Graph Pattern Matching for Dynamic Team Formation

Shuai Ma Jia Li Chunming Hu Xudong Liu Jinpeng Huai
SKLSDE Lab, Beihang University, China
Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China
{mashuai, lijia1108, hucm, liuxd, huaijp}@buaa.edu.cn

ABSTRACT

Finding a list of k teams of experts, referred to as *top-k team formation*, with the required skills and high collaboration compatibility has been extensively studied. However, existing methods have not considered the specific collaboration relationships among different team members, *i.e.,* structural constraints, which are typically needed in practice. In this study, we first propose a novel graph pattern matching approach for top-k team formation, which incorporates both structural constraints and capacity bounds. Second, we formulate and study the *dynamic top-k team formation* problem due to the growing need of a dynamic environment. Third, we develop an unified incremental approach, together with an optimization technique, to handle continuous pattern and data updates, separately and simultaneously, which has not been explored before. Finally, using real-life and synthetic data, we conduct an extensive experimental study to show the effectiveness and efficiency of our graph pattern matching approach for (dynamic) top-k team formation.

1. INTRODUCTION

The *top-k team formation* problem is to find a list of k highly collaborative teams of experts such that every team satisfies the skill requirements of a certain task. Various approaches [7, 9, 15, 22, 25, 34] have been proposed, and fall into two categories in terms of the way to improve the collaborative compatibility of team members: (a) minimizing team communication costs, defined with *e.g.,* the diameter, minimum spanning tree and the sum of pairwise member distances of the induced subgraph [7,9,22,25], and (b) maximizing team communication relations, *e.g.,* the density of the induced subgraph [15,34]. Further, [15] and [34] consider a practical setting by introducing a lower bound on the number of individuals with a specific skill in a team, and an upper bound of the total team members, respectively.

Example 1: Consider a recommendation network G_1 taken from [37] as depicted in Fig. 1, in which (a) a node denotes a person labeled with her expertise, *e.g.,* project manager (PM), software architect (SA), software developer (SD), software tester (ST), user interface designer (UD) and business analyst (BA), and (b) an edge indicates the collaboration relationship between two persons, *e.g.,* (PM_1, UD_1) indicates PM_1 worked well with UD_1 within previous projects.

A headhunt helps to set up a team for a software product by searching proper candidates from G_1 (ignore dashed edges). A desired team has (i) one PM, and one to two BAs, UDs, SAs, SDs and STs, such that (ii) PM should collaborate with SAs, BAs and UDs well, and SDs and STs should collaborate with each other well and both with SAs well.

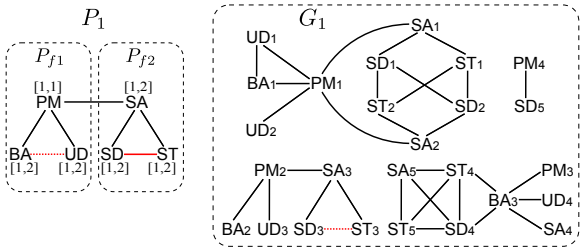


Figure 1: Motivation example

One may verify that existing methods [15, 22, 25, 34], can hardly find a desired team. They only find teams satisfying the skill requirement [15,22,25] and the lower bound capacity requirement [15,34] (condition (i)), and cannot guarantee the specific collaboration relationships among team members, *i.e.,* structural constraints (condition (ii)). □

A natural question is how to further capture the *structural and capacity constraints* in a unified model for team formation? We introduce a revision of *graph pattern matching* for team formation to fill in this gap. Given a pattern graph P and a data graph G, graph pattern matching is to find all subgraphs in G that match P, and has been extensively studied [11,14,19,28,29,38]. Essentially, we utilize patterns to capture the structural constraint, and revise the semantics of graph pattern matching for team formation. For instance, a desired team requirement can be specified by the pattern P_1 (ignore dashed edges) in Fig. 1, in which nodes represent the skill requirements, edges specify the topology constraint, and the bounds on nodes are the capacity constraint.

Another issue lies in that team formation is accompanied with a highly dynamic environment. It typically needs many efforts to find the ideal teams, and is common for professionals to refine patterns (requirements) multiple rounds [18,36]. Further, real-life graphs are often big and constantly evolve over time [13]. We show this with an example.

Example 2: Consider P_1 and G_1 in Example 1 again.
(1) One may find that P_1 is too restrictive to find any sensible match in G_1. Hence, she needs to refine the pattern by updating P_1 with ΔP_1, *e.g.,* an edge deletion $(\mathsf{SD}, \mathsf{ST})^-$.
(2) It is also common that a data update ΔG_1 comes on G_1, *e.g.,* an edge insertion $(\mathsf{SD}_3, \mathsf{ST}_3)^+$.
(3) Finally, it can be the case when pattern update ΔP_1 and data update ΔG_1 come simultaneously on P_1 and G_1. □

This motivates us to study the *dynamic top-k team formation* problem, to handle continuous pattern and data up-

dates, separately and simultaneously. It is known that incremental algorithms avoid re-computing from scratch by re-using previous results [32]. However, incremental algorithms of graph pattern matching for pattern updates has not been investigated, though there exist incremental algorithms of graph pattern matching for dealing with data updates [10, 11, 13]. Further, it is also challenging for incremental algorithms to handle simultaneous pattern and data updates in a unified way.

Contributions. To this end, we introduce a graph pattern matching approach for (dynamic) top-k team formation.

(1) We propose *team simulation*, a revision of traditional graph pattern matching, for top-k team formation (Section 2). It extends existing methods by incorporating the structural and capacity constraints using pattern graphs. To cope with the highly dynamic environment of team formation, we also formulate the dynamic top-k team formation problem (Section 2), for dealing with pattern and data updates, separately and simultaneously.

(2) We develop a batch algorithm for computing top-k teams via team simulation (Section 3). We study the satisfiability problem for pattern graphs, a new problem raised in the presence of capacity bounds for graph pattern matching. We also propose two optimization techniques, handling radius varied balls and density based filtering, for speeding up the process of computations.

(3) We develop a unified approach to handling the need for both pattern and data updates (Sections 4 and 5). Due to the inherent difficulty of the problem, we propose an incremental strategy based on *pattern fragmentation* and *affected balls* by localizing the effects of pattern and data updates, and we develop a unified incremental algorithm for dealing with separate and simultaneous pattern and data updates, with an optimization technique with the *early return* property for incremental top-k algorithms, an analogy of the traditional early termination property.

(4) Using real-life data (CITATION) and synthetic data (SYNTHETIC), we demonstrate the effectiveness and efficiency of our graph pattern matching approach for (dynamic) team formation (Section 6). We find that (a) our method is able to identify more sensible teams than existing team formation methods *w.r.t.* practical measurements, and (b) our incremental algorithm outperforms our batch algorithm, even when changes reach 36% for pattern updates, 34% for data updates and (25%, 22%) for simultaneous pattern and data updates, and when 29% for continuous pattern updates, 26% for continuous data updates and (20%, 18%) for continuously simultaneous pattern and data updates, respectively.

To our knowledge, this work is among the first to study simultaneous pattern and data incremental computations, no previous work has studied pattern updates for incremental pattern matching [10, 13], not to mention continuous and simultaneous pattern and data updates. This is the most general dynamic setting for incremental computations.

All detailed proofs are available in the full version [4].

Related work. Previous work can be classified as follows.

Graph simulation [19] and its extensions have been introduced for graph pattern matching [11, 14, 28, 29], in which *strong simulation* introduces duality and locality into simulation [29], and shows a good balance between its computational complexity and its ability to preserve graph topology. Furthermore, [14] already adopts capacity bounds on the edges of pattern graphs via subgraph isomorphism, and [12] uses graph pattern matching to find single experts, instead of a team of experts. In this study, team simulation is proposed for team formation as an extension of graph simulation and strong simulation on undirected graphs with capacity constraints on the nodes of pattern graphs.

There has been a host of work on team formation by minimizing the communication cost of team members, based on the diameter, density, minimum spanning tree, Steiner tree, and sum of pairwise member distances among others [7, 9, 15, 22, 25, 27, 34], which are essentially a specialized class of keyword search on graphs [6]. Similar to [22], we are to find top-k teams. However, [22] adopted Lawler's procedure [26], and is inappropriate for large graphs. We also adopt *density* as the communication cost, which shows a better performance [15], and further require that all team members are *close to each other* (located in the same balls), along the same line with [7,9,22,25]. Except for simply minimizing the communication cost among team members, [20, 22] consider minimizing the cost among team members and team leaders. Different from these work, we introduce *structural constraints*, in terms of graph pattern matching [11, 29], into team formation, while retaining the capacity bounds on specific team members like [15, 34].

Incremental algorithms (see [10, 32] for a survey) have proven usefulness in a variety of applications, and have been studied for graph pattern matching [11,13] and team formation [7] as well. However, [10, 11, 13, 32] only consider data updates, and [7] only considers continuously coming new tasks. In this work, we deal with both pattern and data updates for team formation, and support both insertions and deletions. To our knowledge, this is the first study on pattern updates, and is the most general and practical dynamic setting considered so far.

Query reformulation (*a.k.a.* query rewriting) is to generate alternative queries that may produce better answers, and has been studied for structured queries [31], keyword queries [40] and graph queries [30]. However, different from our study of handling pattern updates, the focus of query reformulation is not on incremental computations.

Although top-k queries (see [21] for a survey) have been studied for both graph pattern matching and team formation [22], they have never been studied for both team formation and graph pattern matching in a dynamic setting.

2. DYNAMIC TEAM FORMATION

We first propose *team simulation*, a revision of traditional graph pattern matching. We then formally introduce the *top-k team formation* problem via team simulation. We finally present the *dynamic top-k team formation* problem.

2.1 Team Simulation

We first extend pattern graphs of traditional graph pattern matching to carry capacity requirements, and then define team simulation on undirected graphs.

We start with basic notations.

Data graphs. A *data graph* is a labeled undirected graph $G(V, E, l)$, where V and E are the sets of nodes and edges, respectively; and l is a total labeling function that maps each node in V to a set of labels.

Pattern graphs. A *pattern graph* (or simply pattern) is an undirected graph $P(V_P, E_P, l_P, f_P)$, in which (1) V_P and E_P are the set of nodes and the set of edges, respectively; (2) l_P is a total labeling function that maps each node in V_P to a single label; and (3) f_P is a total capacity function such that for each node $u \in V_P$, $f_P(u)$ is a closed interval $[x, y]$, where $x \leq y$ are non-negative integers.

Intuitively, $f_P(u)$ specifies a range bound for node u, indicating the required quantity for the matched nodes in data graphs. Note that for traditional patterns [11, 14, 16, 41], bounds are typically carried on edges, not on nodes. We also also denote data and pattern graphs as $G(V, E)$ and $P(V_P, E_P)$ respectively. The size of G (resp. P), denoted by $|G|$ (resp. $|P|$), is defined to be the total number of nodes and edges in G (resp. P).

We now redefine graph simulation on undirected graphs, which is originally defined on directed graphs [11, 19]. Consider pattern graph $P(V_P, E_P)$ and data graph $G(V, E)$.

Graph simulation. Data graph G *matches* pattern graph P via graph simulation, denoted by $P \prec G$, if there exists a binary *match relation* $M \subseteq V_P \times V$ in G for P such that

(1) for each $(u, v) \in M$, the label of u matches one label in the label set of v, *i.e.,* $l_P(u) \in l(v)$; and

(2) for each node $u \in V_P$, there exists $v \in V$ such that (a) $(u, v) \in M$, and (b) for each adjacent node u' of u in P, there exists a adjacent node v' of v in G such that $(u', v') \in M$.

For any G that matches P, there exists a *unique maximum* match relation via graph simulation [19].

We then introduce the notions of balls and match graphs.

Balls. For a node v in data graph G and a non-negative integer r, the *ball* with *center* v and *radius* r is a subgraph of G, denoted by $\hat{G}[v, r]$, such that (1) all nodes v' are in $\hat{G}[v, r]$, if the number of hops between v' and v, $\mathsf{hop}(v', v)$, is no more than r, and (2) it has exactly the edges appearing in G over the same node set.

Match graphs. The *match* graph *w.r.t.* a binary relation $M \subseteq V_P \times V$ is a subgraph G_s of data graph G, in which (1) a node $v \in V_s$ if and only if it is in M, and (2) it has exactly the edges appearing in G over the same node set.

Intuitively, the match graph G_s *w.r.t.* M is the induced subgraph of G such that its nodes play a role in M.

We are now ready to define team simulation, by extending graph simulation to incorporate the locality constraints enforced by balls, and the capacity bounds carried by patterns.

Team simulation. Data graph G *matches* pattern P via team simulation *w.r.t.* a radius r, denoted by $P \lhd_r G$, if there exists a *ball* $\hat{G}[v, t]$ ($t \in [1, r]$, $t \in Z$) in G, such that

(1) $P \prec \hat{G}[v, t]$, with the maximum match relation M and the match graph G_s *w.r.t.* M; and

(2) for each node u in P, the number of nodes v in G_s with $(u, v) \in M$ falls into $f_P(u)$.

We refer to G_s as a *perfect* subgraph of G *w.r.t.* P.

Intuitively, (1) pattern graphs P capture the structural and capacity constraints, and (2) a perfect subgraph G_s of pattern P corresponds to a desired team, which is required to satisfy the following conditions: (a) G_s itself is located in a ball $\hat{G}[v, t]$ where $t \in [1, r]$ as a match graph; and (b) G_s satisfies the capacity constraints carried over pattern P.

Example 3: Consider pattern P_1 and data graph G_1 in Fig. 1, and team simulation with $r = 2$ is adopted.

One can easily verify that P_1 matches G_1 via team simulation, *i.e.,* $P_1 \lhd_r G_1$, as (a) there is a perfect subgraph in in ball $\hat{G}[\mathsf{PM}_1, 2]$, *i.e.,* the connected component of G_1 containing PM_1, which maps PM, BA, UD, SA, SD and ST in P_1 to PM_1, BA_1, $\{\mathsf{UD}_1, \mathsf{UD}_2\}$, $\{\mathsf{SA}_1, \mathsf{SA}_2\}$, $\{\mathsf{SD}_1, \mathsf{SD}_2\}$ and $\{\mathsf{ST}_1, \mathsf{ST}_2\}$, respectively, and, moreover, (b) the capacity bounds on all pattern nodes are satisfied. □

Remarks. (1) Team simulation differs from graph simulation [19] and strong simulation [29] in the existence of capacity bounds on pattern graphs and its ability to capture matches on undirected graphs.

(2) Different from strong simulation with balls having a fixed radius (*i.e.,* the diameter of a pattern), team simulation adopts a more natural setting that the radius of the balls is flexible, and is only less than a user specified upper bound.

2.2 Top-k Team Formation

Given pattern P, data graph G, and two positive integers r and k, the *top-k team formation* problem, denoted as $\mathrm{kTF}(P, G, k)$, is to find a list L_k of k perfect subgraphs (*i.e.,* teams) with the top-k largest density in G for P, via team simulation.

Here the *density* den_G of graph $G(V, E)$ is $|E|/|V|$, where $|E|$ and $|V|$ are the number of edges and the number of nodes respectively, as commonly used in data mining applications [17, 39]. Intuitively, the larger den_G is, the more collaborative a team is. In this way, not only the two objective functions of existing team formation methods are preserved, *i.e.,* the locality retained by balls and the density function in selecting top-k results, but also the relationships among members and the capacity constraint on patterns.

Example 4: Consider P_1, G_1 in Fig. 1 and $r = 2$. We simply set $k = 1$, as most existing solutions for kTF only compute the best team [7, 9, 15, 25, 34].

One may want to look for candidate teams with existing methods, satisfying the search requirement in Example 1:
(1) by minimizing the *team diameter* [25], which returns the team with $\{\mathsf{BA}_3, \mathsf{PM}_3, \mathsf{UD}_4, \mathsf{SA}_4, \mathsf{SD}_4, \mathsf{ST}_4\}$,
(2) by minimizing the *sum of all-pair distances* of teams [22], which returns exactly the same team as (1) in this case, or
(3) by maximizing the *team density* [15], which returns the team with all the nodes in the two connected components in G_1 with PM_1 and BA_3, except UD_2, PM_3, UD_4, SA_4.

One may already notice that these teams only satisfy the skill requirement, *i.e.,* condition (i) in Example 1, and cannot guarantee the specific collaboration relationships among team members. Indeed, the team found in (1) and (2) is connected by BA_3 only, and the team found in (3) has loose collaborations among its members. That is, existing methods are not appropriate for identifying the the desired teams.

When team simulation is adopted, it returns the perfect subgraph in Example 3 with its density = 1.4, satisfying both conditions (i) and (ii), much better than those teams found by the above existing methods. □

2.3 Dynamic Top-k Team Formation

We now introduce dynamic top-k team formation.

Pattern updates (ΔP). There are five types of pattern updates: (1) *edge insertions* connecting nodes in P, (2) *edge deletions* disconnecting nodes in P, (3) *node insertions* attaching new nodes to P, (4) *node deletions* removing nodes

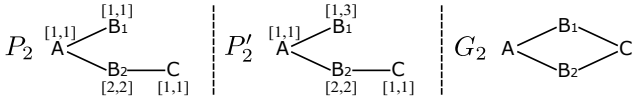


Figure 2: Pattern satisfiability

from P, and (5) *capacity changes* adjusting the node capacities in P, while P remains connected in all cases.

Data updates (ΔG). There are four types of data updates, defined along the same lines as the first four types of pattern updates. Further, different from pattern updates, there is no need to keep G connected for data updates.

Dynamic top-k team formation. Given pattern P, data graph G, positive integers r and k, the list $L_k(P, G)$ of top-k perfect subgraphs for P in G, a set of pattern updates ΔP and a set of data updates ΔG, the *dynamic top-k team formation* problem, denoted by kDTF$(P, G, k, L_k, \Delta P, \Delta G)$, is to find a list of k perfect subgraphs with the top-k largest density for $P \oplus \Delta P$ in $G \oplus \Delta G$, via team simulation.

Here $\oplus$ denotes applying changes ΔP to P and ΔG to G, and $P \oplus \Delta P$ and $G \oplus \Delta G$ denote the updated pattern and data graphs. It is worth mentioning that kDTF covers a broad range of dynamic situations, *i.e.,* handling continuously separate and simultaneous pattern and data updates.

3. FINDING TOP-K TEAMS

In this section, we develop a batch algorithm for top-k team formation. We first study the pattern satisfiability problem for team simulation, then introduce two optimization techniques, and finally we present our batch algorithm.

3.1 Pattern Satisfiability

Different from graph simulation [19] and its extensions [11, 29], there exist patterns that cannot match any data graph via team simulation, due to the presence of capacity constraints on patterns. We illustrate this with an example.

Example 5: (1) For pattern P_2 in Fig. 2, one can verify that there exist no data graphs G such that $P_2 \lhd_r G$ because (a) for any nodes v in G, if v matches with the node labeled with B_2, then it must match with the node labeled with B_1, and, hence, (b) the capacity upper bound on B_1 should not be less than the lower bound on B_2.
(2) However, pattern P_2' in Fig. 2 is satisfiable as $P_2' \lhd_r G_2$, and pattern P_1 in Fig. 1 is also satisfiable as $P_1 \lhd_r G_1$. □

We say that a pattern P is *satisfiable* iff there exists a data graph G such that P matches G via team simulation, *i.e.,* $P \lhd_r G$. The good news is that checking the satisfiability of pattern graphs can be done in low polynomial time.

Proposition 1: *The satisfiability of patterns P can be checked in $O(|P|^2)$ time.* □

By treating P as both data and pattern graphs, compute the maximum match relation M in P for P, via graph simulation. Then pattern P is satisfiable iff for each $(u, v) \in M$ with the capacity bounds $[x_u, y_u]$ on u and $[x_v, y_v]$ on v, respectively, $x_v \leq y_u$ holds. Observe that the size of M is bounded by $|P|^2$, and pattern graphs are typically small.

By Proposition 1, we shall consider satisfiable pattern graphs only in the sequel.

3.2 Batch Algorithm

We then introduce two techniques for optimizing the computation of team simulation.

Handling radius varied balls. kTF is to find top-k teams within balls $\hat{G}[v, t]$, where $v \in V$ and $t \in [1, r]$. However, it is very costly to construct all $r|V|$ balls, and to compute perfect subgraphs in all of them. Indeed it is also not necessary, and it only needs to construct and compute the matches for a number of $|V|$ balls, *i.e.,* the set of balls $\hat{G}[v, r]$ where $v \in V$ and radius is r, and then incrementally computes the perfect subgraphs for balls $\hat{G}[v, t]$ ($t \in [1, r-1]$) from the match graphs for ball $\hat{G}[v, r]$, as shown below.

Theorem 2: *Given P, ball $\hat{G}[v, r]$ and $\hat{G}[v, t]$ ($t \in [1, r-1]$) in G, (1) if $P \prec \hat{G}[v, t]$, then $P \prec \hat{G}[v, r]$; and (2) if G_s (resp. G_s') is the match graph* w.r.t. *the maximum match relation M (resp. M') in $\hat{G}[v, r]$ (resp. $\hat{G}[v, t]$) for P via graph simulation, then $M' \subset M$, and G_s' is a subgraph of G_s.* □

When we have the match graph G_s in $\hat{G}[v, r]$ for P via graph simulation, to compute the perfect subgraph in $\hat{G}[v, t]$ ($t \in [1, r-1]$) for P via team simulation, we need to (1) first identify the subgraph G_s^t in G_s belonging to $\hat{G}[v, t]$, which can be easily identified in the process for constructing $\hat{G}[v, r]$ without extra computation; (2) check whether G_s^t is already a match graph for P in $\hat{G}[v, t]$ via graph simulation; if not, remove the unmatched nodes and edges from G_s^t until find the match graph G_s' for P in $\hat{G}[v, t]$. This can be achieved by executing an efficient incremental process in [13]; and (3) finally check whether capacity bounds are satisfied. If so, G_s' is the perfect subgraph in $\hat{G}[v, r]$ for P via team simulation.

Density based ball filtering. We further reduce the number of balls to speedup the process by adopting the density based filtering technique. The key idea is to tell whether a ball is possible to produce one of the final top-k matches.

Given a ball $\hat{G}[v, r]$, we compute the density upper bound $\mathsf{den}_{\hat{G}_s}$, where $\hat{G}_s$ is a subgraph of $\hat{G}[v, r]$. If the bound is larger than the density of the current k-th result, *i.e.,* there is a possibility for the ball; Otherwise, the ball is simply ignored to avoid redundant computations.

The trick part is how to efficiently compute the upper bound of $\mathsf{den}_{\hat{G}_s}$ for each ball in G. As the best densest subgraph algorithms are in $O(|\hat{G}[v, r]|^3)$ time [17], which is costly, we utilize an important result in [39], shown below.

Lemma 3: *Let den_{H_c} and den_{H_d} be the density of the maximum core H_C and the densest subgraph H_d of graph H. Then (1) $\mathsf{den}_{H_c} \leq \mathsf{den}_{H_d} \leq 2 * \mathsf{den}_{H_c}$; and (2) there exists an algorithm that computes den_{H_c} in $O(|E_H|)$ time [39].* □

Here the maximum core H_C of a graph H is a subgraph of H whose node degree is at least ρ, where ρ is the maximum possible one. By Lemma 3, we use $2 * \mathsf{den}_{H_c}$ as the density upper bound for filtering unnecessary balls.

We are now ready to present our batch algorithm for kTF.

Algorithm batch**.** As shown in Fig. 3, it takes input as P, G, and two integers r and k, and outputs the top-k densest perfect subgraphs in G for P. It firstly checks whether P is satisfiable (line 1). If so, for each ball $\hat{G}[v, r]$ in G, it computes the maximum core $\hat{G}_C$ of $\hat{G}[v, r]$, and checks whether the density based ball filtering condition holds (lines 3-6). If so, it skips the current ball, and moves to the next one; otherwise, it computes the perfect subgraph G_s of P in $\hat{G}[v, r]$ via team simulation by invoking undirgSim (line 7, see full

Input: $G(V, E)$, $P(V_P, E_P)$, and positive integers r and k.
Output: Top-k densest teams.

1. **if** P is unsatisfiable **then return** nil;
2. $L_k := \emptyset$;
3. **for** each ball $\hat{G}[v, r]$ in G **do**
4. compute the maximum core $\hat{G}_C$ of the ball $\hat{G}[v, r]$;
5. **if** $2*\mathsf{den}_{\hat{G}_C} \leq$ the density of the k-th result in L_k **then**
6. **continue**;
7. $G_s := \mathsf{undirgSim}\ (P, \hat{G}[v, r])$;
8. If G_s satisfies capacity bounds on P **then** Insert G_s into L_k;
9. **for** each ball $\hat{G}[v, t]$ with $t \in [1, r-1]$ **do**
10. $G'_s := \mathsf{incSim}\ (G_s, P, \hat{G}[v, t])$;
11. If G'_s satisfies capacity bounds on P **then** Insert G'_s into L_k;
12. **return** $L_k[0 : k-1]$.

Figure 3: Algorithm batch

version [4]), an adaption from graph simulation [11, 19] and checking capacity bounds (line 8). It then computes perfect subgraphs G'_s of P in inner balls $\hat{G}[v, t]$ by invoking incSim, an extension of the data incremental algorithms in [13] and checking capacity bounds (lines 9-11).

Correctness & complexity analyses. The correctness of batch is assured by the following.

(1) The correctness of undirgSim (resp. incSim) can be verified along the same lines as graph simulation [19] (resp. incremental simulation [13]).

(2) Theorem 2 and Lemma 3. It takes $O(|P|^2)$ to check pattern satisfiability, $O(|V||P||G|)$ to compute team simulation, $O(r|V||V_P||E|)$ to incrementally compute matches in inner balls, and $O(|V||E|)$ to compute the density of the maximum core for $|V|$ balls. Thus batch is in $O(|P|^2 + |V||P||G| + r|V||V_P||E|)$. However, actual time is much less due to density based ball filtering and that $O(r|V||V_P||E|)$ is the worst case complexity for incremental process, while r is small, *i.e.,* 2 or 3.

4. A UNIFIED INCREMENTAL SOLUTION

In this section, we first analyze the challenges and design principles of dynamic top-k team formation, and then develop a unified incremental framework for kDTF. For convenience, the notations used are summarized in Table 1.

4.1 Analyses of Dynamic Team Formation

By Theorem 2, pattern P matches a ball $\hat{G}[v, t]$ ($t \in [1, r-1]$), only if P matches ball $\hat{G}[v, r]$ via graph simulation, and the match results for $\hat{G}[v, t]$ can be derived from the matches for $\hat{G}[v, r]$. Therefore, the key of the incremental computation is to deal with the balls $\hat{G}[v, r]$ with radius r. In the sequel, a ball has a radius r by default.

We first analyze the inherent computational complexity of the dynamic top-k team formation.

Incremental complexity analysis. As observed in [32, 33], the complexity of incremental algorithms should be measured by the size $|\mathsf{AFF}|$ of *the changes* in the input and output, rather than the entire input, to measure the amount of work essentially to be performed for the problem.

An incremental problem is said to be *bounded* if it can be solved by an algorithm whose complexity is a function of $|\mathsf{AFF}|$ alone, and is *unbounded*, otherwise. Unsurprisingly, the dynamic top-k team formation problem is unbounded, similar to the other extensions of graph simulation [11, 13].

Notations	Description
P, G	pattern and data graphs
$\hat{G}[v, r]$	a ball in G with center node v and radius r
$L_k(P, G)$	the list of top-k perfect subgraphs in G for P
$\Delta P, \Delta G$	pattern and data updates
$\oplus$	applying updates ΔP and ΔG to P and G
$\mathcal{P}_h = \{P_{fi}, C\}$	pattern fragmentation: h fragments and cut
AffBs	affected balls
$M(P_{fi}, \hat{G}[v, r])$	the maximum match relation in $\hat{G}[v, r]$ for P_{fi}
$M(P, G)$	fragment-ball matches (auxiliary structure)
FS, BS	fragment status, ball status (auxiliary structure)
FBM	fragment-ball-match index, containing FS, BS
BF, UP	ball filter, update planner (auxiliary structure)

Table 1: Notations

Proposition 4: *The* kDTF *problem is unbounded, even for* $k = 1$ *and unit pattern or data updates.* □

We then illustrate the impact of pattern and data updates on the matching results with an example.

Example 6: Continue Example 2 with ΔP_1 and ΔG_1.

(1) For ΔP_1, $\hat{G}[\mathsf{PM}_1, 2]$ already matches P, and may produce more matched nodes for $P_1 \oplus \Delta P_1$, thus a re-computation for perfect subgraphs is needed. For all other balls, ΔP_1 may turn unmatched nodes to matched and may produce perfect subgraphs, thus re-computation is also needed.

(2) For ΔG_1, it produces a new perfect subgraph for P in $G_1 \oplus \Delta G_1$, *i.e.,* the connected component having PM_2. □

We finally discuss the challenges and principles of designing incremental algorithms for kDTF from three aspects.

(1) Impacts of pattern and data updates. Beyond Proposition 4 and Example 6, one can also verify that (a) unit pattern updates are likely to result in the entire change in previous results, such that all balls need to be accessed and all matches need to be re-computed, and (b) the impact of data updates can also be global, such that the entire data graph may need to be accessed to re-compute matches. Hence, the key is to identify and localize the impacts of pattern and data updates.

(2) Maintenance of auxiliary information. Auxiliary data on intermediate or final results for P in G are typically maintained for incremental computation [13,33]. How to design light-weight and effective auxiliary structures is critical. One may want to store $M(P, G)$, the match relations of P for all balls in G, as adopted by existing incremental pattern matching algorithms for data updates [13]. However, the impact of ΔP is global, as shown in Example 6. By storing $M(P, G)$, for pattern edge/node deletions, it has to recompute matches for all balls, *i.e.,* the entire $M(P, G)$. Thus, storing $M(P, G)$ could be useless, not to mention $L_k(P, G)$, the list of top-k perfect subgraphs for P in G *w.r.t.* $M(P, G)$.

(3) Support of continuous pattern and data updates. A practical solution should support continuous pattern and data updates, separately and simultaneously, which further increases difficulties on the design of auxiliary data structures and incremental algorithms.

4.2 A Unified Incremental Framework

Nevertheless, we develop an incremental approach to handling pattern and data updates in a unified framework, by utilizing *pattern fragmentation* and *affected balls* to localize the impacts of pattern and data updates, and to reduce the cost of maintaining auxiliary structures and computations.

(I) Localization with pattern fragmentation. We

say that $\{P_{f1}(V_{f1}, E_{f1}), \ldots, P_{fh}(V_{fh}, E_{fh}), C\}$ is an *h-fragmentation* of pattern $P(V_P, E_P)$, denoted as $\mathcal{P}_h$, if (1) $\bigcup_{i=1}^{h} V_{fi} = V_P$, (2) $V_{fi} \cap V_{fj} = \emptyset$ for any $i \neq j \in [1, h]$, (3) E_{fi} is exactly the edges in P on V_{fi}, and (4) $C = E_P \setminus (E_{f1} \cup \ldots \cup E_{fh})$.

We also say P_{fi} ($i \in [1, h]$) as a *fragment* of P, and C as a *cut* of P, respectively.

Observe that by pattern fragmentation, a pattern update on P is either on a fragment P_{fi} or on the cut C of P, and, in this way, the impact of pattern updates is localized. Moreover, graph simulation holds a nice property on pattern fragmentation, as shown below.

Theorem 5: *Let $\{P_{f1}, \ldots, P_{fh}\}$ be an h-fragmentation of pattern P. For any ball $\hat{G}$ in G, let M_i ($i \in [1, h]$) be the maximum match relation in $\hat{G}$ for P_{fi} via graph simulation, and M be the maximum match relation in $\hat{G}$ for P via graph simulation, respectively, then $M \subseteq \bigcup_{i=1}^{h} M_i$.* □

We also say that M_i is a *partial match relation* in ball $\hat{G}$ for P via graph simulation. By the nature of graph simulation [19], $\bigcup_{i=1}^{h} M_i$ is actually an intermediate result of M. Once we have the maximum match relation M for P in $\hat{G}$, via graph simulation, we can further produce the result for P in $\hat{G}$ via team simulation, by a capacity check.

That is, based on pattern fragmentation, we maintain an auxiliary structure for storing fragment-ball matches for incremental computations, *i.e.,* $\tilde{M}(P, G)$ *w.r.t.* $\mathcal{P}_h$ that is the maximum match relations for all pattern fragments of P in all balls of G, via graph simulation. Moreover, its space cost is light-weight, as will be shown in the experimental study.

By storing $\tilde{M}(P, G)$, we have $\bigcup_{i=1}^{h} M_i$ for each ball $\hat{G}$, and we can simply update M_i while leaving other parts untouched. That is, we indeed compute for $P_{fi} \oplus \Delta P(\hat{G})$, instead of $P \oplus \Delta P(\hat{G})$, and combine all $P_{fi} \oplus \Delta P(\hat{G})$ to derive $P \oplus \Delta P(\hat{G})$. Even better, the updates ΔP on the cut C of P only involve with a simple combination process, avoiding the computation for any pattern fragments.

For a better incremental process, we typically want (1) to avoid skewed updates by balancing the sizes of all fragments, and (2) to minimize the efforts to assemble the partial matches of all fragments. Thus we define and investigate the *pattern fragmentation* problem.

Given pattern P and a positive integer h, it is to find an h-fragmentation of P such that both $\max(|P_{fi}|)$ ($i \in [1, h]$) and $|C|$ are minimized. Intuitively, the bi-criteria optimization problem partitions a pattern into h components of roughly equal size while minimizing the cut size.

The problem is intractable, as shown below.

Proposition 6: *The pattern fragmentation problem is* NP*-complete, even for $h = 2$.* □

However, P and h are typically small in practice [11], *e.g.,* $|P| = 15$ and $h = 3$. In light of this, we give a heuristic algorithm, denoted by PFrag, for the problem, and is shown in the full version [4]. PFrag works by connecting pattern fragmentation to the widely studied (k, ν)-Balanced Partition problem [8], which is not approximable in general, but has efficient and sophisticated heuristic algorithms [23].

(II) Localization with affected balls (AffBs). We further localize the impact of pattern and data updates with *affected balls* to avoid unnecessary computations.

We say that a ball in G is *affected w.r.t.* an incremental algorithm $\mathcal{A}$, and pattern and data updates, if $\mathcal{A}$ accesses the ball again. We use $||$AffBs$||$ and $|$AffBs$|$ to denote the cardinality and total size of AffBs, respectively.

Indeed, AffBs are those balls with a possibility to have final results *w.r.t.* ΔP and ΔG. We only access AffBs, and ignore the rest balls. Specifically, (1) for ΔP, it allows us to avoid computing updated partial relations for an updated fragment in every ball; and (2) for ΔG, the locality property of team simulation supports to localize the update impacts to a set of balls whose structures are changed by ΔG.

(III) Algorithm framework. We now provide a unified incremental algorithm to handle both pattern and data updates, based on pattern fragmentation and affected balls.

Given pattern P with its h-fragmentation $\mathcal{P}_h$, data graph G, two integers r and k, and auxiliary structures (to be introduced in Section 5) such as the partial match relations for all pattern fragments and all balls (radius r), algorithm dynamic consists of three steps for ΔP and ΔG, as follows.

(1) Identifying AffBs. Algorithm dynamic invokes two different procedures to identify AffBs for separate ΔP or ΔG, respectively. For simultaneous ΔP and ΔG, dynamic takes the union of the AffBs produced by the two procedures.

(2) Update partial match relations in AffBs. For a ball affected by ΔP, dynamic updates the partial match relations for the updated pattern fragments with incremental computation; For a ball affected by ΔG, dynamic updates the partial match relations for all pattern fragments; And, for a ball affected by both ΔP and ΔG, dynamic follows the same way as it does for ΔG only. Meanwhile, auxiliary structure FBM (to be seen shortly) is updated for handling continuously separate and simultaneous pattern and data updates.

(3) Combining partial match relations. dynamic combines all partial relations for a subset of AffBs and computes the top-k perfect subgraphs within them and their inner balls.

Observe that dynamic handles pattern and data updates, separately and simultaneously, in a unified way.

5. INCREMENTAL ALGORITHMS

In this section, we introduce the details of our incremental algorithm dynamic, including (a) auxiliary data structures, (b) algorithms dynamicP and dynamicG to handle pattern and data updates, respectively, and (c) dynamic by integrating dynamicP and dynamicG together.

5.1 Auxiliary Data Structures

Auxiliary structures fall into two classes: maintain partial matches and handle pattern incremental computing. Consider an h-fragmentation $\mathcal{P}_h = \{P_{f1}, \ldots, P_{fh}, C\}$ of pattern $P(V_P, E_P)$, data graph $G(V, E)$, and pattern updates ΔP.

(I) Data structures in the first class are as follows.

*(1) Fragment status (*FS*)* consists of 2^h boolean vectors $(b_1, \ldots, b_h)$, referred to as *type code (tc)*, where b_i ($i \in [1, h]$) is either 0 or 1. Recall that h is very small, *e.g.,* 3.

We use FS to classify the match status of balls in G into 2^h types *w.r.t.* $\mathcal{P}_h$. For a ball with type code $(b_1, \ldots, b_h)$, b_i is 1 iff P_{fi} matches the ball via graph simulation.

*(2) Ball status (*BS*)* consists of $|V|$ triples $(bid, cflag, den)$, such that *bid* is the *id* of a ball, $cflag$ is the id of the latest processed unit pattern update for the ball (initially set to 0), and *den* is the density upper bound of subgraphs in the ball.

We use BS to store the basic information for balls in G.

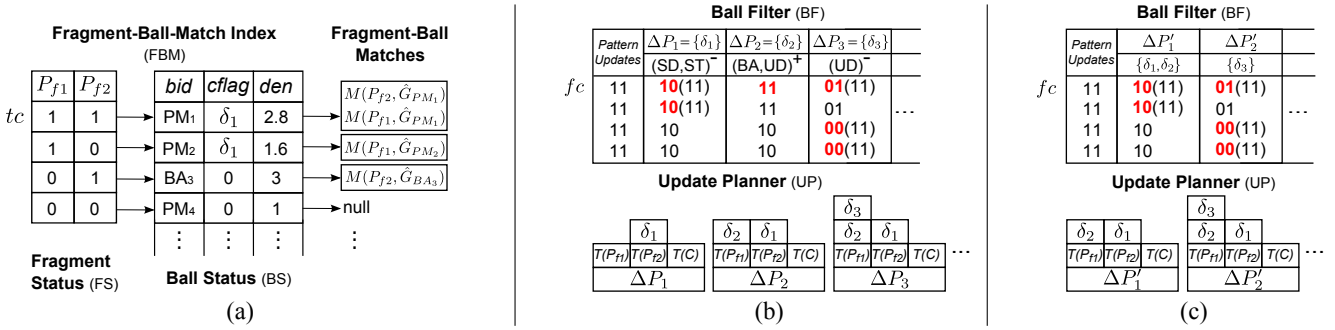


Figure 4: Example auxiliary data structures

(3) Fragment-ball matches of P in G, denote as $\tilde{M}(P,G)$, are $\bigcup_{i\in[1,h],v\in V} M(P_{fi},\hat{G}[v,r])$, such that $M(P_{fi},\hat{G}[v,r])$ is the maximum match relation for P_{fi} in ball $\hat{G}[v,r]$, via graph simulation, and there are in total $|V|$ balls.

Here $\tilde{M}(P,G)$ is used to store match relations for the pattern fragments of P in all balls of G. Instead of storing a single $\tilde{M}(P,G)$, we organize $\tilde{M}(P,G)$ in terms of the match status between pattern fragments and balls, *i.e.,* FS and BS.

*(4) Fragment-ball-match index (*FBM*)* links FS and BS together, to form the *fragment-ball-match index*. Then FBM is linked to $\tilde{M}(P,G)$. The details are shown below.

For each record of ball $\hat{G}[v,r]$ in BS, (a) there is a link from its type code in FS pointing to the record; and (b) there is another link from the record to a set of $M(P_{fi},\hat{G}[v,r])$ $(i\in[1,h])$ in $\tilde{M}(P,G)$, if the type code with which the ball is associated has $b_i=1$, *i.e.,* $M(P_{fi},\hat{G}[v,r])$ is not empty.

Intuitively, FBM indexes the partial match relations $\tilde{M}(P,G)$ based on the match status of balls *w.r.t.* $\mathcal{P}_h$.

Example 7: Consider P_1 and G_1 (both without dashed edges) in Fig. 1, $r=2$, $k=2$, $h=2$, auxiliary structures $\tilde{M}(P,G)$ and FBM that are shown in Fig. 4(a).
(1) Pattern P_1 is divided into fragments P_{f1} and P_{f2} by algorithm PFrag, so there are $2^2=4$ type codes in FS.
(2) For balls linked with tc (1, 1), *e.g.,* ball $\hat{G}[\mathsf{PM}_1,2]$, there are matches to both P_{f1} and P_{f2} in the ball. Besides, there exist balls $\hat{G}[\mathsf{PM}_2,2]$, $\hat{G}[\mathsf{BA}_3,2]$ and $\hat{G}[\mathsf{PM}_4,2]$ linked with tc (1, 0), (0, 1) and (0, 0) respectively. For simplicity, we use these 4 balls only in the following analysis. □

These structures enforce a nice property as follows.

Theorem 7: *With $\tilde{M}(P,G)$ and* FBM w.r.t. *an h-fragmentation of P, given ΔP and ΔG, the incremental algorithm* dynamic *processes ΔP and ΔG in time determined by P, $\tilde{M}(P,G)$ and* AffBs*, not directly depending on G.* □

We shall prove Theorem 7 by providing specific techniques for dynamic and analyzing its time complexity.

(II) Data structures in the second class are as follows.

*(1) Ball filter (*BF*)* consists of 2^h boolean vectors (b_1, …, b_h), referred to as *filtering code (fc)*, such that each fc in BF corresponds to a type code tc in FS. Each b_i $(i\in[1,h])$ in an fc of BF is initially set to 1, and is updated for each unit pattern update δ in ΔP: (a) when δ is an edge deletion or a node deletion to P_{fi}, the i-th bit of all the 2^h filtering codes in BF is set to 0; Otherwise, (b) BF remains intact.

*(2) Update planner (*UP*)* consists of $h+1$ stacks $T(P_{f1})$, …, $T(P_{fh})$, $T(C)$. Stack $T(P_{fi})$ $(i\in[1,h])$ (resp. $T(C)$) records all unit updates in all arrived pattern updates ΔP_1, …, ΔP_N that are applied to fragment P_{fi} (resp. C) of P. Initially, all of them are empty, and are dynamically updated for each unit update in each coming set of pattern updates.

5.2 Dealing with Pattern Updates

We present algorithm dynamicP to handle pattern updates ΔP, following the steps in Section 4.2, and an early return optimization technique for dynamicP.

(I) Identifying affected balls. We first develop procedure IdABall to identify AffBs with structures FBM and BF.

Procedure IdABall. Given an h-fragmentation $\mathcal{P}_h$ of P, ΔP, (1) it updates BF by processing all unit updates in ΔP. (2) For each $j\in[1,2^h]$, it then executes a bitwise AND operation (&) between type code tc_j of FS in FBM and updated filtering code $fc_{j\Delta}$ in BF, *i.e.,* $tc_j\&fc_{j\Delta}$. (3) Finally, if $tc_j\&fc_{j\Delta}=fc_{j\Delta}$, IdABall refers to BS in FBM to mark the balls with type code tc_j as AffBs, and resets $fc_{j\Delta}$ to $(1,\ldots,1)$. The condition $tc_j\&fc_{j\Delta}=fc_{j\Delta}$ holds as long as (a) the i-th $(i\in[1,h])$ bit of $fc_{j\Delta}$ is 0, *i.e.,* there exists an edge/node deletion on P_{fi}, which may produce more matched nodes, or (b) the i-th $(i\in[1,h])$ bit of $fc_{j\Delta}$ and tc_j are both 1, *i.e.,* balls with tc_j already match with P_{fi}, though there are no edge/node deletions on P_{fi}.

Example 8: Consider the input and auxiliary structures in Example 7, and BF in Fig. 4.
(1) ΔP_1 comes with a unit edge deletion δ_1=(SD, ST)$^-$ on P_{f_2}, and all four fc in BF are updated from fc (1, 1) to fc_Δ (1, 0), as shown in the second column of BF in Fig. 4(b). IdABall identifies ball $\hat{G}[\mathsf{PM}_1,2]$ with tc (1, 1) and $\hat{G}[\mathsf{PM}_2,2]$ with tc (1, 0) as AffBs, since $tc(1,1)\&fc_\Delta(1,0)=fc_\Delta(1,0)$ and $tc(1,0)\&fc_\Delta(1,0)=fc_\Delta(1,0)$. Then IdABall resets the two corresponding filtering codes in BF to (1, 1).
(2) Consider another case when $\Delta P_1'$ comes with δ_1 and δ_2, where δ_1 is same as above and $\delta_2=(\mathsf{BA},\mathsf{UD})^+$. BF is updated as shown in the second column of BF in Fig. 4(c), and IdABall identifies the same AffBs as above. □

The correctness of IdABall is ensured by the following.

Proposition 8: *For any ball $\hat{G}[v,r]$ in G, if there exists a perfect subgraph of $P\oplus\Delta P$ in $\hat{G}[v,r]$, then $\hat{G}[v,r]$ must be an affected ball produced by procedure* IdABall. □

Lazy update policy. To reduce computation, dynamicP only updates the partial relations for AffBs in $\tilde{M}(P,G)$ for computing $L_k(P\oplus\Delta P,G)$. However, those partial relations in the filtered balls also need an update for handling future

updates $\Delta P'$, but definitely become outdated *w.r.t.* $P \oplus \Delta P$. Hence, dynamicP needs a smart policy to maintain those match relations in the filtered balls.

To do this, algorithm dynamicP maintains the status of all unit updates applied to P so far, and processes unit updates in ΔP as *late* as possible, while having no effects on future updates $\Delta P'$, *i.e.,* a *lazy update policy.*

Algorithm dynamicP utilizes auxiliary structure UP together with the $cflag$ item in BS. When handling current ΔP, for each ball $\hat{G}$, $\hat{G}$[cflag] records the id of the latest processed unit pattern update for $\hat{G}$, and is initialized to 0. When future $\Delta P'$ comes, for any AffB $\hat{G}$ *w.r.t.* $\Delta P'$ and any fragment P_{fi}, dynamicP computes $M(P_{fi} \oplus \Delta P'_{fi}, \hat{G})$ based on $M(P_{fi}, \hat{G})$ by procedure IncMatch (to be seen shortly), where $\Delta P'_{fi}$ consists of the unit updates stored in $T(P_{fi})$ whose ids are larger than $\hat{G}$[cflag] in BS.

Example 9: Continue Example 8. (1) Balls $\hat{G}[\mathsf{PM}_1, 2]$ and $\hat{G}[\mathsf{PM}_2, 2]$ are AffBs, and UP is shown in Fig. 4(b).
(a) UP is updated *w.r.t.* $\Delta P_1 = \{\delta_1\}$. dynamicP updates the partial relations for P_{f2} *w.r.t.* δ_1 in the two balls, and sets their $cflag$ in BS to δ_1, as the status shown in Fig. 4(a).
(b) Afterwards, ΔP_2 with an edge insertion $\delta_2 = (\mathsf{BA}, \mathsf{UD})^+$ comes. IdABall updates BF and UP as shown in Fig. 4(b) and identifies balls with *tc* $(1, 1)$ as AffBs, *e.g.,* $\hat{G}[\mathsf{PM}_1, 2]$.
(c) Finally, ΔP_3 with a node deletion $\delta_3 = (\mathsf{UD})^-$ comes. IdABall identifies *tc* $(1, 1)$, $(0, 1)$ and $(0, 0)$ as AffBs. Take ball $\hat{G}[\mathsf{BA}_3, 2]$ for example, which is the first time identified as an AffB. By referring to UP, dynamicP updates the partial relations for P_{f1} *w.r.t.* $\{\delta_2, \delta_3\}$, and for P_{f2} *w.r.t.* $\{\delta_1\}$.
(2) In the case when $\Delta P'_i$ contains multiple updates, BF and UP are updated accordingly as shown in Fig. 4(c). □

(II) Updating Fragment-Ball matches. We then update the partial match relations for AffBs in $\tilde{M}(P, G)$ *w.r.t.* ΔP, by procedure IncMatch.

Procedure IncMatch. Given h-fragmentation $\mathcal{P}_h$ of P, G, $\tilde{M}(P, G)$, ΔP, UP and AffBs *w.r.t.* ΔP. IncMatch updates $M(P_{fi}, \hat{G})$ to $M(P_{fi} \oplus \Delta P_{fi}, \hat{G})$ in $\tilde{M}(P, G)$ for each fragment P_{fi} and each AffB $\hat{G}$. Recall that ΔP_{fi} consists of unprocessed unit updates accumulated in UP applied to P_{fi}. We show how to update $M(P_{fi}, \hat{G})$ in different cases.

(1) There exist edge/node deletions in ΔP_{fi}. In this case, IncMatch accesses the AffB $\hat{G}[v, r]$ in G. It simply computes the maximum match relations for $P_{fi} \oplus \Delta P_{fi}$ in $\hat{G}[v, r]$ by procedure undirgSim in $O(|P_{fi} \oplus \Delta P_{fi}||\hat{G}[v, r]|)$ time.

(2) No edge/node deletions in ΔP_{fi}. IncMatch processes updates of the same type together in this case as follows.
(i) Capacity changes in ΔP_{fi} or updates on C. In this case, no computation is needed for maintaining partial relations for AffBs at all, *i.e.,* $M(P_{fi} \oplus \Delta P_{fi}, \hat{G}) = M(P_{fi}, \hat{G})$. Only a capacity check and an inner ball check in the combination procedure are needed (to be seen immediately).
(ii) Edge insertions in ΔP_{fi}. In this case, IncMatch calls procedure patEIns to process edge insertions.

Procedure patEIns. Given $M(P_{fi}, \hat{G}[v, r])$ (also represented by $\mathsf{R}(\cdot)$), $\hat{G}[v, r]$ and an edge insertion $\delta = (u, u')$, patEIns computes $M(P_{fi} \oplus \delta, \hat{G}[v, r])$ *incrementally*, as shown in Fig. 5, along the same lines as for data incremental graph simulation [13]. patEIns first finds the directly affected data

Input: $M(P_{fi}, \hat{G}[v, r])$, $\hat{G}[v, r]$, pattern edge insertion $\delta = (u, u')^+$.
Output: $M(P_{fi} \oplus \delta, \hat{G}[v, r])$.

1. RMv := ∅;
2. **for each** $u \in V_P$ **do** $\mathsf{R}(u) := \{w | (u, w) \in M(P_{fi}, \hat{G}[v, r])\}$;
3. **for each** node $w \in \mathsf{R}(u)$ **do**
4. **if** there exists no $(w, w') \in E_{\hat{G}[v,r]}$ with $w' \in \mathsf{R}(u')$ **then**
5. RMv.push($[u, w]$);
6. **for each** node $w' \in \mathsf{R}(u')$ **do**
7. **if** there exists no $(w', w) \in E_{\hat{G}[v,r]}$ with $w \in \mathsf{R}(u)$ **then**
8. RMv.push($[u', w']$);
9. **while** RMv $\neq \emptyset$ **do**
10. $[u, w]$:= RMv.pop(); $\mathsf{R}(u) := \mathsf{R}(u) \setminus \{w\}$;
11. **for each** $(u, u') \in E_{P_{fi}}$ **do**
12. **for each** $(w, w') \in E_{\hat{G}[v,r]}$ with $w' \in \mathsf{R}(u')$ **do**
13. **if** there is no $(w', w'') \in E_{\hat{G}[v,r]}$ with $w'' \in \mathsf{R}(u)$ **then**
14. RMv.push ($[u', w']$);
15. **if** there is a node $u \in V_{P_{fi}}$ with $|\mathsf{R}(u)| = 0$ **then** $\mathsf{R}(\cdot) := \emptyset$;
16. $M(P_{fi} \oplus \delta, \hat{G}[v, r]) := \{(u, w) | u \in V_P, w \in \mathsf{R}(u)\}$;
17. **return** $M(P_{fi} \oplus \delta, \hat{G}[v, r])$;

Figure 5: Procedure patEIns

nodes that need to be removed from $\mathsf{R}(\cdot)$ due to the edge insertion to P_{fi}, and pushes them along with the matched pattern nodes into RMv (lines 3-8). It then recursively identifies and removes the nodes in $\mathsf{R}(\cdot)$ affected by the previous removed nodes (lines 9-14). The recursive process is executed by utilizing a stack RMv. If there exists a pattern node u with empty $\mathsf{R}(u)$, then $\mathsf{R}(\cdot)$ is set to $\emptyset$ (line 15). Finally, patEIns returns the updated $\mathsf{R}(\cdot)$ for $P_{fi} \oplus \delta$ (lines 16-17).

Example 10: Consider case (1)-(b) in Example 9. Given $\delta_2 = (\mathsf{BA}, \mathsf{UD})^+$, and $M(P_{f1}, \hat{G}_{\mathsf{PM}_1})$, which is composed of nodes PM_1, BA_1 and $\{\mathsf{UD}_1, \mathsf{UD}_2\}$ mapped to nodes PM, BA and UD in P_{f1}. To compute the updated $M(P_{f1} \oplus \delta_2, \hat{G}_{\mathsf{PM}_1})$, patEIns removes UD_2 that is directly affected by δ_2, and finds no other nodes need to be removed. □

(iii) Node insertions in ΔP_{fi}. Node insertions are handled in a similar way as edge insertions, by extending patEIns.

Given a node insertion $\delta = (u, (u, u'))^+$, where u is a newly inserted node, to compute the updated $M(P_{fi} \oplus \delta, \hat{G}[v, r])$, IncMatch firstly computes the set of nodes $\mathsf{R}(u)$ in $\hat{G}[v, r]$ that have the same label with u, and then calls patEIns $(M(P_{fi}, \hat{G}[v, r]), \hat{G}[v, r], (u, u'))$ to get the updated result.

Updating FBM. After updating $\tilde{M}(P, G)$, dynamicP updates FBM for all AffBs, by changing the links according to the updated partial relations in $\tilde{M}(P, G)$, and also updating the cflag item in BS, which is in $O(||\mathsf{AffBs}||)$ time.

(III) Combining Fragment-Ball matches. Algorithm dynamicP finally combines the updated partial match relations in AffBs to get the updated top-k perfect subgraphs $L_k(P \oplus \Delta P, G)$ by procedure combine. Observe that only the balls from AffBs that match with all pattern fragments of $P \oplus \Delta P$ can enter the combination process.

Procedure combine. For an AffB $\hat{G}[v, r]$, combine invokes patEIns$(\bigcup_{i \in [1,h]} M(P_{fi}, \hat{G}[v, r]), \hat{G}[v, r], C \oplus \Delta C)$ to compute the maximum match relations of $P \oplus \Delta P$ for $\hat{G}[v, r]$ incrementally, where ΔC consists of the edge insertions/deletions in ΔP applied to the cut edges C. It then checks whether the capacity bounds, together with the updates on them, are satisfied. If so, it constructs the perfect subgraph *w.r.t.* the match relations above. It then checks the inner balls

Input: P, h-fragmentation $\mathcal{P}_h$, G, integers r and k, ΔP, and auxiliary structures $\tilde{M}(P,G)$, FBM, BF and UP.
Output: Top-k perfect subgraphs for $P \oplus \Delta P$ in G.

1. $L_k := \emptyset$;
2. AffBs := IdABall($\mathcal{P}_h, \Delta P$, FBM, BF);
3. Sort AffBs by $\hat{G}$[den] in non-ascending order;
4. **for each** $\hat{G}[v,r]$ in AffBs **do** /* non-ascending order */
5. **if** $|L_k| \geq k$ **and** $\hat{G}[v,r]$[den] $\leq$ den$_{L_k[k-1]}$ **then**
6. **Output** $L_k[0:k-1]$. **/* early-return optimization*/**
7. IncMatch($M(P_{fi}, \hat{G}[v,r]), \hat{G}[v,r], \Delta P_{fi}$) ($i \in [1,h]$); /* runs in the background */
8. S_{G_s} := combine($\bigcup_{i\in[1,h]} M(P_{fi}, \hat{G}[v,r]), \hat{G}[v,r], C \oplus \Delta C$);
9. Insert the set of perfect subgraphs in S_{G_s} into L_k;
10. **return** $L_k[0:k-1]$.

Figure 6: Algorithm dynamicP

together with the capacity bounds, and finally returns the list of top-k perfect subgraphs $L_k(P \oplus \Delta P, G)$.

Example 11: Continue Example 9-(1), after dynamicP updated partial relations for AffBs *w.r.t.* ΔP_3, balls $\hat{G}[\mathsf{PM}_1, 2]$, $\hat{G}[\mathsf{PM}_2, 2]$ and $\hat{G}[\mathsf{BA}_3, 2]$ enter the combination process.
(1) For $\hat{G}[\mathsf{PM}_1, 2]$ and $\hat{G}[\mathsf{PM}_2, 2]$, as $C \oplus \Delta C = \{(\mathsf{PM}, \mathsf{SA})\}$, combine finds that there is an $\mathsf{SA_i}$ (resp. $\mathsf{PM_j}$) connecting to $\mathsf{PM_j}$ (resp. $\mathsf{SA_i}$), and the capacity bounds are satisfied. For inner balls, based on above results, combine finds that no perfect subgraphs reside in $\hat{G}[\mathsf{PM}_1, 1]$ and $\hat{G}[\mathsf{PM}_2, 1]$. Hence it returns the above two perfect subgraphs in two balls.
(2) For $\hat{G}[\mathsf{BA}_3, 2]$, combine finds no $\mathsf{SA_i}$ connecting to $\mathsf{PM_j}$, and vice versa. Hence, no sensible matches are found. □

(IV) Early return optimization technique. We propose an optimization technique for dynamicP to further speed-up the incremental computations, by making use of the top-k semantics. We first define *early return* for incremental top-k algorithms, analogous to *early termination* for batch top-k algorithms [35].

Early return. An algorithm has the *early return property*, if for pattern P with updates ΔP and for any data graph G, it outputs $L_k(P \oplus \Delta P, G)$ as early as possible without the need to update match relations for every AffB, while the updates can be executed in background.

Proposition 9: *There exists an algorithm for the dynamic top-k team formation problem with early return property.* □

We prove dynamicP retains the early return property. Recall the density based filtering optimization for algorithm batch in Section 3. dynamicP also utilizes density upper bounds for pruning a portion of AffBs. More specifically, given P and G, dynamicP maintains the density upper bound for each ball in the *den* item in BS, *i.e.,* $\hat{G}$[den], calculated according to Lemma 3. Thus, given ΔP, if the top-k densest perfect subgraphs found so far are denser than the density upper bound of the remaining AffBs, dynamicP *outputs* the current top-k densest perfect subgraphs as $L_k(P \oplus \Delta P, G)$, while continuing updating $\tilde{M}(P,G)$ in AffBs in *background*.

Note that the early return optimization is effective for pattern updates, but not for data updates and the case when ΔP contains node insertions with new labels (expertise).

(V) The complete algorithm for pattern updates. Given $\tilde{M}(P,G)$, FBM, BF and UP, for pattern update ΔP, algorithm dynamicP computes the top-k perfect subgraphs for $P \oplus \Delta P$ in G with early return property, and maintains auxiliary structures simultaneously by invoking procedure IdABall, IncMatch and combine one by one.

Algorithm dynamicP. It works as follows, as shown in Fig. 6. For each ΔP, dynamicP firstly sets the result list L_k to empty, and identifies AffBs *w.r.t.* ΔP by IdABall (lines 1-2). It then sorts AffBs by their density upper bounds $\hat{G}$[den] in BS in non-ascending order (line 3), and accesses AffBs sequentially by this order (lines 4-10). Whenever it comes to next AffB $\hat{G}[v,r]$, it firstly checks whether there are already k perfect subgraphs found in L_k, and moreover, the density of the kth (smallest) perfect subgraph in L_k is larger than $\hat{G}[v,r]$[den] (line 5). If so, dynamicP immediately outputs L_k as final results (line 6), and then continues to update $\tilde{M}(P,G)$ for those AffBs in background by IncMatch (line 7); Otherwise, dynamicP updates the partial relations and combines them by combine to get the set of perfect subgraphs S_{G_s} in $\hat{G}[v,r]$ and its inner balls (lines 7-8). It then inserts the set of perfect subgraphs in S_{G_s} into L_k (line 9).

Correctness & complexity analysis. The correctness of dynamicP is assured by the correctness of IdABall (Proposition 8), IncMatch, combine, and early return property (Lemma 3). dynamicP is in $O(\bigcup_{\hat{G}\in\mathsf{AffBs}} \bigcup_{i\in[1,h]}(|M(P_{fi}, \hat{G})| + r|M(P_{fi} \oplus \Delta P_{fi}, \hat{G})|) + r|P \oplus \Delta P||\mathsf{AffBs}| + |\Delta P|)$ time *w.r.t.* ΔP, while r is small, *i.e.,* 2 or 3 (See full version [4]).

5.3 Dealing with Data Updates

We next propose dynamicG to handle ΔG, following the framework in Section 4.2. Given auxiliary structures $\tilde{M}(P,G)$ and FBM, we put together procedures IdABall, IncMatch and combine for computing match results for P in $G \oplus \Delta G$. As procedures IncMatch and combine handle ΔG basically the same as ΔP, so we mainly show how IdABall identifies AffBs *w.r.t.* ΔG.

Procedure IdABall. Given G, ΔG, and FBM, IdABall identifies AffBs according to the lemma as follows.

Lemma 10: *A ball $\hat{G}[v,r]$ with center node v in G is identified as an* AffB w.r.t. ΔG *and* FBM,
(1) for some unit data update δ of ΔG, where (a) δ is an edge insertion/deletion, $(w_1,w_2)^+/(w_1,w_2)^-$ and v is in both $\hat{G}[w_1,r]$ and $\hat{G}[w_2,r]$, or (b) δ is a node insertion/deletion, $(w,(w,w'))^+/(w)^-$ and v is in $\hat{G}[w,r]$; or
(2) when $\hat{G}[v,r]$ has type code $(1,\ldots,1)$ in FBM. □

We say a ball which satisfies condition (1) is a *structural affected* ball, *i.e.,* the structure of the ball is changed due to the exertion of some updates in ΔG.

Proposition 11: *Given P, G and ΔG, if there is a perfect subgraph for P in ball $\widehat{G \oplus \Delta G}[v,r]$ of $G \oplus \Delta G$, then $\hat{G}[v,r]$ must be an affected ball produced by procedure* IdABall. □

Different from pattern updates, procedure IncMatch recomputes partial match relations in $\tilde{M}(P,G)$ for each pattern fragment of P in each *structural affected* ball; and no computation is needed for AffBs that only satisfy condition (2) in Lemma 10. Procedure combine combines the partial relations *w.r.t.* ΔG in the same way as handling ΔP.

Updating FBM. Algorithm dynamicG also updates FBM for all AffBs. In addition to updating the links from FS to BS in FBM as for pattern updates, dynamicG maintains BS by (a) removing (resp. inserting new) entries from (resp. to)

BS corresponding to balls whose center nodes are removed from (resp. inserted to) G, due to node deletions (resp. node insertions); and (b) updating the den item in BS *w.r.t.* ΔG. These updates can be done in $O(|\mathsf{AffBs}|)$ time.

Example 12: Consider P_1 and G_1 (both without dashed edges) in Fig. 1, and FBM in Fig. 4(a). When $\Delta G_1 = (\mathsf{SD}_3, \mathsf{ST}_3)^+$ comes, by Lemma 10, IdABall identifies $\hat{G}[\mathsf{SD}_3, 2]$, $\hat{G}[\mathsf{ST}_3, 2]$, $\hat{G}[\mathsf{SA}_3, 2]$ and $\hat{G}[\mathsf{PM}_2, 2]$ as *structural affected* balls, together with balls with tc $(1, 1)$ in FBM as AffBs, *i.e.,* $\hat{G}[\mathsf{PM}_1, 2]$, while filtering out all other balls. □

Algorithm dynamicG**.** Given $\tilde{M}(P, G)$, FBM and data updates ΔG, dynamicG computes the match results for P in $G \oplus \Delta G$, and maintains auxiliary structures by invoking procedures IdABall, IncMatch and combine sequentially.

Correctness & complexity analyses. The correctness of dynamicG *w.r.t.* ΔG follows from Proposition 11 and the correctness of IncMatch and combine. dynamicG is overall in $O(\bigcup_{\widehat{G \oplus \Delta G} \in \mathsf{AffBs}} \bigcup_{i \in [1,h]} r|M(P_{fi}, \widehat{G \oplus \Delta G})| + r|P||\mathsf{AffBs}| + |\Delta G|)$ time *w.r.t.* ΔG, while r is small, *i.e.,* 2 or 3 (See [4]).

5.4 Unifying Pattern and Data Updates

We are now ready to provide algorithm dynamic, integrating dynamicP and dynamicG, presented in Sections 5.2 and 5.3, respectively, to process continuous pattern and data updates, separately and simultaneously.

Algorithm dynamic is able to handle *simultaneous* ΔP and ΔG, because of the consistency in: (1) the processes for handling ΔP and ΔG, which follow the same steps in Section 4.2; (2) auxiliary data structures for supporting ΔP and ΔG; and (3) the combination procedures, which suffice to support simultaneous pattern and data updates.

Observe that dynamic can handle *continuously* simultaneous ΔP and ΔG, as dynamic incrementally maintains the auxiliary structures for continuously coming ΔP and ΔG.

Remark. Note that the running time of dynamicP and dynamicG is determined by $\{P, \Delta P, \tilde{M}(P, G), \mathsf{AffBs}\}$ and $\{P, \Delta G, \tilde{M}(P, G), \mathsf{AffBs}\}$, respectively, not directly depending on G. From this, we complete the proof of Theorem 7.

6. EXPERIMENTAL STUDY

We conducted four sets of experiments to evaluate the performance of (1) batch for the top-k team formation problem, (2) dynamic for the dynamic top-k team formation problem *w.r.t.* single sets of (a) pattern updates, (b) data updates, and (c) simultaneous pattern and data updates, (3) dynamic *w.r.t.* continuous sets of pattern and data updates, and (4) the extra space cost of auxiliary structures used by dynamic.

6.1 Experimental Settings

We used the following real-life and synthetic datasets.

Real-life graphs. We used two real-life graphs.

(1) CITATION [3] contains 1.39M paper nodes and 3.02M paper-paper citation edges. We used its undirected version, where edges indicate the relevance relationship, and generated 200 labels based on phrase clustering of paper titles.

(2) YOUTUBE [5] contains 2.03M video nodes and 12.2M edges, which represent recommendations between two videos. We used the undirected version, and generated 400 labels based on the built-in categories and ages of videos.

Synthetic graph generator. We generated synthetic graphs (SYNTHETIC) with community structures as existed in real-life, by adopting the LFR-benchmark graph model [24]. It is controlled by three parameters: the number n of nodes, the average degree d of nodes, and the number l of node labels.

Pattern generator. We implemented a generator to produce pattern graphs, controlled by 4 parameters: the number of nodes $|V_P|$, the number of edges $|E_P|$, the label l_P for each node from an alphabet of labels in the corresponding data graphs, and the capacity bound f_P for each node.

Algorithms. We implemented the following algorithms, all in C++: (1) algorithm batch for kTF, (2) incremental algorithm dynamic for kDTF, (3) three compared top-k team formation algorithms minDia, minSum and denAlk, where (a) minDia is to minimize the team diameter [25], which is firstly proposed for the team formation problem, (b) minSum is to minimize the sum of all-pair shortest distances of teams [22], and (c) denAlk is to maximize the team density [15], which has the same goal with our algorithms. Most of the algorithms for kTF, including minDia and denAlk, only compute the best team, while minSum is able to find top-k teams in polynomial time, which is an adaption of Lawler's procedure [26]. Based on this, we extend minDia and denAlk to find top-k teams in polynomial time.

We used a PC with Intel Core i5-4570 CPU and 16GB of memory. We randomly generated 3 sets of input and repeated 5 times for each test. The average is reported here.

6.2 Experimental Results

We present our findings. In all the experiments, we set $k = 10$, $r = 2$, $h = 3$, $(|V_P|, |E_P|)$ to be (10,12), and capacity bounds to be [1,10] by default. When generating synthetic graphs, we fixed $n = 10^7$, $d = 10$ and $l = 200$. All the findings on YOUTUBE are reported in the full version [4].

Exp-1: Efficiency of batch. We firstly evaluated the efficiency of batch vs. minDia, minSum and denAlk. We generated pattern graphs for batch, and the corresponding queries (labels requirements) for minDia, minSum and denAlk.

Algorithms minDia, minSum and denAlk do not scale well on large graphs. Indeed, (1) minDia and minSum took more than 8 hours to finish their preprocessing, *i.e.,* computing all-pair-shortest-paths; and (2) denAlk took more than 24 hours even when $k = 1$ on CITATION. By contrast, batch took around 100 seconds on CITATION by default settings. Hence, we report the effectiveness of these algorithms on a sampled data graph with 10,000 nodes on CITATION only.

Exp-2: Effectiveness of batch. We then evaluated the efficiency of batch vs. minDia, minSum and denAlk by checking the quality of matches returned by them.

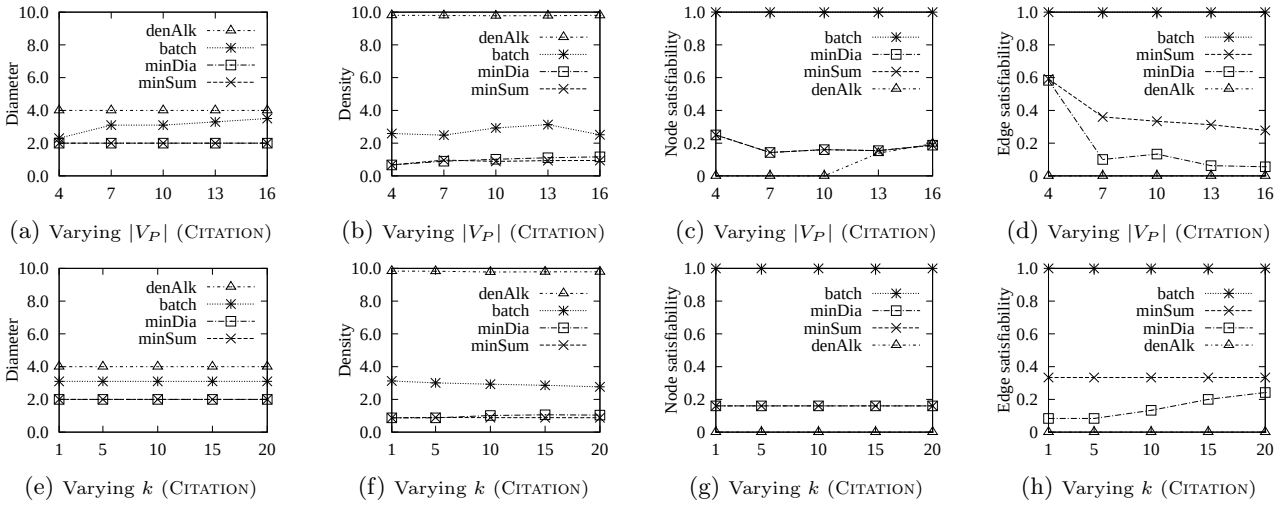
To evaluate the quality of teams found by the above four algorithms for kTF, we defined four quality measures. Consider a matched subgraph G_S and pattern $P(V_P, E_P)$.

(a) [*Diameter*]: the diameter of G_S.

(b) [*Density*]: the density of G_S.

(c) [*Node satisfiability*]: $\eta_V(G_S, P) = \#\mathsf{sat}_V(G_S, P)/|V_P|$, where $\#\mathsf{sat}_V(G_S, P)$ is the number of nodes in P that are satisfied by G_S, in which we say a pattern node u is satisfied by G_S if there are a set V_u of nodes in G_S that match u and moreover, V_u satisfies the capacity constraints on u.

(d) [*Edge satisfiability*]: $\eta_E(G_S, P) = \#\mathsf{sat}_E(G_S, P)/|E_P|$, where $\#\mathsf{sat}_E(G_S, P)$ is the number of edges in P satisfied by G_S, in which we say an edge (u_1, u_2) is satisfied by G_S if for each v_1 in G_S that matches u_1, there exists (v_1, v_1') in G_S



(a) Varying $|V_P|$ (CITATION) (b) Varying $|V_P|$ (CITATION) (c) Varying $|V_P|$ (CITATION) (d) Varying $|V_P|$ (CITATION)

(e) Varying k (CITATION) (f) Varying k (CITATION) (g) Varying k (CITATION) (h) Varying k (CITATION)

Figure 7: Performance evaluation of algorithm batch for top-k team formation

so that v_1' matches u_2, and for each v_2 in G_S that matches u_2, there exists (v_2, v_2') in G_S such that v_2' matches u_1.

Note that (a) and (b) are two traditional quality measures utilized by existing team formation algorithms [15,22,25,34]. Intuitively, $\eta_V(G_S, P)$ (resp. $\eta_E(G_S, P)$) measures how well G_S meets the node capacity requirements (resp. structural constraints) in P, and their values fall in $[0, 1]$.

(i) Impacts of $|V_P|$. Varying the number $|V_P|$ of nodes in P from 4 to 16, we took the average value of top-k teams found by batch, minDia, minSum and denAlk *w.r.t.* four quality measures. The results are reported in Figures 7(a)-7(d).

Observe the following. (1) The diameters of teams found by batch are comparable to those of minDia and minSum, which are in particularly designed to minimize the diameters, as shown in Fig. 7(a). This is ensured by the use of balls in batch. (2) The densities of teams found by batch, though are smaller than denAlk which is specialized for maximizing team densities, are larger than minDia and minSum, as in Fig. 7(b). (3) The node satisfiability of teams found by batch is much higher than minDia, minSum and denAlk, *e.g.,* 1.0 vs. no larger than 0.2 in all cases as in Fig. 7(c). (4) The teams found by batch come with a higher edge satisfiability , *e.g.,* 1.0 in all cases, compared to smaller than 0.6 by minDia, minSum and denAlk, as shown in Fig. 7(d).

(ii) Impacts of k. Varying k from 1 to 20, we report the results in Figures 7(e)-7(h). Observe that the quality of teams found by the four algorithms shows the same rule as varying $|V_P|$, and the quality is not sensitive to k, a desirable property when top-k semantics is concerned.

These verify that batch can effectively preserve structural and capacity constraints for top-k team formation *w.r.t.* edge and node satisfiability, and pertains a good team collaboration compatibility *w.r.t.* diameter and density.

Exp-3: Efficiency of dynamic for single set of updates. We evaluated the efficiency of algorithm dynamic for processing one set of pattern updates, data updates and simultaneous pattern and data updates vs. algorithm batch on CITATION and SYNTHETIC, respectively.

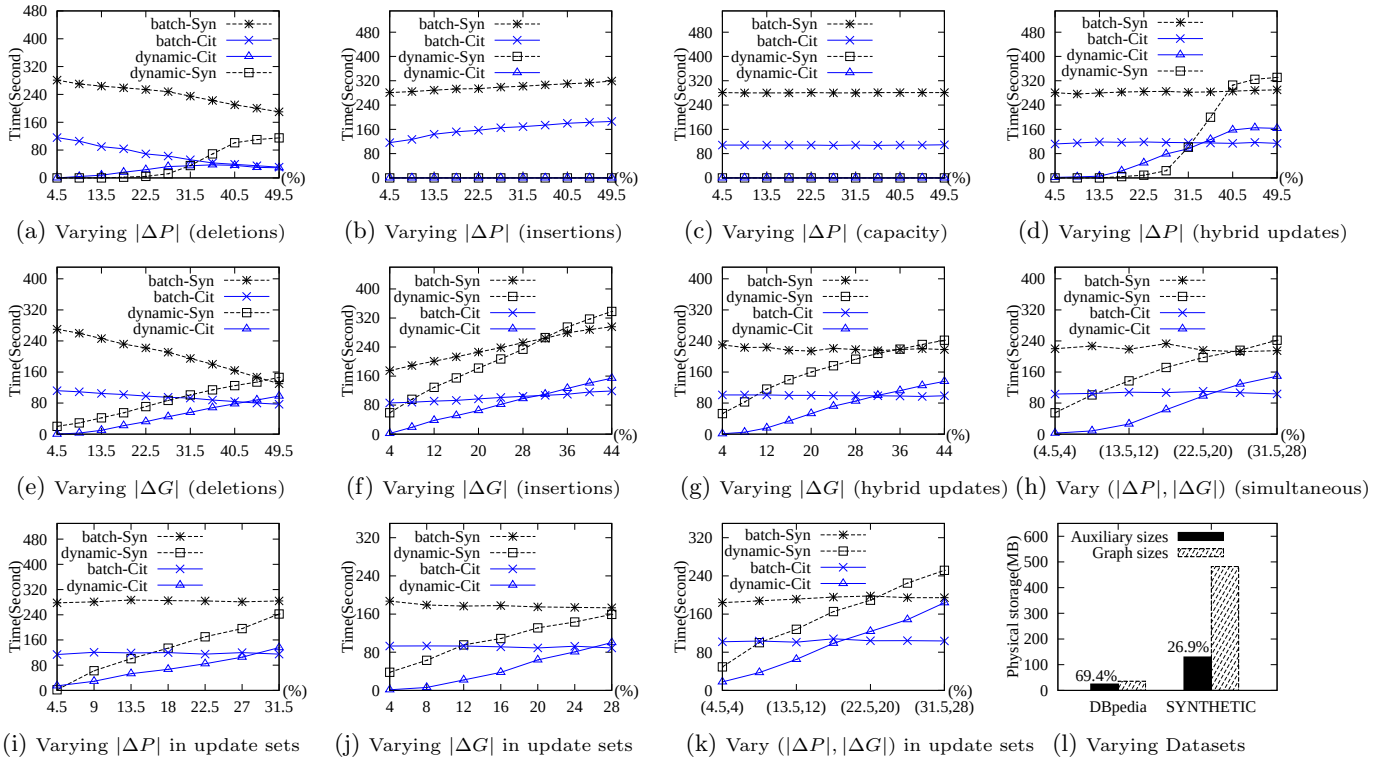
(i) Pattern updates. We fixed ($|V_P|$, $|E_P|$) to be $(10, 12)$, and varied the number $|\Delta P|$ of unit updates from 1 to 11, corresponding to 4.5% to 49.5% in Figs. 8(a), 8(b), 8(c) and 8(d), which show the results when ΔP contains (edge and node) deletions, (edge and node) insertions, capacity changes and hybrid pattern updates (5 types) respectively, while keeping the proportion for each type equal.

We find the following. (1) dynamic outperforms batch even when deletions are no more than 40.5% on CITATION and 49.5% on SYNTHETIC; dynamic consistently does better than batch due to the early-return strategy. (2) dynamic improves batch to a large extent when only processes insertions and capacity changes. (3) For the same $|\Delta P|$, dynamic needs less time to process insertions than deletions. (4) When processes hybrid pattern updates, dynamic outperforms batch when changes are no more than (31.5%, 40.5%) on (CITATION, SYNTHETIC); It is because all balls are identified as AffBs when pattern updates accumulate to a certain extent.

(ii) Data updates. For (edge and node) deletions (resp. insertions) on datasets, *e.g.,* CITATION with $|G| = 4.4M$, we varied $|G|$ from $4.4M$ to $2.22M$ (resp. from $3.05M$ to $4.4M$) in 4.5% decrements (resp. 4% increments) by randomly picking a subset of nodes and edges and removing from G (resp. inserting into G); For hybrid data updates (4 types), we randomly sampled a subgraph G_s and removed from G, obtaining the initial G. We varied $|G|$ by firstly removing a subset of nodes and edges from G and then inserting a subset of nodes and edges from G_s into G, in total 4% updates. The results are shown in Figures 8(e), 8(f) and 8(g).

We find the following. (1) dynamic outperforms batch when insertions are no more than 28% and 32% on CITATION and SYNTHETIC (resp. 40.5% and 45% for deletions). (2) For the same $|\Delta G|$, dynamic needs less time to process deletions than insertions. (3) We have conducted a survey: the user increment on Facebook [1] and Twitter [2] daily reaches 1.23‰ and 2.47‰. Therefore, dynamic is able to handle the increments accumulated in dozens of days on Facebook and Twitter at a high efficiency. (4) dynamic outperforms batch when hybrid data updates are no more than 32% and 36% on CITATION and SYNTHETIC, respectively.

(iii) Simultaneous pattern and data updates. Varying the number of hybrid pattern updates from 1 to 7 and the amount of hybrid data updates from 4% to 28% together, corresponding to (4.5%, 4%) to (31.5%, 28%) for $(\Delta P, \Delta G)$ in Fig. 8(h). We find that dynamic outperforms batch when $(\Delta P, \Delta G)$ is no more than (22.5%, 20%) and (27%, 24%) on



(a) Varying $|\Delta P|$ (deletions) (b) Varying $|\Delta P|$ (insertions) (c) Varying $|\Delta P|$ (capacity) (d) Varying $|\Delta P|$ (hybrid updates)

(e) Varying $|\Delta G|$ (deletions) (f) Varying $|\Delta G|$ (insertions) (g) Varying $|\Delta G|$ (hybrid updates) (h) Vary $(|\Delta P|, |\Delta G|)$ (simultaneous)

(i) Varying $|\Delta P|$ in update sets (j) Varying $|\Delta G|$ in update sets (k) Vary $(|\Delta P|, |\Delta G|)$ in update sets (l) Varying Datasets

Figure 8: Performance evaluation of algorithm dynamic for dynamic top-k team formation (Cit: CITATION, Syn: SYNTHETIC)

CITATION and SYNTHETIC, respectively.

Exp-4: Efficiency of dynamic for continuous sets of updates. We evaluated dynamic for a serial sets of pattern updates, data updates and simultaneous pattern and data updates vs. batch on CITATION and SYNTHETIC.

(i) Pattern updates. We generated 5 sets of hybrid pattern updates, varying the number of updates in each set from 1 to 7. We tested the average time took by dynamic to finish all these sets one by one. The results are reported in Fig. 8(i).

Recall that dynamic adopts a lazy update policy, which definitely affects the processing time of next updates. However, we find dynamic outperforms batch *w.r.t.* average time, when the amount of hybrid pattern updates in each set is no more than (27%, 31.5%) on (CITATION, SYNTHETIC). This verifies the effectiveness of our lazy update policy.

(ii) Data updates. The setting is same as above. Varying the amount of hybrid data updates in each set from 4% to 28%, the results are reported in Fig. 8(j). We find that dynamic outperforms batch when hybrid data updates are no more than (24%, 28%) on (CITATION, SYNTHETIC).

(iii) Simultaneous pattern and data updates. Using the same setting and varying the simultaneous pattern and data updates $(\Delta P, \Delta G)$ from (4.5%, 4%) to (31.5%, 28%) in Fig. 8(k), We find that dynamic outperforms batch when updates in each set are no more than (18%, 16%) and (22.5%, 20%) on CITATION and SYNTHETIC, respectively.

Exp-5: Physical storage of auxiliary structures. As shown in Fig. 8(l), the incremental algorithm dynamic takes (25MB, 130MB) extra space to store all its auxiliary structures on (CITATION, SYNTHETIC), while they need (36MB, 482MB) space to store themselves. That is, the auxiliary structures are light-weight, and only take (69.4%, 26.9%) extra space compared with the original datasets.

Summary. From these tests, we find the following.
(1) Our graph pattern matching approach is effective at capturing the practical requirements of top-k team formation.
(2) Our batch algorithm for top-k team formation is efficient, *e.g.,* it only took 116s when $|V| = 1.39M$ and $|V_P| = 10$.
(3) Our incremental algorithm for dynamic top-k team formation is able to process continuous pattern and data updates, separately and simultaneously, and it is more promising than its batch counterpart, even (a) when changes are 36% for pattern updates, 34% for data updates, and (25%, 22%) for simultaneous pattern and data updates on average, and (b) when 29% for continuous pattern updates, 26% for continuous data updates and (20%, 18%) for continuously simultaneous pattern and data updates on average.

7. CONCLUSION

We have introduced a graph pattern matching approach for (dynamic) top-k team formation problem. We have proposed team simulation, based on which we have developed a batch algorithm for top-k team formation. We have also developed a unified incremental algorithm to handle continuous pattern and data updates, separately and simultaneously. We have experimentally verified the effectiveness and efficiency of the batch and incremental algorithms.

A couple of topics are targeted for future work. First, an interesting topic is to develop distributed algorithms for top-k team formation. Second, the study of dynamic algorithms for query updates is in its infancy, and hence, an important topic is to develop such algorithms for various problems.

8. REFERENCES

[1] http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.

[2] http://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/.

[3] Citation. https://aminer.org/billboard/citation/.

[4] Full version. https://lijia6.github.io/full.pdf.

[5] YouTube. https://http://netsg.cs.sfu.ca/youtubedata/.

[6] C. C. Aggarwal and H. Wang. *Managing and Mining Graph Data*. Springer, 2010.

[7] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Online team formation in social networks. In *WWW*, 2012.

[8] K. Andreev and H. Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[9] S. Datta, A. Majumder, and K. Naidu. Capacitated team formation problem on social networks. In *KDD*, 2012.

[10] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, pages 155–169, 2017.

[11] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.

[12] W. Fan, X. Wang, and Y. Wu. Expfinder: Finding experts by graph pattern matching. In *ICDE*, 2013.

[13] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3):18:1–18:47, 2013.

[14] W. Fan, Y. Wu, and J. Xu. Adding counting quantifiers to graph patterns. In *SIGMOD*, 2016.

[15] A. Gajewar and A. D. Sarma. Multi-skill collaborative teams based on densest subgraphs. In *SDM*, 2012.

[16] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.

[17] A. V. Goldberg. Finding a maximum density subgraph. In *Technical Report CSD-84-171*, 1984.

[18] M. Habibi and A. Popescu-Belis. Query refinement using conversational context: A method and an evaluation resource. In *NLDB*, 2015.

[19] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[20] 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*, 2016.

[21] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

[22] M. Kargar and A. An. Discovering top-k teams of experts with/without a leader in social networks. In *CIKM*, 2011.

[23] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[24] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4), 2008.

[25] T. Lappas, K. L. Sarma, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[26] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401C–405, 1972.

[27] L. Li, H. Tong, N. Cao, K. Ehrlich, Y.-R. Lin, and N. Buchler. Replacing the irreplaceable: Fast algorithms for team member recommendation. In *WWW*, 2015.

[28] G. Liu, K. Zheng, Y. Wang, M. A. Orgun, A. Liu, L. Zhao, and X. Zhou. Multi-constrained graph pattern matching in large-scale contextual social graphs. In *ICDE*, 2015.

[29] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, 2014.

[30] D. Mottin, F. Bonchi, and F. Gullo. Graph query reformulation with diversity. In *KDD*, 2015.

[31] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. *PVLDB*, 6(14):1762–1773, 2013.

[32] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[33] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2):233–277, 1996.

[34] S. Rangapuram, T. Bühler, and M. Hein. Towards realistic team formation in social networks based on densest subgraphs. In *WWW*, 2013.

[35] F. Ronald, L. Amnon, and N. Moni. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[36] H. Sajjad, P. Pantel, and M. Gamon. Underspecified query refinement via natural language question generation. In *COLING*, 2012.

[37] L. G. Terveen and D. W. McDonald. Social matching: A framework and research agenda. *ACM Trans. Comput.-Hum. Interact.*, 12(3):401–434, 2005.

[38] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[39] E. Valari, M. Kontaki, and A. N. Papadopoulos. Discovery of top-k dense subgraphs in dynamic graph collections. In *SSDBM*, 2012.

[40] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. In *ICDE*, 2012.

[41] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.

Appendix A: Detailed Algorithms

1. Procedure undirgSim (Section 3)

Input: Pattern graph $P(V_P, E_P)$ and ball $\hat{G}[v, r]$.
Output: The match graph G_s for P in $\hat{G}[v, r]$.

1. **for** each $u \in V_P$ **do**
2. $\mathsf{R}(u) := \{w | w$ is in $\hat{G}[v, r]$ and $l_P(u) \in l(w)\}$;
3. **while** there are changes **do**
4. **for** each edge (u, u') in E_P and each node $w \in \mathsf{R}(u)$ **do**
5. **if** there is no edge (w, w') in $\hat{G}[v, r]$ with $w' \in \mathsf{R}(u')$ **then**
6. $\mathsf{R}(u) := \mathsf{R}(u) \setminus \{w\}$;
7. **if** $\mathsf{R}(u) = \emptyset$ **then return** $\emptyset$;
8. $M := \{(u, w) | u \in V_P, w \in \mathsf{R}(u)\}$;
9. Construct the match graph G_s *w.r.t.* M;
10. **return** G_s;

Figure 9: Procedure undirgSim

As shown in Fig 9, given P and ball $\hat{G}[v, r]$, undirgSim finds the match graph for P in $\hat{G}[v, r]$. For each node u in V_P, it first computes the set $\mathsf{R}(u)$ of candidate matches w containing the label of u (lines 1-2), and then removes nodes from $\mathsf{R}(u)$ iteratively (lines 3-7). A node w is removed from $\mathsf{R}(u)$ if there is an adjacent node u' of u, but there exists no adjacent node w' of w such that $w' \in \mathsf{R}(u')$. If so, undirgSim constructs the match graph G_s *w.r.t.* the maximum match relation M and returns it (lines 8-10).

2. Algorithm PFrag (Section 4)

PFrag works by connecting the *pattern fragmentation* problem to the (k, ν)-Balanced Partition problem. It is to divide the nodes of a graph into k components such that each component is of size no more than $\nu \cdot \frac{|V|}{k}$, and the number of edges between different components is minimized. As illustrated before, the (k, ν)-balanced partition problem, though is not approximable in general, has a number of sophisticated heuristic algorithms [8].

Input: Pattern graph $P(V_P, E_P)$ and integer h.
Output: An h-fragmentation $\{P_{f1}, \ldots, P_{fh}, C\}$ of P.

1. $k := h$; $\nu := h$; $M_P := |P|$; $M_C := 0$;
2. $(P^\nu_{f1}, \ldots, P^\nu_{fh}, C^\nu) := \mathsf{BalanceP}(k, \nu)$;
3. $M^\nu_P := \mathsf{max}\{|P^\nu_{f1}|, \ldots, |P^\nu_{fh}|\}$; $M^\nu_C := |C^\nu|$;
4. **while** $\mathsf{max}\{M_P, M_C\} > \mathsf{max}\{M^\nu_P, M^\nu_C\}$ **do**
5. $P_{f1} := P^\nu_{f1}, \ldots, P_{fh} := P^\nu_{fh}$;
6. $C := C^\nu$; $M_P := M^\nu_P$; $M_C := M^\nu_C$;
7. **if** $M^\nu_P \geq M^\nu_C$ **then** $\nu := \frac{\nu}{2}$ **else** $\nu := \frac{3\nu}{2}$;
8. $(P^\nu_{f1}, \ldots, P^\nu_{fh}, C) := \mathsf{BalanceP}(k, \nu)$;
9. $M^\nu_P := \mathsf{max}\{|P^\nu_{f1}|, \ldots, |P^\nu_{fh}|\}$; $M^\nu_C := |C^\nu|$;
10. **return** $\{P_{f1}, \ldots, P_{fh}, C\}$;

Figure 10: Algorithm PFrag

As shown in Fig. 10, given P and integer h, PFrag finds an h-fragmentation for P by recursively invoking algorithm $\mathsf{BalanceP}(k, \nu)$ for the (k, ν)-Balanced partition problem ($\nu \geq 1$) with different ν, such that the final returned h-fragmentation strikes a balance between the size of each fragment P_{fi} and the size of the cut C. More specifically, PFrag maintains M_P and M_C as the size of the largest fragment P_{fi} and the size of the cut respectively. Initially, it sets M_P to $|P|$ and M_C to 0 (line 1). It then invokes BalanceP with both k and ν being h, *i.e.,* has no constraints on the size of fragments of P (line 2). After that, it iteratively checks whether the generated h-fragmentation can be improved (line 4) by adjusting ν in a *binary search style* (lines 4-9). If the current size M^ν_P of the largest fragment is no smaller than the size M^ν_C of the cut, it invokes BalanceP with h and $\frac{\nu}{2}$, or with h and $\frac{3\nu}{2}$ otherwise (line 7). It returns the h-fragmentation if it cannot be improved anymore (line 10).

Correctness & Complexity. The correctness is obvious as PFrag always returns an h-fragmentation. Algorithm PFrag runs in $O(\log h \cdot t_{\mathsf{BalanceP}})$, where t_{BalanceP} is the complexity of the algorithm for the (k, ν)-Balanced Partition problem. Indeed, PFrag calls at most $\log h$ times BalanceP, as $\nu \geq 1$.

Appendix B: Detailed Proofs

1. Proof of Proposition 1

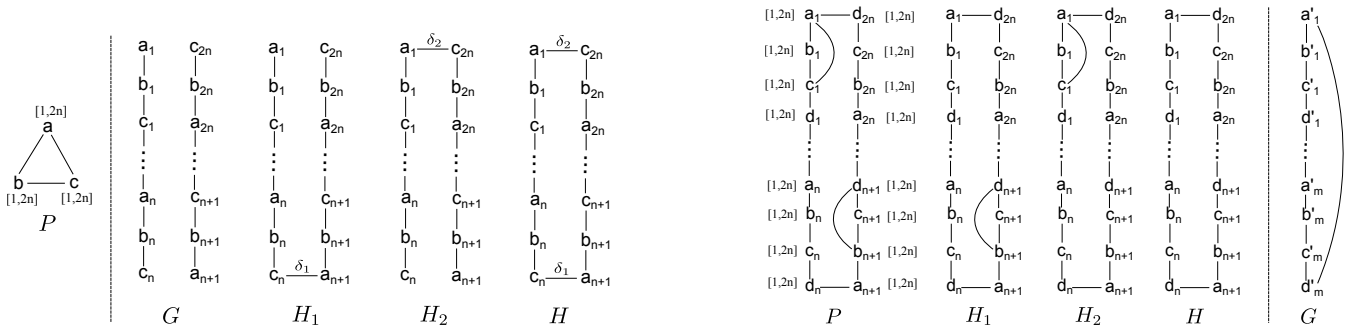
We will prove the proposition by providing the $O(|P|^2)$ time algorithm, along with its correctness proof.

The satisfiability of patterns P can be checked by the following algorithm: (a) Compute the maximum match relation M of $P \prec P$; (b) Check for each $(u, v) \in M$ with the capacity bounds $[x_u, y_u]$ on u and $[x_v, y_v]$ on v, respectively, whether $x_v \leq y_u$ holds. One can verify that step (a) is in $O(|P|^2)$ time by invoking procedure undirgSim; and step (b) can be checked in $O(|P|^2)$ time as the size of M is bounded by $|P|^2$. Thus the algorithm is overall in $O(|P|^2)$ time.

We next prove the correctness of the algorithm.

(I) We firstly prove that when the algorithm outputs "yes", then pattern P is indeed satisfiable. This is correct since if the algorithm outputs "yes", *i.e.,* $P \prec P$ *w.r.t.* M and capacity bounds in M are satisfied, there must exist a data graph G such that $P \lhd_r G$. G can be derived by (i) computing equivalent classes of nodes in P based on M, such that u and w are in the same class iff both $(u, w) \in M$ and $(w, u) \in M$; (ii) creating a set of nodes for each equivalent class, where the cardinality of the set falls in the intersection of capacity bounds on nodes in the equivalent class; and (iii) connecting the nodes belong to two equivalent classes iff there exist edges in P connecting two pattern nodes which belong to the same two equivalent classes. We also set r to be the radius of G. One can verify that $P \lhd_r G$.

(II) We then prove that when the algorithm outputs "no", then pattern P is unsatisfiable. If the algorithm outputs "no", then either (i) $P \not\prec P$ or (ii) $P \prec P$ but capacity bounds are not satisfied. We will prove this by contradiction. We consider case (i) $P \not\prec P$ and assume that P is satisfiable. For convenience, we use $P_1 \not\prec P_2$ to denote $P \not\prec P$, while P_1, P_2 and P are exactly same. From that P is satisfiable, we know that there exists a data graph G such that $P_1 \lhd_r G$, $P_2 \lhd_r G$ and $P_1 \lhd_r G_s$, where G_s is the perfect subgraph in G. By the definition of team simulation and graph simulation, from $P_2 \lhd_r G$, we know that $G_s \prec P_2$; from $P_1 \lhd_r G_s$, we know that $P_1 \prec G_s$; and from $P_1 \prec G_s$ and $G_s \prec P_2$, we know that $P_1 \prec P_2$. This contradicts the assumption. We then consider case (ii), $P \prec P$ *w.r.t.* M while there exists $(u, v) \in M$ with the capacity bounds $[x_u, y_u]$ on u and $[x_v, y_v]$ on v, respectively, and $x_v > y_u$, and assume that P is satisfiable. From P is satisfiable, we know that there exists a data graph G such that $P \lhd_r G$. From $P \prec P$ *w.r.t.* M and $(u, v) \in M$, we know that for any nodes w in G, if w matches the pattern node v, then it must match the pattern node u, that is, the number of data nodes match u is larger than that of data nodes match v. However, this contradicts that the upper bound of u (y_u) is smaller than the lower bound of v (x_v).



(a) Unit data update (b) Unit pattern update

Figure 11: Unboundedness of unit data and pattern update

2. Proof of Theorem 2

We will prove Theorem 2 (1) firstly. We know that $P \prec \hat{G}[v,t]$, and suppose that M_t is the maximum match relation for P in $\hat{G}[v,t]$. Since $\hat{G}[v,t]$ is a subgraph of $\hat{G}[v,r]$, there must exist a binary relation M_r such that for any $(u,v) \in M_t$, where u is a pattern node and v is the matched data node of u in $\hat{G}[v,t]$, $(u,v) \in M_r$ holds, where v is the matched data node of u in $\hat{G}[v,r]$, *i.e.,* $M_t \subset M_r$. Since M_t is the maximum match relation for P in $\hat{G}[v,t]$ and $M_t \subset M_r$, by the definition of graph simulation, then $P \prec \hat{G}[v,r]$ and M_r is a match relation for P in $\hat{G}[v,r]$.

We will prove Theorem (2) using the lemma below.

Lemma 12: *For any data graphs G_1 and G_2, and pattern graph P, if G_1 is a subgraph of G_2, then $M_1 \subset M_2$, where M_i $(i = 1, 2)$ is the maximum match relation for P in G_i via graph simulation.* □

We next prove the lemma and then use the lemma to prove Theorem 2 (2).

Proof of Lemma 12: We will prove the lemma by contradiction. We know that G_1 is a subgraph of G_2, and suppose $M_1 \not\subset M_2$. That is, there exists a pair of nodes $(u,v) \in M_1$, where u is a pattern node and v is the matched data node of u, but $(u,v) \notin M_2$. By the definition of graph simulation, from $(u,v) \in M_1$, we know that for any child node u' of u in P, there exists a child node v' of v in G_1 such that $(u',v') \in M_1$. From $(u,v) \notin M_2$, we know that there exists a child node u'' of u in P, but there exist no child nodes v'' of v in G_2, such that $(u'',v'') \prec M_2$. The process to compute maximum match relations for graph simulation is a recursive process to remove unmatched nodes from the initialized match relations M. Because there exist no u'' in G_2 such that $(u'',v'') \prec M_2$, (u,v) is removed from M_2. Since G_1 is a subgraph of G_2, (u,v) should also be removed from M_1, such that $(u,v) \notin M_1$. This contradicts the assumption that $(u,v) \in M_1$, and we get the lemma proved. □

By Lemma 12, since $\hat{G}[v,t]$ is a subgraph of $\hat{G}[v,r]$, then $M' \subset M$. By the definition of match graphs, since $M' \subset M$, then G'_s is a subgraph of G_s.

3. Proof of Proposition 4

Incremental complexity is defined in terms of LP (locally persistent) graph algorithms [34]. We also adopt the notion to prove unboundedness of graph algorithms for kDTF.

The proofs below strictly follows the one in [34].

*(I) k*DTF *is unbounded for unit data update.* Consider the following pattern and data graph, as shown in Fig. 11(a), and unit data updates. Let data graph G consist of two chains (a_1, b_1, c_1, …, a_n, b_n, c_n) and (a_{n+1}, b_{n+1}, c_{n+1}, …, a_{2n}, b_{2n}, c_{2n}) where a_i, b_i and c_i have labels A, B and C respectively. Let pattern graph be a triangle with nodes a, b and c with labels A, B and C respectively. Consider two unit edge insertions $\delta_1 = (c_n, a_{n+1})^+$ and $\delta_2 = (c_{2n}, a_1)^+$, and set $k = 1$ and $r = 6n$. Let H_1 and H_2 denote the graphs $G \oplus \delta_1$ and $G \oplus \delta_2$, respectively. Obviously, $L_k(P,G) = L_k(P,H_1) = L_k(P,H_2) = \emptyset$, while $L_k(P, H_1 \oplus \delta_2) \neq \emptyset$. Assume that there exists a locally persistent incremental algorithm $\mathcal{A}$ for kDTF. Let $Trace(G', \delta')$ denote the sequence of steps executed by $\mathcal{A}$ in processing some update δ' to some graph G'. Now consider the following two instances: the application of update δ_2 to G and the application of update δ_2 to graph H_1. Obviously, the update process must behave differently in these two cases, and $Trace(G, \delta_2)$ must be different from $Trace(H_1, \delta_2)$ (because many nodes of $H_1 \oplus \delta_2$ are affected, while no node in $G \oplus \delta_2$ is affected). Since a locally persistent algorithm makes no use of global storage, this can happen only both $Trace(G, \delta_2)$ and $Trace(H_1, \delta_2)$ include a visit to some node w that contains different information in the graphs G and H_1. However, H_1 was obtained from G by applying update δ_1. Hence the information at node w must have been changed during the updating of applying δ_1 to G. Therefore, $Trace(G, \delta_1)$ must also contain a visit to node w. As a characteristic of locally persistent algorithms is that if a node w is visited during the updating of applying change δ' to graph G', then every node on some path in G' from a modified node of δ' to w must have been visited. Therefore, $Trace(G, \delta_1)$ and $Trace(G, \delta_2)$ both contain a visit to w, from the nodes in δ_1 and δ_2, respectively. Thus, $Trace(G, \delta_1)$ and $Trace(G, \delta_2)$ include visits to every node on the path from c_n or a_{n+1} to c_{2n} or a_1 respectively. Hence, the time taken for processing update δ_1 to G plus the time taken for processing update δ_2 to G must be no smaller than the distance between c_n or a_{n+1} to c_{2n} or a_1, *i.e., n*, which is not a constant. However, $|\mathsf{AFF}|$ in both cases are 1, such that the complexity of the incremental algorithm $\mathcal{A}$ cannot be measured by a function of $|\mathsf{AFF}|$. Thus, $\mathcal{A}$ is not a bounded locally persistent incremental algorithm.

That is, kDTF is unbounded even for $k = 1$ and unit data update.

*(II) k*DTF *is unbounded for unit pattern update.* Consider the following pattern and data graph, as shown in Fig. 11(b), and unit pattern updates. Let data graph G be a cycle (a'_1, b'_1, c'_1, d'_1, …, a'_m, b'_m, c'_m, d'_m, a'_1), where a'_i, b'_i, c'_i

and d'_i have labels A, B, C and D respectively. Let pattern graph be a cycle (a_1, b_1, c_1, d_1, ..., a_n, b_n, c_n, d_n, a_{n+1}, b_{n+1}, c_{n+1}, d_{n+1}, ..., a_{2n}, b_{2n}, c_{2n}, d_{2n}, a_1) and two extra edges (a_1, c_1) and (b_{n+1}, d_{n+1}), where a_i, b_i, c_i and d_i have labels A, B, C and D respectively. Consider two unit edge deletions $\delta_1 = (a_1, c_1)^-$ and $\delta_2 = (b_{n+1}, d_{n+1})^-$, and set $k = 1$ and $r = 4m$. Let H_1 and H_2 denote the graphs $P \oplus \delta_1$ and $P \oplus \delta_2$, respectively. Obviously, $L_k(P, G) = L_k(H_1, G) = L_k(H_2, G) = \emptyset$, while $L_k(H_1 \oplus \delta_2, G) \neq \emptyset$. Assume there exists a locally persistent incremental algorithm $\mathcal{A}$ for kDTF. Let $Trace(P', \delta')$ denote the sequence of steps executed by $\mathcal{A}$ in processing some update δ' to some pattern P'. Now consider two instances: the application of update δ_2 to P and the application of update δ_2 to H_1. Obviously, the update process must behave differently in these two cases, and $Trace(P, \delta_2)$ must be different from $Trace(H_1, \delta_2)$ (because many nodes in G for $H_1 \oplus \delta_2$ are affected, while no node in G for $P \oplus \delta_2$ is affected). Since a locally persistent algorithm makes no use of global storage, this can happen only both $Trace(P, \delta_2)$ and $Trace(H_1, \delta_2)$ include a visit to some node w that contains different information in P and H_1. However, H_1 was obtained from P by applying δ_1. Hence the information at node w must have been changed during the updating of applying δ_1 to P. Therefore, $Trace(P, \delta_1)$ must also contain a visit to node w. According to the characteristics of locally persistent algorithms as illustrated in the data updates case, $Trace(P, \delta_1)$ and $Trace(P, \delta_2)$ both contain a visit to w, from the nodes in δ_1 and δ_2, respectively. Thus, $Trace(P, \delta_1)$ and $Trace(P, \delta_2)$ include visits to every node on the path from a_1 or c_1 to b_{n+1} or d_{n+1} respectively. Hence, the time taken for processing δ_1 to P plus the time taken for processing δ_2 to P must be no smaller than the distance between a_1 or c_1 to b_{n+1} or d_{n+1}, *i.e.,* $4n$, which is not a constant. However, $|\mathsf{AFF}|$ in both cases are 1, such that the complexity of algorithm $\mathcal{A}$ cannot be measured by a function of $|\mathsf{AFF}|$. Thus, $\mathcal{A}$ is not a bounded locally persistent incremental algorithm. That is, kDTF is unbounded even for $k = 1$ and unit pattern update.

(1) and (2) together prove that kDTF is unbounded even for $k = 1$ and unit pattern or data update.

4. Proof of Theorem 5

We will prove the theorem by induction. Given pattern $P(V_P, E_P)$ and its fragmentation $\{P_{f1}, \ldots, P_{fh}, C\}$, we use $P^C(V_{PC}, E_{PC})$ to denote the pattern with V_{PC}=V_P and E_{PC}=E_P/C. Graph simulation is an iterative process to remove unmatched nodes from the candidate nodes, as illustrated in procedure undirgSim in Fig. 9. We utilize M_C^k (resp. M_i^k, M^k) to denote the match relation for P^C (resp. P_{fi}, P) in $\hat{G}$ in the kth iteration. By the definition of graph simulation, we have $M_C^k = \bigcup_{i=1}^h M_i^k$. To prove $M \subseteq \bigcup_{i=1}^h M_i$, we next prove $M^k \subseteq M_C^k$ for each iteration instead.

(1) For $k = 0$, *i.e.,* the initialization step of graph simulation algorithm, the algorithm computes the set of candidate matches for each pattern node. As P and P^C have exactly the same node set, we have $M^0 = M_C^0$.

(2) For $k = n$ ($n \geq 0$), if $M^n \subseteq M_C^n$ holds, we prove $M^{n+1} \subseteq M_C^{n+1}$ holds in the $(n+1)th$ iteration. Suppose both $(u, w) \in M^n$ and $(u, w) \in M_C^n$ hold, and in the $(n+1)th$ iteration, (u, w) is removed from M_C^n if there is an adjacent node u' of u in P_C, but there exists no adjacent node w' of w in G such that $(u', w') \in M_C^n$. Therefore, (u, w) must be removed from M^n as $E_{PC} \subseteq E_P$, that is, the edge (u, u') in E_{PC} must belong to E_P. Thus, we have $M^{n+1} \subseteq M_C^{n+1}$.

By (1) and (2), we have proven that $M \subseteq \bigcup_{i=1}^h M_i$.

5. Proof of Proposition 6

The decision version of the *pattern fragmentation* problem, denoted by $\mathsf{dOFGP}(P, h, r_1, r_2)$, is to decide whether there exists a fragmentation $\{P_{f1}, \ldots, P_{fh}, C\}$ such that, (a) $\max_i |P_{fi}| \leq r_1 \frac{|P|}{h}$ and (b) $|C| \leq r_2|P|$.

Upper bound. We show the NP upper bound by providing an NP algorithm to determine whether there exists an h-fragmentation of P. Given P, the algorithm works as follows.
(1) Guess an h-fragmentation $\mathcal{P}_h$ of P.
(2) Check whether it satisfies restrictions of r_1 and r_2 (conditions (a) and (b) in the definition of dOFGP). If so, return yes, otherwise go to the first step and guess another instance.

The algorithm is in NP since step (2) can be checked in PTIME (liner time, indeed).

Lower bound. We next show that $\mathsf{dOFGP}(P, h, r_1, r_2)$ is NP-hard by reduction from the 3SAT problem. An instance of 3SAT is a formula $\psi = C_1 \wedge \ldots \wedge C_m$, where each C_i is a disjunction of three literals, *i.e.,* $C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i (i \in [1, m])$, in which $\ell_j^i (j \in [1, 3])$ is either a variable x_k or the negation of variable $\bar{x}_k (k \in [1, n])$ from a universal set $U = \{x_1, \ldots, x_n\}$ of variables. Given a ψ, 3SAT is to decide whether ψ is satisfiable, *i.e.,* there exists a truth assignment μ to variables in U such that ψ is *true* under μ. It is known that 3SAT is NP-complete [32].

Given ψ of 3SAT, we next construct an instance of dOFGP, *i.e.,* a pattern graph P, the number of fragments h, and two ratios r_1 and r_2, such that ψ is satisfiable if and only if $\mathsf{dOFGP}(P, h, r_1, r_2)$ is *true*.

(1) Pattern graph P. Pattern P is constructed in two steps: (1.a) construct a graph P'; and (1.b) expand P' to P. We below describe the two steps.

(1.a) P' is constructed as follows. First, for each $C_i (i \in [1, m])$, construct a set V_i of three nodes u_1^i, u_2^i, and u_3^i, yielding $3m$ nodes in V_1, ..., V_m. Intuitively, node $u_j^i (i \in [1, m], j \in [1, 3])$ is to encode literal ℓ_j^i. Then connect nodes as follows: for each $i, j \in [1, m]$ and $i \neq j$, connect u_s^i in V_i and u_t^j in V_j $(s, t \in [1, 3])$ if they do not encode complement literals, *i.e.,* $\ell_t^j \neq \bar{\ell}_s^i$. In other words, u_t^j and u_s^i are not connected only when $\ell_s^i = x$ and $\ell_t^j = \bar{x}$ or $\ell_s^i = \bar{x}$ and $\ell_t^j = x$ for some $x \in U$.

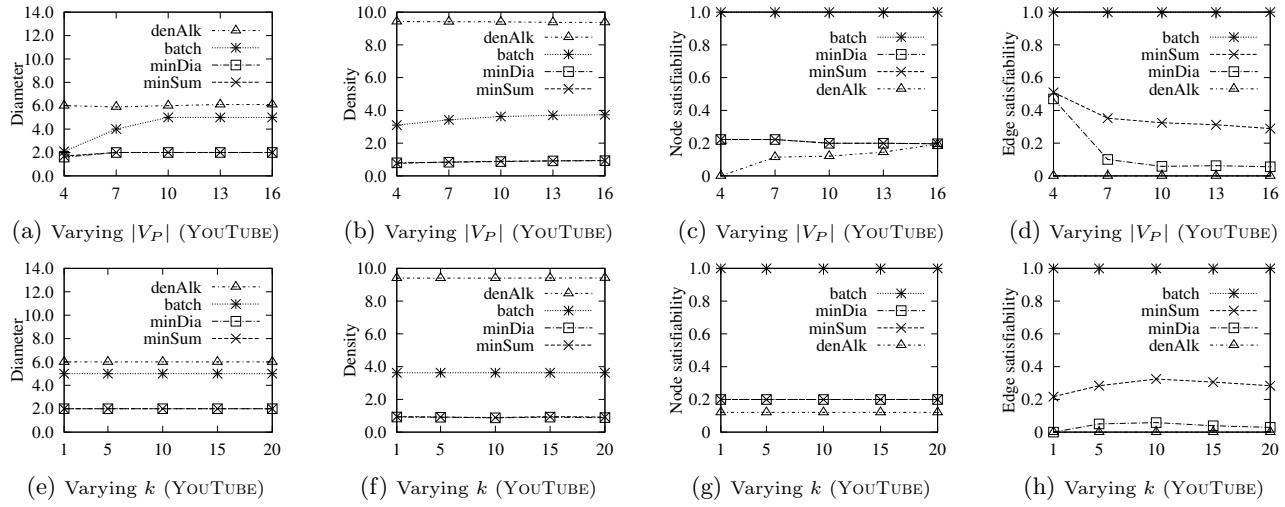
(1.b) We next expand P' to P as follows. For each node set V_i constructed in P', connect it with an $(m-2)$-clique K_i, such that each node $u_j^i (j \in [1, 3])$ in V_i is connected to every node in the clique K_i.

(2) h. Let $h = m + 1$.

(3) r_1. Let $r_1 = (m+1) * (C_m^2 + m)/|P|$, *i.e.,* $r_1 * \frac{|P|}{h} = C_m^2 + m$ is the size of a m-clique.

(4) r_2. Let $r_2 = \frac{|E_P| - (C_m^2 + m) - m * (C_m^2 + m - 1)}{|P|}$.

One can verify that ψ is satisfiable if and only if P has an h-fragmentation $P_1, \ldots, P_h$ such that $max_i |P_i| \leq r_1 * \frac{|P|}{h} (i \in [1, h])$ and $|C| \leq r_2 * |P|$. This indeed is verified by using the following properties of the above construction:



(a) Varying $|V_P|$ (YouTube) (b) Varying $|V_P|$ (YouTube) (c) Varying $|V_P|$ (YouTube) (d) Varying $|V_P|$ (YouTube)

(e) Varying k (YouTube) (f) Varying k (YouTube) (g) Varying k (YouTube) (h) Varying k (YouTube)

Figure 12: Performance evaluation of batch for top-k team formation problem

- ψ is satisfiable if and only if P' has an m-clique;
- P has a unique $m+1$ fragmentation satisfying that (i) an m-clique is the largest fragment and (ii) cut is no larger than $|E_P|-(C_m^2+m)-m*(C_m^2+m-1)$: the m-clique fragment must come from P', which leaves the m extended $(m-2)$-clique as the remaining m fragments, each of size equal to that of an m-clique minus 1.

6. Proof of Proposition 8

The proposition is correct as when ΔP comes, the perfect subgraphs must reside in the balls that match all pattern fragments of $P \oplus \Delta P$. IdABall identifies AffBs who already match all fragments of P; Or else, if there exists a fragment P_{fi} of P that an AffB cannot match, there must exist an edge/node deletion in ΔP on P_{fi}, such that the ball may match the updated fragment $P_{fi} \oplus \Delta P$. Thus, IdABall filters out the balls that cannot not match at least one fragment of P, and there are no deletion updates on the fragment.

7. Proof of Proposition 11

The proof is similar to the one of Proposition 8. IdABall filters out the balls that cannot match all pattern fragments, and there are no data updates on the ball.

Appendix C: Detailed Correctness & Complexity Analysis

1. Algorithm dynamicP (Section 5.2)

The correctness of dynamicP *w.r.t.* ΔP is assured by the correctness of (a) IdABall (by Proposition 8); (b) IncMatch, which is assured by the correctness of undirgSim that has been proved in Section 3, and the correctness of of patEIns. Indeed, patEIns only removes nodes that are no longer valid matches in each $M(P_{fi}, \hat{G}[v,r])$; (c) combine, whose analysis follows the same way as that of patEIns; and (d) the early return property (by Lemma 3).

Procedure IdABall runs in $O(2^h \cdot (|\Delta P|+h)+||\text{AffBs}||)$ time, where it takes $O(2^h \cdot |\Delta P|)$ time to update BF, $O(2^h \cdot h)$ to do the AND-operation, and $O(||\text{AffBs}||)$ to identify AffBs. As h is typically small, *e.g.,* 2 to 5, thus IdABall is in $O(|\Delta P| + ||\text{AffBs}||)$; patEIns is in $O(|M(P_{fi}, \hat{G}[v,r])| + |V_{P_{fi}}||E_{\hat{G}[v,r]}|)$ time. Indeed, the recursive process for checking invalid nodes in $M(P_{fi}, \hat{G}[v,r])$ is bounded by $O(|V_{P_{fi}}||E_{\hat{G}[v,r]}|)$, and the process to update $M(P_{fi}, \hat{G}[v,r])$ is bounded by the size of its changes, which is monotonically decreasing; According to patEIns, combine is in $O(|\bigcup_{i=1}^{h} rM(P_{fi} \oplus \Delta P_{fi}, \hat{G}[v,r])| + r|V_{P\oplus\Delta P}||E_{\hat{G}[v,r]}|)$ time.

Putting these together, algorithm dynamicP is in $O(\bigcup_{\hat{G}\in\text{AffBs}} \bigcup_{i\in[1,h]} (|M(P_{fi}, \hat{G})| + r|M(P_{fi} \oplus \Delta P_{fi}, \hat{G})|) + r|P \oplus \Delta P||\text{AffBs}| + |\Delta P|)$ time *w.r.t.* ΔP.

2. Algorithm dynamicG (Section 5.3)

The correctness of dynamicG *w.r.t.* ΔG is assured by the correctness of IdABall (by Proposition 11) and procedures IncMatch and combine, which can be proved along the same lines as for pattern updates.

One can verify that IdABall is in $O(|\text{AffBs}| + |\Delta G|)$, IncMatch for all fragments and all AffBs is in $O(|P||\text{AffBs}|)$ time, and combine is in $O(\bigcup_{i\in[1,h]} r|M(P_{fi}, \widehat{G \oplus \Delta G})| + r|V_P||E_{\widehat{G\oplus\Delta G}}|)$ time. Therefore, algorithm dynamicG is in $O(\bigcup_{\widehat{G\oplus\Delta G}\in\text{AffBs}} \bigcup_{i\in[1,h]} r|M(P_{fi}, \widehat{G \oplus \Delta G})| + r|P||\text{AffBs}| + |\Delta G|)$ time *w.r.t.* ΔG.

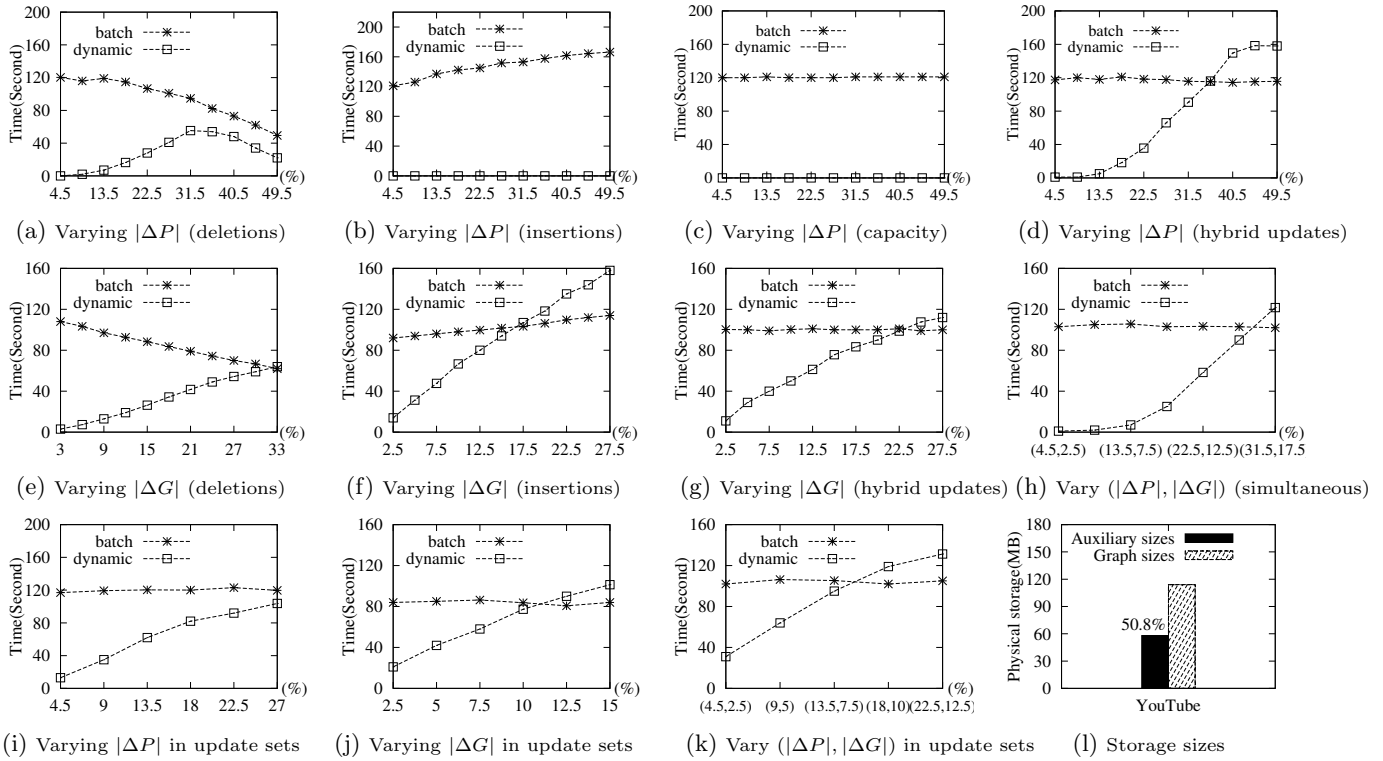
Appendix D: Extra Experiments

The experimental results on YouTube are reported here.

Exp-1: Performance of batch. We firstly evaluated the performance of batch vs. minDia, minSum and denAlk on YouTube *w.r.t.* four quality measures. The results are reported in Fig. 12(a) to 12(h). We find batch strikes a balance at capturing the practical requirements.

Exp-2: Efficiency of dynamic for single set of updates. Varying the amount of updates in one update set from 4.5% to 49.5% for pattern updates, 2.5% to 27.5% (resp. 3% to 33%) for data insertions and hybrid data updates (resp. data deletions), and (4.5%, 2.5%) to (31.5%, 17.5%) for simultaneous pattern and data updates, the results are reported in Fig. 13(a) to 13(h). We find that dynamic outperforms batch when ΔP, ΔG and (ΔP, ΔG) are no more than 36%, 22.5% and (27%, 15%), respectively.

Exp-3: Efficiency of dynamic for continuous sets of updates. We generated 5 sets of hybrid updates, varying



(a) Varying $|\Delta P|$ (deletions) (b) Varying $|\Delta P|$ (insertions) (c) Varying $|\Delta P|$ (capacity) (d) Varying $|\Delta P|$ (hybrid updates)

(e) Varying $|\Delta G|$ (deletions) (f) Varying $|\Delta G|$ (insertions) (g) Varying $|\Delta G|$ (hybrid updates) (h) Vary $(|\Delta P|, |\Delta G|)$ (simultaneous)

(i) Varying $|\Delta P|$ in update sets (j) Varying $|\Delta G|$ in update sets (k) Vary $(|\Delta P|, |\Delta G|)$ in update sets (l) Storage sizes

Figure 13: Performance evaluation of dynamic for dynamic top-k team formation problem (YouTube)

the amount of updates in each set from 4.5% to 27% for pattern updates, 2.5% to 15% for data updates, and (4.5%, 2.5%) to (22.5%, 12.5%) for simultaneous pattern and data updates. We tested the average time took by dynamic to process all these sets one by one. The results are reported in Fig. 13(i) to 13(k). We find dynamic outperforms batch when changes are no more than 27%, 10% and (13.5%, 7.5%) for continuous pattern, data and simultaneous pattern and data updates, respectively.

Exp-4: Physical storage of auxiliary structures. As shown in Fig. 13(l), it takes 58MB extra space to store all auxiliary structures utilized by dynamic on YouTube, which needs 114MB space to store itself, *i.e.,* 50.8% compared with the original dataset.