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 do not consider the specific collaboration relationships among team members, *i.e.,* structural constraints, which are typically needed in practice. In this paper, we first propose a novel graph pattern matching approach for top-k team formation, which incorporates structural constraints and capacity bounds. Furthermore, we formulate and study the dynamic top-k team formation problem, on account of that team formation is accompanied with a highly dynamic environment, to continuously process both user input and data updates, which as far as we know, has never been considered by previous study. We then develop an unified incremental approach with an optimization to handle continuous pattern and data updates, separately and simultaneously. Using real-life and synthetic data, we finally demonstrate the effectiveness and efficiency of our graph pattern matching approach for (dynamic) top-k team formation.

1. INTRODUCTION

Given a task and a social network $G(V, E)$, where V is a set of experts labeled with skills and E reflects the collaboration relations among the experts, the *top-k team formation* problem (kTF) is to find a list of k highly collaborative teams of experts in G such that each satisfies the skill requirements of the task. Various approaches [6,8,14,21,24,33] have been proposed for kTF, and fall into two categories in terms of the way to improve the collaborative compatibility of team members: (1) minimizing team communication costs, defined with *e.g.,* the diameter, minimum spanning tree and the sum of pairwise member distances of the induced subgraph [6, 8, 21, 24], and (2) maximizing team communication relations, *e.g.,* the density of the induced subgraph [14, 33]. Further, [14] and [33] consider a practical setting that introduces a lower bound on the number individuals with a specific skill in a team, and an upper bound of the team members, respectively.

Example 1: Consider a recommendation network G_1 taken from [36] 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

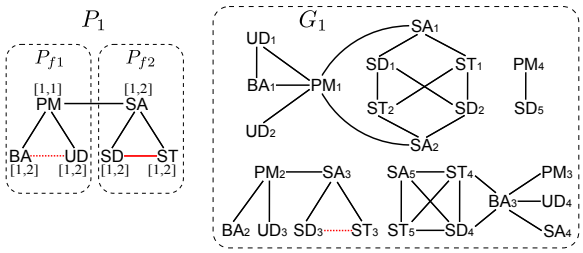


Figure 1: Motivation example

PM_1 worked well with UD_1 within previous projects.

An HR manager may help to set up a team for developing a new software product, and she searches G_1 (without dashed edges) for proper candidates. The desired team has (1) one PM, and one to two BAs, UDs, SAs, SDs and STs, such that (2) the PM should collaborate with SAs, BAs and UDs well, and the SDs and STs should collaborate with each other well and both with SAs well.

One can verify that none of existing methods suffice to identify the above desired team. They can only find teams satisfying the skill requirement [14, 21, 24] and the lower bound capacity requirement [14, 33] in search condition (1), while cannot guarantee the specific collaboration relationships among team members, *i.e.,* structural constraints in search condition (2) (see details in Example 4). □

A natural question is how to further capture the *structural and capacity constraints* in a unified model for team formation? We essentially 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 [10, 13, 18, 27, 28, 37]. Intuitively, patterns P capture the structural constraint, and we extend graph pattern matching semantics to solve the team formation problem. As shown in Fig. 1, the desired team requirement can be expressed by the pattern graph P_1 (ignore dashed edges), such that the nodes represent the skill requirements, the edges specify the topology constraint, and the bounds on nodes are the capacity constraint.

Finding desired teams typically needs a lot of labor efforts, and it is even common for professionals to refine search conditions multiple times in order to find ideal teams [17, 35]. Moreover, real-life graphs are often big and constantly evolve over time [12]. That is, team formation is accompanied with a highly dynamic environment. However, it is often too costly to recompute top-k teams from scratch in response to pattern and data updates. These highlight the need for incremental algorithms to compute the updated top-k team-

s. As opposed to batch algorithms, incremental algorithms avoid re-computing from scratch by re-using previous results. Traditional incremental algorithms for graph queries including graph pattern matching queries are proposed to compute changes in response to data updates, which has a lot of work [9, 10, 12]. However, the study on incremental algorithms for graph pattern updates is in its vacancy.

Example 2: Recall P_1 and G_1 in Example 1 and continue. (1) When the manager get the match result, *i.e.,* the top-k teams $P_1(G_1)$ to P_1 in G_1, she may find P_1 is too restrictive to find possible matches. Therefore, she enters a pattern update ΔP_1 on P_1, composed of an edge deletion $(\mathsf{SD}, \mathsf{ST})^-$. We denote $P_1 \oplus \Delta P_1$ the updated pattern. It is much better to derive $P_1 \oplus \Delta P_1(G_1)$ from $P_1(G_1)$ than a computation for $P_1 \oplus \Delta P_1$ in G_1 from scratch.

(2) When a data update ΔG_1 comes on G_1, composed of an edge insertion $(\mathsf{SD}_3, \mathsf{ST}_3)^+$. Similar with (1), it is better to compute $P_1(G_1 \oplus \Delta G_1)$ from $P_1(G_1)$ incrementally.

(3) Even better, when it comes to the case pattern ΔP_1 and data updates ΔG_1 come together, it is desirable to compute $P_1 \oplus \Delta P_1(G_1 \oplus \Delta G_1)$ from $P_1(G_1)$ incrementally. □

This motivates us to study the *dynamic top-k team formation* problem (kDTF), to handle continuous pattern and data updates, separately and simultaneously. As a basis of kDTF, we also investigate the incremental problem for graph pattern queries on pattern updates. The definition can be formulated along the same lines with the existing data incremental problem. However, devising pattern incremental algorithms is quite challenging as the impact of pattern updates is global, due to the inherent hardness of the problem. Despite of this, we devise an effective unified incremental approach for the kDTF problem.

To our knowledge, no previous work has studied pattern updates for incremental pattern matching [9,12], not to mention continuous and simultaneous pattern and data updates. It is the most general dynamic setting considered so far.

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 problem. It extends existing methods by incorporating the structural and capacity constraints using pattern graphs (Section 2). We formulate the dynamic top-k team formation problem, to handle separate and simultaneous pattern and data updates, in order to cope with the highly dynamic environment of team formation (Section 2). To the best of our knowledge, this work gives the first investigation on pattern and simultaneous pattern and data incremental computations.

(2) We develop a quadruplicate batch algorithm with two optimization techniques for computing top-k teams via team simulation. We also study the satisfiability problem for pattern graphs, a new problem raised in the presence of capacity bounds for graph pattern matching (Section 3). We develop a unified approach to handling the need for both pattern and data updates. We first prove that the dynamic top-k team formation problem is unbounded even for single pattern or data updates, by extending the notion of incremental boundedness [31]. In light of this, we then propose an incremental strategy based on *pattern fragmentation* and *affected balls* to localize the effects of pattern and data updates (Sections 4). We finally 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 (Sections 5).

(3) 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 34% for pattern updates, 31% 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 continuous simultaneous pattern and data updates, respectively.

All detailed proofs are available in the Appendix.

Related work. Previous work can be classified as follows.

Graph simulation [18] and its extensions have been introduced for graph pattern matching [10, 13, 27, 28], in which *strong simulation* introduces duality and locality into simulation [28], and shows a good balance between its computational complexity and its ability to preserve graph topology. Furthermore, [13] already adopts capacity bounds on the edges of pattern graphs via subgraph isomorphism, and [11] 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 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 [6, 8, 14, 21, 24, 26, 33], which are essentially a specialized class of keyword search on graphs [5]. Similar to [21], we are to find top-k teams. However, [21] adopted Lawler's procedure [25], and is inappropriate for large graphs. We also adopt *density* as the communication cost, which shows a better performance [14], and further require that all team members are *close to each other* (located in the same balls), along the same lines as [6, 8, 21, 24]. Except for simply minimizing the communication cost among team members, [19, 21] consider minimizing the cost among team members and team leaders. Different from these work, we introduce *structural constraints*, in terms of graph pattern matching [10, 28], into team formation, while retaining the capacity bounds on specific team members like [14, 33].

Incremental algorithms (see [9, 32] for a survey) have proven usefulness in a variety of applications, and have been studied for graph pattern matching [10, 12] and team formation [6] as well. However, [9, 10, 12, 32] only consider data updates, and [6] only consider continuous 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 /modification) is to generate alternative queries that may produce better answers, and has been studied for structured queries [30], keyword queries [39] and graph queries [29]. However, dif-

ferent from our study of handling pattern updates, the focus of query reformulation is not on incremental computations.

Although top-k queries (see [20] for a survey) have been studied for both graph pattern matching and team formation [21], 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*, an extension 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 Pattern Matching for Team Formation

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 [10, 13, 15, 40], 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 [10, 18]. 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 [18].

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 $u \in V_P$, the number of nodes $v \in V_s$ 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 P_1 and G_1 in Fig. 1. When team simulation with $r = 2$ is adopted, P_1 matches G_1 as there is a perfect subgraph in G_1, *i.e.,* the connected component of G_1 containing PM_1, which resides in ball $\hat{G}[\mathsf{PM}_1, 2]$. It maps PM, BA, UD, SA, SD and ST in P_1 to PM_1, BA_1, {UD_1, UD_2 }, {SA_1, SA_2 }, {SD_1, SD_2 } and {ST_1, ST_2 }, respectively.

One can easily verify that (a) $P_1 \prec \hat{G}[\mathsf{PM}_1, 2]$, and the ball is exactly the match graph here, and (b) the capacity bounds on all pattern nodes are satisfied. □

Remarks. (1) Team simulation differs from graph simulation and strong simulation in the existence of capacity bounds on pattern graphs and its ability to capture matches on undirected graphs. (2) As for locality constraint, different from strong simulation that requires the matched nodes to locate in a ball with a fixed given radius, team simulation adopts a more natural setting that the radius of the balls is flexible, only less than a user specified upper bound.

2.2 Top-k Team Formation

Given P, G and two positive integers r, k, the *top-k team formation* problem, denoted as 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 [16, 38]. 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 algorithms for kTF only compute the best team, instead of top-k teams.

One may want to look for candidate teams with existing methods, satisfying the search conditions in Example 1: (a) by minimizing the *team diameter* [24], which returns the team with {BA_3, PM_3, UD_4, SA_4, SD_4, ST_4}; (b) by minimizing the *sum of all-pair distances* of teams [21], which returns exactly the same team as (a) in this case; or (c) by maximizing the *team density* [14], which returns the team

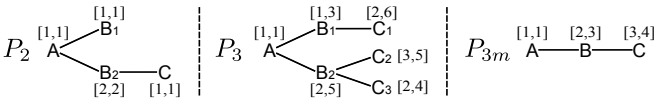


Figure 2: Pattern graphs

consisting of the nodes of the two connected components in G_1 with PM_1 and BA_3 (excluding UD_2, PM_3, UD_4, SA_4).

However, one may notice that these teams only satisfy the skill requirement, *i.e.,* the search condition (1) in Example 1, and cannot guarantee the specific collaboration relationships among team members, *i.e.,* structural constraints. Indeed, the team found in (a) and (b) is connected by BA_3 only, and the team found in (c) 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 search condition (1) and (2), better than the teams found by the existing methods as shown above. □

2.3 Dynamic Top-k Team Formation

We next introduce dynamic top-k team formation.

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

Data updates. There are four types of data updates, which can be 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 P, G, two positive integers r, 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 $\text{kDTF}(P, G, k, L_k, \Delta P, \Delta G)$, 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.

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

3. FINDING TOP-K TEAMS

We devise a quadruplicate time algorithm batch for kTF, with one novel feature and two optimizations, *i.e.,* pattern satisfiability check, and incremental computing for radius varied balls and density based filtering optimization, compared with traditional algorithms for strong simulation [28].

Pattern satisfiability. Given a pattern P, batch firstly checks whether P is satisfiable. 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.,* $P \lhd_r G$.

Different from graph simulation [18] and its extensions [10, 28], pattern graphs may be unsatisfiable for team simulation, due to the presence of capacity constraints enforced on patterns. We illustrate this with an example below.

Example 5: (1) Consider 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 u in G, if u 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) Pattern P_1 in Fig. 1 is satisfiable as $P_1 \lhd_r G_1$. □

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.* □

Proof sketch: By treating P as both data and pattern graphs, compute the maximum match relation M in P for P, via graph simulation. Indeed, 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 $|P|^2$. □

Incremental computing for radius varied balls. Recall that given G, P, k and r, 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 extremely costly to construct all such balls, *i.e., $r|V|$* balls, to compute perfect subgraphs residing in them. Instead, batch only constructs and computes 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 $\hat{G}[v, r]$ who shares the same center node. The correctness of batch is assured by the theorem 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 get 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 [12]; 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 filtering. Although we have reduced a significant amount of computation by only computing team simulation for a number of $|V|$ balls by the incremental optimization above, we can further optimize the process by adopting the density based filtering technique. It allows batch to compute team simulation for much less than $|V|$ balls. The crucial issue is how can tell whether a ball has the possibility to have one of the final top-k results in itself or in its inner balls with the same center node. The idea is, given a ball $\hat{G}[v, r]$, we calculate the upper bound of $\mathsf{den}_{\hat{G}_s}$, where $\hat{G}_s$ is a subgraph of $\hat{G}[v, r]$. If the bound is larger than the current k-th result, *i.e.,* there is possibility the final answer resides in the ball or the inner balls, compute team simulation for it; Otherwise, skip the ball to the remaining balls to avoid redundant team simulation computing.

The problem here is how to efficiently compute the upper bound of $\mathsf{den}_{\hat{G}_s}$ for each ball in G. As the best densest

Input: $G(V, E)$, $P(V_P, E_P)$, and positive integers k, r.
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** $2*\mathsf{den}_{\hat{G}_C}$ < the density of the k-th result in L_k **then**
6. **return** $L_k[0 : k-1]$;
7. $G_s := \mathsf{undirgSim}\ (P, \hat{G}[v, r])$;
8. If G_s satisfy 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 satisfy capacity bounds on P **then** Insert G'_s into L_k;
12. **return** $L_k[0 : k-1]$.

Figure 3: Algorithm batch

subgraph algorithms are in $O(|\hat{G}[v, r]|^3)$ time [16], which is costly, we utilize an important result in [38], 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 [38].* □

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 algorithm batch 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 early termination condition holds (lines 3-6). If so, it returns the current top-k densest perfect subgraphs as top-k teams; otherwise, it computes the perfect subgraph G_s of P in $\hat{G}[v, r]$ via team simulation by invoking undirgSim (line 7, see Appendix), an adaption from graph simulation [10, 18] 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 incremental algorithm in [12] and checking capacity bounds (lines 9-11, see Appendix).

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 for simulation [18] (resp. incremental simulation [12]). (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 early termination 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 METHOD

Based on the dynamic top-k team formation problem defined in Section 2.3, in this section, we first theoretically analyze the challenges and design principles, and then develop a unified incremental framework for kDTF. For convenience, the notations used are summarized in Table 1.

By Theorem 2, we know that P matches a ball $\hat{G}[v, t]$ ($t \in [1, r-1]$), only when 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, we only concern about balls $\hat{G}[v, r]$ in the incremental process, and finally compute the matches for balls $\hat{G}[v, t]$ after we get the matches for ball $\hat{G}[v, r]$. Note that in the following when we refer to a ball, we indicate the ball with radius r by default.

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

Incremental complexity analysis. As observed in [31], the complexity of an incremental algorithm 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 absolutely necessary 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 *unbounded*, otherwise. Unfortunately, kDTF is unbounded, similar to the observations in [10, 12].

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

Example 6: Continue Example 2, and consider updates ΔP_1 or ΔG_1 that obviously may introduce new matches. (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 in the ball 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 containing PM_2. The question is how to directly find new and maintain previous answers without redundant computation. □

We next analyze the difficulties and principles of designing incremental algorithms for kDTF from three aspects.

(1) Impacts of pattern and data updates. It is necessary to identify and localize the impacts of pattern/data updates. As indicated by Proposition 4 and Example 6, one can verify that (a) a unit pattern update is 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) data updates directly change the structure of data graphs, and its impact may be global, such that the entire data graph may need to be accessed to re-compute matches.

(2) Maintenance of auxiliary information. Auxiliary data on intermediate or final results for P in G are typically maintained for incremental computation [12,31]. How to design light-weight and effective auxiliary structures is the key. 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 [12]. 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 raises difficulties on the design of auxiliary data structures and on incremental algorithms.

We next develop an incremental approach to handling pattern and data updates in a unified framework, based on *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.

Pattern fragmentation. We say that $\{P_{f1}(V_{f1}, V_{f1}), \ldots, P_{fh}(V_{fh}, V_{fh}), C\}$ 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 refer to 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.

Graph simulation has a nice property as follows.

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 relations* in ball $\hat{G}$ for P via graph simulation. By the nature of graph simulation [18], $\bigcup_{i=1}^{h} M_i$ is actually intermediate results for computing M. After 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 store and maintain $\tilde{M}(P,G)$ *w.r.t.* $\mathcal{P}_h$ for the incremental process, *i.e.,* 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 shown in the experiments section.

We already know that pattern updates ΔP on P can be treated as the updates ΔP on P_{fi} or C of P. By storing $\tilde{M}(P,G)$, we can get $\bigcup_{i=1}^{h} M_i$ for a ball $\hat{G}$. Then we can do an update computation on M_i while leaving other parts unchanged. That is, to compute $P_{fi} \oplus \Delta P(\hat{G})$ instead of $P \oplus \Delta P(\hat{G})$, and to combine $P_{fi} \oplus \Delta P(\hat{G})$ with the other parts to derive $P \oplus \Delta P(\hat{G})$. Even better, the updates ΔP on C of P only involve with the combination process, avoiding the computation for any pattern fragments.

Intuitively, we 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 the *pattern fragmentation* problem. Given 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. The bi-criteria optimization problem partitions a pattern into h components of roughly equal size while minimizing the cut size between pattern fragments.

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 [10], *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 Appendix. PFrag works by connecting pattern fragmentation to the widely studied (k,ν)-Balanced Partition problem [7], which is not approximable in general, but has efficient and sophisticated heuristic algorithms [22].

So far, we have restricted the impact of pattern updates from the entire P to a fraction of P. We further localize the impact of pattern and data updates with *affected balls* to avoid computing and updating match relations for all balls.

Affected balls (AffBs). We say 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.

Remark. By pattern fragmentation and affected balls, we essentially localize the impact of pattern and data updates from the entire to a fraction of P and from all to a subset of balls in G for team simulation.

Algorithm dynamic. We now provide a unified incremental algorithm dynamic to handle both pattern and data updates.

Given P with its h-fragmentation $\mathcal{P}_h$, G, two integers k, r, and auxiliary structures such as the partial match relations for all pattern fragments and all balls (radius r), while auxiliary structures are to be presented in next section, the algorithm 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 ΔG only. Meanwhile, auxiliary structure FBM (to be seen shortly) is updated for handling continuous 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/data updates in a unified way. Better still, it holds 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 in Section 5.

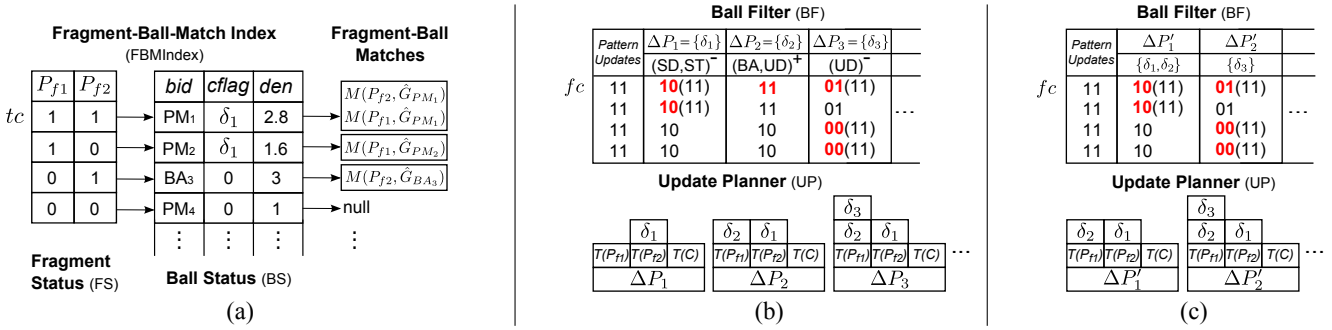


Figure 4: Example auxiliary data structures

5. INCREMENTAL ALGORITHMS

We next present auxiliary data structures for dynamic, and then present algorithm dynamicP and dynamicG to handle pattern and data updates, respectively. We finally present 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 $\{P_{f1}, \ldots, P_{fh}, C\}$ 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.

(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). P_1 is divided into fragments P_{f1} and P_{f2} by algorithm PFrag, so there are $2^2 = 4$ type codes in FS. 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. We only consider these 4 balls in the sequel. □

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

*(1) Ball filter (*BF*)* consists of 2^h boolean vectors $(b_1, \ldots, b_h)$, 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 $T(P_{f1}), \ldots, T(P_{fh}), T(C)$. Stack $T(P_{fi})$ ($i \in [1, h]$) (resp. $T(C)$) records all unit updates in all arrived pattern updates $\Delta P_1, \ldots, \Delta P_N$ that are applied to fragment P_{fi} (resp. C) of P. Initially, all of them are empty. They are dynamically updated for each unit update in each set of coming pattern updates.

5.2 Dealing with Pattern Updates

We present algorithm dynamicP to handle pattern updates ΔP, followed the steps in Section 4, and an early return optimization technique for dynamicP.

5.2.1 Identifying Affected Balls

We first develop procedure IdABall to identify AffBs, which utilizes auxiliary 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 $i \in [1, 2^h]$, it then executes a bitwise AND operation (&) between type code tc_i of FS in FBM and updated $fc_{i\Delta}$ in BF, *i.e.,* $tc_i \& fc_{i\Delta}$. (3) Finally, if $tc_i \& fc_{i\Delta} = fc_{i\Delta}$, IdABall refers to BS in FBM to mark the balls with type code tc_i as AffBs, and resets $fc_{i\Delta}$ to $(1, \ldots, 1)$.

Example 8: Consider the input and auxiliary structures in Example 7, and BF in Fig. 4.

(1) When ΔP_1 comes with a unit edge deletion δ_1=(SD, ST)$^-$, BF is updated as shown in the second column of BF in Fig. 4(b). IdABall identifies balls with type code $(1, 1)$ and $(1, 0)$ as AffBs, *i.e.,* $\hat{G}[\mathsf{PM}_1, 2]$ and $\hat{G}[\mathsf{PM}_2, 2]$. 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 δ_2 = (BA, UD)$^+$. BF is updated as shown in the second column of BF in Fig. 4(c), and IdABall identifies the same AffBs as above. □

In real-life networks, the quantity of balls with type code $(0,\ldots,0)$ is several orders of magnitude larger than that of balls who match with at least one pattern fragment. By filtering out balls with *e.g.,* $tc=(0,\ldots,0)$ or $(1,0,\ldots,0)$, it can reduce a large amount of redundant computations.

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. □

The proof is deferred to Appendix.

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 needs 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 see in Section 5.2.2), 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). □

5.2.2 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.

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

Figure 5: Procedure patEIns

(2) No edge/node deletions in ΔP_{fi}. IncMatch processes updates of the same type together in this case.

(2).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 results check in the combination procedure are needed (to see in Section 5.2.3).

(2).i 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 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 [12]. patEIns first finds the directly affected data nodes which need to be removed from 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 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 R(u), then R($\cdot$) is set to ∅ (line 15). Finally, patEIns returns the updated 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 which is directly affected by δ_2, and finds no other nodes need to be removed. □

(2).ii Node insertions in ΔP_{fi}. Node insertions are handled in the same 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 R(u) in $\hat{G}[v,r]$ which 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.

5.2.3 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 which 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 patElns($\bigcup_{i\in[1,h]} M(P_{fi}, \hat{G}[v, r]), \hat{G}[v, r], C \oplus \Delta C$) 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 results 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 SA_i (resp. PM_j) connecting to PM_j (resp. 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 above two perfect subgraphs in two balls.
(2) For $\hat{G}[\mathsf{BA}_3, 2]$, combine finds no SA_i connecting to PM_j, and vice versa. So no results are in it and its inner balls. □

5.2.4 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 [34].

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).

5.2.5 The Complete Algorithm for Pattern Updates

Given $\tilde{M}(P, G)$, FBM, BF and UP, for pattern update ΔP, algorithm dynamicP computes the maximum match relations of $P \oplus \Delta P$ in G with early return property, and maintains

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$_{L_k[k]}$ **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

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 subfgraph 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 Appendix).

Remarks. The running time of dynamicP is determined by P, ΔP, $\tilde{M}(P, G)$ and AffBs, not directly depending on G. Hence the part for pattern updates in Theorem 7 is proved.

5.3 Dealing with Data Updates

We next propose dynamicG to handle ΔG, followed the steps in Section 4. Given auxiliary structures $\tilde{M}(P, G)$ and FBM, we show how to combine procedures IdABall, IncMatch and combine for computing match results for P in $G \oplus \Delta G$. 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 $v \in V_{\hat{G}[w_1,r]} \cap V_{\hat{G}[w_2,r]}$, or (b) δ is a node insertion/deletion, $(w, (w, w'))^+/(w)^-$, and $v \in V_{\hat{G}[w,r]}$; 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. □

The proof is deferred to Appendix.

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 $O(\bigcup_{\widehat{G \oplus \Delta G} \in \mathsf{AffBs}} \bigcup_{i \in [1,h]} r|M(P_{fi}, \widehat{G \oplus \Delta G})| + r|P||\mathsf{AffBs}|)$ time *w.r.t.* ΔG, while r is small, *i.e.,* 2 or 3 (See Appendix).

Remarks. Note that the running time of dynamicG is determined by P, $\tilde{M}(P, G)$ and AffBs, not directly depending on G, and hence the part for data updates in Theorem 7 is proved. This also completes the proof of Theorem 7.

5.4 Unifying Pattern and Data Updates

We are now ready to provide algorithm dynamic, integrating dynamicP and dynamicG, which are presented in Section 5.2 and Section 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) 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 *continuous* simultaneous ΔP and ΔG, as dynamic incrementally maintains the auxiliary structures for continuous coming ΔP and ΔG.

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 set 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.

Experimental Settings. We use the following settings.

Data graphs. We used a real-life and a synthetic dataset. (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. We generated 200 labels based on phrase clustering of paper titles. (2) We adopted the LFR-benchmark graph model [23] to generate synthetic graphs (Synthetic) with community structure as existed in real-life networks. 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 $|E_P|$, label l_P for each node from an alphabet of labels in the corresponding data graphs, and 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 [24]; (b) minSum is to minimize the sum of all-pair shortest distances of teams [21]; and (c) denAlk is to maximize the team density [14]. Among algorithms for kTF, minSum is able to find top-k teams in polynomial time, which is an adaption of Lawler's procedure. 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. All the findings on YouTube are reported in the Appendix.

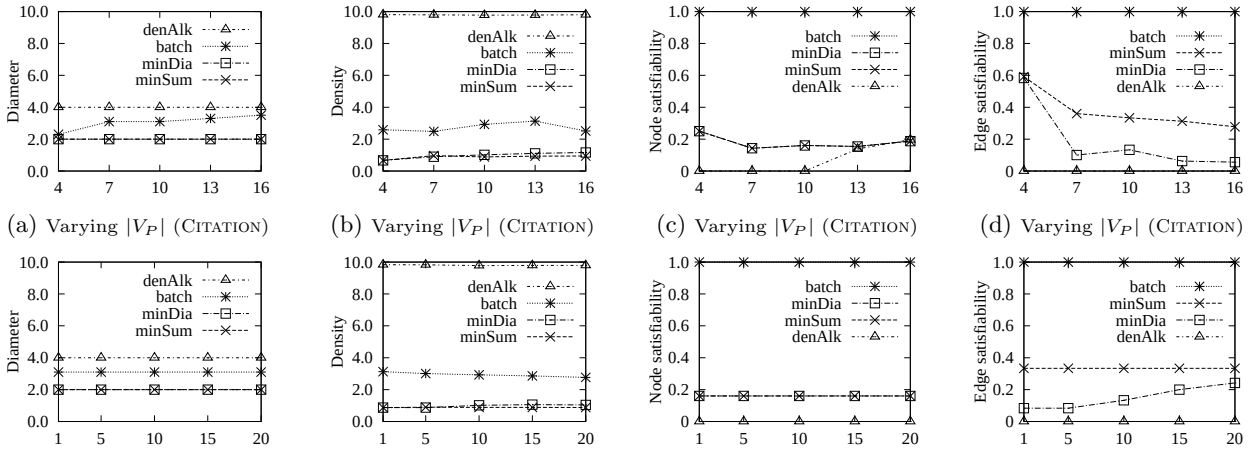
Experimental Results. In all the experiments, we set $k = 10$, $r = 2$, $h = 3$, $(|V_P|, |E_P|)$ to be (10,12), and capacity bounds to be [1,10] by default. When generating synthetic graphs, we fixed $n = 10^7$, $d = 10$ and $l = 200$.

Exp-1: Efficiency of batch. We firstly evaluated the performance of batch vs. minDia, minSum and denAlk.

Algorithms minDia, minSum and denAlk do not scale well on large graphs. Indeed, (1) minDia and minSum take more than 8 hours to finish their preprocessing, *i.e.,* computing all-pair-shortest-paths, and (2) denAlk takes more than 24 hours even when $k = 1$ on Citation. By contrast, batch takes 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 generated pattern graphs for batch, and the corresponding queries (labels requirements) for minDia, minSum and denAlk, and manually checked 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



(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

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 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. 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.* five quality measures. The results are reported in Figures 7(a)-7(d).

Observe the following. (1) In Fig. 7(a), the diameter of teams found by batch is less than minDia and minSum, but the teams are with a small diameter, *e.g.,* less than denAlk, because of the ball restriction of batch; (2) in Fig. 7(b), the density of teams found by batch is less than denAlk, while is higher than minDia and minSum; (3) in Fig. 7(c), the node satisfiability of teams found by batch, *i.e.,* 1.0 in all cases, is much higher than minDia, minSum and denAlk, *i.e.,* less than 0.2; and (4) in Fig. 7(d), the teams found by batch come with a higher edge satisfiability , *i.e.,* 1.0 in all cases, compared to less than 0.6 by minDia, minSum and denAlk.

(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 capacity satisfiability, and pertains a good team collaboration compatibility *w.r.t.* density and diameter.

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.

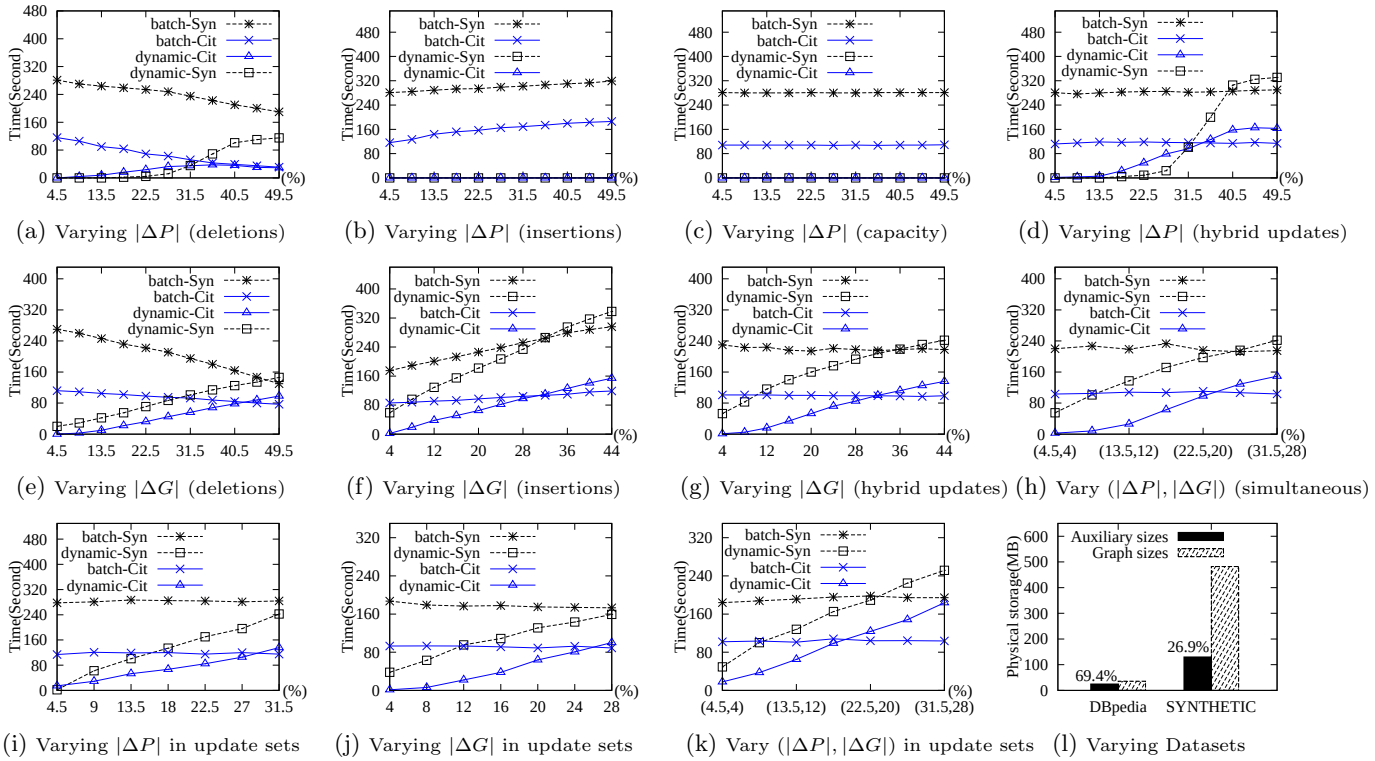
(i) Pattern updates. We fixed $(|V_P|, |E_P|)$ to be $(10, 12)$, and varied the number $|\Delta P|$ of unit updates from 1 to 11, corresponding to 4.5% to 49.5% in Figs. 8(a), 8(b), 8(c) and 8(d), which show the results when ΔP contains (edge and node) deletions, (edge and node) insertions, capacity 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



(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)

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 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 of 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 number 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 the amount of hybrid data updates in each set from 4% to 24%, 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 (27%, 24%) 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), which 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 took 110s 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 34% for pattern updates, 31% 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, and also developed a unified incremental framework 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] YouTube. https://http://netsg.cs.sfu.ca/youtubedata/.

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

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

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

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

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

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

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

[12] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3), 2013.

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

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

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

[16] A. V. Goldberg. Finding a maximum density subgraph. In *TR CSD-84-171*, 1984.

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

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

[19] 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.

[20] 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), 2008.

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

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

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

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

[25] 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), 1972.

[26] 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.

[27] 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.

[28] 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), 2014.

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

[30] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A probabilistic optimization framework for the empty-answer problem. In *PVLDB*, 2013.

[31] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.

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

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

[34] F. Ronald, L. Amnon, and N. Moni. Optimal aggregation algorithms for middleware. *JCSS*, 66(4), 2003.

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

[36] L. G. Terveen and D. W. McDonald. Social matching: A framework and research agenda. In *ACM Trans. Comput.-Hum. Interact.*, pages 401–434, 2005.

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

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

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

[40] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1), 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 perfect subgraph G_s of $\hat{G}[v, r]$ *w.r.t.* P.

1. **for** each $u \in V_P$ **do**
2. $\quad \mathsf{R}(u) := \{w | w$ is in $\hat{G}[v, r]$ and $l_P(u) \in l(w)\}$;
3. **while** there are changes **do**
4. $\quad$ **for** each edge (u, u') in E_P and each node $w \in \mathsf{R}(u)$ **do**
5. $\quad\quad$ **if** there is no edge (w, w') in $\hat{G}[v, r]$ with $w' \in \mathsf{R}(u')$ **then**
6. $\quad\quad\quad \mathsf{R}(u) := \mathsf{R}(u) \setminus \{w\}$;
7. **if** there exists $|\mathsf{R}(u)| \notin f_P(u)$ **then return** $\emptyset$;
8. $M := \{(u, w) | u \in V_P, w \in \mathsf{R}(u)\}$;
9. Construct the perfect subgraph 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 perfect subgraph of P and $\hat{G}[v, r]$. 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 3-6). A node w is removed from $\mathsf{R}(u)$ if there is a adjacent node u' of u, but there exists no adjacent node w' of w such that $w' \in \mathsf{R}(u')$. Finally, it checks whether the capacity bound on each node u is satisfied (line 7). If so, undirgSim constructs the perfect subgraph G_s *w.r.t.* the maximum match relation M and returns it (lines 9-10).

3. 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 [7].

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. $\quad P_{f1} := P^{\nu}_{f1}, \ldots, P_{fh} := P^{\nu}_{fh}$;
6. $\quad C := C^{\nu}$; $M_P := M^{\nu}_P$; $M_C := M^{\nu}_C$;
7. $\quad$ **if** $M^{\nu}_P \geq M^{\nu}_C$ **then** $\nu := \frac{\nu}{2}$ **else** $\nu := \frac{3\nu}{2}$;
8. $\quad (P^{\nu}_{f1}, \ldots, P^{\nu}_{fh}, C) := \mathsf{BalanceP}(k, \nu)$;
9. $\quad M^{\nu}_P := \mathsf{max}\{|P^{\nu}_{f1}|, \ldots, |P^{\nu}_{fh}|\}$; $M^{\nu}_C := |C^{\nu}|$;
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 Theorem 2

2. Proof of Proposition 4

Incremental complexity is defined in terms of LP (locally persistent) graph algorithms [31]. We also adopt the notion to prove unboundedness of graph algorithms for kDTF.

The proofs below strictly follows the one in [31].

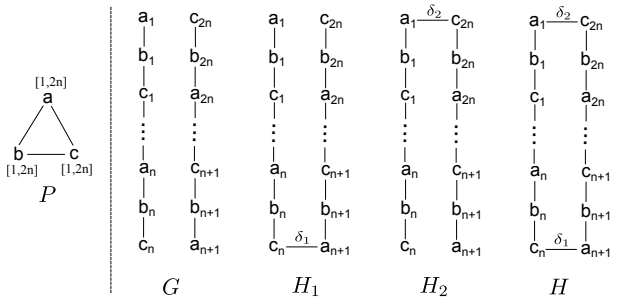


Figure 11: Unboundedness of unit data update

*(I) k*DTF *is unbounded for unit data update.* Consider the following pattern and data graph, 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.

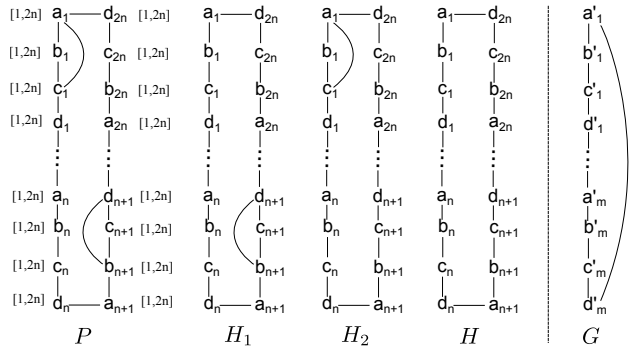


Figure 12: Unboundedness of unit pattern update

*(II) k*DTF *is unbounded for unit pattern update.* Consider the following pattern and data graph, 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.

3. 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 $P^C(V_{P^C}, E_{P^C})$ to denote the pattern with $V_{P^C}{=}V_P$ and $E_{P^C}{=}E_P/C$. Graph simulation is an iterative process to remove unmatched nodes from the candidate nodes, as illustrated in Fig. 9 (lines 1-6). 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 with the same label. 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 $M^n \subseteq M_C^n$ exists, we prove $M^{n+1} \subseteq M_C^{n+1}$ exists in the $(n{+}1)th$ iteration. Suppose both $(u, w) \in M^n$ and $(u, w) \in M_C^n$ exist, and in the $(n+1)th$ iteration, (u, w) is removed from M_C^n if there is a 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 $E_{P^C} \subseteq E_P$, that is, the edge (u, u') in E_{P^C} must belong to E_P. Thus, we have $M^{n+1} \subseteq M_C^{n+1}$.

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

4. 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) $\max_i |P_{fi}| \leq r_1 \cdot \frac{|P|}{h}$ 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.

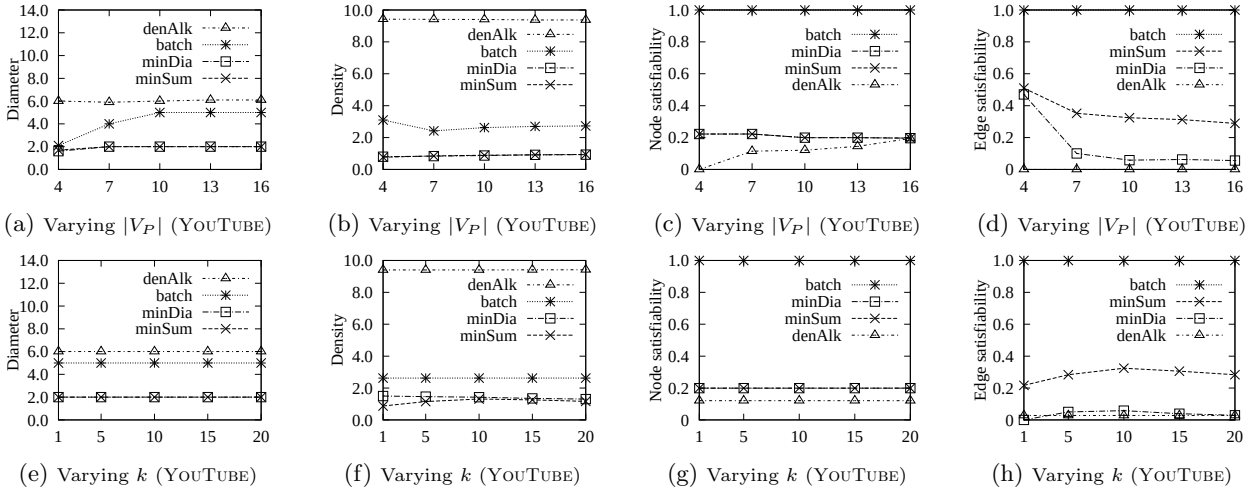
(1) Guess an h-fragmentation $\mathcal{P}_h$ of P.

(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 it is NP-hard to determine whether there exists an h-fragmentation and remains NP-hard even when $h = 2$, by reduction from the *minimum cut* problem, which is known NP-complete[1].

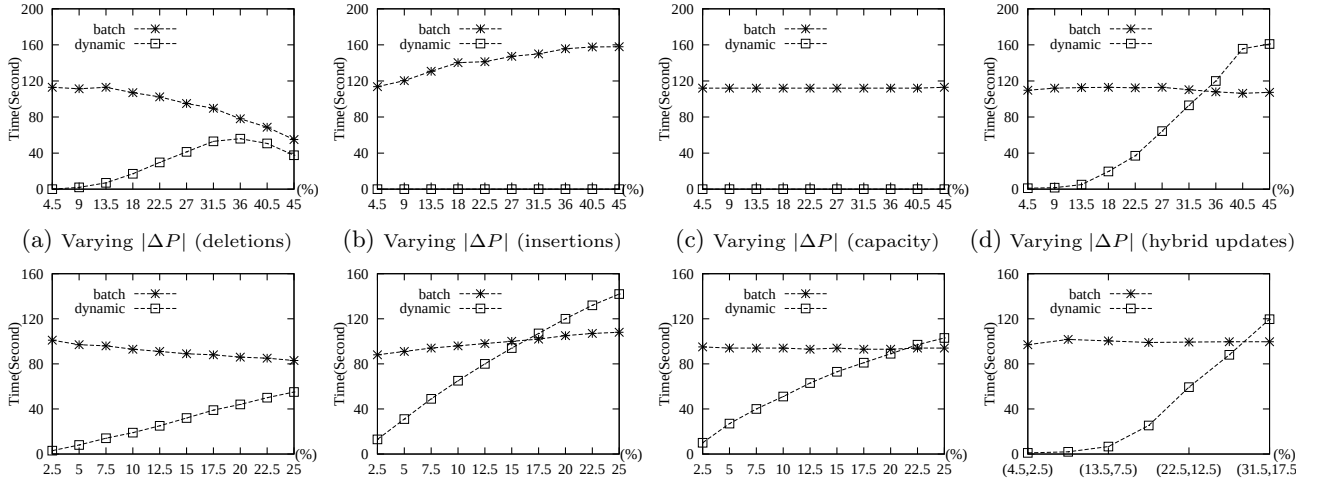
[1] http://www.nada.kth.se/∼viggo/wwwcompendium/node85.html



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

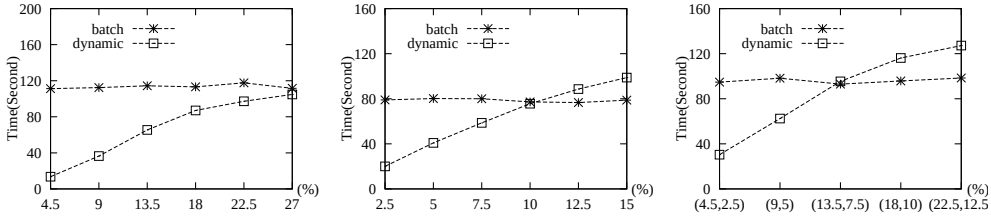
(e) Varying k (YouTube) (f) Varying k (YouTube) (g) Varying k (YouTube) (h) Varying k (YouTube)

Figure 13: Performance evaluation of batch for top-k team formation problem



(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) Varying $(|\Delta P|, |\Delta G|)$ in update sets

Figure 14: Performance evaluation of dynamic for dynamic top-k team formation problem (YouTube)

An instance of the minimum cut problem is given an undirected graph $H = (V_H, E_H)$ and a positive integer m, to find a set $S \subseteq E_H$ of edges whose removal leaves two disjoint connected components, with $|S| \leq m$.

Given an instance of the minimum cut problem, we construct an instance of h-fragmentation problem. It can be achieved by taking H as P, and setting $h = 2$, $r_1 = h$ and $r_2 = m/|P|$. Indeed, the minimum cut problem is a special case of the of h-fragmentation problem.

5. Proof of Proposition 8

We will prove the proposition by contradiction. When ΔP comes, the perfect graphs must reside in the balls that match with each pattern fragment of $P \oplus \Delta P$. IdABall identifies AffBs who already match with each fragment of P; Or else, if there exists a fragment P_{fi} of P that an AffB cannot match, there must exist an update in ΔP on P_{fi}, such that the ball may match with the updated fragment $P_{fi} \oplus \Delta P$. Therefore, IdABall filters out the balls that do not match with a fragment of P, and there are no pattern updates on the fragment.

6. Proof of Proposition 11

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

Appendix C: Extra Experiments

We also used another real-life dataset YouTube [4] for experimental verification. It 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.

The experimental results on YouTube are reported here.

Exp-1: Performance of batch. We firstly evaluated the performance of batch vs. minDia, minSum, denAlk on YouTube *w.r.t.* four quality measures. The results are reported in Fig. 13(a) to 13(h). We find batch strikes a balance at capturing the practical requirements.

Exp-2: Efficiency of dynamic for one set of updates. Varying the amount of updates in one update set from 4.5% to 45% for pattern updates, 2.5% to 25% for data updates, and (4.5%, 2.5%) to (31.5%, 17.5%) for simultaneous updates, the results are reported in Fig. 14(a) to 14(h). We find dynamic outperforms batch when ΔP, ΔG and (ΔP, ΔG) are no more than 31.5%, 20% and (27%, 15%).

Exp-3: Efficiency of dynamic for continuous sets of updates. We generated 5 sets of updates, varying the amount of updates 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 evaluated the average time took by dynamic to process these sets of updates one by one. The results are reported in Fig. 14(i) to 14(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 updates respectively.

Exp-4: Physical storage of auxiliary structures. It takes 58MB extra space to store all auxiliary structures utilized by dynamic on YouTube, which need 114MB space to store itself, *i.e.,* 50.8% compared with the original dataset.