in NS''$, then $v \in \Delta NS^+$. Since $NS' = NS \cup \Delta NS^+$, then $v \in NS'$ which contradicts $v \notin NS'$. Therefore, for each index node, $NS' = NS''$. That is, the updated CCI is correct.

Theorem 7.2: Taking ΔG_L^-, ΔG_E^-, ΔG_N^- or ΔG_W^+ as input, our incremental maintenance strategies can maintain the CCI correctly.

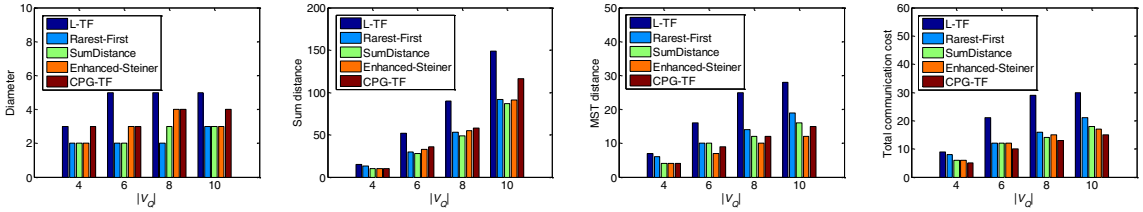
Proof: Suppose $NS' \neq NS''$, then there is at least one node v in the updated G such that (1) $v \in NS' \wedge v \notin NS''$ or (2) $v \in NS'' \wedge v \notin NS'$. If (1) is true, since $NS \supseteq NS''$, then a) $v \notin NS$ or b) $v \in NS\text{-}NS''$. If a) is true, since $NS' = NS\text{-}\Delta NS^-$, then $v \notin NS'$ which contradicts $v \in NS'$. If b) is true, then v is a node not satisfying R_1, R_2 or the communication constraints, so $v \in \Delta NS^-$. Since $NS' = NS\text{-}\Delta NS^-$, then $v \notin NS'$ which contradicts $v \in NS'$. If (2) is true, since $NS \supseteq NS''$, then $v \in NS$. Since $NS' = NS\text{-}\Delta NS^-$ and $v \notin NS'$, then $v \in \Delta NS^-$. So v is a node not satisfying R_1, R_2 or the communication constraints. It contradicts $v \in NS''$. Therefore, for each index node, $NS' = NS''$. That is, the updated CCI is correct.

VIII. EXPERIMENTS

We implement the experiments on a Server with Intel(R) Xeon(R) CPU E7-4820 v4 @ 2.00GHz, 2.0TB main memory and 11.5TB hard disk, running 64bit CentOS release 6.9 (Final). We use two real-world social graphs, Ego-Facebook (4039 nodes and 88234 edges) and Email-Enron (36692 nodes and 183831 edges), which are available at *snap.stanford.edu*. Like the well-known existing works in team formation [10, 11, 13], we consider 20 classes of labels and randomly set the labels of nodes in our data sets. The weight of each edge is randomly set as an integer from 1 to 3. We generate different CPGs by changing structure constraints and communication constraints. Specifically, these CPGs are generated by controlling the number of nodes (i.e. $|V_Q|$, set as 4, 6, 8 and 10 respectively), the number of edges (set between 4 and 10) and communication constraints (set as an integer between 1 and 3). We set K (the number of returned teams) to 3.



(a) Ego-Facebook



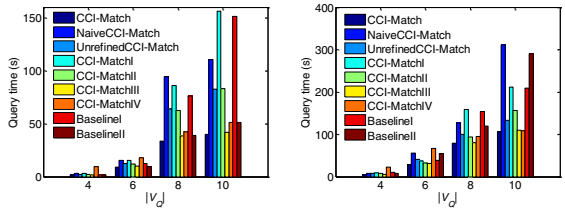
(b) Email-Enron

Fig. 8. Performance comparison of different team formation algorithms

Effectiveness evaluation of team formation: We use the diameter, sum distance (i.e. the sum of the communication cost between each pair of skill holders), the minimum spanning tree (MST) distance, and the total communication cost (i.e. the sum of communication cost between two adjacent members in the CPG) to evaluate the quality of the teams produced by the following different algorithms. (1) Label based team formation (L-TF): the nodes satisfying each kind of label requirements in the CPG constitute a group of candidates. We randomly select one from each group to form the team. Besides the required labels, other methods also consider the communication cost. (2) Rarest-First [1]: the goal is to minimize the diameter of the team. (3) SumDistance [5]: the goal is to minimize the sum distance. (4) Enhanced-Steiner [1]: the goal is to minimize the MST distance. (5) CPG based team formation (CPG-TF): the goal is to minimize the total communication cost.

Results: As in Fig. 8, L-TF only considers the required skills, so the quality of teams found by it is low. The diameters (or the sum distances) of teams found by CPG-TF are comparable to those of Rarest-First (or SumDistance)

which is in particularly designed to minimize the diameters (or the sum distances). The MST distances of teams found by CPG-TF are larger than Enhanced-Steiner which is specialized for minimizing MST distances, but are smaller than others. However, the total communication cost of teams found by CPG-TF is smaller than all the others. These results verify that CPG-TF can effectively find high quality teams. For MST distance and total communication cost, the experiments on Ego-Facebook and Email-Enron exhibit similar trends. For diameter and sum distance, CPG-TF is worse relatively on Email-Enron than on Ego-Facebook. That is because Email-Enron is sparser than Ego-Facebook in network structures, resulting in larger diameter and sum distance.

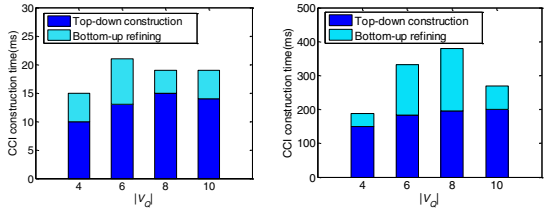


(a) Ego-Facebook (b) Email-Enron

Fig. 9. Query time comparison of different node matching algorithms

Performance evaluation of node matching: We evaluate the following methods. (1) CCI-Match: CCI-based node matching algorithm. (2) NaiveCCI-Match: the CCI-Match algorithm where the CCI is constructed only based on the label requirements in the CPG. (3) UnrefinedCCI-Match: the CCI-Match algorithm where the CCI is not refined. (4) CCI-MatchI: CCI-Match algorithm with a random matching order of nodes in the CCI. None of the three heuristic rules are applied. (5) CCI-MatchII: CCI-Match algorithm with a matching order based on the core first matching rule. (6) CCI-MatchIII: CCI-Match algorithm with a matching order based on both the core first matching rule and the pruning power first matching rule. (7) CCI-MatchIV: CCI-based node matching (without CPG normalization). (8) BaselineI: node matching is performed based on the shortest path length index [15]. (9) BaselineII: path matching constrained by a path length is performed based on S-Index [17].

Results: The query performance of the effect of different CCI construction strategies and different node matching strategies is illustrated in Fig. 9. With the increase of $|V_Q|$, the query time of CCI-Match is always less than that of two Baseline methods in both of the two datasets. Also, with the increase of the size of the datasets, the advantage is even more obvious. That is because BaselineI only considers the shortest path length between nodes, but ignores the matching cost based on a matching order of nodes. BaselineII focuses on the data graph that consists of multiple smaller subgraphs. All nodes in the data graph need to be indexed. When the size of data graph increases, the cost of node matching and index traversal increases sharply. Only labels are considered by NaiveCCI-Match, so lots of false-positive candidates are stored in the CCI which need more query time. UnrefinedCCI-Match improves upon NaiveCCI-Match by checking the CCI in a top-down way according to the input constraints. CCI-MatchI treats all nodes equally when matching without distinguishing the nodes in core part or in noncore part. CCI-MatchII improves upon CCI-MatchI, and CCI-MatchIII further improves upon CCI-MatchII by adopting our proposed heuristic rules. CCI-MatchIV confuses the redundant edges with the non-tree edges, resulting in longer time for matching the nodes connected by such edges than CCI-MatchIII. CCI-Match further refines the CCI in a bottom-up way and applies all of the three heuristic rules, leading to the best performance.



(a) Ego-Facebook (b) Email-Enron

Fig. 10. Time cost of constructing a CCI

TABLE II. STORAGE COST (KB) OF CONSTRUCTING A CCI

Dataset	Algorithms	$\|V_Q\|$=4	$\|V_Q\|$=6	$\|V_Q\|$=8	$\|V_Q\|$=10
Ego-Facebook	NaiveCCI	7401	8885	10194	11560
	UnrefinedCCI	5067	5416	5645	5384
	RefinedCCI	5062	5409	5534	5183
Email-Enron	NaiveCCI	29012	36552	38501	218695
	UnrefinedCCI	17602	19125	19949	21103
	RefinedCCI	13641	19104	19638	11701

Time cost of constructing a CCI: As described in Subsection V.B, CCI construction includes top-down construction and bottom-up refining. We measure the time of the two fashions respectively.

Results: As in Fig. 10, compared with bottom-up refining, top-down construction takes a larger proportion of time. When $|V_Q|$ reaches a certain level, the time for CCI construction might be less because the constraints are stricter and the number of candidates in each index node gets less, resulting in less time for refining.

Storage cost of constructing a CCI: The storage cost of different CCI construction algorithms are tested. (1) NaiveCCI: only the label requirements in the CPG are considered. Neither top-down construction nor bottom-up refining is performed. (2) UnrefinedCCI: only top-down construction is performed. (3) RefinedCCI: our CCI construction algorithm which includes both top-down construction and bottom-up refining.

Results: As in Table II, UnrefinedCCI improves upon NaiveCCI, and RefinedCCI improves upon UnrefinedCCI by removing the candidates that violate the constraints from the CCI. When $|V_Q|$ is 10, the storage cost decreases a little. This is because, when $|V_Q|$ reaches a certain level, the number of nodes in G satisfying the requested constraints gets less, resulting in lower storage cost than before.

Performance evaluation of incremental maintenance: We compare the performance of our incremental maintenance strategies (IM) with the non-incremental maintenance strategies (NIM) and the incremental node matching strategy (INC) proposed in [15], aiming at ΔG_L^+, ΔG_L^-, ΔG_E^+ and ΔG_E^-. The other situations are similar (see Section VII). For ΔG_L^+ (or ΔG_L^-), we randomly choose m (set as 40 and 60 respectively) nodes in G and add (or delete) one label to (or from) each. The value of the label is random. For ΔG_E^+ (or ΔG_E^-), we randomly add (or delete) m (set as 40 and 60 respectively) edges into (or from) G.

Results: The variation trend of experimental results under different sizes of CPGs are similar. For simplicity, we only show the results regarding to the CPG with six nodes (Table III). Since NIM needs to perform the whole process of query, its average processing time is always more than others. INC does not support incremental maintenance for the changes of labels. As for ΔG_E^+ (or ΔG_E^-), although INC considers the incremental maintenance of the shortest path length between nodes, it ignores the incremental maintenance of the node order for matching. IM performs the incremental maintenance for both the shortest path length matrix and the CCI, which can effectively limit the scope of modification.

TABLE III. THE AVERAGE QUERY TIME (S) BASED ON DIFFERENT SIZE OF ΔG_L^+, ΔG_L^-, ΔG_E^+ AND ΔG_E^- RESPECTIVELY

Dataset	ΔG	m=40			m=60		
		NIM	INC	IM	NIM	INC	IM
Ego-Facebook	ΔG_L^+	9.53		0.55	9.76		0.72
	ΔG_L^-	9.02		0.02	8.86		0.03
	ΔG_E^+	9.52	8.69	2.04	9.64	8.98	3.93
	ΔG_E^-	9.04	6.14	1.89	8.72	6.41	2.91
Email-Enron	ΔG_L^+	29.39		1.51	29.79		2.49
	ΔG_L^-	28.63		0.07	28.38		0.06
	ΔG_E^+	29.58	26.22	8.38	29.81	26.73	13.52
	ΔG_E^-	28.64	23.58	5.15	28.58	22.26	7.90

IX. CONCLUSION

We present an effective and efficient team formation method based on CPG, which takes into account both structure constraints and communication constraints. First, we propose a CPG preprocessing method and design an index structure, i.e. CCI, to speed up the matching between the CPG and the data graph. Then we propose a CCI-based node matching algorithm to minimize the total number of intermediate results. Also some incremental maintenance strategies are proposed. The experiments demonstrate the effectiveness and the efficiency of our proposed method. In our future work, we will work on multi-objective team formation methods, and incremental maintenance strategies for changes of a CPG.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (U1811261, 61672142).

REFERENCES

[1] T. Lappas, K. Liu, and E. Terzi, "Finding a team of experts in social networks," in *SIGKDD2009*, 2009, pp. 467-476.

[2] M. Sozio, and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *SIGKDD2010*, 2010, pp. 939-948.

[3] M. Kargar, and A. An, "Discovering top-k teams of experts with/without a leader in social networks," in *CIKM2011*, 2011, pp. 985-994.

[4] J. Huang, Z. Lv, Y. Zhou, H. Li, H. Sun, and X. Jia, "Forming grouped teams with efficient collaboration in social networks," *The Computer Journal*, vol. 60, no. 11, pp. 1545-1560, 2017.

[5] C. T. Li, M. Y. Huang, R. Yan, and S. D. Lin, "On team formation with expertise query in collaborative social networks," *World Wide Web Journal*, vol. 21, pp. 939-959, 2018.

[6] C. Ding, F. Xia, G. Gopalakrishnan, W. Qian, and A. Zhou, "TeamGen: an interactive team formation system based on professional social network," in *WWW2017*, 2017, pp. 195-199.

[7] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, vol. 1, no. 1, pp. 364-375, 2008.

[8] W. Han, J. Lee, and J. H. Lee, "Turbo$_{iso}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *SIGMOD2013*, 2013, pp. 337-348.

[9] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing Cartesian products," in *SIGMOD2016*, 2016, pp. 1199-1214.

[10] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *PVLDB*, vol. 3, no. 1-2, pp. 264-275, 2010.

[11] W. Fan, X. Wang, and Y. Wu, "ExpFinder: Finding experts by graph pattern matching," in *ICDE2013*, 2013, pp. 1316-1319.

[12] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu, "Scalable supergraph search in large graph databases," in *ICDE2016*, 2016, pp. 157-168.

[13] W. Fan, Y. Wu, and J. Xu, "Adding counting quantifiers to graph patterns," in *SIGMOD2016*, 2016, pp. 1215-1230.

[14] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Strong simulation: Capturing topology in graph pattern matching," *ACM Transactions on Database Systems (TODS)*, vol. 39, no. 1, pp. 1-46, 2014.

[15] G. Sun, G. Liu, Y. Wang, M. A. Orgun, and X. Zhou, "Incremental graph pattern based node matching," in *ICDE2018*, 2018, pp. 281-292.

[16] H. Rahman, S. Roy, S. Thirumuruganathan, S. Amer, and G. Das, "Optimized group formation for solving collaborative tasks," *PVLDB*, vol. 28, no. 1, pp. 1-23, 2019.

[17] W. Lin, X. Xiao, J. Cheng, and S. Bhowmick, "Efficient algorithms for generalized subgraph query processing," in *CIKM2012*, 2012, pp. 325-334.

[18] S. Ma, J. Li, C. Hu, X. Liu, and J. Huai, "Graph pattern matching for dynamic team formation," *CoRR*, abs/1801.01012, 2018.

[19] Q. Shi, G. Liu, K. Zheng, A. Liu, Z. Li, L. Zhao, and X. Zhou, "Multi-constrained top-K graph pattern matching in contextual social graphs," in *ICWS2017*, 2017, pp. 588-595.

[20] D. Gao, Y. Tong, J. She, T. Song, L. Chen, and K. Xu, "Top-k team recommendation in spatial crowdsourcing," in *WAIM2016*, 2016, pp. 194-204.

[21] J. R. Ullmann, "An algorithm for subgraph isomorphism," *JACM*, vol. 23, no. 1, pp. 31-42, 1976.

[22] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 3, pp. 1-47, 2013.

[23] E. Abdelhamid, M. Canim, M. Sadoghi, B. Bhatta, Y. Chang, and P. Kalnis, "Incremental frequent subgraph mining on large evolving graphs," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 29, no. 12, pp. 2710-2723, 2017.

[24] K. Kim, I. Seo, W. Han, J. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "TurboFlux: a fast continuous subgraph matching system for streaming graph data," in *SIGMOD2018*, 2018, pp. 411-426.

[25] B. Du, S. Zhang, N. Cao, and H. Tong, "First: Fast interactive attributed subgraph matching," in *SIGKDD2017*, 2017, pp. 1447-1456.

[26] G. Ramalingam, and T. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1, pp. 233-277, 1996.