Making Graphs Compact by Lossless Contraction

Wenfei Fan[1,2,3], Yuanhao Li[1], Muyang Liu[1], Can Lu[2]
[1]University of Edinburgh [2]Shenzhen Institute of Computing Sciences [3]Beihang University
United Kingdom[1] China[2,3]
{wenfei@inf.,yuanhao.li@,muyang.liu@}ed.ac.uk,lucan@sics.ac.cn

ABSTRACT

This paper proposes a scheme to reduce big graphs to small graphs. It contracts obsolete parts, stars, cliques and paths into supernodes. The supernodes carry a synopsis $S_{\mathcal{Q}}$ for each query class $\mathcal{Q}$ to abstract key features of the contracted parts for answering queries of $\mathcal{Q}$. The contraction scheme provides a compact graph representation and prioritizes up-to-date data. Better still, it is generic and lossless. We show that the same contracted graph is able to support multiple query classes at the same time, no matter whether their queries are label-based or not, local or non-local. Moreover, existing algorithms for these queries can be readily adapted to compute exact answers by using the synopses when possible, and decontracting the supernodes only when necessary. As a proof of concept, we show how to adapt existing algorithms for subgraph isomorphism, triangle counting and shortest distance to contracted graphs. We also provide an incremental contraction algorithm in response to updates. We experimentally verify that on average, the contraction scheme reduces graphs by 71.2%, and improves the evaluation of these queries by 1.53, 1.42 and 2.14 times, respectively.

CCS CONCEPTS

• **Information systems** → **Graph-based database models**.

KEYWORDS

Graph data management; Graph contraction; Graph algorithms

ACM Reference Format:
Wenfei Fan, Yuanhao Li, Muyang Liu, Can Lu. 2021. Making Graphs Compact by Lossless Contraction. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452797

1 INTRODUCTION

There has been prevalent use of graphs in artificial intelligence, knowledge bases, search, recommendation, business transactions, fraud detection and social network analysis. Graphs in the real world are often big, *e.g.,* transaction graphs in e-commerce companies easily have billions of nodes and trillions of edges. Worse still, graph computations are often costly, *e.g.,* graph pattern matching via subgraph isomorphism is intractable. These highlight the need for developing techniques for speeding up graph computations.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452797

There has been a host of work on the subject, either by making graphs compact, *e.g.,* graph summarization [37] and compression [7], or speeding up query answering by building indices [46]. The prior work typically targets a specific class of queries, *e.g.,* query-preserving compression [22] and 2-hop labeling [13] are for reachability queries. In practice, however, multiple applications often run on the same graph at the same time. It is infeasible to switch compression schemes between different applications. It is also too costly to build indices for each and every query class in use.

Another challenge stems from obsolete data. As a real-life example, consider graphs converted from IT databases at a telecommunication company. The databases were developed in stages over years, and have a large schema with hundreds of attributes. About 80% of the attributes were copied from earlier versions and have not been touched for years. No one can tell what these attributes are for, but no one has the gut to drop them in the fear of information loss. As a result, a large bulk of the graphs is obsolete. As another example, there are a large number of zombie accounts in Twitter. As reported by The New York Times, 71% of Lady Gaga's followers are fake or inactive, and it is 58% for Justin Bieber. The obsolete data incurs heavy time and space costs, and often obscures query answers.

The challenges give rise to several questions. Is it possible to find a compact representation of graphs that is *generic* and *lossless*? That is, we want to reduce big graphs to a substantially smaller form. Moreover, using the *same* representation, we want to compute *exact answers* to *different* classes of queries *at the same time*. In addition, can the representation separate up-to-date data from obsolete components without loss of information? Can we adapt existing query evaluation algorithms to the compact form, without the need for redeveloping the algorithms starting from scratch? Furthermore, can we efficiently and incrementally maintain the representation in response to updates to the original graphs?

Contributions & organization. This paper proposes a new approach to tackling these challenges, by extending graph contraction.

(1) A contraction scheme (Section 2). We propose a scheme to reduce big graphs into smaller ones. It contracts obsolete components, stars, cliques, paths into supernodes, and prioritizes up-to-date data. For each query class $\mathcal{Q}$, supernodes carry a synopsis $S_{\mathcal{Q}}$ that records key features needed for answering queries of $\mathcal{Q}$. As opposed to graph summarization and compression, the scheme is generic and lossless. A contracted graph retains the same topological structure for all query classes $\mathcal{Q}$, and the same synopses $S_{\mathcal{Q}}$ work for all queries in the same $\mathcal{Q}$. Only $S_{\mathcal{Q}}$ may vary for different classes $\mathcal{Q}$.

(2) Proof of concept (Sections 3). We show that existing query evaluation algorithms can be readily adapted to contracted graphs. In a nutshell, we extend the algorithms to handle supernodes. For a query Q in $\mathcal{Q}$, we make use of the synopsis $S_{\mathcal{Q}}$ of a supernode if

it carries sufficient information for answering Q, and decontract the supernode only when necessary. We pick three different query classes: subgraph isomorphism (SubIso), triangle counting (TriC) and shortest distance (Dist), based on the following dichotomies:

- label-based (SubIso) vs. non-label based (TriC, Dist);
- local (SubIso, TriC) vs. non-local (Dist); and
- topological constraints (Dist $\prec$ TriC $\prec$ SubIso).

We show how easy to adapt existing algorithms for these queries to contracted graphs, without increasing their complexity. Better still, all these queries can be answered *without decontraction of topological structures* except some supernodes for obsolete parts.

(3) Incremental contraction (Section 4). We develop an incremental algorithm for maintaining contracted graphs in response to updates to original graphs, which may change both the topological structures and timestamps. We show that the algorithm is *bounded* [44], *i.e.,* it takes at most $O(|\mathsf{AFF}|^2)$ time, where $|\mathsf{AFF}|$ is the size of areas affected by updates, not the size of the entire (possibly big) graph.

(4) Empirical evaluation (Section 5). Using 9 real-life graphs, we experimentally verify the following. On average, (a) the contraction scheme reduces graphs by 71.2%. (b) Contraction makes SubIso, TriC and Dist1.53, 1.42 and 2.14 times faster, respectively. (c) The total space cost of our contraction scheme for the three accounts for 9.8% of indices for Turboiso [26], HINDEX [43] and PLL [3]. It is 6.1% when MC [38] and kNN [50] also run on the same graph. The synopses for each take 7.3% of the space. Hence the scheme is scalable with the number of applications on the same graph. (d) Contracting obsolete data improves the efficiency of conventional queries and temporal queries by 1.23 and 1.88 times on average, respectively. (e) Our (incremental) contraction scheme scales well with graphs and updates, *e.g.,* taking 103s on graphs with 110M nodes and edges.

We discuss related work in Section 6 and future work in Section 7.

2 A GRAPH CONTRACTION SCHEME

Preliminaries. We start with basic notations. Assume two infinite sets Θ and Γ for labels and timestamps, respectively.

Graphs. We consider undirected graphs $G = (V, E, L, T)$, where (a) V is a finite set of nodes, (b) $E \subseteq V \times V$ is a bag of edges, (c) for each node $v \in V$, $L(v)$ is a label in Θ; and (d) T is a partial function: for each node $v \in V$, if $T(v)$ is defined, it is a timestamp in Γ that indicates the time when v or its adjacent edges were last updated.

Queries. A graph query is a computable function from a graph G to another object, *e.g.,* a Boolean value, a graph, and a relation. For instance, a *graph pattern matching* query is a graph pattern Q to find the set of subgraphs in G that are isomorphic to Q, denoted by $Q(G)$. *Triangle counting* is a constant query to find all triangles in G.

A *query class* $\mathcal{Q}$ is a set of queries of the same "type", *e.g.,* all graph patterns. We also refer to $\mathcal{Q}$ as an *application.* In practice, multiple applications run on the same graph G *simultaneously.*

2.1 Contraction Scheme

A *graph contraction scheme* is a triple $\langle f_C, \mathcal{S}, f_D \rangle$, where (1) f_C is a *contraction function* such that given a graph G, $G_c = f_C(G)$ is a graph deduced from G by contracting certain subgraphs H into supernodes v_H; we refer to H as the *subgraph contracted to* v_H, and G_c as the *contracted graph* of G by f_C; (2) $\mathcal{S}$ is a set of *synopsis functions* such that for each query class $\mathcal{Q}$ in use, there exists $S_{\mathcal{Q}} \in \mathcal{S}$ that annotates each supernode v_H of G_c with a *synopsis* $S_{\mathcal{Q}}(v_H)$; and (3) f_D is a *decontraction function* that restores each supernode v_H in G_c to its contracted subgraph H.

Example 1: Graph G in Fig. 1(a) is a fraction of Twitter network. A node denotes a user (u), a tweet (t), a keyword (k), or a feature of a user such as id (i), name (n), number of followers (f) and link to other account (l). An edge indicates the following: (1) (u, u'), a user follows another; (2) (u, t), a user posts a tweet; (3) (t, t'), a tweet retweets another; (4) (t, k), a tweet tags a keyword; (5) (k, k'), two keywords are highly related; (6) (u, k), a user is interested in a keyword; or (7) a user has a feature, *e.g.,* (i, l). In G, subgraphs in dashed rectangles are contracted into supernodes, yielding a contracted graph G_c shown in Fig. 1(b). Synopses S_{SubIso} for SubIso are shown in Fig. 1(d) and will be elaborated in Section 3.1. □

Before we formally define $f_C, \mathcal{S}, f_D$, observe the following.

(1) The contraction scheme is *generic.* (a) Note that f_C, G_c and f_D are *application independent*, *i.e.,* they remain the same no matter what query classes $\mathcal{Q}$ run on the contracted graphs. (b) While $\mathcal{S}$ is application dependent, it is *query independent*, *i.e.,* all queries $Q \in \mathcal{Q}$ use the same synopses annotated by $S_{\mathcal{Q}}$.

(2) The contraction scheme is *lossless* due to synopses $\mathcal{S}$ and decontraction function f_D. As will be seen in Section 3, an existing algorithm $\mathcal{A}$ for a query class $\mathcal{Q}$ can be readily adapted to contracted graph and computes exact query answers. When evaluating a query $Q \in \mathcal{Q}$ at a supernode v_H, $\mathcal{A}$ checks whether the synopsis $S_{\mathcal{Q}}(v_H)$ at v_H has enough information for Q; it uses $S_{\mathcal{Q}}(v_H)$ without decontraction if so, and decontracts v_H by restoring its subgraph via f_D otherwise. No answer to Q is lost or twisted in either case.

We next give the details of f_C, $\mathcal{S}$ and f_D. We aim to strike a balance between space cost and query evaluation cost. When a graph is *over-contracted*, *i.e.,* when the subgraphs contracted to individual supernodes are too large or too small, the decontraction cost goes up although the contracted graph G_c may take less space. Moreover, the more detailed synopses are, the less likely decontraction is needed, but the higher space overhead is incurred.

(1) Contraction function. Function f_C contracts subgraphs in G into supernodes in G_c. To simplify the discussion, we contract the following basic structures as a proof of concept.

(a) *Obsolete component*: a connected subgraph consisting of nodes whose timestamps are earlier than threshold t_0.

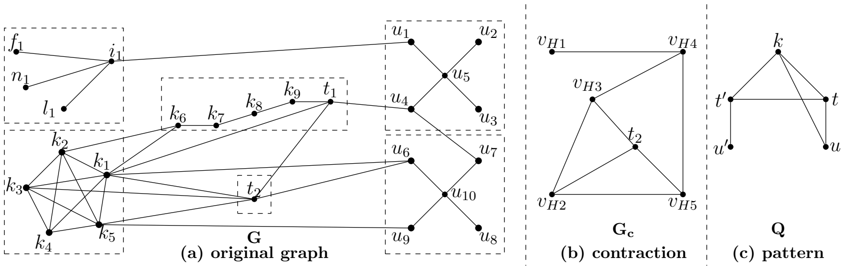
(b) *Topological component*: clique, star and path as examples.

We contract subgraphs with the number of nodes in the range $[k_l, k_u]$ to avoid over-contraction (see Section 5 for the choices).

Function f_C maps each node v in graph G to a supernode in contracted graph G_c, which is either v_H if v falls in one of the subgraphs H in (a) or (b), or node v itself otherwise.

In Example 1, function f_C maps nodes in each dashed rectangle to its corresponding supernode, *e.g.,* $f_C(i_1) = f_C(n_1) = f_C(f_1) = f_C(l_1) = v_{H1}$, $f_C(k_1) = \ldots = f_C(k_5) = v_{H2}$ and $f_C(t_2) = t_2$.

Intuitively, obsolete components help us prioritize up-to-date



(a) original graph (b) contraction (c) pattern

node v_H	v_H.type	
v_{H1}	obsolete	
v_{H2}	clique	
v_{H3}	path	v_{H2}.**list**=$\langle k_6, \ldots, k_9, t_1\rangle$
v_{H4}	star	$v_{H3}.c = u_5$
v_{H5}	star	$v_{H4}.c = u_{10}$

$\mathbf{S_{SubIso}}$

(d) synopses for SubIso(timestamp not shown)

Figure 1: Graph contraction

data, and topological ones reduce unnecessary checking when answering queries. As will be seen in Section 5, cliques, stars, paths and obsolete components contribute 12.9%, 19.4%, 0.5% and 67.2% to the contraction ratio, and 37.8%, 21.4%, 2.2% and 38.4% to the speedup of query answering process, respectively.

(2) *Contracted graph*. For a graph G, its *contracted graph by f_C* is $G_c = f_C(G) = (V_c, E_c, f'_C)$, where (a) V_c is a set of supernodes mapped from G as remarked above; (b) $E_c \subseteq V_c \times V_c$ is a bag of superedges, where a *superedge* $(v_{H1}, v_{H2}) \in E_c$ if there exist nodes v_1 and v_2 such that $f_C(v_1) = v_{H1}$, $f_C(v_2) = v_{H2}$ and $(v_1, v_2) \in E$; and (c) f'_C is the reverse function of f_C, *i.e.,* $f'_C(v_H) = \{(v, L(v)) \mid f_C(v) = v_H\}$.

In Example 1, f'_C maps each supernode in G_c of Fig. 1(b) back to the nodes in the corresponding rectangle in Fig. 1(a), *e.g.,* $f'_C(v_{H1}) = \{(i_1, \text{id}), (n_1, \text{name}), (f_1, \text{follower}), (l_1, \text{link})\}$.

(3) *Synopsis*. For each query class Q in use, a synopsis function S_Q is in $\mathcal{S}$, to retain features necessary for answering queries in Q. For instance, when Q is the class of graph patterns, at each supernode v_H, $S_Q(v_H)$ consists of the type of v_H and the most distinguished features of $f_D(v_H)$, *e.g.,* the central node of a star and the sorted node list of a path. We will give more details about S_Q in Section 3. As will also be seen there, f'_C and synopses S_Q taken together often suffice to answer queries in Q, without decontraction.

Note that not every S_Q has to reside in memory. We load S_Q to memory only if its corresponding application Q is in use.

Decontraction. Function f_D restores the subgraph contracted to a supernode. More specifically, for a supernode v_H, $f_D(v_H)$ restores the edges between the nodes in $f'_C(v_H)$, *i.e.,* the subgraph induced by $f'_C(v_H)$. For a *superedge* (v_{H1}, v_{H2}), $f_D(v_{H1}, v_{H2})$ restores the edges between $f'_C(v_{H1})$ and $f'_C(v_{H2})$, *i.e.,* the bipartite subgraph with node set $f'_C(v_{H1}) \cup f'_C(v_{H2})$ and edge set $f'_C(v_{H1}) \times f'_C(v_{H2}) \cap E$.

That is, the contracted subgraphs and edges are not dropped. They can be restored by f_D when necessary. In light of f_D, the contraction scheme is guaranteed lossless.

For example, function f_D restores the subgraph in Fig. 1(a) from supernodes, *e.g.,* $f_D(v_{H4})$ is a star with central node u_5 and leaves u_1, u_2, u_3 and u_4. It also restores edges from superedges, *e.g.,* $f_D(v_{H2}, v_{H3}) = \{(t_1, k_1), (k_1, k_6), (k_2, k_6)\}$.

2.2 Contraction algorithm

We next present an algorithm to contract a given graph G, denoted as GCon. It first contracts all obsolete data to prioritize up-to-date data. Each obsolete component is a connected subgraph that contains only nodes with timestamps earlier than a threshold t_0. It is extracted by bounded breadth-first-search (BFS) that stops at non-obsolete nodes. The remaining nodes are contracted into topological components such as paths, stars, cliques, or are left as singletons.

Algorithm GCon
Input: A graph G, timestamp threshold t_0, range $[k_l, k_u]$.
Output: Contraction function f_C and decontraction function f_D.

```
1. contract obsolete components;
2. T(G) := ordered set of regular structures of G;
3. for each t ∈ T(G) do
4.   contract topological components ([k_l, k_u]) of type t into supernodes;
5. deduce f_C and f_D;
6. return f_C and f_D;
```

Figure 2: Algorithm GCon

Different types of graphs have different dominating regular structures, *e.g.,* cliques are ubiquitous in social networks while paths are more prevalent in road networks. Hence we identify the order of topological components to contract for different types of graphs G, denote as $T(G)$. That is, we employ a *deterministic* order to ensure that important structures are contracted earlier and preserved.

More specifically, (1) for social networks and collaboration graphs, $T(G)$ = [clique, path, star]; (2) for Web graphs, $T(G)$ = [star, clique, path]; and (3) for road networks, $T(G)$ = [star, path, clique].

Putting these together, we present the main driver of algorithm GCon in Fig. 2. Given a graph G, a timestamp threshold t_0 and range $[k_l, k_u]$, it constructs functions f_C and f_D of the contraction scheme. It first contracts nodes with timestamps earlier than threshold t_0 into obsolete components (line 1). It then recalls the ordered set $T(G)$ of topological components to contract based on the type of G (line 2). Next, GCon contracts topological components into supernodes following the order $T(G)$, and deduces functions f_C and f_D accordingly (lines 3-5). More specifically, it does the following.

(1) For paths, it first extracts intermediate nodes that have only two neighbors and the neighbors are disconnected. For each path containing only intermediate nodes, it constructs a path component along with two neighbors of the endpoints.

(2) For cliques, it repeatedly selects an uncontracted node that connects to all selected ones, and extracts a clique.

(3) For stars, it first picks a central node. It then repeatedly selects an uncontracted node as a leaf that is (a) connected to the center and (b) disconnected from all selected leaves; it makes these into a star.

As remarked earlier, the remaining nodes that cannot be contracted into any component are mapped to themselves by f_C.

Example 2: Assume that timestamp threshold t_0 for graph G of Fig. 1(a) is larger than timestamps of nodes i_1, n_1, f_1 and l_1, but is smaller than those of remaining nodes. Algorithm GCon works as follows. (1) It first triggers bounded BFS, and contracts i_1, n_1, f_1 and l_1 into an obsolete component v_{H1} in G_c. (2) Since G is a social network, it contracts clique, path and star in this order. It builds a clique v_{H2} with nodes $k_1, \ldots, k_5$. (3) It finds k_7, k_8, k_9, u_1, u_7 and u_9 as candidate intermediate nodes for paths, and contracts k_7, k_8, k_9 into a path v_{H3} with endpoints k_6 and t_1. Nodes u_1, u_7 and u_9 cannot make paths due to lower bound $k_l = 4$. (4) It picks u_5 and u_{10} as central nodes for stars, and makes two stars v_{H4} and v_{H5}. (5) Node t_2 is left singleton, and is mapped to itself by f_C. □

Complexity. (1) Obsolete components can be contracted in $O(|G|)$ time via edge-disjoint bounded BFSs; (2) paths can be built in $O(|G|)$ time; (3) contracting each clique takes $O(|G|)$ time and all cliques can be handled in $O(|G|^2)$; and (4) similarly, all stars can be contracted in $O(|G|^2)$. Thus GCon costs at most $O(|G|^2)$ time.

Properties. Observe the following about the contraction scheme. (1) It is *lossless* and is able to compute exact query answers. (2) It is *generic* and supports multiple applications at the same time. This is often necessary since on average 10 classes of queries run on a graph simultaneously in GDB benchmarks [19]. (3) It *prioritizes up-to-date data* by separating it from obsolete data. (4) It improves performance. (a) As will be seen in Section 5, $|G_c| \ll |G|$. (b) Decontraction is often not needed, *e.g.,* SubIso does not need to decontract topological components, and for TriC and Dist, even obsolete supernodes do not need decontraction (Section 3).

2.3 Parallel Contraction Algorithm

We next parallelize algorithm GCon, to speed up the contraction process. Note that contraction is conducted once offline, and is then incrementally maintained in response to updates (Section 4).

The idea is to leverage data-partitioned parallelism. Given n available machines and a graph G, we partition G into fragments $(F_1, \ldots, F_n)$ and distribute them to n machines. All the machines first run GCon on its local fragment in parallel since after all, each F_i is a graph itself. They then contract "border nodes", *i.e.,* nodes with edges across fragment. We ensure that each node v is contracted into at most one supernode v_H by function f_C. More specifically, we outline the parallel algorithm, denoted by PCon, as follows.

(1) Partition G "evenly" using a parallel edge-cut partitioner, *e.g.,* ParMETIS [31], such that each node of G is in a single fragment.

(2) Each machine runs GCon on its local fragment, in parallel.

(3) For each border node v, if v is not yet contracted into a supernode, build its k_u-neighbor, *i.e.,* the subgraph with only uncontracted nodes within k_u hops of v. Neighbors are identified in parallel, coordinated by a machine M_0. Coordinator M_0 merges overlapped neighbors into one, and distributes disjoint ones to n machines.

(4) Each machine contracts its assigned subgraphs in parallel.

When some neighbors in step (3) are too big, they are edge-cut partitioned again and processed following steps (1) and (2).

One can verify that each node v in G is contracted into at most one supernode v_H. The graph G_c contracted by PCon may be slightly different from that of GCon since border nodes may be contracted in different orders. One can fix this by repeating steps (1)-(4) for each of clique, star and path following the order $T(G)$. Nonetheless, we experimentally find that the differences are not substantial enough to worth the extra cost. Moreover, the contracted graphs of PCon are already compact, *i.e.,* they cannot be contracted further.

3 PROOF OF CONCEPT

We next show that existing query evaluation algorithms can be readily adapted to contracted graphs. As a proof of concept, we pick three query classes: (1) subgraph isomorphism (labeled queries with locality); (2) triangle counting (non-labeled queries with locality); and (3) shortest distance (non-labeled and non-local queries).

Informally, for a query $Q \in \mathcal{Q}$, we check whether the synopsis $S_Q(v_H)$ at a supernode v_H has enough information for Q; it uses $S_Q(v_H)$ directly if so; otherwise it decontracts superedges adjacent to v_H or restores the subgraph of v_H via decontraction function f_D. As will be seen shortly, $S_Q(v_H)$ often provides enough information either to process Q at v_H as a whole or safely skip v_H. Thus it suffices to answer queries in the three classes by decontracting superedges, without decontracting any topological components.

3.1 Graph Pattern Matching with Contraction

We start with graph pattern matching in contracted graphs.

Pattern. A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$, where (1) V_Q (resp. E_Q) is a set of *pattern nodes* (resp. *edges*), and (2) L_Q is a function that assigns a label $L_Q(u)$ to each $u \in V_Q$.

We also investigate *temporal pattern* (Q, t), where Q is a pattern as above and t is a given timestamp (see details shortly).

To simplify the discussion, we consider connected patterns Q. This said, our algorithm can be adapted to disconnected ones.

Pattern matching. A *match* of pattern Q in graph G is a subgraph $G' = (V', E', L', T')$ of G that is isomorphic to Q, *i.e.,* there exists a *bijective function* $h : V_Q \to V'$ such that (1) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (2) $e = (u, u')$ is an edge in pattern Q iff (if and only if) $(h(u), h(u'))$ is an edge in graph G and $L_Q(u') = L(h(u'))$. We denote by $Q(G)$ the set of all matches of Q in G.

A *match* of a temporal pattern (Q, t) in graph G is a match G' in $Q(G)$ such that for each node v in G', $T'(v) > t$, *i.e.,* a match of (conventional) pattern Q in which all nodes have timestamps later than t. We denote by $Q(G, t)$ all matches of (Q, t) in G.

The *graph pattern matching problem*, denoted by SubIso, is to compute, given a pattern Q and a graph G, the set $Q(G)$ of matches. Similarly, the *temporal matching problem* is to compute $Q(G, t)$ for a given (Q, t) and a graph G, denoted by SubIso_t.

Note that (1) patterns Q are *labeled, i.e.,* nodes are matched by labels. Moreover, (2) Q has the *locality*, *i.e.,* for any match G' of Q in G and any nodes v_1 and v_2 in G', v_1 and v_2 are within d_Q hops when treating G' as an undirected graph. Here d_Q is the *diameter* of Q, *i.e.,* the maximum shortest distance between any two nodes in Q.

The decision problem of pattern matching is NP-complete (cf. [24]); similarly for temporal matching. Several algorithms are in place for SubIso, notably Turboiso [26] with indices and VF2 [16] without index. Both can be adapted to contracted graphs.

Theorem 1: *Using a* ***linear*** *synopsis function, both* Turboiso *and* VF2 *can be adapted to compute exact answers for* SubIso *on* G_c*, which decontract only supernodes of obsolete components and superedges between supernodes,* ***not topological components****.* □

We give a constructive proof for Turboiso, because (1) it is one of the most efficient algorithms for subgraph isomorphism and is followed by other SubIso algorithms *e.g.,* [9, 45], and (2) it employs indexing to reduce redundant matches; we show that the indices

Algorithm Turboiso
Input: A graph G and a graph pattern Q.
Output: The set $Q(G)$ of all matches of Q in G.
1. $Q(G) := \emptyset$; $v_s := ChooseStartN(Q, G)$
2. $Q' := RewriteToNEC(Q, v_s)$;
3. **for each** $x_s \in \{x \mid x \in V \wedge L(x) = L(v_s)\}$ **do**
4. $\quad CR := ExploreCR(v_s, x_s)$;
5. $\quad$ **if** $CR \neq \emptyset$ **then**
6. $\quad\quad$ compute matching order $O(x_s, CR)$;
7. $\quad\quad Q(G) := Q(G) \cup SGSearch((x_s, v_s), Q, Q', G, O)$;
8. return $Q(G)$;

Figure 3: Algorithm Turboiso

Algorithm $\mathsf{SubA_c}$
Input: Contracted G_c, scheme $\langle f_C, S_{\mathsf{SubIso}}, f_D\rangle$, function f'_C and pattern Q.
Output: The set $Q(G)$ of all matches of Q in G.
1. $Q(G) := \emptyset$; $v_s := ChooseStartN(Q, G_c)$
2. $Q' := RewriteToNEC(Q, v_s)$;
3. **for each** $x_s \in \{x \mid x \in V_c \wedge L(v_s) \subseteq L(x)\}$ **do**
4. $\quad CR := ExploreCR(v_s, x_s, f'_C, S_{\mathsf{SubIso}})$;
5. $\quad$ **if** $CR \neq \emptyset$ **then**
6. $\quad\quad$ compute matching order $O(x_s, CR)$;
7. $\quad\quad Q(G) := Q(G) \cup SGSearch((x_s, v_s), Q, Q', G_c, O, f'_C, S_{\mathsf{SubIso}}, f_D)$;
8. return $Q(G)$;

Figure 4: Algorithm $\mathsf{SubA_c}$

for SubIso can be inherited by contracted graphs, *i.e.,* contraction and indexing complement each other.

Below we first present synopses for SubIso (Section 3.1.1), which are the same for both VF2 and Turboiso. We then show how to adapt Turboiso to contracted graphs (Section 3.1.2).

3.1.1 *Contraction for* SubIso.

The idea of synopses is to store the types and key features of regular structures so that we could check pattern matching without decontracting topological components.

The synopsis of a supernode v_H for SubIso is defined as follows:

- clique: v_H.type = clique;
- star: v_H.type = star, $v_H.c$ records its central node;
- path: v_H.type = path, v_H.list = $\langle u_1, \ldots, u_{|v_c|}\rangle$, storing all the nodes on the path in order;
- obsolete component: v_H.type = obsolete; and
- each component maintains $v_H.t = \max\{T(v) \mid v \in f'_C(v_H)\}$, *i.e.,* the largest timestamp of its nodes.

For instance, the synopsis $S_{\mathsf{SubIso}}(v_H)$ for each supernode v_H in the contracted graph G_c of Fig. 1(b) is given in Fig. 1(d).

The synopses in S_{SubIso} have two properties. (1) Taken with the reverse function f'_C of f_C, the synopsis of a supernode v_H suffices to recover topological component H contracted to v_H. For instance, given the central node and leaf nodes, a star can be uniquely determined. As a result, no supernode decontraction is needed for topological components. (2) The synopses can be constructed during the traversal of G for constructing G_c, as a byproduct.

We remark that the design of synopses needs domain knowledge. This said, (1) users only need to develop synopses for their applications in use, not exhaustively for all possible query classes; and (2) synopsis design is no harder than developing indexing structures.

3.1.2 *Subgraph Isomorphism.*

We first review algorithm Turboiso, and then show how to adapt Turboiso to contracted graphs.

Turboiso. As shown in Fig. 3, given a graph G and a pattern Q, Turboiso computes $Q(G)$ as follows. It first rewrites pattern graph Q into a tree Q' by performing BFS from a start vertex v_s (lines 1-2). Here each vertex in Q' is a *neighborhood equivalence class* (NEC). Then, for each start vertex x_s of each region, Turboiso constructs a candidate region (CR), *i.e.,* an index maintaining candidates for each NEC vertex in Q', via DFS from x_s (lines 3-4). If valid candidates are found, *i.e., $CR \neq \emptyset$*, Turboiso enumerates all possible matches that map x_s to v_s following a matching order O (lines 5-6). It expands $Q(G)$ with valid matches identified in the process (line 7).

Algorithm $\mathsf{SubA_c}$. Turboiso can be easily adapted to contracted graph G_c, denoted by $\mathsf{SubA_c}$. As shown in Fig. 4, $\mathsf{SubA_c}$ adopts the same logic as Turboiso except minor adaptations in ExploreCR (line 4) and SGSearch (line 7) to deal with supernodes. To see these, let H be the subgraph contracted to a supernode v_H.

(1) ExploreCR. It adds a supernode v_H as a candidate for a node u in Q if some node in v_H can match u, which is checked by $S_{\mathsf{SubIso}}(v_H)$ and $f'_C(v_H)$. It also prunes CR based on v_H.type, *e.g.,* a node u in Q matches intermediate nodes on a path only if its degree is no larger than 2. No supernodes or superedges are decontracted.

(2) SGSearch. Checking the existence of an edge (x, y) that matches edge $(v_x, v_y) \in Q$ is easy with synopses S_{SubIso} and functions f'_C and f_D. Here x (resp. y) denotes a node in supernode $v_H = f_C(x)$ (resp. $v_H = f_C(y)$) in the candidates of v_x (resp. v_y). When $f_C(x) = f_C(y) = v_H$, (a) if v_H.type=star, (x, y) exists only if $x = v_H.c$ or $y = v_H.c$; (b) if v_H.type = clique, (x, y) always exists; and (c) if v_H.type=path, (x, y) exists if x and y are next to each other in v_H.list. Hence no topological component is decontracted by f_D. (d) If v_H.type=obsolete, it checks whether none of the labels in Q is in $f'_C(v_H)$; it safely skips v_H if so, and decontracts v_H by f_D to check the existence of (x, y) otherwise. If x and y match distinct supernodes, it suffices to decontract superedge $(f_C(x), f_C(y))$ by f_D.

Example 3: Query Q in Fig. 1(c) is to find potential friendships based on retweets and keywords. Nodes x and x' in Q both have label x. Given Q, algorithm $\mathsf{SubA_c}$ first chooses k as the start node, to which only v_{H2} and v_{H3} can match. For v_{H2}, ExploreCR adds v_{H3} and t_2 as candidates for t and t', v_{H5} for u, and v_{H4}, v_{H5} for u'. Note that for obsolete supernode v_{H1}, none of the labels in Q is covered by $f'_C(v_{H1})$; hence, v_{H1} can be safely skipped. SGSearch finds that t_2 matches t since there is no edge connecting v_{H3} and v_{H5}. Thus it matches k, t, u, t', u' with k_1, t_2, u_6, t_1, u_4, respectively. Similarly, for v_{H3}, ExploreCR adds v_{H3} and t_2 as candidates for t and t', v_{H4} as candidate for u, and v_{H4}, v_{H5} as candidates for u, u'. Next, SGSearch finds that u_4 and t_1 match u and t by decontracting superedge (v_{H3}, v_{H4}); then k_9 matches k. However, since k_9 is an intermediate node of path v_{H3}, no match for t' can be found. Hence, k, t, u, t', u' match k_1, t_2, u_6, t_1, u_4. □

Analyses. $\mathsf{SubA_c}$ is correct since it has the same logic as Turboiso albeit pruning strategies. While the two have the same worst-case complexity, $\mathsf{SubA_c}$ operates on G_c, which is much smaller than G (see Section 5); moreover, its ExplorCR saves traversal cost and SGSearch saves validation cost by pruning invalid matches.

Temporal pattern matching. Algorithm $\mathsf{SubA_c}$ can also take a temporal pattern (Q, t) as part of its input, instead of Q. The only major difference is at CR construction (line 4), where a supernode v_H is safely pruned if $v_H.t \leq t$, when v_H.type is obsolete or not. It skips a match if it contains a node v with $T(v) \leq t$.

3.2 Triangle Counting with Contraction

We next study triangle counting [14, 28]. In graph G, a *triangle* is a clique of three vertices. The *triangle counting problem* is to find the total number of triangles in G, denoted by TriC.

Similar to SubIso, TriC is local with diameter 1. In contrast to SubIso, it consists a single query and is not labeled.

We adapt TriA [14] for TriC to contracted graphs, since it is one of the most efficient TriC algorithms [28], and it does not use indexing (as a different example from TurboIso).

Theorem 2: *With a* ***linear*** *synopsis function,* TriA *can be adapted to* G_c *for* TriC*, which* ***decontracts superedges only*** *but decontracts no supernodes, neither topological nor obsolete components.* □

3.2.1 Contraction for TriC. Observe that contraction function f_C on G is equivalent to node partition of G, such that two nodes are in the same partition if they are contracted into the same supernode. The idea of synopses is to pre-count triangles with at least two nodes in the same partition, without enumerating them. As will be seen shortly, this allows us to avoid supernode decontraction for both topological and obsolete components. Consider a triangle (u, v, w) in G that is mapped to G_c via f_C. We have the following cases.

(1) If $f_C(u) = f_C(v) = f_C(w) = v_H$, where v_H contracts a subgraph H with vertex set $V(H)$, then (a) when H is a clique, there are $\binom{|V(H)|}{3}$ triangles inside H; (b) when H is an obsolete component, then the number of triangles inside H can be pre-calculated, denoted by t_H; and (c) there are no triangles inside H otherwise.

(2) If $f_C(u) = f_C(v) = v_I$, $f_C(w) = v_J$, where v_I and v_J contract subgraphs I and J, respectively, then (a) when I is a clique, then w leads to $\binom{k}{2}$ triangles, where k is the number of neighbors of w in I. Denote by t^I_w the number of such triangles in a clique neighbor I of w. (b) When I is an obsolete component or a star, then u and v together lead to k triangles, where k is the number of common neighbors of u, v in J. We denote by $t^J_{u,v}$ the number of such triangles in a common neighbor J of u, v. Note that I cannot be a path.

(3) If $f_C(u) = v_I$, $f_C(v) = v_J$, $f_C(w) = v_K$, we count such triangles online and it suffices to decontract only superedges.

The synopsis $S_{\mathsf{TriC}}(v_H)$ of a supernode v_H for TriC extends $S_{\mathsf{SubIso}}(v_H)$ with an extra tag tc, which records the number of triangles pre-calculated as above. More specifically, v_H.tc is computed as follows. In the definition below, u and v range over nodes in $V(H)$, I ranges over clique neighbors of u, J ranges over common neighbors of u, v, and t^I_u, t_H and $t^J_{u,v}$ are defined as above:

- clique: $v_H.\text{tc} = \binom{|V(H)|}{3} + \Sigma_u \Sigma_I t^I_u$;
- star: $v_H.\text{tc} = \Sigma_u \Sigma_I t^I_u + \Sigma_u \Sigma_J t^J_{v_H.c,u}$;
- path: $v_H.\text{tc} = \Sigma_I t^I_{u_1} + \Sigma_I t^I_{u_{|V(H)|}}$, where u_1 and $u_{|V(H)|}$ are the first and last node on the path, respectively; and
- obsolete: $v_H.\text{tc} = t_H + \Sigma_u \Sigma_I t^I_u + \Sigma_{u,v} \Sigma_J t^J_{u,v}$, where u and v are connected nodes in subgraph H contracted by v_H.

Synopses S_{TriC} also share the properties of S_{SubIso}.

Example 4: In contracted graph G_c of Fig. 1(b), only v_{H2} contracts a clique, denoted by I. Synopsis $S_{\mathsf{TriC}}(v_H)$ of a supernode v_H extends $S_{\mathsf{SubIso}}(v_H)$ with v_H.tc: (1) for v_{H1}, (a) $H1$ contracted to v_{H1} contains no triangles; thus $t_{H1} = 0$; (b) I is not a neighbor of any node u in $V(H1)$; thus $t^I_u = 0$; and (c) nodes in $V(H1)$ have no common neighbors, *i.e.,* no J exist for any connected $u, v \in V(H1)$; thus $t^J_{u,v} = 0$. Hence $v_{H1}.\text{tc} = 0$. (2) For v_{H2}, v_{H2}.type=clique, $|V(H2)| = 5$ and no other supernodes in G_c are cliques. Hence $v_{H2}.\text{tc} = 10$. (3) For v_{H3}, the first and last elements k_6 and t_1 have 2 and 1 neighbors in I, respectively. Thus $t^I_{k_6} = 1$, $t^I_{t_1} = 0$, and $v_{H3}.\text{tc} = 1$. (4) Similarly, $v_{H4}.\text{tc} = v_{H5}.\text{tc} = 0$, and $t_2.\text{tc} = 3$. □

3.2.2 Triangle counting. We now adapt algorithm TriA [14] to contracted graphs. The adapted algorithm is referred to as $\mathsf{TriA_c}$.

TriA. Given a graph G, TriA assigns distinct numbers to all the nodes in G. It then enumerates triangles for each edge (u, v) by counting the common neighbors w of u and v such that $w < u$ and $w < v$.

Algorithm $\mathsf{TriA_c}$. On a contracted graph G_c with superedges decontracted, $\mathsf{TriA_c}$ works in the same way as TriA except that at a supernode v_H (for either topological and obsolete component), it simply accumulates v_H.tc without decontraction or enumeration.

Example 5: From synopsis S_{TriC}, $\mathsf{TriA_c}$ directly finds 14 triangles. In G_c, it finds two additional triangles (u_6, t_2, k_1) and (t_1, t_2, k_1) by restoring superedges. Thus it finds 16 triangles in G. No supernodes of either topological or obsolete components are decontracted. □

Analyses. $\mathsf{TriA_c}$ is correct since it counts all triangles in G once and only once. It speeds up TriA since it works on a smaller contracted G_c, and reduces the cost by leveraging pre-calculated triangles.

Temporal triangle counting. $\mathsf{TriA_c}$ can be adapted to count triangles with timestamp later than a given time t. It prunes a supernode v_H if $v_H.t \leq t$, and drops a triangle if it has a node v with $T(v) \leq t$.

3.3 Shortest Distance with Contraction

We next study the shortest distance problem, denoted by Dist.

Shortest distance. Consider an undirected weighted graph $G = (V, E, L, T, W)$ with additional weight W; for each edge e, $W(e)$ is a positive number for the length of the edge. The length of a *path* $p = (v_0, \ldots, v_k)$ in G is simply $\text{sum}_{i \in [1,k]} W(v_{i-1}, v_i)$.

The problem is to compute, given a pair (u, v) of nodes in G, the shortest distance between u and v, denoted by $d(u, v)$ [3, 13, 18].

As opposed to SubIso, shortest distance queries are *unlabeled*, *i.e.,* the value of query answer $d(u, v)$ does not depend on labels. In contrast to SubIso and TriC, Dist is non-local, *i.e.,* there exists no d independent of the input graph G such that $d(u, v) < d$.

We adapt Dijkstra's algorithm [18] to contracted graphs, denoted by Dijkstra, which is one of the best known algorithms for Dist.

Theorem 3: *With a* ***linear*** *synopsis function,* Dijkstra *for* Dist *can be adapted to contracted graph* G_c*; it decontracts superedges but* ***no supernodes****, neither topological nor obsolete components.* □

3.3.1 Contraction for Dist. A path between nodes u and v can be decomposed into (1) edges between supernodes, and (2) paths within a supernode. The idea of synopses is to pre-compute the shortest distances within supernodes to avoid supernode decontraction, for both topological and obsolete components. Edges between supernodes are recovered by superedge decontraction when necessary. Suppose that v_1 and v_2 are nodes mapped to supernode v_H by f_C, *i.e.,* $f_C(v_1) = f_C(v_2) = v_H$. We compute the shortest distance for (v_1, v_2) within the subgraph H contracted to v_H, denoted by $d_{v_H}(v_1, v_2)$. The synopsis $S_{\text{Dist}}(v_H)$ extends $S_{\text{SubIso}}(v_H)$ with a tag dis that is a set of triples $(v_1, v_2, d_{v_H}(v_1, v_2))$ for a path between v_1 and v_2 within v_H, based on v_H.type:

- clique: v_H.dis=$\{(v_1, v_2, d_{v_H}(v_1, v_2))\}$ for $v_1, v_2 \in f'_C(v_H)$;
- path: v_H.dis = $\{(u_1, u_{|f'_C(v_H)|}, \Sigma_{1 \leq i < |f'_C(v_H)|} W(u_i, u_{i+1}))\}$, *i.e.*, it records the path itself;
- obsolete: v_H.dis=$\{(v_1, v_2, d_{v_H}(v_1, v_2)) \mid v_1, v_2 \in f'_C(v_H)\}$.

We can compute $S_{\text{Dist}}(v_H)$ in constant time as $|f'_C(v_H)| \leq k_u$ if v_H.type is clique or obsolete. If v_H.type is star, we can find $d_{v_H}(v_1, v_2)$ for two nodes by using the synopsis, f'_C and W.

Example 6: Assume that $W(u, v) = 1$ for all edges (u, v) in graph G of Fig. 1(a). Then for supernodes in Fig. 1(b), (1) v_{H1}.dis = $\{(i_1, f_1, 1), (i_1, n_1, 1), (i_1, l_1, 1), (f_1, n_1, 2), (f_1, l_1, 2), (n_1, l_1, 2)\}$; (2) v_{H2}.dis = $\{(k_i, k_j, 1)\}$ for $1 \leq i < j \leq 5$; and (3) v_{H3}.dis = $\{(k_6, t_1, 4)\}$. □

3.3.2 Shortest distance. We adapt algorithm Dijkstra to contracted graphs G_c, and refer to the adapted algorithm as DisAc.

Dijkstra. Given a graph G and a pair (u, v) of nodes, Dijkstra finds the shortest distances from u to nodes in G in ascending order, and terminates as soon as $d(u, v)$ is determined. It maintains a set S of nodes whose shortest distances from u are known; it initializes distance estimates $\overline{d}(u) = 0$, and $\overline{d}(w) = \infty$ for other nodes. At each step, Dijkstra moves a node w from $V \setminus S$ to S that has minimal $\overline{d}(w)$, and updates distance estimates of nodes adjacent to w accordingly.

Algorithm DisAc. DisAc is the same as Dijkstra except minor changes to updating distance estimates. When moving a node w having $f_C(w) = v_H$, from $V \setminus S$ to S, DisAc updates distance estimates $\overline{d}(w')$ for $w' \in f'_C(v_H)$ as follows: (1) if v_H.type is clique or obsolete, update $\overline{d}(w')$ by $\overline{d}(w) + d_{v_H}(w, w')$ using v_H.dis; (2) if v_H.type = star, update $\overline{d}(w')$ by $\overline{d}(w) + d_{v_H}(w, w')$, where $d_{v_H}(w, w')$ can be easily computed by synopsis; (3) if v_H.type = path, update $\overline{d}(w')$ by $\overline{d}(w) + d_{v_H}(w, w')$ for the other endpoint w' using v_H.dis; in these cases, no supernode of either topological or obsolete components is decontracted. In addition, DisAc updates $\overline{d}(w')$ by $\overline{d}(w) + W(w, w')$ for all edges (w, w') where $f_C(w) \neq f_C(w')$, by decontracting superedge $(f_C(w), f_C(w'))$ at worst, the same as Dijkstra.

Example 7: Given query (u_2, k_5) on G_c of Fig. 1(b), DisAc works in steps: (1) initially, $S = \emptyset$, $\overline{d}(u_2) = 0$, and $\overline{d}(v) = \infty$ for all other nodes; (2) $S = \{u_2\}$, $\overline{d}(u_5) = 1$, $\overline{d}(u_1) = \overline{d}(u_3) = \overline{d}(u_4) = 2$ by f'_C and $S_{\text{Dist}}(v_{H4})$ (v_{H4} contracts a star); (3) $S = \{u_2, u_5, u_1, u_3, u_4\}$, $\overline{d}(t_1) = 3$ by edge (u_4, t_1), and $\overline{d}(k_6) = \overline{d}(t_1) + d_{v_{H3}}(k_6, t_1) = 7$ by v_{H3}.dis; $\overline{d}(i_1) = 3$ by edge (u_1, i_1), and $\overline{d}(f_1) = \overline{d}(n_1) = \overline{d}(l_1) = 4$ by v_{H1}.dis; similarly, $\overline{d}(u_7) = 3$ and $\overline{d}(u_{10})$=4, $\overline{d}(u_6) = \overline{d}(u_8) = \overline{d}(u_9) = 5$ by f'_C and $S_{\text{Dist}}(v_{H5})$; (4) $S = \{u_2, u_5, u_1, u_3, u_4, t_1, i_1, u_7\}$, $\overline{d}(t_2) = 4$ by edge (t_1, t_2); (5) $S = \{u_2, u_5, u_1, u_3, u_4, t_1, i_1, u_7, f_1, n_1, l_1, t_2\}$, $\overline{d}(k_1) = \overline{d}(k_3) = \overline{d}(k_5) = 5$ by edges (t_2, k_1), (t_2, k_3), (t_2, k_5). When DisAc moves k_5 to S, it gets $d(k_5) = 5$. It returns $d(u_2, k_5) = 5$. □

Analyses. By induction on the length of shortest paths, we can verify that DisAc is correct. In particular, for each node w' in G, when $\overline{d}(w')$ is updated by node w that is mapped to the same supernode, the update is equivalent to a series of Dijkstra updates. Moreover, DisAc works on smaller contracted graphs G_c and saves traversal cost inside contracted components without any decontraction.

Temporal shortest distance. Similar to temporal SubIso and TriC, we consider *temporal* Dist *queries* (u, v, t), where (u, v) is a pair of nodes as in Dist, and t is a timestamp. It is to compute the shortest length of paths p from u to v such that for each node w on p, $T(w) > t$. It is also to prioritize frequently visited nodes in a graph.

Algorithm DisAc can be adapted to temporal Dist, by skipping nodes v with $T(v) \leq t$. It safely ignores a supernode v_H if $v_H.t \leq t$.

4 INCREMENTAL CONTRACTION

We next develop an incremental algorithm to maintain contracted graphs in response to updates ΔG to graph G. We start with batch update ΔG, which is a sequence of edge insertions and deletions. We formulate the problem (Section 4.1), present the incremental algorithm (Sections 4.2, 4.3), and discuss vertex updates (Section 4.4).

4.1 Problem

Given a contraction scheme $\langle f_C, \mathcal{S}, f_D \rangle$, a contracted graph $G_c = f_C(G)$, and batch update ΔG, the *incremental contraction problem*, denoted as ICP, is to compute (a) changes ΔG_c to G_c such that $G_c \oplus \Delta G_c = f_C(G \oplus \Delta G)$, *i.e.*, to get the contracted graph of the updated graph $G \oplus \Delta G$, where $G_c \oplus \Delta G_c$ applies ΔG_c to G_c; (b) the updated synopses of supernodes; and (c) functions $f_C \oplus \Delta f_C$ and function $f_D \oplus \Delta f_D$ *w.r.t.* the new contracted graph $G_c \oplus \Delta G_c$.

ICP studies the maintenance of contracted graphs in response to update ΔG that may change both the topological structures of contracted graph G_c, and refresh timestamps of nodes. As a consequence, obsolete nodes may be promoted to be non-obsolete ones if they are touched by edges in ΔG, among other things.

Criterion. Following [44], we measure the complexity of incremental algorithms in terms of the size of the *affected area*, denoted by AFF. Here AFF includes (a) changes ΔG to the input, (b) changes ΔG_c to the output, and (c) edges with at least an endpoint in (a) or (b).

An incremental algorithm is said to be *bounded* if its complexity is determined by |AFF|, not by size $|G|$ of graph G.

Intuitively, ΔG is typically small in practice. When ΔG is small, so is ΔG_c. Hence when ΔG is small, a bounded incremental algorithm is often far more efficient than a batch algorithm that recomputes G_c starting from scratch, since the cost of the latter depends on the size of possibly big G, as opposed to |AFF| of the former.

An incremental problem is *bounded* if there exists a bounded incremental algorithm for it, and is *unbounded* otherwise.

Challenges. Problem ICP is nontrivial. (1) Topological components are fragile, *e.g.*, when inserting an edge between two leaves of a star H, H is no longer a star, and its nodes may need to be merged into other topological components. (2) Refreshing timestamps may make some obsolete nodes "fresh", and force us to reorganize obsolete and topological components. (3) When contracted graph G_c is changed, so are their associated synopses and decontraction function.

Main result. Despite the challenges, we show that bounded incremental contraction is within reach in practice.

Theorem 4: *Problem* ICP *is bounded for* SubIso, TriC *and* Dist, *and takes at most* $O(|\text{AFF}|^2)$ *time.* □

We give a constructive proof of Theorem 4 consisting of two parts: (1) the maintenance of the contracted graph G_c and its associated decontraction function f_D (Section 4.2); and (2) the maintenance of the synopses of affected supernodes (Section 4.3).

4.2 Incremental Contraction

An incremental algorithm is shown in Fig. 5, denoted by IncCR. It has three steps: *preprocessing* to initialize affected areas, *updating* to maintain contracted graph G_c, and *contracting* to process refreshed singleton nodes. To simplify the discussion we focus on how to update G_c in response to ΔG; the handling of f_D is similar.

(a) Preprocessing. Algorithm IncCR first identifies an initial area affected by update ΔG (lines 1-2). It removes "unaffecting" updates from ΔG that have no impact on G_c (line 1), *i.e.,* edges in ΔG that are between two supernodes when none of their nodes is an intermediate node of a path. These updates are made to corresponding subgraphs of G that are maintained by f_D. It then refreshes timestamps of nodes u touched by edges $e = (u, v)$ in ΔG (line 2). Suppose that u is mapped by f_C to supernode v_H with v_H.type = obsolete. Then v_H is decomposed into singleton nodes, u is non-obsolete and is mapped to itself by f_C. Such singleton nodes are collected in a set V_s, as the initial area affected by ΔG. Node v is treated similarly.

Note that an unaffecting update would not become affecting later on. All changes in ΔG are applied in G in the given order.

(b) Updating. IncCR then updates G_c (lines 3-8). For each update $e = (u, v)$, IncCR invokes procedure IncCR$^+$ (resp. IncCR$^-$) to update G_c when e is to be inserted (resp. deleted) (lines 4-7). Updating G_c may make some updates in ΔG unaffecting, which are further removed from ΔG (line 8). Moreover, some nodes may become "singleton" when a topological component is decomposed by the updates, *e.g.,* leaves of a star. It collects such nodes in the set V_s.

More specifically, to insert an edge $e = (u, v)$, IncCR$^+$ updates G_c and adds new singleton nodes to V_s. Suppose that u (resp. v) is mapped by f_C to supernode v_{H1} (resp. v_{H2}) (line 1). IncCR$^+$ decomposes v_{H1} and v_{H2} into the regular structures of topological components (line 2). For instance, if v_{H1} and v_{H2} are the same star, u and v make a triangle with the central node; thus IncCR$^+$ decomposes the star into singleton nodes. When v_{H1}.type = clique and v_{H2}.type = path, v_{H_2} is divided into two shorter paths. Note that components with less than k_l nodes are decomposed into singleton nodes. All such singleton nodes are added to V_s (line 3).

(c) Contracting. Finally, algorithm IncCR processes nodes in V_s (line 10). It (a) merges nodes into neighboring supernodes; or (b) builds new components with these nodes, if possible; otherwise (c) it leaves nodes v as singleton, *i.e.,* by letting $f_C(v) = v$.

Example 8: Consider inserting four edges into G of Fig. 1(a): (1) (n_1, f_1): nodes n_1 and f_1 are mapped to obsolete component v_{H1}, and v_{H1} is decomposed into singleton nodes, one for each of n_1, f_1, i_1 and l_1; then (n_1, f_1) is removed from ΔG; (2) (k_1, u_4): it is unaffecting since $f_C(k_1) \neq f_C(u_4)$ and neither k_1 nor u_4 is an intermediate node of a path; (3) (k_1, u_{10}): it is also unaffecting; and (4) (u_1, u_4): as it makes a new triangle (u_1, u_4, u_5), v_{H4} is decomposed into singletons. Edge deletions are handled similarly. □

Analyses. Algorithm IncCR takes $O(|\mathsf{AFF}|^2)$ time: (a) the preprocessing step is in $O(|\Delta G|)$ time; (b) the updating step takes $O(|\mathsf{AFF}|)$ time, in which updating f_D is the dominating part; and (3) the cost of contracting V_s into topological components is in $O(|\mathsf{AFF}|^2)$.

Algorithm IncCR
Input: A graph contraction scheme $\langle f_C, \mathcal{S}, f_D \rangle$, a contracted graph G_c of a graph G and updates ΔG to G.
Output: New contracted graph $G_c \oplus \Delta G_c$.
1. reduce ΔG; $V_s := \emptyset$;
2. refresh nodes u in ΔG;
3. **for each** update $e = (u, v) \in \Delta G$ **do**
4.   **if** e is an edge insertion
5.   **then** IncCR$^+(G_c, e)$;
6.   **else** if e is an edge deletion
7.   **then** IncCR$^-(G_c, e)$;
8.   reduce ΔG;
9. Contract (V_s, G_c);
10. return G_c;

Procedure IncCR$^+$
*Input:*a contracted graph G_c, edge insertion $e = (u, v)$.
Output: An updated G_c.
1. $v_{H1} := f_C(u)$; $v_{H2} := f_C(v)$;
2. Divide (v_{H1}, v_{H2});
3. add singleton nodes into V_s;

Figure 5: Algorithm IncCR

The algorithm is (a) bounded [44], since its cost is determined by $|\mathsf{AFF}|$ alone, and (b) *local* [21], *i.e.,* the changes are confined only to affected supernodes and their neighbors in G_c.

4.3 Maintenance of Synopses

We next show that for SubIso, TriC and Dist, (a) the number of supernodes whose synopses are affected is at most $O(|\mathsf{AFF}|)$, and (2) the synopsis for each supernode can be updated in $O(|\mathsf{AFF}|)$ time. Hence incremental synopses maintenance for each of SubIso, TriC and Dist takes at most $O(|\mathsf{AFF}|^2)$ time.

To see these, consider a supernode v_H in G_c. (a) For SubIso, recall that $S_{\mathsf{SubIso}}(v_H)$ stores the type and key features of v_H (Section 3.1). It is easy to see that the number of supernodes whose synopses are affected is at most $|\Delta G_c|$, and $S_{\mathsf{SubIso}}(v_H)$ for each such v_H can be updated in $O(1)$ time. Thus the maintenance of S_{SubIso} is bounded in $O(|\mathsf{AFF}|)$ time. (b) For TriC, synopsis $S_{\mathsf{TriC}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ with v_H.tc. Note that v_H.tc is updated by (i) clique neighbors I of nodes u in v_H where $I \in \mathsf{AFF}$; (ii) v_H itself if v_H.type=clique or v_H.type=obsolete; and (iii) common neighbors J of connected nodes u, v in v_H for $J \in \mathsf{AFF}$. Thus supernodes affected are enclosed in AFF, which covers ΔG, ΔG_c and their neighbors. Moreover, $S_{\mathsf{TriC}}(v_H)$ for each affected v_H can be updated in $|\mathsf{AFF}|$ time. Thus the maintenance of S_{TriC} is bounded in $O(|\mathsf{AFF}|^2)$ time. (c) For Dist, $S_{\mathsf{Dist}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ with v_H.dis, which is confined to v_H and can be updated in $O(1)$ time since $|f'_C(v_H)| \leq k_u$. Thus the incremental maintenance of S_{Dist} is bounded in $O(|\mathsf{AFF}|)$ time.

Example 9: Continuing with Example 8, we show how to maintain v_H.tc in $S_{\mathsf{TriC}}(v_H)$ for supernodes v_H in G_c; $S_{\mathsf{SubIso}}(v_H)$ and $S_{\mathsf{Dist}}(v_H)$ are simpler since their affected synopses are confined to ΔG_c. (1) For edge insertion (n_1, f_1), v_{H1} is decomposed into four singletons, for which synopses are defined as $n_1.\text{tc} = f_1.\text{tc} = l_1.\text{tc} = i_1.\text{tc} = 0$. (2) For (unaffecting) edge insertion (k_1, u_4), $v_H.tc$ remains the same for all $v_H \in G_c$. (3) For (unaffecting) insertion (k_1, u_{10}),

Graph	$\|V\|$, $\|E\|$	k_u	CR	clique	star	path	obsolete
Twitter	81K, 1.3M	100	0.184/0.299	8.95/35.12	17.78/64.88	0.00/0.00	73.27/0
LiveJournal	4M, 35M	500	0.397/0.558	12.08/37.27	21.52/62.71	0.01/0.02	66.40/0
LivePokec	1.6M, 22M	500	0.472/0.689	4.45/11.35	35.81/88.65	0.00/0.00	59.73/0
Google	876K, 4.3M	200	0.193/0.294	17.75/52.20	17.72/47.76	0.02/0.04	64.52/0
NotreDame	325K, 1.1M	200	0.279/0.47	9.36/30.47	22.89/69.32	0.01/0.21	67.68/0
DBLP	204K, 382K	100	0.140/0.172	33.56/71.87	13.18/28.11	0.02/0.03	53.24/0
Hollywood	1.1M, 56M	500	0.246/0.561	16.36/81.40	5.69/18.60	0.00/0.00	77.95/0
citHepTh	28K, 352K	50	0.278/0.396	14.50/41.29	21.90/58.59	0.02/0.13	63.57/0
Traffic	24M, 29M	500	0.401/0.750	0.01/0.03	15.05/78.40	4.25/21.60	80.70/0

Table 1: Contraction ratio

Graph	SubIso		TriC		Dist	
	RE	EX	RE	EX	RE	EX
Twitter	6.96	4.08	5.95	3.16	4.18	3.51
LiveJournal	8.17	6.56	6.92	2.11	4.57	5.03
LivePokec	10.47	6.76	6.23	4.0	3.06	4.01
Google	3.39	5.98	2.44	2.67	2.36	5.22
NotreDame	10.89	4.64	1.8	4.74	4.02	4.82
DBLP	4.09	6.58	4.98	6.45	3.46	4.02
Hollywood	5.49	4.75	4.3	7.07	2.38	5.42
citHepTh	6.67	4.92	4.38	4.3	3.61	4.09
Traffic	9.48	5.31	5.61	5.11	5.62	4.32

Table 2: Slowdown (%) by RE and EX orders

k_1 becomes a common neighbor of u_{10} and u_6; then $t^{H2}_{u_{10},u_6} = 1$ and v_{H5}.tc = 1. (4) When inserting (u_1, u_4), v_{H4} is decomposed into singletons, whose synopses are u_1.tc = ... = u_5.tc = 0. □

4.4 Vertex Updates

Vertex updates are a dual of edge updates [32]. More specifically,

(1) when inserting a new node v, v is first treated as a singleton and collected in set V_s; it is then contracted into a topological structure in the contracting step of algorithm IncCR (line 9).

(2) When deleting a node v that is contracted into a supernode v_H, there are three cases to consider: (a) if v is the central node of a star, v_H is removed and all nodes in $f'_C(v_H)$ except v are treated as singletons and collected in set V_s, as in the updating step of algorithm IncCR (line 5); the singletons are then contracted as above; (b) if v is an intermediate node of a path, v_H is replaced by two supernodes that contract two shorter paths, as in the updating step (line 5); otherwise (c) v is removed directly as in the preprocessing step (line 1). Note that deleting v may remove some superedges adjacent to v_H, which are maintained by function f_D.

Similar to edge updates, contracting nodes in V_s dominates the cost. One can verify that it can be done in $O(|\mathsf{AFF}|^2)$ time. Similarly, synopsis maintenance also takes $O(|\mathsf{AFF}|^2)$ time. Hence incremental contraction remains bounded in the presence of vertex updates.

5 EXPERIMENTAL STUDY

Using real-life graphs, we experimentally evaluated (1) the reduction ratio and (2) the speedup of the contraction scheme, (3) the impact of contracting each topological component and obsolete component; (4) the space cost of the contraction scheme compared to existing indexing methods; (5) the efficiency of the (incremental) contraction algorithm; and (6) the parallel scalability of the scheme.

Experiment setting. We used the following datasets.

(1) Graphs. We used 9 real-life graphs: three social networks Twitter [40], LiveJournal [51] and LivePokec [6]; two Web graphs Google [35] and NotreDame [4]; three collaboration networks DBLP [2], Hollywood [10] and citHepTh [34]; and a road network Traffic [1]. Their sizes are shown in Table 1. We randomly generated a time series to simulate obsolete attributes, at most 70% (it is 80% for the IT data of our industry collaborator). We also tested obsolete components with random (temporal) queries generated on all datasets.

We also generated synthetic graphs with up to 10M nodes and 100M edges, to test the scalability of the contraction algorithm.

Updates. We randomly generated ΔG, controlled by size $|\Delta G|$ and a ratio ρ of edge insertions to deletions. We kept $\rho = 1$ unless stated otherwise, *i.e.,* the size of $G \oplus \Delta G$ remains stable.

(2) Graph patterns. We generated pattern queries controlled by the number V_Q of query nodes, the number E_Q of edges, and labels L_Q.

(3) Implementation. We implemented the following, all in C++. (1) Algorithms $\mathsf{SubA_c}$ (Section 3.1.2), $\mathsf{TriA_c}$ (Section 3.2.2), DisAc (Section 3.3.2) and $\mathsf{VF2_c}$ for SubIso by adapting VF2 [16] to contracted graphs. (2) Our contraction algorithm GCon (Section 2.2) and its parallel version PCon (Section 2.3), and incremental algorithm IncCR for batch updates (Section 4). (3) The baselines include: (a) Turboiso [26] and TurboIsoBoosted [45] with indexing, and VF2 [16] without indexing for SubIso; (b) graph compression DeDense [39] for SubIso; (c) TriA [28] for TriC; and (d) Dijkstra for Dist. We did not compare with summarization since it does not support any exact algorithm for the three applications.

(4) Environment. The experiments were run on a single processor machine powered by Xeon 3.0 GHz with 32G memory, running Linux. We simulated up to 20 distributed parallel machines using two machines, each with 12 cores powered by Xeon 3.0 GHz, 64GB RAM, and 10Gbps NIC. Each simulated machine has a single core with 4GB RAM, and communication is only via message passing. Each experiment was run 5 times. The average is reported.

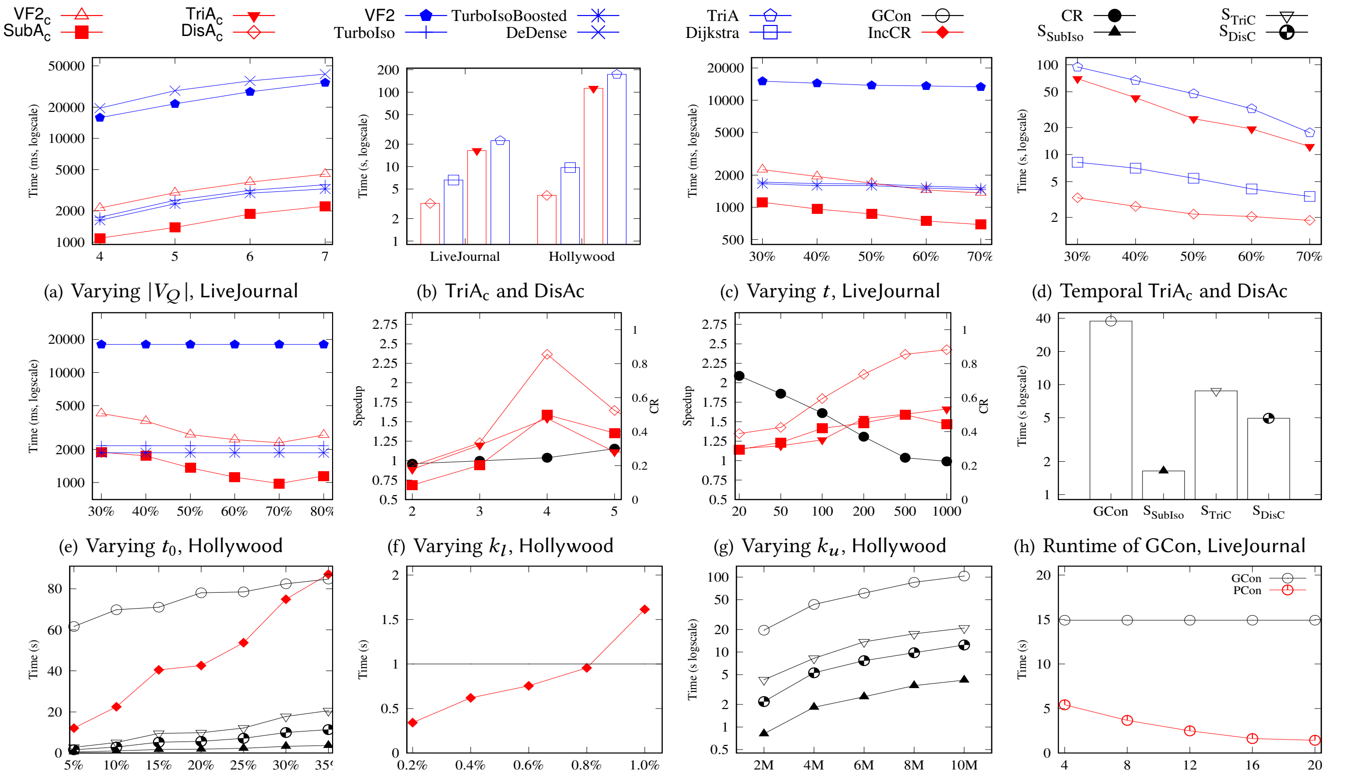
Experimental results. We now report our findings.

Exp-1: Effectiveness: Contraction ratio. We first tested the *contraction ratio* of our contraction scheme, defined as $CR = |G_c|/|G|$. Note that for each query class $\mathcal{Q}$, CR is the same for all queries in $\mathcal{Q}$. Moreover, all applications on G share the same contracted graph G_c albeit different synopses. We also report the impact of each topological component and obsolete component for each dataset .

As remarked in Section 2, we limit the nodes of contracted subgraphs within $[k_l, k_u]$. We fixed $k_l = 4$ and varied k_u based on the size of each graph. We considered two settings: (a) when obsolete data is taken into account, with threshold $t_0 = 50\% t_m$, where t_m denotes the maximum timestamp in each dataset; and (b) when we do not separate obsolete data, *i.e.,* when $t_0 = 0$. The results are reported in Table 1 for all the real-life graphs (in which each column indicates either CR or percentage of contribution to CR with/without obsolete mark). We can see the following.

(1) When $t_0 = 50\% t_m$, CR is on average 0.288, *i.e.,* contraction reduces these graphs by 71.2%. When $t_0 = 0$, *i.e.,* if obsolete data is not considered, CR is 0.465. These show that real-life graphs can be effectively contracted in the presence and absence of obsolete data.

(2) When obsolete data is present, the average CR is 0.351, 0.236, 0.221 and 0.401 in social networks, Web graphs, collaboration networks and road networks, respectively. When obsolete data is ab-



(a) Varying $|V_Q|$, LiveJournal (b) TriA$_c$ and DisAc (c) Varying t, LiveJournal (d) Temporal TriA$_c$ and DisAc

(e) Varying t_0, Hollywood (f) Varying k_l, Hollywood (g) Varying k_u, Hollywood (h) Runtime of GCon, LiveJournal

(i) Varying $|\Delta G|$, LiveJournal (j) Varying $|\Delta G|$/s, LiveJournal (k) Varying $|G|$, GCon, Synthetic (l) Varying k, PCon, Hollywood

Figure 6: Performance evaluation

sent, *CR* is 0.515, 0.382, 0.377 and 0.75. The contraction scheme performs the best on collaboration networks in both settings, since such graphs exhibit evident inhomogeneities and community structures.

(3) When obsolete data is absent, on average a component of path, clique and star contains 4.81, 5.34 and 8.41 nodes and contribute 2.5%, 40.1% and 57.4% to CR, respectively. When obsolete mark is taken into account, their contribution is 0.5%, 12.9% and 19.4% to CR, respectively. This is because nodes from these components may be moved to obsolete components. Cliques and paths bear smaller impact than stars, due to their regular structures and size bound k_l. Hence stars contribute more substantially to CR in this case.

(4) We also studied the impact of the contraction order on query evaluation. Taking the order proposed in Section 2 as the baseline, we tested the impact of (a) RE, by reversing the order, and (b) EX, by exchanging between different types of graphs, *e.g.,* we use the order for road networks to contract social graphs. On average the CR of RE and EX is decreased by 5.3% and 5.1%, respectively. As shown in Table 2, the average slowdown of RE and EX is (a) 7.3% and 5.5% for SubIso, (b) 4.7% and 4.4% for TriC, and (c) 3.7% and 4.5% for Dist, respectively. These justify the order of Section 2.

Exp-2: Effectiveness: query processing. We next evaluated the speedup of the scheme, measured by query evaluation time over original and contracted graphs. We report results on some graphs for the lack of space; the results on the other graphs are consistent.

Subgraph isomorphism. Varying $|V_Q|$ from 4 to 7, we tested VF2, Turboiso and TurboIsoBoosted on LiveJournal as G, DeDense [39] on the compressed graph, and SubA$_c$ and VF2$_c$ on the contracted graph G_c of G. As shown in Fig. 6(a), (1) on average, SubA$_c$ on G_c is 1.68, 19.1 and 1.55 times faster than Turboiso, DeDense and TurboIsoBoosted, respectively; (2) VF2$_c$ beats DeDense by 9.36 times; (3) VF2$_c$ without indices is only 18.1% slower than Turboiso with indices, while Turboiso and TurboIsoBoosted are 9.1 and 9.8 times faster than VF2, respectively; and (4) the speedup is bigger on collaboration networks, *e.g.,* 1.71 times on Hollywood.

Triangle counting. As shown in Fig. 6(b), the results for TriC are consistent with the results on subgraph isomorphism: (1) TriA$_c$ on the contracted G_c is on average 1.42 times faster than TriA on their original graphs G. (2) The speedup is more evident in collaboration networks: *e.g.,*TriA$_c$ on Hollywood is 1.47 times faster than TriA.

Shortest distance. As also shown in Fig. 6(b), algorithm DisAc is 2.06 and 2.36 times faster than Dijkstra on LiveJournal and Hollywood, respectively, by reducing search and employing synopses.

Temporal queries. Fixing $|Q| = 4$ and varying t from $30\%t_m$ to $70\%t_m$, we evaluated temporal queries SubIso$_t$, TriC$_t$ and Dist$_t$ on LiveJournal. As shown in Figures 6(c)-6(d), (1) SubA$_c$ is on average 1.86 and 1.79 times faster than Turboiso and TurboIsoBoosted, respectively; VF2$_c$ outperforms VF2 by 8.08 times. (2) The average speedup for TriC and Dist is 1.52 and 2.35 times, respectively. (3) The speedup is larger for temporal queries than for conventional ones, as expected. (4) It is more substantial for larger t on SubIso$_t$.

The results verify that our contraction scheme (a) speeds up evaluation for all three applications, and (b) can be used together with existing algorithms, with indexing (*e.g.,* Turboiso) or not (*e.g.,* VF2$_c$). (c) It is effective by separating up-to-date data from obsolete.

Graph	SubIso			TriC			Dist		
	clique	star	path	clique	star	path	clique	star	path
Twitter	0.44	0.09	0	0.16	0.19	0	0.27	0.27	0
LiveJournal	0.08	0.03	0	0.17	0.02	0.01	0.44	0.13	0.03
LivePokec	0.58	0.11	0	0.03	0.23	0	0.3	0.24	0.002
Google	0.45	0.2	0	0.33	0.18	0.03	0.42	0.15	0
NotreDame	0.71	0.19	0.02	0.5	0.36	0	0.47	0.26	0
DBLP	0.72	0.17	0.02	5.75	2.19	0	0.26	0.37	0.01
Hollywood	0.13	0.02	0.01	0.23	0.11	0	0.24	0.26	0.01
citHepTh	0.56	0.16	0	0.17	0.08	0	0.32	0.23	0.03
Traffic	0.11	0.25	0.04	0.01	0.21	0.1	0.001	0.09	0.06

Table 3: Slowdown(s) by disabling topological component

Exp-3: Impact of each component. We next evaluated the impact of contracting each of clique, star and path.

Impact of topological components. We took contraction of all the 3 topological components as the baseline (unit 1), and tested the impact of each component in query evaluation time by disabling it, using all the datasets. As shown in Table 3, the average slowdown in evaluation time by disabling clique, star and path is: (a) 27.3%, 9.9% and 0.6% for SubIso, (b) 19.6%, 11.5% and 0.2% for TriC, and (c) 21.7%, 17.2% and 3.0% for Dist, respectively. We can see that clique has the biggest impact on SubIso due to its high pruning power.

Impact of obsolete components. We tested the impact of obsolete data on conventional queries. Fixing $|Q| = 4$ and varying x for timestamp threshold $t_0 = x\% t_m$, Figure 6(e) reports the runtime of SubIso on Hollywood. We find that (1) the speedup is bigger for larger t_0 when $t_0 \leq 70\%$, *i.e.,* more nodes are contracted into obsolete components; (2) obsolete components speed up SubIso, TriC and Dist by 1.32, 1.16 and 1.21 times, respectively; and (3) the speedup for SubIso gets smaller when $t_0 \geq 80\%$ due to the overhead of decontracting obsolete components. The results are consistent for Dist and TriC, except that their speedup does not go down when t_0 gets larger since they do not decontract obsolete supernodes.

Impact of k_l and k_u. We also tested the impact of k_l and k_u on the contraction ratio and efficiency. Fixing $k_u = 500$ (resp. $k_l = 4$) and varying k_l (resp. k_u) from 2 to 6 (resp. 20 to 1000), Figure 6(f) (resp. 6(g)) reports the *CR* (right y-axis) and speedup (left y-axis) of $\mathrm{SubA_c}$, $\mathrm{TriA_c}$ and DisAc on Hollywood. The CR decreases with the decrease of k_l and increase of k_u. Moreover, query evaluation is slowed down when $k_l \leq 3$ or $k_u \geq 500$ because of excessive superedge decontractions or overlarge components. Thus, we find that the best k_l and k_u for Hollywood are 4 and 500, respectively. The results on the other graphs are consistent (not shown).

Exp-4: Space cost. We next studied the space cost of our contraction scheme compared with indexing cost. The space cost includes the sizes of the contracted graph $|G_c|$, decontraction function $|f_D|$ and the sizes of synopses for active applications; as shown in Section 3, $\mathrm{SubA_c}$, $\mathrm{TriA_c}$ and DisAc do not need to decontract topological components; hence we only uploaded f_D for obsolete components into memory. Space cost of $\mathrm{SubA_c}$ also includes the size of adopted indexes I_{SubIso}. We compared with the three indices used by Turboiso, HINDEX [43] and PLL [3].

Table 4 shows how the space cost increases when more applications run on Google as G. We find the following. (1) Our contraction scheme takes totally 941MB for SubIso, TriC and Dist, much smaller than 9.58GB taken by Turboiso, PLL and HINDEX. (2) With the

Application	Contraction scheme		Indexing	
	detail	space cost	detail	space cost
Shared parts	G_c, f_D	837MB	G	727MB
+SubIso	$\mathcal{S}_{\mathrm{SubIso}}, I_{\mathrm{SubIso}}$	875MB	Turboiso	1.07GB
+TriC	$\mathcal{S}_{\mathrm{TriC}}$	901MB	+HINDEX	2.1GB
+Dist	$\mathcal{S}_{\mathrm{Dist}}$	941MB	+PLL	9.58GB
+MC	$\mathcal{S}_{\mathrm{MC}}$	1.05GB	+RMC	12.9GB
+kNN	$\mathcal{S}_{\mathrm{kNN}}$	1.18GB	+Antipole	19.4GB

Table 4: Total space cost of applications run on Google

contraction scheme, graph G is no longer needed. That is, compared to G, the scheme uses only 29.4% additional space for the supernodes/edges in G_c and synopses for three applications. The scheme trades affordable space for speedup. (3) Synopses S_{SubIso}, S_{TriC} and S_{Dist} take 11.1% of the total space of contraction, *i.e.,* G_c and f_D dominate the space cost, which are shared by all applications. Hence the more applications are supported, the more substantial the improvement of the contraction scheme is over indices. To verify this, we further adapted existing algorithms for maximum clique (MC) [38] and k-nearest neighbors (kNN) [50]. The total space cost of the contraction scheme for the five applications is 1.18GB, *i.e.,* 25% increment. It accounts for only 6.1% of the indices for Turboiso, PLL, HINDEX, RMC [38] of MC and Antipole [11] of kNN.

Exp-5: Efficiency of (incremental) contraction. We next evaluated the efficiency of both GCon and IncCR. We also studied the impact of the order and varied rates of updates on IncCR.

Efficiency of GCon. We first report the efficiency of GCon on liveJournal. As shown in Fig. 6(h), (1) on average it takes 37.7s to contract the graph. (2) It takes on average 1.63s, 8.75s, 4.93s only to compute the synopses for SubIso, TriC and Dist, respectively; *i.e.,* computing synopses only takes on average 13.5% of the time of GCon.

Efficiency of IncCR. We next tested the efficiency of IncCR, by varying $|\Delta G|$ from $5\%|G|$ to $35\%|G|$. As shown in Fig. 6(i) on liveJournal, (1) on average IncCR is 1.8 times faster than GCon, up to 5.1 times when $|\Delta G| = 5\%|G|$. It takes on average 13.3% time to update the synopses for 5% updates on the three applications. (2) IncCR beats GCon even when $|\Delta G|$ is up to $30\%|G|$. This justifies the need for incremental contraction. (3) IncCR is sensitive to $|\Delta G|$; it takes longer for larger $|\Delta G|$. Results are consistent on the other graphs.

Impact of update order. We tested the impact of the orders of edge insertions and deletions in ΔG on IncCR. Fixing $|\Delta G| = 10\%$, we varied the order of updates by (1) random (RO), (2) insertion-first (IF) and (3) deletion-first (DF). On average RO, IF and DF have a performance difference less than 3.6% on Hollywood. That is, IncCR is *stable* on batch updates, regardless of the order of single edges.

Impact of update rates. We also evaluated the efficiency of IncCR against real-time updates, measured by the updates coming in 1s intervals, *i.e.,* $|\Delta G|$/s. Varying $|\Delta G|$/s from $0.2\%|G|$/s to $1\%|G|$/s, Figure 6(j) show the following on LiveJournal. (1) On average it takes only 0.88s to update the graph. (2) The update time is less than 1s even when the updates are up to $0.8\%|G|$. Thus IncCR can handle $0.8\%|G|$ of "burst" updates on graph with 40M nodes and edges.

Exp-6: Scalability. Finally, we evaluated (1) the scalability of our contraction algorithm GCon with graph size $|G|$, and (2) the parallel scalability of algorithm PCon with the number of machines.

Scalability on $|G|$. Varying the size $|G| = (|V|, |E|)$ of synthetic graphs from $(2M, 20M)$ to $(10M, 100M)$, we tested the scalability of GCon. As shown in Fig. 6(k), GCon scales well when G grows. It takes 103s even when G has 10M nodes and 100M edges.

Scalability of PCon. We tested the scalability of algorithm PCon with the number k of machines, by varying k from 4 to 20. As shown in Fig. 6(l) on Hollywood, PCon scales well with k by improving 3.8 times. The results on other graphs are consistent.

Summary. We find the following over 9 real-life graph. On average, (1) the contraction scheme reduces graphs by 71.2%. The contraction ratio is 0.351, 0.236, 0.221 and 0.401 in social networks, Web graphs, collaboration networks and road networks, respectively. (2) It improves the evaluation of SubIso, TriC and Dist by 1.53, 1.42 and 2.14 times, respectively. Existing algorithms can be adapted to the scheme, with indices or not. (3) Cliques, stars and paths improve the query evaluation by 22.9%, 12.9% and 1.3%, respectively. (4) Contracting obsolete data improves the efficiency of both conventional queries and temporal queries, by 1.23 and 1.88 times on average, respectively. (5) Its total space cost on SubIso, TriC and Dist is only 9.8% of indexing costs of Turboiso, PLL and HINDEX. The synopses for the three query classes take only 11.1% of the total space. Thus our contraction scheme scales with the number of applications. (6) Algorithms GCon, PCon and IncCR scale well with graphs and updates. GCon takes 103s when G has 110M edges and nodes, and PCon takes 9.7s with 20 machines. IncCR is 5.1 times faster than GCon when $|\Delta G|$ is $5\%|G|$, and is still faster up to $30\%|G|$.

6 RELATED WORK

Contraction. As a traditional graph programming technique [25], node contraction merges nodes, and subgraph contraction replaces connected subgraphs with supernodes. It is used in *e.g.,* single source shortest paths [30], connectivity [25] and spanning tree [23].

In contrast, we extend contraction with synopses to build a compact representation of graphs as a generic optimization scheme, which is a departure from programming techniques.

Compression. Graph compression has been studied for, *e.g.,* social network analysis [15], subgraph isomorphism [20, 39], graph simulation [22], reachability and shortest distance [29]. It computes query-specific equivalence relations by merging equivalent nodes into a single node. Some compression methods are query preserving (*i.e.,* lossless), *e.g.,* [22, 29, 39], and can answer particular types of queries on compressed graphs without decompression.

Our contraction scheme differs from compression in the following. (a) It allows multiple applications to share the same contracted graph. In contrast, compressed graphs are query dependent; no one supports different applications to run on the same compressed graph. (b) Contraction guarantees to be lossless, while some compression schemes are lossy, *e.g.,* [20]. (c) Existing algorithms can be readily adapted to contracted graphs. In contrast, compression often needs to develop new algorithms, *e.g.,* [39] demands a decompose-and-join algorithm for subgraph isomorphism.

Summarization. Graph summarization aims to produce an abstraction or summary of a large graph by aggregating nodes or subgraphs (see [37] for a survey), classified as follows. (1) Node aggregation, *e.g.,* GraSS [33] merges node clusters into supernodes labeled with the number of edges within and between the clusters; it is developed for adjacency, degree and centrality queries. SNAP [49] generates an approximate summary of a graph structure by aggregating nodes based on attribute similarity. (2) Edge aggregation, *e.g.,* [42] generates a summary by aggregating edges into superedges, with a bounded number of edges different from the original graph. (3) Simplification: instead of aggregating nodes and edges, OntoVis [47] drops low-degree nodes, duplicate paths and unimportant node labels. Most summarization methods are lossy, *e.g.,* GraSS and SNAP retain part of attributes, and OntoVis drops nodes, edges and labels.

Incremental maintenance of summarization has been studied in [17, 27, 48]. It depends on update intervals [48]; short-period summarization is space-costly, while long-interval summarization may miss updates. To cope with these, [27] aggregates updates into a graph of "frequent" nodes and edges, and computes an approximate summary based on all historical updates on entire graph.

Both summarization and contraction schemes aim to provide a generic graph representation to speed up graph analyses. However, (1) the contraction scheme is *lossless* and allows exact answers to be computed for various classes of queries. In contrast, summarization is typically lossy and supports at best certain aggregate or approximate queries only. (2) Existing algorithms for query answering can be readily adapted to contracted graphs, while new algorithms often have to be developed on top of graph summaries. (3) Contracted graphs can be incrementally maintained with boundedness and locality, while summarization maintenance requires historical updates and often operates on the entire graph [27].

Indexing. A variety of indices have been studied for, *e.g.,* subgraph isomorphism [8, 9, 16, 26, 41], reachability [5, 12, 29, 52] and shortest distance [13, 36]. Indices are query specific and take extra space.

Our contraction scheme differs from indexing in that it supports multiple applications on the same contracted graph, while a separate index has to be built for each query class. Moreover, it is more efficient to maintain contracted graphs than indices. This said, the contraction scheme can be complemented with indices for further speedup, as demonstrated by SubIso (Section 3.1).

7 CONCLUSION

We have proposed a contraction scheme to make big graphs small, as a generic optimization scheme for multiple applications to run on the same graph at the same time. We have shown that the scheme is generic and lossless. Moreover, it prioritizes up-to-date data by separating it from obsolete data. In addition, existing query evaluation algorithms can be readily adapted to compute exact answers, often without decontracting topological components. Our experimental results have verified that our scheme is effective.

One topic for future work is to explore what topological structures to contract for various types of graphs, besides path, star and clique. Another topic is to recursively apply the contraction scheme, and build a contraction hierarchy.

Acknowledgements. Fan, Li and Liu are supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/180014. Liu is also supported in part by EPSRC EP/L01503X/1, EPSRC CDT in Pervasive Parallelism at the University of Edinburgh. Lu is supported in part by NSFC 62002236.

REFERENCES

[1] 2006. Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml*.
[2] 2012. DBLP. *https://snap.stanford.edu/data/com-DBLP.html*.
[3] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*.
[4] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. The diameter of the World Wide Web. *CoRR* cond-mat/9907038 (1999).
[5] Shikha Anirban, Junhu Wang, and Md Saiful Islam. 2019. Multi-level graph compression for fast reachability detection. In *DASFAA*.
[6] Seung-Hee Bae, Daniel Halperin, Jevin D West, Martin Rosvall, and Bill Howe. 2017. Scalable and efficient flow-based community detection for large-scale graph analysis. *TKDD* 11, 3 (2017), 1–30.
[7] Maciej Besta and Torsten Hoefler. 2018. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *CoRR* abs/1806.01799 (2018).
[8] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *SIGMOD*.
[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*.
[10] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression techniques. In *WWW*. 595–602.
[11] Domenico Cantone, Alfredo Ferro, Alfredo Pulvirenti, Diego Reforgiato Recupero, and Dennis Shasha. 2005. Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. *TKDE* 17, 4 (2005), 535–550.
[12] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*.
[13] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SICOMP* 32, 5 (2003).
[14] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–1.
[15] Sara Cohen. 2016. Data management for social networking. In *SIGMOD*.
[16] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI* 26, 10 (2004), 1367–1372.
[17] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. 2001. Communities of interest. In *International Symposium on Intelligent Data Analysis*.
[18] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[19] David Dominguez-Sal, Norbert Martinez-Bazan, Victor Muntes-Mulero, Pere Baleta, and Josep Lluis Larriba-Pey. 2010. A discussion on the design of graph database benchmarks. In *Technology Conference on Performance Evaluation and Benchmarking*. 25–40.
[20] J. Fairey and L. Holder. 2016. StarIso: Graph Isomorphism Through Lossy Compression. In *2016 Data Compression Conference (DCC)*.
[21] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *SIGMOD*.
[22] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD*.
[23] Harold N Gabow, Zvi Galil, and Thomas H Spencer. 1984. Efficient implementation of graph algorithms using contraction. In *FOCS*.
[24] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.
[25] Jonathan Gross and Jay Yellen. 1998. *Graph Theory and its applications*. CRC Press.
[26] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turbo$_{\text{iso}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*.
[27] Shawndra Hill, Deepak K Agarwal, Robert Bell, and Chris Volinsky. 2006. Building an effective representation for dynamic networks. *Journal of Computational and Graphical Statistics* 15, 3 (2006), 584–608.
[28] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *SIGMOD*.
[29] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*.
[30] Roozbeh Karimi, David M Koppelman, and Chris J Michael. 2019. GPU road network graph contraction and SSSP query. In *ICS*.
[31] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SISC* 20, 1 (1998), 359–392.
[32] Walter Kropatsch. 1996. Building irregular pyramids by dual-graph contraction. In *Vision Image and Signal Processing*.
[33] Kristen LeFevre and Evimaria Terzi. 2010. GraSS: Graph structure summarization. In *SDM*.
[34] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *SIGKDD*.
[35] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR* abs/0810.1355 (2008).
[36] Yongjiang Liang and Peixiang Zhao. 2017. Similarity search in graph databases: A multi-layered indexing approach. In *ICDE*.
[37] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. 2018. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv.* 51, 3 (2018), 62:1–62:34.
[38] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the maximum clique in massive graphs. *PVLDB* 10, 11 (2017), 1538–1549.
[39] Antonio Maccioni and Daniel J Abadi. 2016. Scalable pattern matching over compressed graphs via dedensification. In *SIGKDD*.
[40] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NIPS*.
[41] Han Myoungji, Kim Hyunjoon, Gu Geonmo, Park Kunsoo, and Han Wook-Shin. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *SIGMOD*.
[42] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. 2008. Graph summarization with bounded error. In *SIGMOD*.
[43] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *IEEE High Performance Extreme Computing Conference*. 1–7.
[44] Ganesan Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *TCS* 158, 1-2 (1996), 233–277.
[45] Xuguang Ren and Junhu Wang. 2015. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *PVLDB* 8, 5 (2015), 617–628.
[46] Sherif Sakr and Ghazi Al-Naymat. 2010. Graph indexing and querying: a review. *IJWIS* 6, 2 (2010), 101–120.
[47] Zeqian Shen, Kwan-Liu Ma, and Tina Eliassi-Rad. 2006. Visual analysis of large heterogeneous social networks by semantic and structural abstraction. *TVCG* 12, 6 (2006), 1427–1439.
[48] Sucheta Soundarajan, Acar Tamersoy, Elias B Khalil, Tina Eliassi-Rad, Duen Horng Chau, Brian Gallagher, and Kevin Roundy. 2016. Generating graph snapshots from streaming edge data. In *WWW*.
[49] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. 2008. Efficient aggregation for graph summarization. In *SIGMOD*.
[50] Yubao Wu, Ruoming Jin, and Xiang Zhang. 2016. Efficient and exact local search for random walk based top-k proximity query in large graphs. *TKDE* 28, 5 (2016), 1160–1174.
[51] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*.
[52] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *PVLDB* 3, 1–2 (2010), 276–284.