The VLDB Journal
https://doi.org/10.1007/s00778-022-00731-7

REGULAR PAPER

Making graphs compact by lossless contraction

Wenfei Fan[1,2,3] · Yuanhao Li[1] · Muyang Liu[1] · Can Lu[2]

Received: 4 April 2021 / Revised: 27 September 2021 / Accepted: 30 December 2021

Abstract
This paper proposes a scheme to reduce big graphs to small graphs. It contracts obsolete parts and regular structures into supernodes. The supernodes carry a synopsis $S_\mathcal{Q}$ for each query class $\mathcal{Q}$ in use, to abstract key features of the contracted parts for answering queries of $\mathcal{Q}$. Moreover, for various types of graphs, we identify regular structures to contract. 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, shortest distance, connected component and clique decision to contracted graphs. We also provide a bounded incremental contraction algorithm in response to updates, such that its cost is determined by the size of areas affected by the updates alone, not by the entire graphs. We experimentally verify that on average, the contraction scheme reduces graphs by 71.9% and improves the evaluation of these queries by 1.69, 1.44, 1.47, 2.24 and 1.37 times, respectively.

Keywords Graph data management · Graph contraction · Graph algorithms · Incremental computation

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 (cf. [42]). These highlight the need for developing techniques for speeding up graph computations.

✉ Yuanhao Li
yuanhao.li@ed.ac.uk

Wenfei Fan
wenfei@inf.ed.ac.uk

Muyang Liu
muyang.liu@ed.ac.uk

Can Lu
lucan0811@gmail.com

[1] University of Edinburgh, Edinburgh, UK

[2] Shenzhen Institute of Computing Sciences, Shenzhen, China

[3] BDBC, Beihang University, Beijing, China

There has been a host of work on the subject, either by making graphs compact, *e.g.,* graph summarization [67] and compression [12,82], or speeding up query answering by building indices [81]. The prior work often targets a specific class of queries, *e.g.,* query-preserving compression [37] and 2-hop labeling [25] 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 or summaries 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 incur heavy time and space costs and often obscure 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 *queries of different classes 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 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 and organization. In this paper, we propose a new approach to tackling these challenges, by extending the idea of graph contraction.

(1) A contraction scheme (Sect. 2). We propose a contraction scheme to reduce big graphs into smaller ones. It contracts obsolete components and regular structures 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 conventional 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 class $\mathcal{Q}$. Only $S_{\mathcal{Q}}$ may vary for different query classes $\mathcal{Q}$. We identify regular structures to contract in different types of graphs, and develop a (parallel) contraction algorithm.

(2) Proof of concept (Sect. 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. When answering 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 five different query classes: subgraph isomorphism (SubIso), triangle counting (TriC), shortest distance (Dist), connected component (CC) and clique decision (CD) based on the following dichotomies:

- label-based queries (SubIso) versus non-label based ones (TriC, Dist, CC, CD);
- local queries (SubIso, TriC, CD) versus non-local ones (Dist, CC); and
- various degrees of topological constraints (Dist$\prec$ CC $\prec$ TriC $\prec$ CD$\prec$ SubIso).

We show how easy to adapt existing algorithms for these query classes 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 (Sect. 4). We develop an incremental algorithm for maintaining contracted graphs in response to updates to original graphs. Such updates may change both the topological structures and timestamps (obsolete data). We show that the algorithm is *bounded* [77], *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. We parallelize the algorithm to scale with large graphs.

(4) Empirical evaluation (Sect. 5). Using 10 real-life graphs, we experimentally verify the following. On average, (a) the contraction scheme reduces graphs by 71.9%, up to 86.0%. (b) Contraction makes SubIso, TriC, Dist, CC and CD 1.69, 1.44, 1.47, 2.24 and 1.37 times faster, respectively. (c) The total space cost of our contraction scheme for the five accounts only for 12.6% of indices for TurboIso [44], HINDEX [75], PLL [4] and RMC [68]. It is 9.0% when kNN [92] also runs on the same graph. The synopses for each take 9.7% 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 1.64 and 1.78 times on average, respectively. (e) Our (incremental) contraction scheme scales well with graphs, *e.g.,* it takes 33.1s to contract graphs of 1.8B edges and nodes with 20 cores.

We survey related work in Sect. 6 and identify research topics for future work in Sect. 7.

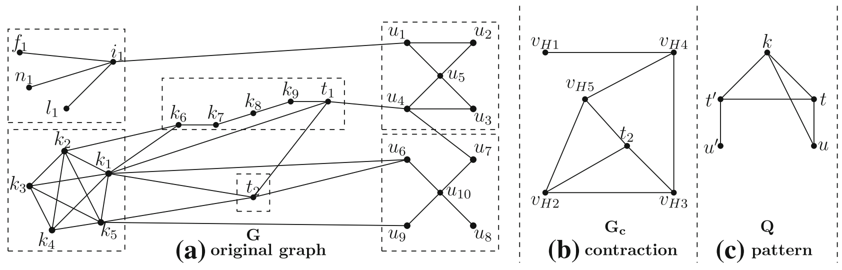
2 A graph contraction scheme

In this section, we first present the graph contraction scheme (Sect. 2.1). We then identify topological components to contract for different types of real-life graphs (Sect. 2.2). Moreover, we develop a contraction algorithm (Sect. 2.3) and its parallelization (Sect. 2.4).

Preliminaries. We start with basic notations.

Graphs. Assume two infinite sets Θ and Γ for labels and timestamps, respectively. 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 such that 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 number, a graph, or a relation. For instance, a *graph pattern matching* query is a graph pattern Q to find the set of subgraphs in



node v_H	v_H.type	
v_{H1}	obsolete	
v_{H2}	clique	
v_{H3}	star	$v_{H3}.c = u_{10}$
v_{H4}	butterfly	$v_{H4}.s = u_5$, $v_{H4}.e = (u_1, u_2), (u_3, u_4)$
v_{H5}	path	$v_{H5}.\mathbf{list}=\langle k_6, \ldots, k_9, t_1\rangle$

$\mathbf{S_{SubIso}}$

(d) synopses for SubIso(timestamp not shown)

Fig. 1 Graph contraction

G that are isomorphic to pattern Q, denoted by $Q(G)$. A *query class* $\mathcal{Q}$ is a set of queries of the same "type," *e.g.,* all graph pattern queries. 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. 1a 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 accounts of the same user in other social networks (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; (7) (i, l), a user has a feature; or (8) (i, f), a user has f followers.

In G, subgraphs in dashed rectangles are contracted into supernodes, yielding a contracted graph G_c shown in Fig. 1b. Synopses S_{SubIso} for SubIso are shown in Fig. 1d and are elaborated in Sect. 3.1. □

Before we formally define functions f_C, f_D and synopsis $\mathcal{S}$, 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 shown in Sect. 3, an existing algorithm $\mathcal{A}$ for a query class $\mathcal{Q}$ can be readily adapted to contracted graph and computes exact query answers.

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 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.

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

(b) *Topological component*: a subgraph with a regular structure, *e.g.,* clique, star, path and butterfly.

Different types of graphs have different regular substructures, *e.g.,* cliques are ubiquitous and effective in social networks while paths are only effective in road networks. In Sect. 2.2, we will identify what regular structures H to contract in different types of graphs.

We contract subgraphs with the number of nodes in the range $[k_l, k_u]$ to avoid over-contraction (see Sects. 2.3 and 5 for the choices).

Contraction function f_C maps each node v in graph G to a supernode in contracted graph G_c, which is either a supernode 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 $f_C(t_2) = t_2$.

Obsolete components help us prioritize up-to-date data, and topological ones reduce unnecessary checking when answering queries. As shown in Sect. 5, on average the first three regular structures and obsolete components contribute 18.3%, 14.9%, 2.8% and 63.1% to the contraction ratio, and speeds up query answering by 1.61, 1.44, 1.04 and 1.71 times, 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, function f'_C maps each supernode in contracted graph G_c of Fig. 1b back to the nodes in the corresponding rectangle in Fig. 1a, *e.g.,* $f'_C(v_{H1}) = \{(i_1, \text{id}), (n_1, \text{name}), (f_1, \text{follower}), (l_1, \text{link})\}$.

Intuitively, the reverse function f'_C recovers the contracted nodes and their associated labels, while the decontraction function f_D restores the topological structures of the contracted subgraphs.

(3) Synopsis. For each query class $\mathcal{Q}$ in use, a synopsis function S_Q is in $\mathcal{S}$, to retain features necessary for answering queries in $\mathcal{Q}$. For instance, when $\mathcal{Q}$ 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 Sect. 3. As will also be seen there, f'_C and synopses S_Q taken together often suffice to answer queries in $\mathcal{Q}$, without decontraction.

Note that not every synopsis S_Q has to reside in memory. We load S_Q to memory only if its corresponding application $\mathcal{Q}$ is currently in use.

(4) Decontraction. Function f_D restores contracted subgraphs. For *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 *superedge* (v_{H1}, v_{H2}), $f_D(v_{H1}, v_{H2})$ recovers the edges between $f'_C(v_{H1})$ and $f'_C(v_{H2})$.

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 scheme is guaranteed lossless.

For example, decontraction function f_D restores the subgraph in Fig. 1a from supernodes, *e.g.,* $f_D(v_{H3})$ is a star with central node u_{10} and leaves u_6, u_7, u_8 and u_9. It also restores edges from superedges, *e.g.,* $f_D(v_{H2}, v_{H5}) = \{(t_1, k_1), (k_1, k_6), (k_2, k_6)\}$.

2.2 Identifying regular structures

We now identify what regular structures to contract for *different types* of real-life graphs.

Different types of graphs. We investigated the following 10 different types of graphs: (1) social graphs: Twitter [70] and LiveJournal [94]; (2) communication networks: WikiTalk [62]; (3) citation networks: HepTh [63] and Patent [63]; (4) Web graphs: Google [64] and NotreDame [5]; (5) knowledge graphs: DBpedia [61] and WordNet [71]; (6) collaboration networks: DBLP [2] and Hollywood [15]; (7) biomedical graphs: Mimic [51]; (8) economic networks: Poli [80]; (9) chemical graphs: Enzymes [80]; and (10) road networks: Traffic [1].

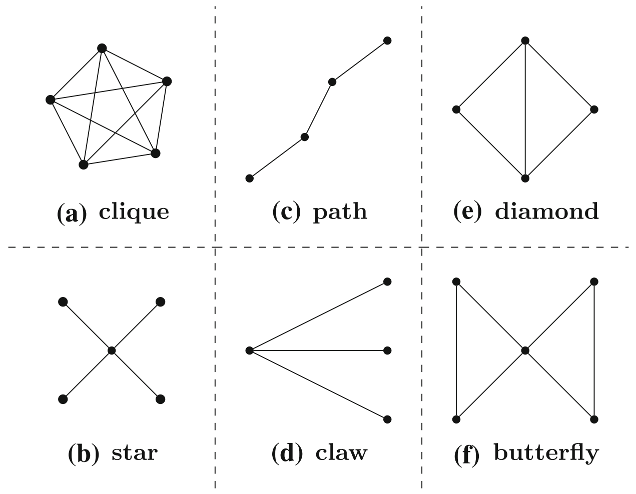


Fig. 2 Frequent regular structures

Regular structures. For a certain type of graphs G, we apply a subgraph mining model $\mathcal{M}$ to G. It returns a set of frequent subgraphs $\mathcal{M}(G) = \{g_1, g_2, ...\}$ of G together with the *support* of each g_i. Support metrics may vary in different mining models, *e.g.,* GRAMI [33] adopts *minimum image based* metric [19]. We pick subgraphs whose supports are above a threshold t_s.

As an example, we adopt a subgraph miner GRAMI [33] as $\mathcal{M}$. GRAMI discovers all the frequent subgraphs in G that have a support above a predefined threshold, which are then manually inspected. We pick g_i's with at least 4 nodes to avoid over-contraction.

As shown in Fig. 2, we found the following 6 structures in the 10 types of graphs: (a) clique: a fully-connected graph; (b) star: a single central node with neighbors; (c) path: a sequence of connected nodes with no edges between the head and tail (its two endpoints); (d) claw: a special star in which the central node has exactly 3 neighbors, denoted as its *leaves*; claws are quite frequent and are hence treated separately; (e) diamond: two triangles that share two endpoints; and (f) butterfly: two triangles sharing a single node.

Note that within these structures H, the only edges allowed are those that form H. Moreover, edges are allowed from each node in H to nodes outside of H. The only exception is that for a path, only the two endpoints can connect to other nodes in the graph.

Table 1 Common structures in different types of graphs

Graph type	Regular structure
Social graphs	Clique, star, diamond, butterfly, path
Communication networks	Star
Citation networks	Clique, star, diamond, butterfly
Web graphs	Star, clique, diamond
Knowledge graphs	Star, claw
Collaboration networks	Clique, star, diamond
Biomedical graphs	Star, clique, path
Economic networks	Star
Chemical graphs	Claw, path
Road networks	Star, claw, path

We summarize how these structures appear in the 10 *types* of graphs in Table 1, ordered by supports and importance from high to low. Note that different graphs have different frequent regular substructures. Cliques, stars and diamonds often occur in social graphs, while in road networks, stars, claws and paths are frequent.

Note that frequent pattern mining is conducted *once for each type* of graphs offline, *not* for each input graph. For instance, we always contract cliques, stars, diamonds, butterflies and paths for social graphs.

```
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.  $T(G)$ := precomputed regular-structure list by the type of $G$;
3.  for each $t \in T(G)$ do
4.    contract topological components ($[k_l, k_u]$) of type $t$;
5.  deduce $f_C$ and $f_D$;
6.  return $f_C$ and $f_D$;
```

Fig. 3 Algorithm GCon

2.3 Contraction algorithm

We next present an algorithm to contract a given graph G, denoted as GCon and shown in Fig. 3.

A tricky issue is that the contracted graphs depend on the order on the regular structures contracted. For example, if we contract diamonds first in the Twitter graph G_0 of Fig. 1a, then it contracts $\{t_2, k_1, k_5, k_3\}$ as a diamond; after this there are no cliques in G_0. In contrast, if cliques are contracted first, then $\{k_1, k_2, k_3, k_4, k_5\}$ is extracted. As suggested by $\mathcal{M}$, cliques "dominate" in social graphs and hence should be "preserved" when contracting G_0.

We adopt a *deterministic* order to ensure that important structures are contracted earlier and hence preserved. We order the importance of different types of regular structures in a graph G by their supports: the higher the support is, the more important the topology is. We denote by $T(G)$ its ordered set of regular structures to contract in Table 1. Note that $T(G)$ is determined by *the type* of G, *e.g.,* social graphs, and is learned *once offline* regardless of individual G.

Given a graph G, algorithm GCon first contracts all obsolete data into components 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 then either contracted into topological components, or are left as singletons.

Putting these together, we present the main driver of algorithm GCon in Fig. 3. 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 t_0 into obsolete components (line 1). It then recalls the list $T(G)$ of topological components to contract based on the type of graph G (line 2). Next, GCon contracts topological components into supernodes following order $T(G)$, and deduces f_C and f_D accordingly (lines 3-5). Each topological component consists of only uncontracted nodes. More specifically, it does the following.

(1) It extracts a clique by repeatedly selecting an un-contracted node that connects to all selected ones, subject to pre-selected size bounds k_l and k_u (see below).

(2) It extracts a star by first picking a central node v_c, and then repeatedly selecting an un-contracted node as a leaf that is (a) connected to v_c and (b) disconnected from all selected leaves, again subject to k_l and k_u.

(3) For paths, it first extracts intermediate nodes having only two neighbors that are not linked by an edge. It then finds a path consisting of only the intermediate nodes, along with two neighbors of the endpoints.

(4) For diamonds, it first selects an edge (u, v) and then picks x and y that are (a) connected to both u and v, and (b) pairwise disconnected.

(5) For butterflies, it first selects a node v that has a degree at least 4. It then checks whether there exist four neighbors u, x, y, z of node v such that exactly (u, x, v) and (y, z, v) form two triangles.

(6) For claws, it selects nodes with exactly 3 neighbors, and there is no edge between any two neighbors.

As remarked earlier, the remaining nodes that cannot be contracted into any component as above are treated as singleton, *i.e.,* mapped to themselves by f_C.

Example 2 Assume that timestamp threshold t_0 for graph G of Fig. 1a 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, star, diamond, butterfly and path in this order. (3) It builds a clique v_{H2} with nodes $k_1, \ldots, k_5$. (4) It picks u_{10} and u_5 as central nodes for a star, and makes a star v_{H3} consisting of $u_6, u_7, u_8, u_9, u_{10}$. Nodes u_5, u_1, u_3 cannot make a star due to lower bound $k_l = 4$. (5) No diamond exists. (6) It picks u_5 as central node for a butterfly and makes a butterfly v_{H4}. (7) It finds k_7, k_8 and k_9 as candidate intermediate nodes for paths, and contracts them into a path v_{H5} with endpoints k_6 and t_1. (8) Node t_2 is left as a singleton, and is mapped to itself by f_C. □

Range $[k_l, k_u]$. We contract an (obsolete/topological) component H such that the number of its nodes is in the range $[k_l, k_u]$. The reason is twofold. (1) If H is too small, a contracted graph would have an excessive number of supernodes; this leads to over-contraction with high overhead for possible decontraction and low contraction ratio. Thus, we set a lower bound k_l. (2) We set an upper bound k_u to avoid overlarge components and excessive superedge decontraction. We experimentally find that the best k_l and k_u are 4 and 500, respectively.

Diamonds, butterflies and claws have a fixed size with 4, 5 and 4 nodes, respectively, in the range above.

Complexity. Algorithm GCon takes at most $O(|G|^2)$ time. Indeed, (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) it takes $O(|G|)$ time to contract each clique and $O(|G|^2)$ time for all cliques; and (4) similarly, the other regular structures can be contracted in $O(|G|^2)$ 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 on the same contracted graph at the same time. This is often necessary. For instance, on average 10 classes of queries run on a graph simultaneously in GDB benchmarks [32]. (3) It *prioritizes up-to-date data* by separating it from obsolete data. (4) It improves performance. (a) As discussed in Sect. 5, $|G_c| \ll |G|$. In particular, each obsolete component is contracted into a single supernode. (b) Decontraction is often not needed. As shown in Sect. 3, none of SubIso, CD, TriC, Dist and CC needs to decontract any topological component, and for TriC, Dist and CC, even obsolete components do not need decontraction.

```
Algorithm PCon
Input: A edge-cut fragment $F_i(V_i, E_i)$, timestamp threshold $t_0$,
    range $[k_l, k_u]$.
Output: Contraction function $f_C$, decontraction function $f_D$.
1. conduct GCon locally in parallel with $n$ workers;
2. for each uncontracted border node $v$ do
3.   build uncontracted neighbors of at most $k_u$ nodes;
4. master $M_0$ merges uncontracted subgraphs by overlapping;
5. $M_0$ distributes disjoint uncontracted neighbors to workers;
6. conduct GCon on assigned uncontracted neighbors;
7. return $f_C$ and $f_D$ deduced by GCon;
```

Fig. 4 Algorithm PCon

2.4 Parallel contraction algorithm

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

Parallel setting. Assume a master (processor) M_0 and n workers (processors) $P_1, \ldots, P_n$. Graph G is partitioned into n fragments $F_1, \ldots, F_n$ by an edge-cut partitioner [17,55], and the fragments are distributed to n workers $P_1, \ldots, P_n$, respectively. We adopt the BSP model [88], which separates iterative computations into supersteps and synchronizes states after each superstep.

Parallel contraction algorithm PCon. As shown in Fig. 4, the idea of PCon is to leverage data-partitioned parallelism. PCon first conducts GCon locally on each fragment in parallel, and then contracts uncontracted "border nodes," *i.e.,* nodes with edges crossing fragments, by building neighbors of at most k_u uncontracted nodes, referred to as *uncontracted neighbors*, which are subgraphs with uncontracted nodes.

More specifically, algorithm PCon works as follows.

(1) All workers run GCon on its local fragment in parallel (line 1), since after all, each fragment F_i is a graph itself.

In contrast with single-thread GCon, workers do not contract mirror nodes, *i.e.,* nodes assigned to other fragments with edges linked to the local fragment. Adopting edge-cut partition, each node of G is assigned to a single fragment and is contracted at most once during GCon.

(2) PCon contracts "border nodes" (line 2-3). For each border node v, if v is not contracted, PCon builds it uncontracted neighbors. Such neighbors are identified in parallel, coordinated by master M_0.

(3) Master M_0 merges overlapped neighbors into one, and distributes disjoint ones to n workers (line 4-5). In this way, PCon reduces communication cost and speeds up the process when contracting border nodes.

(4) Each worker contracts its assigned uncontracted-neighbors of border nodes, in parallel (line 6).

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 topological components 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 ensured *compact*, *i.e.,* they cannot be contracted further.

3 Proof of concept

In this section, we show that existing query evaluation algorithms can be readily adapted to the contracted graphs. As a proof of concept, we pick five query classes: (1) graph pattern matching SubIso via subgraph isomorphism (labeled queries with locality); (2) triangle counting TriC (un-labeled queries with locality); (3) shortest distance Dist (un-labeled and non-local queries); (4) connected component CC (un-labeled queries without locality); and (5) clique decision CD (un-labeled queries with locality). Among these, subgraph isomorphism and clique decision are intractable (cf. [42]).

Informally, when answering 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 to process Q at v_H as a whole or safely skip v_H. Thus, it suffices to answer queries in the five classes by decontracting superedges, without decontracting any topological components. Here *decontraction* $f_D(v_{H1}, v_{H2})$ of a superedge (v_{H1}, v_{H2}) restores the edges between $f'_C(v_{H1})$ and $f'_C(v_{H2})$ (Sect. 2).

The main result of this section is as follows.

Theorem 1 *Using **linear** synopsis functions,*

(1) for each of SubIso *and* CD*, there are existing algorithms that can be adapted to compute exact answers on contracted graphs G_c, which decontract only supernodes of obsolete components and superedges between supernodes, **not any topological components**;*

(2) for TriC *and* Dist*, there are existing algorithms that can be adapted to G_c and decontract no supernodes, **neither topological nor obsolete components**; and*

(3) for CC*, there are existing algorithms that can be adapted to G_c and decontract **neither supernodes (topological and obsolete) nor superedges**.* □

Below we provide a constructive proof for Theorem 1 by adapting existing algorithms of the five query classes to contracted graphs one by one.

3.1 Graph pattern matching with contraction

We start with graph pattern matching (SubIso).

Preliminaries. We first review basic notations.

<u>*Pattern*</u>. A *graph pattern* is defined as a graph $Q = (V_Q, E_Q, L_Q)$, where (1) V_Q is a set of *pattern nodes*, (2) E_Q is a set of *pattern edges*, and (3) L_Q is a function that assigns a label $L_Q(u)$ to each $u \in V_Q$.

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

To simplify the discussion, we consider connected patterns Q. This said, our algorithm can be adapted to disconnected ones. We denote by u, v pattern nodes in pattern Q, and by x, y nodes in graph G. A *neighbor* of node v is a node such that $(u, v) \in E_Q$.

<u>*Pattern matching*</u>. 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. We denote by $Q(G)$ the set of all matches of pattern Q in graph 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 temporal pattern (Q, t) and a graph G, denoted by SubIso$_t$.

Graph pattern matching is widely used in graph queries [6,40,79,90] and graph dependencies [36,39].

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, v_2 in G', v_1 and v_2 are within d_Q hops when treating G' as undirected. 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. [42]); similarly for temporal matching. A variety of algorithms have been developed for SubIso, notably TurboIso [44] with indices and VF2 [28] without index. Both TurboIso and VF2 can be adapted to contracted graphs as characterized in Theorem 1.

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.,* [14,78], and (2) it employs indexing to reduce redundant matches; by adapting TurboIso we show that the indices for SubIso can be inherited by contracted graphs, *i.e.,* contraction and indexing complement each other. The same algorithm works for temporal matching. The proof for VF2 is simpler (not shown).

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

3.1.1 Contraction for SubIso

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

The synopsis of a supernode v_H for query class 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;
- diamond: v_H.type = diamond, $v_H.s_1$ and $v_H.s_2$ store the two share nodes of the two triangles;
- butterfly: v_H.type =butterfly, $v_H.s$ records the node shared by the two triangles, and $v_H.e$ stores the two disjoint edges;
- claw: v_H.type =claw, $v_H.c$ stores the central node and $v_H.s_i$ ($i \in [1, 3]$) record its three neighbors;
- 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.

Node labels are stored in the reverse function f'_c of the contraction function f_c (see Sect. 2.1).

For instance, the synopsis $S_{\mathsf{SubIso}}(v_H)$ for each supernode v_H in the contracted graph G_c of Fig. 1b is given in Fig. 1d. Note that S_{SubIso} only stores the synopses of the regular structures contracted in a graph.

Properties. 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 contracted graph G_c, as a byproduct.

```
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.      $CR_0$ := ExploreCR($v_s, x_s$);
5.      if $CR_0 \neq \emptyset$ then
6.          compute matching order $O(x_s, CR_0)$;
7.          $Q(G) := Q(G) \cup$ SGSearch($(x_s, v_s), Q, Q', G, O$);
8.  return $Q(G)$;
```

Fig. 5 Algorithm TurboIso

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

Below we first review algorithm TurboIso [44] and then show how to adapt TurboIso to contracted graphs.

TurboIso. As shown in Fig. 5, 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) that contains pattern nodes in Q having identically matching data vertices. Then, for each start vertex x_s of each region, TurboIso constructs a candidate region (CR_0), *i.e.,* an index that maintains candidates for each NEC vertex in Q', via DFS from x_s (lines 3-4). If valid candidates are found, *i.e.,* $CR_0 \neq \emptyset$, TurboIso enumerates all possible matches that map x_s to v_s following a matching order O (lines 5-6). The matching order O is decided by sorting the leaf NEC vertices based on the number of their candidate vertices. It expands $Q(G)$ with valid matches identified in the process (line 7).

Algorithm SubA$_\mathsf{c}$. TurboIso can be easily adapted to contracted graph G_c, denoted by SubA$_\mathsf{c}$. As shown in Fig. 6, SubA$_\mathsf{c}$ 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_0 based on v_H.type, *e.g.,* a node u in Q cannot match intermediate nodes on paths if u is in some triangle in Q; and u

```
Algorithm SubA_c
Input: Contracted $G_c$, contraction 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.      $CR_0 := ExploreCR(v_s, x_s, f'_C, S_{\mathsf{SubIso}})$;
5.      if $CR_0 \neq \emptyset$ then
6.        compute matching order $O(x_s, CR_0)$;
7.        $Q(G) := Q(G) \cup$
              $SGSearch((x_s, v_s), Q, Q', G_c, O, f'_C, S_{\mathsf{SubIso}}, f_D)$;
8.  return $Q(G)$;
```

Fig. 6 Algorithm $\mathsf{SubA_c}$

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 or claw, (x, y) exists only if $x = v_H.c$ or $y = v_H.c$; (b) if v_H.type = clique, (x, y) always exists; (c) if v_H.type=path, (x, y) exists if x and y are next to each other in v_H.list; (d) if v_H.type=diamond, (x, y) exists if at least one of x and y is the shared node $v_H.s_1$ or $v_H.s_2$; and (e) if v_H.type=butterfly, (x, y) exists if x and y are not endpoints of the two disjoint edges in $v_H.e$ simultaneously. Hence, no topological component is decontracted by f_D. (f) 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. 1c is to find potential friendship between users based on the retweet and shared keywords in their posted tweets. Nodes u and u' both have the same label u. Given Q, $\mathsf{SubA_c}$ first chooses k as the start node, to which only v_{H2} and v_{H5} can match. For v_{H2}, ExploreCR adds v_{H5} and t_2 as candidates for t and t', v_{H3} as candidate for u, and v_{H3} and v_{H4} as candidates 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 exists no edge between v_{H3} and v_{H5}. Thus, it matches k, t, u, t', u' with k_1, t_2, u_6, t_1, u_4.

Similarly for v_{H5}, ExploreCR adds v_{H5} and t_2 as candidates for t and t', v_{H4} as candidate for u, and v_{H3} and v_{H4} for u and 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. One can easily verify that $\mathsf{SubA_c}$ is correct since it has the same logic as TurboIso except that it incorporates pruning strategies. While they have the same worst-case complexity, $\mathsf{SubA_c}$ operates on G_c, much smaller than G (see Sect. 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_0 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 [26,47], which has been used in clustering [91], cycle detection [48] and transitivity [74]. 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, it consists of a single query and is not labeled.

We adapt algorithm TriA of [26] for TriC to contracted graphs, since it is one of the most efficient TriC algorithms [47], and it does not use indexing (as a different example from TurboIso). We show that for TriC, the adapted algorithm needs to decontract no supernodes, neither topological components nor obsolete parts.

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 for TriC 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 supernode v_H contracts a subgraph H with node set $V(H)$, *i.e.,* when the three nodes of a triangle are contracted into the same supernode, then (a) when H is a clique, there are $\binom{|V(H)|}{3}$ triangles inside H; (b) when H is a diamond or a butterfly, there are 2 triangles inside H; (c) when H is an obsolete component, then the number of triangles inside H can be pre-calculated, denoted by t_H; and (d) 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, *i.e.,* if two nodes of a triangle are contracted into the same supernode, then (a) when I is a clique, then w leads to $\binom{k}{2}$ triangles, where k is the

number of the neighbors of w in I. Denote by t^I_w the number of such triangles in a clique neighbor I of w. (b) Subgraph I cannot be a path since intermediate nodes on a path are not allowed to connect to nodes outside I. (c) Otherwise, nodes u and v yield k triangles, where k is the number of common neighbors of u and v in J. We denote by $t^J_{u,v}$ the number of such triangles in a common neighbor J of u and v.

(3) If $f_C(u) = v_I$, $f_C(v) = v_J$, $f_C(w) = v_K$, *i.e.,* when the three nodes of a triangle are contracted into different supernodes, we count such triangles online and it suffices to decontract only superedges, not supernodes.

Synopsis $S_{\mathsf{TriC}}(v_H)$ of 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. Below we use u and v to range over nodes in $V(H)$, I to range over clique neighbors of u, and J to range over common neighbors of u, v. We define t^I_u, t_H and $t^J_{u,v}$ as above.

In a clique H, there are (1) $\binom{|V(H)|}{3}$ triangles; (2) each node $u \in H$ has t^I_u triangles with its clique neighbor I; hence, $v_H.\mathsf{tc} = \binom{|V(H)|}{3} + \Sigma_u \Sigma_I t^I_u$. We can calculate v_H.tc similarly for other regular structures. Thus,

- clique: $v_H.\mathsf{tc} = \binom{|V(H)|}{3} + \Sigma_u \Sigma_I t^I_u$;
- star: $v_H.\mathsf{tc} = \Sigma_u \Sigma_I t^I_u + \Sigma_u \Sigma_J t^J_{v_H.c,u}$;
- path: $v_H.\mathsf{tc} = \Sigma_I t^I_{u_1} + \Sigma_I t^I_{u_{|V(H)|}}$, where u_1 and $u_{|V(H)|}$ are the first and last node on the path;
- claw: $v_H.\mathsf{tc} = \Sigma_u \Sigma_I t^I_u + \Sigma_{u,v} \Sigma_J t^J_{u,v}$;
- diamond and butterfly: $v_H.\mathsf{tc} = 2 + \Sigma_u \Sigma_I t^I_u + \Sigma_{u,v} \Sigma_J t^J_{u,v}$,
- obsolete: $v_H.\mathsf{tc} = t_H + \Sigma_u \Sigma_I t^I_u + \Sigma_{u,v} \Sigma_J t^J_{u,v}$.

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

Example 4 In the contracted graph G_c of Fig. 1b, 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 exists for any connected $u, v \in V(H1)$; thus, $t^J_{u,v} = 0$. Hence, $v_{H1}.\mathsf{tc} = 0$. (2) For v_{H2}, v_{H2}.type=clique, $|V(H2)| = 5$ and no other supernodes in G_c are cliques. Hence, $v_{H2}.\mathsf{tc} = 10$. (3) For v_{H3}, u_6 and u_9 have only 1 neighbor in clique I; thus, $t^I_u = 0$; similarly, no J exists for any leaf u and $v_{H3}.c$; thus, $t^J_{v_{H3}.c,u} = 0$. Hence, $v_{H3}.\mathsf{tc} = 0$. (4) Similarly, $v_{H4}.\mathsf{tc} = 2$, $v_{H5}.\mathsf{tc} = 1$ and $t_2.\mathsf{tc} = 1$. □

3.2.2 Triangle counting

We now adapt algorithm TriA [26] to contracted graphs. The adapted algorithm is referred to as $\mathsf{TriA_c}$.

Algorithm 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 both topological and obsolete components), it simply accumulates v_H.tc without decontraction or enumeration. It only restores superedges when necessary.

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. One can verify that $\mathsf{TriA_c}$ 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.

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

3.3 Shortest distance with contraction

We next study the shortest distance problem.

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. In a graph G, a *path* p from v_0 to v_k is a sequence $\langle v_0, v_1, \ldots, v_k\rangle$ of nodes such that $(v_i, v_{i+1}) \in E$ for all $0 \le i < k$. The length of a path $p = (v_0, \ldots, v_k)$ in G is simply $\mathsf{sum}_{i\in[1,k]} W(v_{i-1}, v_i)$.

The *shortest distance problem*, denoted by Dist, is to compute, given a pair (u, v) of nodes in G, the shortest distance between u and v, denoted by $d(u, v)$ [4,25,31].

Shortest distance has a wide range of applications, *e.g.,* socially-sensitive search [89,93], influential community detection [9,56] and centrality analysis [16,18].

As opposed to SubIso, shortest distance queries are *unlabeled*, *i.e.,* the value of a query answer $d(u, v)$ does not depend on labels. In contrast with 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 [31] to contracted graphs, denoted by Dijkstra, which is one of the best known algorithms for Dist. Just like TriC, the adapted algorithm for Dist decontracts no supernodes, neither topological components nor obsolete parts.

3.3.1 Contraction for Dist

A path between nodes u and v can be decomposed into (1) edges between supernodes, and (2) edges within a supernode. The idea of synopses for Dist 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 $S_{\mathsf{Dist}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ 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: $v_H.\mathsf{dis} = \{(v_1, v_2, d_{v_H}(v_1, v_2))\}$ for all pairs of $v_1, v_2 \in f'_C(v_H)$;
- path: $v_H.\mathsf{dis} = \{(u_1, u_{|f'_C(v_H)|}, \Sigma_{1\le i<|f'_C(v_H)|} W(u_i, u_{i+1}))\}$, *i.e.,* it records the path itself;
- diamond, butterfly and obsolete components: $v_H.\mathsf{dis} = \{(v_1, v_2, d_{v_H}(v_1, v_2)) \mid v_1, v_2 \in f'_C(v_H)\}$.

In practice, the number of nodes in most contracted subgraphs is far below the upper bound k_u. Indeed, diamonds and butterflies have a constant size, and we find that a clique (resp. star, path and obsolete component) typically contains 6.5 (resp. 7.3, 4.1 and 49.2) nodes. Hence, the size of a synopsis is fairly small. Note that the upper bound k_u should be larger than typical sizes of components, since large components exist and may be more powerful for accelerating computations.

Example 6 Assume $W(u, v) = 1$ for all edges (u, v) in graph G of Fig. 1a. Then, for supernodes in the contracted graph of Fig. 1b, (1) $v_{H1}.\mathsf{dis} = \{(i_1, f_1, 1), (i_1, n_1, 1), (i_1, l_1, 1), (f_1, n_1, 2), (f_1, l_1, 2), (n_1, l_1, 2)\}$; (2) $v_{H2}.\mathsf{dis} = \{(k_i, k_j, 1) \mid 1 \le i < j \le 5\}$; (3) $v_{H4}.\mathsf{dis} = \{(u_1, u_2, 1), (u_1, u_5, 1), (u_1, u_3, 2), (u_1, u_4, 2), \ldots\}$; and finally, (4) $v_{H5}.\mathsf{dis} = \{(k_6, t_1, 4)\}$. □

3.3.2 Shortest distance

We adapt algorithm Dijkstra [31] to contracted graphs G_c, and refer to the adapted algorithm as $\mathsf{DisA_c}$.

Algorithm 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 $\mathsf{DisA_c}$. $\mathsf{DisA_c}$ is the same as Dijkstra except minor changes to updating distance estimates. When moving a node w from $V \setminus S$ to S, suppose that v_H is the supernode to which w is mapped, *i.e.,* $f_C(w) = v_H$. $\mathsf{DisA_c}$ updates distance estimates $\overline{d}(w')$ for $w' \in f'_C(v_H)$ as follows: (1) if v_H.type is clique, butterfly, diamond or obsolete, update $\overline{d}(w')$ by $\overline{d}(w) + d_{v_H}(w, w')$ using $v_H.\mathsf{dis}$; (2) if v_H.type = star or claw, 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 $v_H.\mathsf{dis}$; in these cases, no supernode (for topological or obsolete components) is decontracted. $\mathsf{DisA_c}$ updates $\overline{d}(w')$ by $\overline{d}(w) + W(w, w')$ for all edges (w, w') where $f_C(w) \ne f_C(w')$, by decontracting superedge $(f_C(w), f_C(w'))$ at worst, in the same way as Dijkstra.

Example 7 Given Dist query (u_2, k_5) on the contracted graph G_c of Fig. 1b, $\mathsf{DisA_c}$ works in the following steps: (1) initially, $S = \emptyset$, $\overline{d}(u_2) = 0$, and $\overline{d}(v) = \infty$ for all other nodes; (2) $S = \{u_2\}$, $\overline{d}(u_1) = \overline{d}(u_5) = 1$, $\overline{d}(u_3) = \overline{d}(u_4) = 2$ by using $S_{\mathsf{Dist}}(v_{H4})$; (3) $S = \{u_2, u_1, u_5, u_3, u_4\}$, $\overline{d}(t_1) = 3$ by edge (u_4, t_1), and $\overline{d}(k_6) = \overline{d}(t_1) + d_{v_{H3}}(k_6, t_1) = 7$ by $v_{H5}.\mathsf{dis}$; $\overline{d}(i_1) = 2$ by edge (u_1, i_1), and $\overline{d}(f_1) = \overline{d}(n_1) = \overline{d}(l_1) = 3$ by $v_{H1}.\mathsf{dis}$; 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 making use of reverse function f'_C and synopsis $S_{\mathsf{Dist}}(v_{H3})$ (note that v_{H3} contracts a star); (4) $S = \{u_2, u_1, u_5, u_3, u_4, i_1, t_1, u_7\}$, $\overline{d}(t_2) = 4$ by edge (t_1, t_2); and (5) $S = \{u_2, u_1, u_5, u_3, u_4, i_1, t_1, u_7, f_1, n_1, l_1, t_2\}$, $\overline{d}(k_1) = \overline{d}(k_3) = \overline{d}(k_5) = 5$ by edges (t_2, k_1), (t_2, k_3), (t_2, k_5). When $\mathsf{DisA_c}$ moves node k_5 to S, it gets $d(k_5) = 5$. The algorithm returns $d(u_2, k_5) = 5$. □

Analyses. By induction on the length of shortest paths, one can verify that $\mathsf{DisA_c}$ is correct. In particular, for each node w' in G, when $\overline{d}(w')$ is updated by a node w that is mapped to the same supernode, the update is equivalent to a series of Dijkstra updates. Moreover, $\mathsf{DisA_c}$ works on smaller contracted graphs G_c and saves traversal cost inside contracted components without any decontraction, neither topological nor obsolete.

Temporal shortest distance. Similar to temporal SubIso and TriC, we study *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, $T(w) > t$.

Algorithm $\mathsf{DisA_c}$ can be easily adapted to temporal Dist, by skipping nodes v with $T(v) \le t$. In particular, it safely ignores a supernode v_H if $v_H.t \le t$.

3.4 Connected component with contraction

We next study the connected component problem [29,85]. In a graph G, a *connected component* is a maximal subgraph of G in which any two nodes are connected to each other via a path. The *connected component problem*, denoted as CC, is to compute the set of pairs (s, n) for a given graph G, where

(s, n) indicates that there are n connected components in G that consist of s nodes.

Given a graph G, CC returns the numbers of connected components of various sizes in G. Similar to Dist, CC is a *non-local* query, *i.e.,* it has to traverse the entire graph when answering the query. It is also *un-labeled*, *i.e.,* labels have no impact on its query answer.

This form of CC is used in pattern recognition [45,53], graph partition [86] and random walk [49].

We adapt algorithm CCA of [85] for CC to contracted graphs, since it is one of the most efficient CC algorithms. Better still, we show that the adapted algorithm decontracts neither supernodes nor superedges.

3.4.1 Contraction for CC

The synopsis S_{SubIso} for SubIso suffices for us to answer CC queries. Observe that each subgraph H contracted to a supernode v_H is connected, no matter whether H is a topological component or an obsolete component. We can regard a supernode v_H as a whole when evaluating CC queries, and leverage $S_{\mathsf{SubIso}}(v_H)$ and f'_C to compute the size of connected components. We need neither additional synopses nor any decontraction.

3.4.2 Connected component

We now adapt algorithm CCA [85] to contracted graphs. The adapted algorithm is referred to as $\mathsf{CCA_c}$.

Algorithm CCA. We first review how CCA works. (1) Starting from each unvisited node v in graph G, CCA performs a depth-first-search (DFS) and collects all unvisited nodes reached in the traversal. These nodes are connected to v and are marked as visited. When no more nodes are unvisited, all visited nodes and v form a connected component. CCA records its size s. (2) After all nodes in G are visited, CCA groups connected components by size s and returns the aggregate (s, n).

Algorithm $\mathsf{CCA_c}$. On the contracted graph G_c, $\mathsf{CCA_c}$ works in the same way as CCA except that (1) it only performs DFS on G_c, without decontracting any supernodes or superedges; and (2) the size of each connected component is aggregated as the sum of the size $|f'_C(v_H)|$ of all supernodes v_H in the component.

Example 8 On the contracted graph in Fig. 1b, $\mathsf{CCA_c}$ finds a connected component that consists of supernodes v_{H1}, v_{H2}, v_{H3}, v_{H4}, v_{H5} and t_2. The size s of this component is simply the sum $|f'_C(v_{H1})| + \cdots + |f'_C(v_{H5})| + |f'_C(t_2)|$, *i.e.,* $s = 25$. Since all the supernodes in G_c have been visited, $\mathsf{CCA_c}$ outputs (25, 1). □

Analyses. $\mathsf{CCA_c}$ is correct since it follows the same logic as CCA and all contracted subgraphs are guaranteed to be connected. The algorithm takes at most $O(|G_c|)$ time while CCA takes $O(|G|)$ time. Since G_c is much smaller than G, $\mathsf{CCA_c}$ always outperforms CCA.

Temporal connected component. $\mathsf{CCA_c}$ can be adapted to compute connected components with timestamp later than a given time t, by skipping nodes v with $T(v) \leq t$. It safely ignores a supernode v_H if $v_H.t \leq t$.

3.5 Clique decision with contraction

We next study a decision problem for clique. A *clique* in a graph G is a subgraph C in which there are edges between any two nodes; it is a k-clique if the number of nodes in C is k (*i.e.,* $|V(C)| = k$) . We consider the *clique decision problem* [20,57], denoted by CD, to find whether there exists a k-clique in G for a given number k. CD is being widely used in community search [76], team formation [59] and anomaly detection [11,65].

Similar to Dist and CC, CD is un-labeled. In contrast with Dist and CC, but similar to SubIso, it is local, *i.e.,* all nodes in a clique are within 1 hop of each other.

The clique decision problem is known NP-complete (cf. [42]). A variety of algorithms have been developed for CD, notable CDA of [57], which we will adapt next.

3.5.1 Contraction for CD

Observe the following. (1) Cliques in G contracted into supernodes in G_c can help us find an initial maximum clique (see below). (2) The degree of a node can be used as an upper bound of the maximum clique containing it.

In light of these, we extend synopsis $S_{\mathsf{SubIso}}(v_H)$ with tags cs and md. For a subgraph H that is contracted to a supernode v_H, the two tags record the maximum clique found in H and the maximum degree of the nodes in H, respectively. Specifically, $v_H.\mathsf{cs}$ is based on $v_H.\mathsf{type}$:

- clique: $v_H.\mathsf{cs} = |f'_C(v_H)|$;
- diamond and butterfly: $v_H.\mathsf{cs} = 3$;
- star, path and claw: $v_H.\mathsf{cs} = 2$; and
- obsolete component: we find a k-clique in an obsolete component online.

and $v_H.\mathsf{md}$ is by aggregation:

- node v: $v.\mathsf{md} = |\{u \mid (u, v) \in E\}|$; and
- supernode v_H: $v_H.\mathsf{md} = \max\{v.\mathsf{md} \mid f_C(v) = v_H\}$.

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

Example 9 In the contracted graph G_c of Fig. 1b, $S_{\mathsf{CD}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ with tags cs and md as follows. Since v_{H2} contracts a clique, $v_{H2}.\mathsf{cs} = 5$; $v_{H4}.\mathsf{cs} = 3$ since v_{H4}

contracts a butterfly, and v_H.cs $= 2$ for supernodes v_{H3} (star) and v_{H5} (path). For tag md, v_{H1}.md $= i_1$.md $= 4$; similarly, v_{H2}.md $= 8$, v_{H3}.md $= 4$, v_{H4}.md $= 4$, v_{H5}.md $= 4$, and t_2.md $= 4$. □

3.5.2 Clique decision

We adapt CDA [57] to G_c, denoted as $\mathsf{CDA_c}$.

Algorithm CDA. We first review CDA. Given a graph G, algorithm CDA checks the existence of a k-clique in G by branch-and-bound. It branches from each node in G. Denote by C the current clique in the search, and by P the set of common neighbors of the nodes in C. CDA (1) bounds the search from C if $|C| + |P| < k$, or (2) branches from each node u in P to expand C. More specifically, it iteratively adds a node u from P to C and removes all those nodes in P that are not neighbors of u, enlarging C and shrinking P until P is empty. If $|C| \geq k$, then C contains a k-clique and CDA terminates with true; it returns false if no k-clique is found after all branches are searched.

Algorithm $\mathsf{CDA_c}$. $\mathsf{CDA_c}$ adopts the same logic as CDA except the following: (1) it picks the maximum synopsis v_H.cs among all supernodes v_H in G_c; a k-clique is found directly if v_H.cs $\geq k$; and (2) it skips a supernode v_H in G_c if v_H.md $< k - 1$. Superedges adjacent to v_H are skipped as well since no k-clique contains any node contracted to v_H. Otherwise, it checks the synopsis of v_H if v_H contracts a topological component, or restores obsolete component H contracted to v_H, to check cliques in the original graph G. Note that $\mathsf{CDA_c}$ initiates the search with the largest clique contracted, by checking the synopses. Hence, cliques play a more important role than the other regular structures for CD.

Example 10 For query with $k = 5$, by $S_{\mathsf{CD}}(v_{H2})$ of Fig. 1b, $\mathsf{CDA_c}$ finds a 5-clique and returns true.

For query with $k = 6$, all supernodes except v_{H2} are skipped by synopses. Their adjacent superedges are skipped as well. Since v_{H2} only contracts a 5-clique, $\mathsf{CDA_c}$ fails to find a 6-clique and returns false. □

Analyses. One can verify that $\mathsf{CDA_c}$ is correct since it follows the same logic as CDA except that it adopts pruning strategies that are possible because of the use of synopses. While the two algorithms have the same worst-case complexity, $\mathsf{CDA_c}$ starts with a supernode with a maximum clique and may find a k-clique directly; moreover, it skips a supernode as a whole by synopses, which reduces unnecessary search and validation.

Temporal k-clique. Algorithm $\mathsf{CDA_c}$ can be adapted to find a k-clique with timestamp later than a given time t, by skipping nodes v with $T(v) \leq t$. Like $\mathsf{SubA_c}$ and $\mathsf{TriA_c}$, 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 graphs G. We start with batch update ΔG, which is a sequence of edge insertions and deletions. We formulate the problem (Sect. 4.1), present the incremental algorithm (Sects. 4.2–4.3), discuss vertex updates (Sect. 4.4), and parallelize the algorithm (Sect. 4.5).

4.1 Incremental contraction problem

Updates to a graph G, denoted by ΔG, consists of (1) node updates, *i.e.,* node insertions and deletions; and (2) edge updates, *i.e.,* edge insertions and deletions.

Given a contraction scheme $\langle f_C, \mathcal{S}, f_D \rangle$, a contracted graph $G_c = f_C(G)$, and updates 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 affected supernodes; and (c) functions $f_C \oplus \Delta f_C$ and $f_D \oplus \Delta f_D$ *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 both change 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 ΔG, among other things.

Criterion. Following [77], we measure the complexity of incremental algorithms with 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* [77] if its complexity is determined by |AFF|, not by the size $|G|$ of the entire (possibly big) 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 G, as opposed to |AFF| of the former.

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

Challenges. Problem ICP is nontrivial. (1) Topological components are fragile. For instance, 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 by a query Q 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 challenges, we show that bounded incremental contraction is within reach in practice.

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

We first give a constructive proof of Theorem 2 for edge updates, consisting of two parts: (1) the maintenance of the contracted graph G_c and its associated decontraction function f_D (Sect. 4.2); and (2) the maintenance of the synopses of affected supernodes (Sect. 4.3). We then give a constructive proof of Theorem 2 for vertex updates (Sect. 4.4), which is simpler.

4.2 Incremental contraction algorithm

An incremental algorithm is shown in Fig. 7, 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, where ΔG consists of edge insertions and deletions; the handling of f_D is similar.

(a) Preprocessing. Algorithm IncCR first identifies an initial area affected by edge 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 node 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 update" later on. All changes in ΔG are applied to graph G in the given order.

(b) Updating. Algorithm IncCR then updates contracted graph 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 $v_{H1} = v_{H2}$, and v_{H1}.type =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, supernode v_{H_2} is divided into two shorter paths. Note that components with less than k_l nodes due to updates are decomposed into singleton nodes. All such singleton nodes are added to the set V_s (line 3).

(c) Contracting. Finally, algorithm IncCR processes nodes in the set 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 node v as a singleton, *i.e.,* by letting $f_C(v) = v$.

Example 11 Consider inserting four edges into graph G of Fig. 1a: (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): v_{H4} is not a butterfly any longer, and 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 $O(|\mathsf{AFF}|^2)$ time.

The algorithm is (a) bounded [77], since its cost is determined by |AFF| alone, and (b) *local* [35], *i.e.,* the changes are confined only to affected supernodes and their neighbors in the contracted graph G_c.

4.3 Maintenance of synopses

We next show that for SubIso, TriC, Dist, CC and CD, (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, Dist, CC and CD 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 (Sect. 3.1). One can 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 due to bounds $[k_l, k_u]$.

(b) For TriC, synopsis $S_{\mathsf{TriC}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ with v_H.tc, which is updated by (i) clique neighbors I of nodes u in v_H when $I \in$ AFF; (ii) v_H itself if v_H.type is clique or obsolete; and (iii) common neighbors J of connected nodes u, v

```
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 edge updates $\Delta G$ to $G$.
Output: New contracted graph $G_c \oplus \Delta G_c$.
1.  reduce $\Delta G$; $V_s := \emptyset$;
2.  refresh nodes $u$ in $\Delta G$;
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 $\Delta G$;
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 $V_s$;
```

Fig. 7 Algorithm IncCR

in v_H for $J \in$ 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 |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 $|f'_C(v_H)| \le k_u$. Thus, the incremental maintenance of S_{Dist} is bounded in $O(|\mathsf{AFF}|)$ time.

(d) For CC, recall that the synopsis S_{SubIso} suffices to answer CC queries. Hence, as in case (a), $S_{\mathsf{CC}}(v_H)$ for each supernode v_H can be updated in $O(1)$ time, and the maintenance of S_{CC} is bounded in $O(|\mathsf{AFF}|)$ time.

(e) For CD, $S_{\mathsf{CD}}(v_H)$ extends $S_{\mathsf{SubIso}}(v_H)$ with v_H.cs and v_H.md. Here v_H.cs is confined to v_H and can be updated in $O(1)$ time; v_H.md is confined to v_H and its neighbors, and can be updated in $O(|\mathsf{AFF}|)$ time. Thus, the maintenance of S_{CD} is in $O(|\mathsf{AFF}|^2)$ time.

Example 12 Continuing with Example 11, we show how to maintain v_H.tc in $S_{\mathsf{TriC}}(v_H)$ for supernodes v_H in G_c; $S_{\mathsf{SubIso}}(v_H)$, $S_{\mathsf{Dist}}(v_H)$, $S_{\mathsf{CC}}(v_H)$ and $S_{\mathsf{CD}}(v_H)$ are simpler since their affected synopses are confined to ΔG_c.

More specifically, (1) for edge insertion (n_1, f_1), supernode v_{H1} is decomposed into four singletons, for which synopses are defined as n_1.tc $= f_1$.tc $= l_1$.tc $= i_1$.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) edge insertion (k_1, u_{10}), k_1 becomes a common neighbor of u_{10} and u_6; let H denote the subgraph contracted by v_{H2}; then, $t^H_{u_{10},u_6} = 1$ and v_{H3}.tc $= 1$. (4) When inserting edge (u_1, u_4), v_{H4} is decomposed into singletons. During the contraction phase, nodes u_1, u_2, u_5, u_4 are contracted into a diamond v'_{H4} with v'_{H4}.tc $= 2$. Node u_3 is left singleton, with u_3.tc $= 0$. □

```
Algorithm IncCR$_V$
Input: A graph contraction scheme $\langle f_C, \mathcal{S}, f_D\rangle$, a contracted
        graph $G_c$ of a graph $G$ and vertex updates $\Delta G$ to $G$.
Output: New contracted graph $G_c \oplus \Delta G_c$.
1.  $V_s := \emptyset$;
2.  for each update $u \in \Delta G$ do
3.      if $u$ is an vertex insertion
4.      then add $u$ into $V_s$;
5.      else if $u$ is an vertex deletion
6.          then IncCR$^-_V(G_c, u)$ ;
7.  Contract $(V_s, G_c)$;
8.  return $G_c$;

Procedure IncCR$^-_V$
Input: A contracted graph $G_c$ and vertex deletion $u$.
Output: An updated $G_c$.
1.  $v_H := f_C(u)$;
2.  if $v_H$.type=clique;
3.  then update $v_H$ directly;
4.  else if $v_H$.type =claw, butterfly, obsolete;
5.      then decontract $v_H$, add singleton nodes into $V_s$;
6.  else if $v_H$.type =path, diamond, star;
7.      then process $v_H$, add singleton nodes into $V_s$;
```

Fig. 8 Algorithm IncCR$_V$

4.4 Vertex updates

Vertex updates are a dual of edge updates [58], and can be processed accordingly. More specifically, we present incremental algorithm IncCR$_V$ in Fig. 8, to deal with vertex updates. Consider node insertions and deletions.

(1) When inserting a new node u, algorithm IncCR$_V$ first treats u as a singleton and collects it in set V_s (lines 3-4); the node u is then contracted into a topological structure in the contracting step (line 7).

(2) When deleting a node u that is contracted into a supernode v_H, there are three cases to consider, elaborated in IncCR^-_V of Fig. 8: (a) if v_H is a clique, v_H remains unchanged except that u is removed (lines 2-3); (b) if v_H is a claw, a butterfly or an obsolete component, v_H is decontracted and all nodes in $f'_C(v_H)$ except u are treated as singletons and are collected in set V_s (lines 4-5); and (c) otherwise, we process u and v_H by synopsis and add resulting singleton nodes into V_s (lines 6-7). For instance, consider the case when v_H contracts a star, (i) if u is the central node $v_H.c$, v_H is decontracted in the same way as case (b); and (ii) otherwise, v_H remains to be a star, similar to case (a).

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

```
Algorithm IncPC
Input: A contracted fragment $F_c$, updates $\varDelta F_i$, and
   corresponding contraction scheme $\langle f_C, \mathcal{S}, f_D \rangle$.
Output: New contracted fragment $F_c \oplus \varDelta F_i$.
1.  preprocess updated crossing edges;
2.  conduct IncPC locally in parallel;
3.  collect refreshed singleton nodes in $V_s$;
4.  for each refreshed singleton node $v$ in $V_s$ do
5.    build uncontracted neighbors of at most $k_u$ nodes;
6.  master $M_0$ merges overlapped neighbors;
7.  $M_0$ distributes disjoint uncontracted neighbors to workers;
8.  conduct GCon on assigned uncontracted neighbors;
9.  return $F_c \oplus \varDelta F_i$;
```

Fig. 9 Algorithm IncPC

4.5 Parallel incremental contraction algorithm

We parallelize incremental algorithm IncCR, to speed up the incremental maintenance process.

Parallel setting. Similar to PCon, we use a master M_0 and n workers, A contracted graph G_c is edge-partitioned and is distributed to n workers. Each fragment F_i consists of a part of the contracted graph G_c and its corresponding (partial) decontraction function and synopses. For a *crossing superedge* (v_{H1}, v_{H2}) between two fragments, *i.e.,* when v_{H1} and v_{H2} are assigned to two distinct fragments, the decontraction function $f_D(v_{H1}, v_{H2})$ is maintained in both fragments.

Parallel incremental contraction. The parallel incremental algorithm is denoted by IncPC and shown in Fig. 9. To simplify the discussion, we focus on edge updates; node updates are processed similarly. It works under BSP [88]. In a nutshell, it preprocesses crossing (super)edges (line 1). Then, all the workers run IncCR on its local fragment in parallel (line 2). After that, IncPC contracts refreshed singleton nodes V_s into supernodes (lines 3-8) along the same lines as algorithm PCon. Here, each fragment has its local set V_s and all refreshed singleton nodes in V_s can be coordinated and distributed by the master M_0. Each node v is guaranteed to be contracted into one supernode v_H. More specifically, algorithm IncPC works as follows.

(1) IncPC preprocess updated edges $e = (u, v)$ between two fragments (line 1), *i.e.,* when u and v are contracted into supernodes v_{H1} and v_{H2}, and v_{H1} and v_{H2} are in two distinct fragments. Such updates are unaffecting as long as neither u nor v is an intermediate node of a path, and these updates are maintained by f_D. Otherwise, the supernode of type *path* may be affected and is decomposed into singleton nodes; such refreshed singleton nodes are collected in a set V_s as the initial area affected by ΔG. In the same way as IncCR, we refresh timestamps of obsolete nodes touched by updates.

(2) Each worker locally runs IncCR in parallel (line 2). Refreshed singleton nodes that cannot be contracted into supernodes are collected in V_s (line 3).

(3) For each refreshed singleton node v in V_s, IncPC build its uncontracted neighbors (of at most k_u nodes) in parallel, similar to step (2) in PCon (lines 4-5).

(4) Master M_0 merges overlapped neighbors into one and distributes disjoint ones to n workers (lines 6-7).

(5) Each worker contracts its assigned subgraphs, *i.e.,* uncontracted neighbors, in parallel (line 8).

One can verify that each node v in G is contracted into one supernode v_H (including v itself), and the contracted graph G_c cannot be further contracted.

5 Experimental study

Using ten real-life graphs, we experimentally evaluated (1) the contraction ratio; (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 algorithms; and (6) the parallel scalability of the (incremental) contraction algorithms.

Experiment setting. We used the following datasets.

(1) Graphs. We used ten real-life graphs: three social networks Twitter [70], LiveJournal [94] and LivePokec [10]; three Web graphs Google [64], NotreDame [5] and GSH [3]; three collaboration networks DBLP [2], Hollywood [15] and citHepTh [63]; and a road network Traffic [1]. Their sizes are shown in Table 2. We randomly generated a time series to simulate obsolete attributes, at most 70% (it is 80% for IT data of our industry collaborator). We tested obsolete components with random (temporal) queries generated on all datasets.

We also generated synthetic graphs with up to 250 M nodes and 2.5 B edges, to test the parallel scalability of the (incremental) contraction algorithms.

Updates. We randomly generated edge updates ΔG, controlled by the 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. In the same manner, we generated vertex updates ΔG.

(2) Graph patterns. We implemented a generator for graph pattern queries controlled by three parameters: the number V_Q of pattern nodes, the number E_Q of pattern edges, and a set L_Q of labels for queries Q.

(3) Implementation. We implemented the following algorithms, all in C++. (1) Algorithms $\mathsf{SubA_c}$ (Sect. 3.1.2), $\mathsf{TriA_c}$ (Sect. 3.2.2), $\mathsf{DisA_c}$ (Sect. 3.3.2), $\mathsf{CCA_c}$ (Sect. 3.4.2), $\mathsf{CDA_c}$ (Sect. 3.5.2), $\mathsf{VF2_c}$ for SubIso by adapting VF2 [28] to contracted graphs; in addition, $\mathsf{PLL_c}$ for Dist by adapting PLL [4] to contracted graphs. (2) Our contraction algorithm GCon (Sect. 2.3) and its parallel version PCon (Sect. 2.4), incremental algorithm IncCR for batch updates and its parallel version IncPC (Sect. 4). (3) The baselines include existing query eval-

Table 2 Contraction ratio (each column: CR or % of contribution to CR with/without obsolete mark)

Graph	$\|V\|, \|E\|$	k_u	CR	1st	2nd	3rd	Obsolete
Twitter	81K, 1.3M	100	0.176/0.286	7.78/27.7	15.44/50.71	4.29/14.39	69.69/–
LiveJournal	4M, 35M	500	0.378/0.527	11.46/30.3	20.41/51.4	3.74/9.7	60.99/–
LivePokec	1.6M, 22M	500	0.467/0.651	4.46/9.91	35.91/77.76	2.32/4.83	54.4/–
Google	876K, 4.3M	200	0.193/0.294	19.36/51.47	19.33/47.04	0.58/1.49	60.74/–
NotreDame	325K,1.1M	200	0.274/0.441	23.16/60.64	9.47/26.95	4.56/12.4	62.81/–
GSH	68M, 1.8B	500	0.325/0.493	29.32/77.33	5.31/21.78	0.75/0.89	64.62/–
DBLP	204K, 382K	100	0.14/0.172	36.21/71.65	14.22/28.32	0.02/0.03	49.54/–
Hollywood	1.1M, 56M	500	0.239/0.534	17.36/71.76	6.05/16.46	3.21/11.79	73.38/–
citHepTh	28K, 352K	50	0.26/0.362	21.42/51.93	14.18/36.71	4.6/11.36	59.81/–
Traffic	24M, 29M	500	0.365/0.59	12.37/49.72	9.42/36.74	3.5/13.54	74.7/–

uation algorithms: (a) TurboIso [44] and TurboIsoBoosted [78] with indexing for SubIso, and VF2 [28] without indexing; (b) graph compression DeDense [69] for SubIso; (c) TriA [47] for TriC; (d) Dijkstra without indexing and PLL [4] with indexing for Dist [31]; (e) CCA [85] for CC; and (f) CDA [57] for CD. We did not compare with summarization since it does not support any algorithm to compute exact answers for the five applications.

(4) Experimental environment. The experiments were conducted on a single-processor machine powered by Xeon 3.0 GHz with 64GB memory, running Linux. Since GSH and synthetic graphs ran out of 32 GB memory without contraction, we used a machine with 64 GB memory. For parallel (incremental) contraction, we used 4 machines, each with 12 cores powered by Xeon 3.0 GHz, 32GB RAM, and 10Gbps NIC. Each experiment was run 5 times, and the average is reported here.

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 while incorporating different synopses. In addition, we report the impact of each of the first three topological components and obsolete component for each dataset, in the presence and absence of obsolete data.

As remarked in Sect. 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 are 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 2 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 0.281, *i.e.,* contraction reduces these graphs by 71.9%. When $t_0 = 0$, *i.e.,* if obsolete data are not considered, CR is 0.435. These show that real-life graphs can be effectively contracted in the presence and absence of obsolete data. Compared with the results of [38], by considering more regular structures, the contraction scheme improves the contraction ratio CR by 2.49% and 6.90% in the presence and absence of obsolete data, respectively.

(2) When obsolete data are present, the average CR is 0.34, 0.264, 0.213 and 0.365 in social networks, Web graphs, collaboration networks and road networks, respectively. When obsolete data are absent, CR is on average 0.488, 0.409, 0.356 and 0.59. 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 are absent, on average the first three regular structures contribute 50.2%, 39.4% and 8.0% to CR, respectively. When obsolete mark is taken into account, their contribution is 18.3%, 14.9% and 2.8%, respectively. This is because nodes from these components may be moved to obsolete components.

(4) We also studied the impact of the contraction order on query evaluation. Topological components have different impacts on different types of graphs, *e.g.,* stars, claws and paths are effective in Traffic, and cliques, stars and butterflies work better than the others in collaboration networks. Taking the order of Table 1 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 9.42% and 7.05%, respectively. As shown in Table 3, the average slowdown of RE and EX is (a) 7.24% and 5.58% for SubIso, (b) 5.55% and 5.46% for TriC, (c) 3.89% and 4.30% for Dist, (d) 7.34% and 34.7% for CC, and (e) 2.38% and 19.1% for CD,

Table 3 Slowdown (%) by RE and EX orders

Graph	SubIso		TriC		Dist		CC		CD	
	RE	EX	RE	EX	RE	EX	RE	EX	RE	EX
Twitter	8.04	3.98	7.41	3.66	5.27	2.86	8.22	19.2	6.73	24.9
LiveJournal	9.46	5.52	8.26	5.09	2.71	5.32	9.03	61.1	5.49	18.3
LivePokec	8.67	6.46	3.15	3.12	4.49	2.48	6.10	51.7	8.23	20.2
Google	5.17	7.54	6.07	3.75	1.02	3.8	7.19	38.7	−4.17	12.5
NotreDame	11.9	5.76	4.20	6.46	5.95	4.93	3.72	44.5	−4.33	15.3
GSH	3.52	6.22	4.59	6.08	2.78	4.15	4.25	32.1	−5.53	16.4
DBLP	2.13	5.53	11.3	14.2	4.38	5.31	18.8	19.6	5.05	34.2
Hollywood	6.32	6.39	2.25	4.73	3.89	5.81	5.75	30.3	3.02	29.3
citHepTh	7.48	3.24	3.98	4.91	2.56	3.23	7.43	35.5	7.92	17.1
Traffic	9.69	5.11	4.29	2.56	5.78	5.11	2.94	14.2	1.39	2.87

respectively. These justify that the order of Table 1 is effective for most applications and most types of graphs. There are also exceptions, *e.g.,* reversing the order for Web graphs improves the efficiency of CD. Recall that we contract stars, cliques and butterflies for Web graphs. For CD in particular, however, cliques play a more important role than the other two (Sect. 3.5); hence, contracting cliques first may work better for CD.

Exp-2: Effectiveness: query processing. We next evaluated the speedup of query processing introduced by the contraction scheme, measured by query evaluation time over original and contracted graphs.

Subgraph isomorphism. Varying the size $|V_Q|$ of pattern queries from 4 to 7, we tested VF2, TurboIso and TurboIsoBoosted on GSH and Hollywood as G, DeDense [69] on the compressed graph, and $\mathsf{SubA_c}$ and $\mathsf{VF2_c}$ on the contracted graph G_c of G. For each query, we output the first 10^8 matches. As shown in Fig. 10a, b, (1) on average, $\mathsf{SubA_c}$ on G_c is1.69, 1.49 and 18.85 times faster than TurboIso, TurboIsoBoosted and DeDense, respectively; (2) $\mathsf{VF2_c}$ beats DeDense by 9.31 times; (3) $\mathsf{VF2_c}$without indices is only 19.1% slower than TurboIso with indices, while TurboIsoBoosted and TurboIso are 10.1 and 8.97 times faster than VF2, respectively; and (4) the speedup is more substantial on collaboration networks, *e.g.,* 2.11 times on Hollywood, because cliques are prevalent in such graphs and are the most effective structure for SubIso due to the high capacity in pruning invalid matches.

Triangle counting. As shown in Fig. 10c, the results for TriC are consistent with the results on subgraph isomorphism: (1) $\mathsf{TriA_c}$ on the contracted G_c is on average 1.44 times faster than TriA on their original graphs G. (2) The speedup is more evident in collaboration networks: *e.g.,* $\mathsf{TriA_c}$ on Hollywood is 1.57 times faster than TriA while it is 1.47, 1.45 and 1.28 times on LiveJournal, Google and Traffic, respectively. TriA spends more than 1000 seconds on GSH (hence not shown).

Shortest distance. The results for Dist are consistent with the results on SubIso. As reported in Fig. 10d, $\mathsf{DisA_c}$ is 1.64 and 1.36 times faster than Dijkstra on GSH and Hollywood, respectively, by reducing search space and employing synopses. PLL could not build indices on GSH within 64G memory, while $\mathsf{PLL_c}$ successfully builds indices on (smaller) contracted GSH. On average, $\mathsf{PLL_c}$ spends 94.2μs to evaluate a query on GSH. On other smaller datasets, in contrast, $\mathsf{PLL_c}$ is 18% slower than PLL due to overhead on supernodes.

Connected component. As shown in Fig. 10e over LiveJournal, GSH, Hollywood, and Traffic for social graphs, Web graphs, collaboration networks and road networks, respectively, the results for CC are consistent with the results on SubIso and TriC: (1) algorithm $\mathsf{CCA_c}$ on contracted graph G_c is on average 2.24 times faster than CCA on the original graph G, since $\mathsf{CCA_c}$ operates on the smaller G_c without decontracting supernodes or superedges. (2) The speedup is more evident in collaborations networks: *e.g.,* $\mathsf{CCA_c}$ on Hollywood is 2.87 times faster than CCA, since the contraction scheme performs the best on such graphs and the time complexity of $\mathsf{CCA_c}$ is linear in the size of the contracted graph.

Clique decision. As also shown in Fig. 10f, (1) algorithm $\mathsf{CDA_c}$ is 1.32, 1.54, 1.52 and 1.08 times faster than CDA on LiveJournal, Hollywood, GSH and Traffic, respectively, by using synopses to start with an initial maximum clique that may find a k-clique directly. (2) The speedup is less evident in road networks. For road networks, the contraction scheme contracts stars, claws and paths into supernodes; hence, we can only find a 2-clique (an edge) as the initial maximum clique by using synopses, which is trivial and useless.

The results on the other graphs are consistent.

Temporal queries. Fixing pattern size $|Q| = 4$ and varying timestamp t in temporal queries from 30%t_m to 70%t_m, we tested SubIso_t, TriC_t, Dist_t, CC_t and CD_t. As shown in Fig. 10g–k on LiveJournal, (1) $\mathsf{SubA_c}$ is on average 1.81 and 1.77 times faster than TurboIsoBoosted and TurboIso, respectively; $\mathsf{VF2_c}$ outperforms VF2 by 7.83 times. (2) The average speedup for TriC, Dist, CC and CD is 1.58, 2.31, 1.66

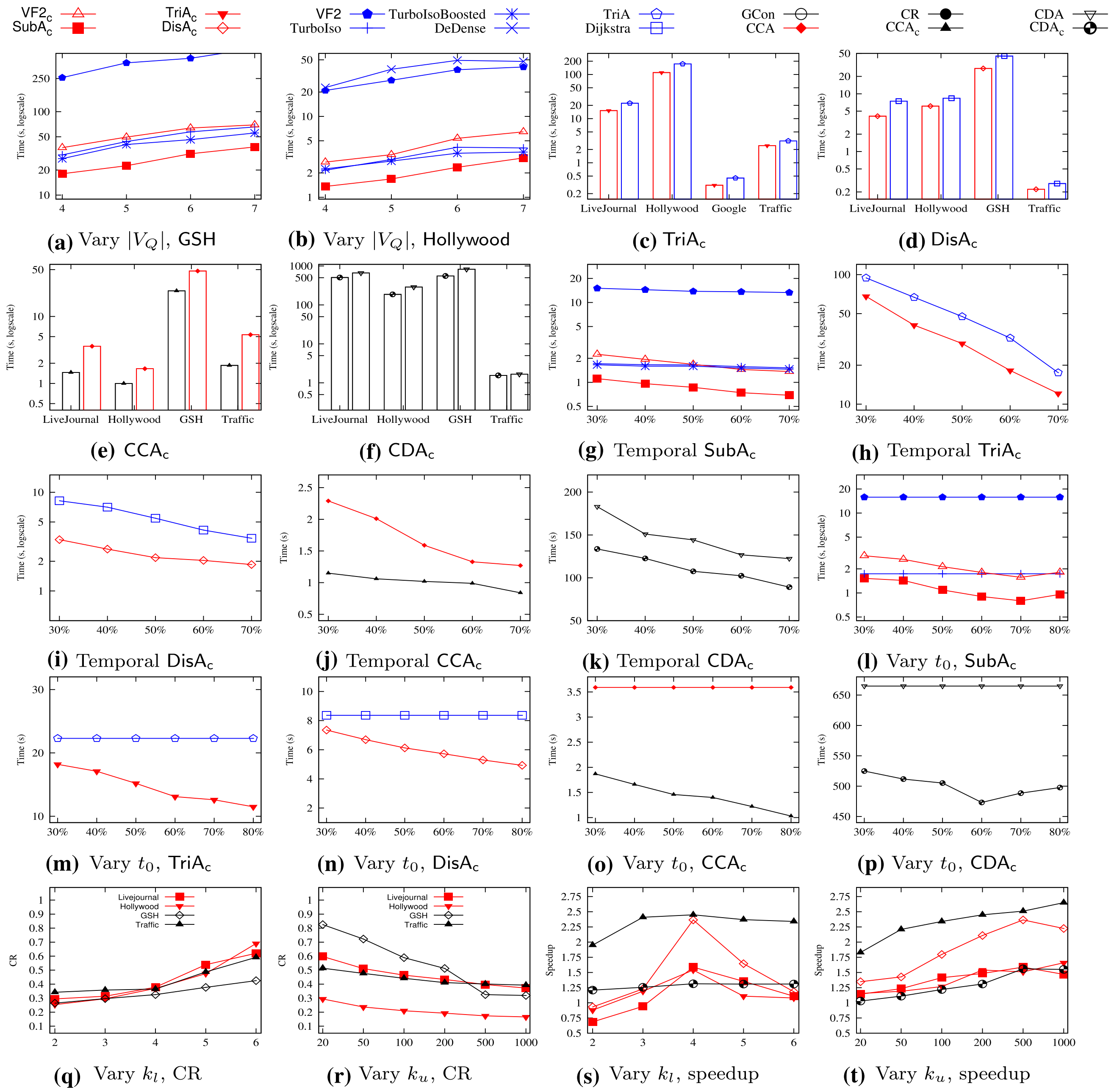


Fig. 10 Performance evaluation

and 1.31 times, respectively. (3) The speedup is larger for temporal queries than for conventional ones since temporal information maintained in synopsis provides additional capacity to skip more supernodes, as expected. (4) It is more substantial for larger t on SubIso$_t$.

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

We remark that our contraction scheme aims to make a generic optimization for multiple applications to run on the same graph at the same time. When a new application is considered, adding a specific synopsis suffices for our scheme. In contrast, a separate indexing structure has to be built for indexing approaches. Better still, it is much easier to develop synopses than indices. Moreover, existing indexing structures can be inherited by contracted graphs, to improve performance from contraction in addition to from indexing.

Table 4 Slowdown(%) by disabling certain topological component

Graph	SubIso			TriC			Dist			CC			CD		
	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd	1st	2nd	3rd
Twitter	45.8	10.9	4.7	16.4	19.1	2.1	28.2	28.7	5.7	85.1	141.6	22.8	42.7	4.1	0.3
LiveJournal	46.3	16.7	3.0	17.5	3.9	1.4	44.3	13.2	7.1	68.4	95.5	14.5	27.5	3.3	0.9
LivePokec	45.5	13.5	2.1	5.5	22.1	0.7	29.5	23.6	4.4	18.0	69.5	11.3	39.0	5.2	1.3
GSH	11.7	32.2	0.4	5.4	18.2	1.1	15.9	33.1	0.7	41.7	10.8	0.4	4.9	52.2	0.2
Google	19.6	40.6	2.5	8.7	20.3	2.9	18.3	44.6	5.8	107.1	70.8	5.6	5.4	57.8	2.7
NotreDame	15.2	42.3	3.3	29.5	41.2	0.4	27.7	47.8	4.9	55.4	50.2	8.0	2.1	20.6	0.5
DBLP	66.6	17.0	0.8	572.1	216.6	1.7	23.2	29.5	0.4	631.7	450.2	0.1	65.1	7.9	0.1
Hollywood	40.3	13.4	5.1	22.6	10.9	1.5	24.0	26.3	5.4	80.1	64.3	5.9	51.7	3.8	0.2
citHepTh	54.5	15.7	2.4	15.4	7.2	0.5	32.3	22.6	7.3	280.7	222.4	25.7	35.5	1.7	0.6
Traffic	30.1	24.3	5.7	10.1	3.5	9.4	40.2	18.7	10.6	41.7	8.7	5.2	4.3	2.5	0.3

Exp-3: Impact of each component. We next evaluated the impact of contracting each of the topological components identified in Sect. 2.2.

Impact of topological components. Based on Table 1, we took contraction of the first three types of regular structures as the baseline, and tested the impact of each component on the efficiency of query answering by disabling it, using all the ten real-life datasets.

As shown in Table 4, the average slowdown in evaluation time by disabling each of the first three structures is (a) 37.6%, 22.7% and 3.02% for SubIso, (b) 70.3%, 36.3% and 2.0% for TriC, (c) 28.4%, 28.8% and 5.2% for Dist, (d) 141.0%, 118.4% and 9.9% for CC, and (e) 27.8%, 15.9% and 0.7% for CD, respectively. In particular, the impact of each regular structure is mostly consistent with the contraction order. This said, for specific application and graphs, the impact of each regular structure may be slightly different. For CD on Web graphs, the average slowdown in evaluation time by disabling the first structure (star) and the second structure (clique) is 4.1% and 43.5%, respectively, since cliques dominate the effectiveness of the synopses for CD.

Impact of obsolete components. We tested the impact of contracting obsolete components on the efficiency of answering conventional queries. Fixing $|Q| = 4$ and varying x for timestamp threshold such that $t_0 = x\% t_m$, Fig. 10i–p reports the results of SubIso, TriC, Dist, CC and CD on LiveJournal, respectively. 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, Dist, CC and CD by 1.56, 1.53, 1.39, 2.49 and 1.33 times, respectively; and (3) the speedup for SubIso and CD gets smaller when $t_0 \geq 80\%$ due to the overhead of decontracting obsolete components. The results are consistent for Dist, TriC and CC, except that their speedup does not go down when t_0 gets larger since they do not need to decontract obsolete components.

Impact of k_l and k_u. We also tested the impact of k_l and k_u on the contraction ratio CR and efficiency. As remarked in Sect. 2.3, diamonds, butterflies and claws have a fixed size, while cliques, stars and paths vary. Fixing $k_u = 500$ (resp. $k_l = 4$) and varying k_l (resp. k_u) from 2 to 6 (resp. 20 to 1000), Fig. 10q (resp. Fig. 10r) reports the CR on LiveJournal, Hollywood, GSH and Traffic, respectively. As shown there, CR decreases when k_l decreases or k_u increases. Similarly, Fig. 10s (resp. Fig. 10t) reports the speedup of SubA_c, TriA_c, DisA_c, CCA_c and CDA_c on Hollywood. Query evaluation is slowed down when $k_l \leq 3$ or $k_u \geq 500$ for all algorithms except CCA_c and TriA_c due to excessive superedge decontractions or overlarge components. Recall that CCA_c decontracts neither supernodes nor superedges, and TriA_c precalculates triangles in both topological components and obsolete parts; hence, it prefers large k_u. We find that the best k_l and k_u for the datasets tested are around 4 and 500, respectively.

The results on the other graphs are consistent.

Exp-4: Space cost. We next studied the space cost of our contraction scheme compared with indexing cost. We consider six algorithms: SubA_c, TriA_c, DisA_c, CCA_c, CDA_c and PLL_c. The space cost includes the sizes of the contracted graph $|G_c|$, decontraction function $|f_D|$ and the sizes of synopses; as shown in Sect. 3, SubA_c, TriA_c, DisA_c, CCA_c and CDA_c do not need to decontract topological components; thus, we only uploaded f_D for obsolete components and superedges into memory. In particular, CCA_c requires no decontraction (Theorem 1) and thus incurs no cost for storing f_D at all. We compared the space cost with the indices used by TurboIso, HINDEX [75], PLL [4] and RMC [68].

Table 5 shows how the space cost increases when more applications run on Google (*i.e.,* graph G). We find the following. (1) Our contraction scheme takes totally 1.62GB for SubIso, TriC, Dist, CC and CD, much smaller than 12.9GB taken by TurboIso, PLL, HINDEX and RMC. (2) With the contraction scheme, graph G is no longer needed. That is,

Table 5 Total space cost of applications run on Google

Application	Contraction		Indexing	
	Detail	Space	Detail	Space
Shared parts	G_c, f_D	837MB	G	727MB
+SubIso	$\mathcal{S}_{\mathsf{SubIso}}$	848MB	TurboIso	1.07GB
+TriC	$\mathcal{S}_{\mathsf{TriC}}$	874MB	+HINDEX	2.1GB
+Dist	$\mathcal{S}_{\mathsf{Dist}}$	1.51GB	+PLL	9.58GB
+CC	–	1.51GB	–	9.58GB
+CD	$\mathcal{S}_{\mathsf{CD}}$	1.62GB	+RMC	12.9GB
+kNN	$\mathcal{S}_{\mathsf{kNN}}$	1.75GB	+Antipole	19.4GB

compared to G, the scheme uses 0.89GB additional space for the supernodes/edges in G_c and synopses for all five applications. It trades affordable space for speedup. (3) Synopses $\mathcal{S}_{\mathsf{SubIso}}$, $\mathcal{S}_{\mathsf{TriC}}$, $\mathcal{S}_{\mathsf{Dist}}$, $\mathcal{S}_{\mathsf{CC}}$ and $\mathcal{S}_{\mathsf{CD}}$ take 48.3% 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 in the contraction scheme is over indices.

To inherit the indexing structures of [44] and PLL, we use 1.14GB additional space to build a compact index for $\mathsf{PLL_c}$ and on average 26MB for $\mathsf{SubA_c}$ on Google. in addition to synopses $\mathcal{S}_{\mathsf{Dist}}$ and $\mathcal{S}_{\mathsf{SubIso}}$.

To verify the scalability with applications, we further adapted existing algorithms for k-nearest neighbors (kNN) [92]. The total space cost of the scheme for the six applications is 1.75GB, *i.e.,* 18.1% increment for each. It accounts for only 9.0% of the indices for TurboIso, PLL, HINDEX, RMC and Antipole [22] of kNN.

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

Efficiency of GCon. We first report the efficiency of GCon. As shown in Fig. 11a–d on LiveJournal, Hollywood, GSH and Traffic, respectively, (1) on average GCon takes 109.7s to contract the graph, without the time of the computation for synopses. (2) It takes on average 4.13s, 21.2s, 18.1s, 0s and 3.38s only to compute the synopses for SubIso, TriC, Dist, CC and CD, respectively; *i.e.,* computing synopses of the five only takes on average 37.3% of the time of GCon. Recall that the synopses for SubIso suffice for us to answer CC queries; hence, it is unnecessary to compute synopses for CC.

Efficiency of IncCR. We tested the efficiency of IncCR, by varying $|\Delta G|$ from $5\%|G|$ to $35\%|G|$. As shown in Fig. 11e–h on LiveJournal, Hollywood, GSH and Traffic, respectively, (1) on average IncCR is 2.1 times faster than GCon, up to 6.3 times when $|\Delta G| = 5\%|G|$. It takes on average 26.6% time to update the synopses for 5% updates on the five 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 $|\Delta G|$.

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 we find that RO, IF and DF have a performance difference less than 3.5% on Hollywood. That is, IncCR is *stable* on batch updates, regardless of the order on the updates. Similarly, we find that RO, IF and DF have a performance difference less than 3.7% on Hollywood for vertex updates.

Impact of update rates. We also tested 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, Fig. 11i shows the following on LiveJournal. (1) On average it takes 0.88s to update contracted graphs, *i.e.,* IncCR is able to efficiently maintain the contracted graphs in real life. (2) The update time is less than 1s even when the updates are up to $0.8\%|G|$. IncCR can handle $0.8\%|G|/s$ of "burst" updates on graph with 40M nodes and edges.

The results are consistent on the other graphs.

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

Scalability on $|G|$. Varying the size $|G| = (|V|, |E|)$ of synthetic graphs from $(50M, 0.5B)$ to $(250M, 2.5B)$, we tested the scalability of GCon using a single machine. As shown in Fig. 11j, GCon scales well with G. It takes 1325s when graph G has $2.75B$ nodes and edges.

Scalability of PCon *and* IncPC. Fixing $|\Delta G| = 10\%|G|$, we tested the scalability of parallel PCon and IncPC with the number k of cores. As shown in Fig. 11k and l on GSH, (1) PCon scales well with k: it is 10.1 times faster when using $k = 20$ cores versus $k = 1$ (single core), and it is 4.3 times faster when k varies from 4 to 20. (2) IncPC is on average 1.9 times faster than PCon. (3) IncPC scales well with k; it is 3.7 times faster when k varies from 4 to 20, across 4 machines.

The results on other graphs are consistent.

Summary. We find the following over 10 real-life graphs. On average, (1) the contraction scheme reduces graphs by 71.9%. The contraction ratio is 0.34, 0.264, 0.213 and 0.365 in social networks, Web graphs, collaboration networks and road networks, respectively. (2) It improves the evaluation of SubIso, TriC, Dist, CC and CD by 1.69, 1.44, 1.47, 2.24 and 1.37 times, respectively. Existing algorithms can be adapted to the scheme, with indices or not. (3) On average, contracting the first three types of regular structures improves the efficiency of query evaluation by 1.61, 1.44 and 1.04 times, respectively. (4) Contracting obsolete data improves the efficiency of both conventional queries and temporal queries,

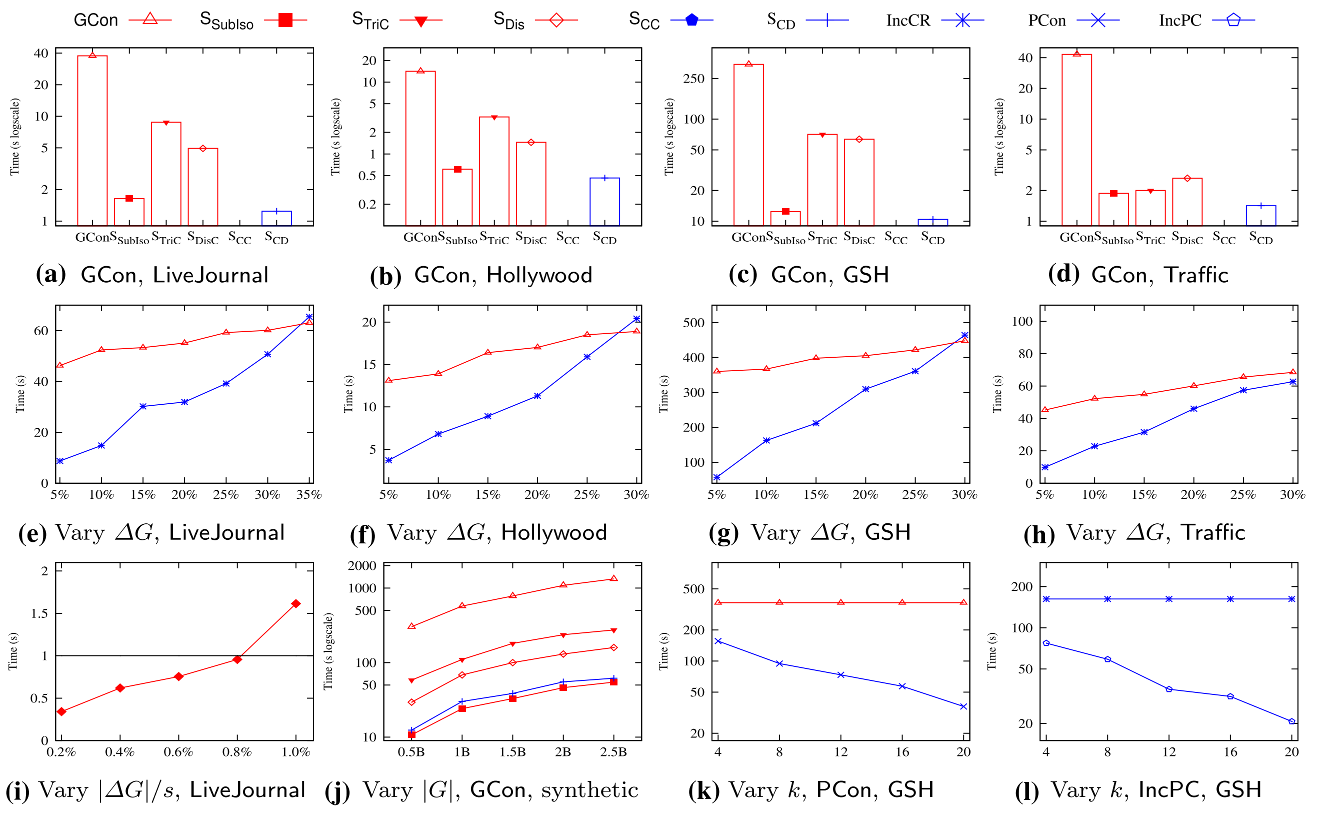


Fig. 11 Efficiency of (incremental) contraction

by 1.64 and 1.78 times on average, respectively. (5) Its total space cost on SubIso, TriC, Dist, CC and CD is only 12.7% of indexing costs of TurboIso, PLL, HINDEX and RMC. The synopses for the five query classes take only 48.3% of the total space of the contraction scheme. Thus, our contraction scheme scales with the number of applications. (6) Algorithms GCon, PCon, IncCR and IncPC scale well with graphs and updates. GCon takes 344s when G has 1.8B edges and nodes, and PCon takes only 33.1s with 20 cores, across 4 machines. IncCR is 4.9 times faster than GCon when $|\Delta G|$ is 5%$|G|$, and is still faster when $|\Delta G|$ is up to 30%$|G|$. (7) PCon and IncPC scale well with the number k of machines. When $|\Delta G| = 10\%|G|$, PCon is 4.3 times faster and IncPC is 3.7 times faster when k varies from 4 to 20.

6 Related work

This paper extends its conference version [38] as follows. (1) We identify a variety of frequent regular structures in different types of graphs, develop their synopses and contract graphs based on their types (Sect. 2.2). In contrast, [38] adopts an one-size-fit-all solution and contracts only cliques, paths and stars for all types of graphs. (2) In light of new regular structures, all examples and algorithms have been extended (Sects. 2–4). (3) We provide the pseudo code and details of a parallel contraction algorithm (Sect. 2.4). (4) We study two new query classes, namely, (non-local) connected component and (intractable) clique decision, for proof of concept (Sects. 3.4 and 3.5). We also extend the algorithms for the three other cases to cope with newly studied topological components (Sects. 3.1–3.3). (5) We extend the study of incremental contraction by presenting vertex updates and parallel incremental maintenance algorithm (Sect. 4). (6) The experimental study is almost entirely new and evaluates the contraction scheme *w.r.t.* different regular structures to contract as well as its effectiveness on new big graphs and new query classes of Sects. 3.4 and 3.5 (Sect. 5).

We discuss the other related work as follows.

Contraction. As a traditional graph programming technique [43], node contraction merges nodes, and subgraph contraction replaces connected subgraphs with supernodes. It is used in *e.g.*, single source shortest paths [54], connectivity [43] and spanning tree [41].

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

Compression. Graph compression has been studied for social network analysis [27], community queries [21], subgraph isomorphism [34,69], graph simulation [37], reachability and shortest distance [50], and GPU-based graph traversal [82]. It often computes query-specific equivalence relations by merging equivalent nodes into a single node or replacing frequent patterns by virtual nodes. Some are query preserving (lossless), *e.g.,* [37,50,69], and can answer certain types of queries on compressed graphs without decompression.

Another category of compression aims to minimize the number of bits required to represent a graph. WebGraph [15] exploits the inner redundancies of Web graphs; [8] proposes an encoding scheme based on node indices assigned by the BFS order; [24] approximates the optimal encoding with MinHash; and [52] removes the hub nodes for an scheme to have better locality.

Our contraction scheme differs from graph compression in the following. (a) It optimizes performance of multiple applications with the same contracted graph. In contrast, many compression schemes are query dependent and require different structures for different query classes. While some methods serve generic queries [8,15,24], they may incur heavy recovering cost. (b) Contraction is lossless, while some compression schemes are lossy, *e.g.,* [34]. (c) For a number of query classes, their existing algorithms can be readily adapted to contracted graphs, while compression often requires to develop new algorithms *e.g.,* [69] 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 [67] for a survey), classified as follows. (1) Node aggregation, *e.g.,* GraSS [60] 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 [87] generates an approximate summary of a graph structure by aggregating nodes based on attribute similarity. (2) Edge aggregation, *e.g.,* [73] generates a summary by aggregating edges, with a bounded number of edges different from the original graph. (3) Simplification: instead of aggregating nodes and edges, OntoVis [83] drops low-degree nodes, duplicate paths and unimportant labels. Most summarization methods are lossy, *e.g.,* GraSS and SNAP only retain part of attributes, and OntoVis drops nodes, edges and labels.

Incremental maintenance of summarization has been studied [30,46,84]. It depends on update intervals [84]; short-period summarization is space-costly, while long-interval summarization may miss updates. To handle these, [46] aggregates updates into a graph of "frequent" nodes and edges and computes a 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, contraction differs from summarization in the following. (1) The contraction scheme is *lossless* and returns exact answers for various classes of queries. In contrast, summarization is typically lossy and supports at best certain aggregate or approximate queries only. (2) Many 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) For a number of query classes studied, contracted graphs can be incrementally maintained with boundedness and locality, while summarization maintenance requires historical updates and often operates on the entire graph [46].

Indexing. Indices have been studied for, *e.g.,* subgraph isomorphism [13,14,28,44,72], reachability [7,23,50,95] and shortest distance [25,66]. They are query specific, and take space and time to store and maintain.

Our contraction scheme differs from indexing as it supports multiple applications on the same contracted graph, while a separate indexing structure 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, by building indices on smaller contracted graphs, as demonstrated in Sect. 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 the contraction scheme is effective.

A topic for future work is to build a hierarchy of contracted graphs by iteratively contracting regular structures into supernodes, until the one at the top fits into the memory; the objective is to make large graphs small enough to fit into the memory of a single machine, and make it possible to process large graphs under limited resources. Another topic is to study the capacity of a single multi-core machine for big graph analytics, by leveraging both contraction and multi-core parallelism.

Acknowledgements Fan, Li and Liu are supported in part by ERC 652976 and Royal Society Wolfson Research Merit Award WRM/R1/18 0014. 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. Traffic. http://www.dis.uniroma1.it/challenge9/download.html (2006)
2. DBLP. https://snap.stanford.edu/data/com-DBLP.html (2012)
3. Gsh host. http://law.di.unimi.it/webdata/gsh-2015-host (2015)
4. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD (2013)
5. Albert, R., Jeong, H., Barabási, A.: The diameter of the World Wide Web. CoRR **cond-mat/9907038** (1999)
6. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: SIGMOD, pp. 1421–1432 (2018)
7. Anirban, S., Wang, J., Islam, M.S.: Multi-level graph compression for fast reachability detection. In: DASFAA (2019)
8. Apostolico, A., Drovandi, G.: Graph compression by bfs. Algorithms **2**(3), 1031–1044 (2009)
9. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: SIGKDD, pp. 44–54 (2006)
10. Bae, S.H., Halperin, D., West, J.D., Rosvall, M., Howe, B.: Scalable and efficient flow-based community detection for large-scale graph analysis. TKDD **11**(3), 1–30 (2017)
11. Berry, N., Ko, T., Moy, T., Smrcka, J., Turnley, J., Wu, B.: Emergent clique formation in terrorist recruitment. In: AAAI Workshop on Agent Organizations (2004)
12. Besta, M., Hoefler, T.: Survey and taxonomy of lossless graph compression and space-efficient graph representations. CoRR arXiv:1806.01799 (2018)
13. Bhattarai, B., Liu, H., Huang, H.H.: CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In: SIGMOD (2019)
14. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing cartesian products. In: SIGMOD (2016)
15. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: WWW, pp. 595–602 (2004)
16. Bonacich, P.: Power and centrality: a family of measures. Am. J. Sociol. **92**(5), 1170–1182 (1987)
17. Bourse, F., Lelarge, M., Vojnovic, M.: Balanced graph edge partition. In: SIGKDD, pp. 1456–1465 (2014)
18. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Sociol. **25**(2), 163–177 (2001)
19. Bringmann, B., Nijssen, S.: What is frequent in a single graph? In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 858–863 (2008)
20. Bron, C., Kerbosch, J.: Algorithm 457: finding all cliques of an undirected graph. CACM **16**(9), 575–577 (1973)
21. Buehrer, G., Chellapilla, K.: A scalable pattern mining approach to web graph compression with communities. In: WSDM, pp. 95–106 (2008)
22. Cantone, D., Ferro, A., Pulvirenti, A., Recupero, D.R., Shasha, D.: Antipole tree indexing to support range search and k-nearest neighbor search in metric spaces. TKDE **17**(4), 535–550 (2005)
23. Cheng, J., Huang, S., Wu, H., Fu, A.W.C.: TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In: SIGMOD (2013)
24. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: SIGKDD, pp. 219–228 (2009)
25. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SICOMP **32**(5) (2003)
26. Cohen, J.: Trusses: Cohesive subgraphs for social network analysis. Natl. Secur. Agency Tech. Rep. **16**(3.1) (2008)
27. Cohen, S.: Data management for social networking. In: SIGMOD (2016)
28. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. TPAMI **26**(10), 1367–1372 (2004)
29. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
30. Cortes, C., Pregibon, D., Volinsky, C.: Communities of interest. In: IDA (2001)
31. Dijkstra, E.W., et al.: A note on two problems in connexion with graphs. Numer. Math. **1**(1) (1959)
32. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P., Larriba-Pey, J.L.: A discussion on the design of graph database benchmarks. In: TPCTC, pp. 25–40 (2010)
33. Elseidy, M., Abdelhamid, E., Skiadopoulos, S., Kalnis, P.: GRAMI: frequent subgraph and pattern mining in a single large graph. PVLDB **7**(7), 517–528 (2014)
34. Fairey, J., Holder, L.: Stariso: Graph isomorphism through lossy compression. In: DCC (2016)
35. Fan, W., Hu, C., Tian, C.: Incremental graph computations: Doable and undoable. In: SIGMOD (2017)
36. Fan, W., Jin, R., Liu, M., Lu, P., Tian, C., Zhou, J.: Capturing associations in graphs. PVLDB **13**(11) (2020)
37. Fan, W., Li, J., Wang, X., Wu, Y.: Query preserving graph compression. In: SIGMOD (2012)
38. Fan, W., Li, Y., Liu, M., Lu, C.: Making graphs compact by lossless contraction (2021). SIGMOD
39. Fan, W., Wu, Y., Xu, J.: Functional dependencies for graphs. In: SIGMOD (2016)
40. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An evolving query language for property graphs. In: SIGMOD (2018)
41. Gabow, H.N., Galil, Z., Spencer, T.H.: Efficient implementation of graph algorithms using contraction. In: FOCS (1984)
42. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York (1979)
43. Gross, J., Yellen, J.: Graph Theory and its Applications. CRC Press, Boca Raton (1998)
44. Han, W.S., Lee, J., Lee, J.H.: Turbo$_{\rm iso}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD (2013)
45. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-component labeling. Pattern Recogn. **42**(9) (2009)
46. Hill, S., Agarwal, D.K., Bell, R., Volinsky, C.: Building an effective representation for dynamic networks. J. Comput. Graph. Stat. **15**(3), 584–608 (2006)
47. Hu, X., Tao, Y., Chung, C.W.: Massive graph triangulation. In: SIGMOD (2013)
48. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SICOMP **7**(4), 413–423 (1978)

49. Jaakkola, M.S.T., Szummer, M.: Partially labeled classification with markov random walks. NIPS **14** (2002)
50. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD (2008)
51. Johnson, A.E., Pollard, T.J., Shen, L., Li-Wei, H.L., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L.A., Mark, R.G.: MIMIC-III, a freely accessible critical care database. Sci. Data **3**(1), 1–9 (2016)
52. Kang, U., Faloutsos, C.: Beyond'caveman communities': Hubs and spokes for graph compression and mining. In: ICDM, pp. 300–309 (2011)
53. Kang, U., McGlohon, M., Akoglu, L., Faloutsos, C.: Patterns on the connected components of terabyte-scale graphs. In: ICDM, pp. 875–880 (2010)
54. Karimi, R., Koppelman, D.M., Michael, C.J.: GPU road network graph contraction and SSSP query. In: ICS (2019)
55. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. JPDC **48**(1), 96–129 (1998)
56. Kempe, D., Kleinberg, J., Tardos, É.: Maximizing the spread of influence through a social network. In: SIGKDD, pp. 137–146 (2003)
57. Koch, I.: Enumerating all connected maximal common subgraphs in two graphs. TCS **250**(1–2), 1–30 (2001)
58. Kropatsch, W.: Building irregular pyramids by dual-graph contraction. In: Vision Image and Signal Processing (1996)
59. Lappas, T., Liu, K., Terzi, E.: Finding a team of experts in social networks. In: KDD (2009)
60. LeFevre, K., Terzi, E.: Grass: Graph structure summarization. In: SDM (2010)
61. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia. Semantic Web **6**(2), 167–195 (2015)
62. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Predicting positive and negative links in online social networks. In: WWW, pp. 641–650 (2010)
63. Leskovec, J., Kleinberg, J.M., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: SIGKDD (2005)
64. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. CoRR arXiv:0810.1355 (2008)
65. Leung, K., Leckie, C.: Unsupervised anomaly detection in network intrusion detection using clusters. In: ACSW (2005)
66. Liang, Y., Zhao, P.: Similarity search in graph databases: a multi-layered indexing approach. In: ICDE (2017)
67. Liu, Y., Safavi, T., Dighe, A., Koutra, D.: Graph summarization methods and applications: A survey. ACM Comput. Surv. **51**(3), 62:1-62:34 (2018)
68. Lu, C., Yu, J.X., Wei, H., Zhang, Y.: Finding the maximum clique in massive graphs. PVLDB **10**(11) (2017)
69. Maccioni, A., Abadi, D.J.: Scalable pattern matching over compressed graphs via dedensification. In: SIGKDD (2016)
70. McAuley, J., Leskovec, J.: Learning to discover social circles in ego networks. In: NIPS (2012)
71. Miller, G.A.: WordNet: a lexical database for English. Commun. ACM **38**(11), 39–41 (1995)
72. Myoungji, H., Hyunjoon, K., Geonmo, G., Kunsoo, P., Wook-Shin, H.: Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together. In: SIGMOD (2019)
73. Navlakha, S., Rastogi, R., Shrivastava, N.: Graph summarization with bounded error. In: SIGMOD (2008)
74. Newman, M.E., Watts, D.J., Strogatz, S.H.: Random graph models of social networks. PNAS **99**(suppl 1), 2566–2572 (2002)
75. Pandey, S., Li, X.S., Buluc, A., Xu, J., Liu, H.: H-index: Hash-indexing for parallel triangle counting on GPUs. In: HPCS, pp. 1–7 (2019)
76. Papadopoulos, S., Kompatsiaris, Y., Vakali, A., Spyridonos, P.: Community detection in social media. Data Min. Knowl. Discov. **24** (2012)
77. Ramalingam, G., Reps, T.: On the computational complexity of dynamic graph problems. TCS **158**(1–2), 233–277 (1996)
78. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. PVLDB **8**(5), 617–628 (2015)
79. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: A property graph query language. In: GRADES (2016)
80. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015)
81. Sakr, S., Al-Naymat, G.: Graph indexing and querying: a review. IJWIS **6**(2), 101–120 (2010)
82. Sha, M., Li, Y., Tan, K.: Gpu-based graph traversal on compressed graphs. In: SIGMOD, pp. 775–792 (2019)
83. Shen, Z., Ma, K.L., Eliassi-Rad, T.: Visual analysis of large heterogeneous social networks by semantic and structural abstraction. TVCG **12**(6), 1427–1439 (2006)
84. Soundarajan, S., Tamersoy, A., Khalil, E.B., Eliassi-Rad, T., Chau, D.H., Gallagher, B., Roundy, K.: Generating graph snapshots from streaming edge data. In: WWW (2016)
85. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
86. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From" think like a vertex" to" think like a graph". PVLDB **7**(3), 193–204 (2013)
87. Tian, Y., Hankins, R.A., Patel, J.M.: Efficient aggregation for graph summarization. In: SIGMOD (2008)
88. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103–111 (1990)
89. Vieira, M.V., Fonseca, B.M., Damazio, R., Golgher, P.B., Reis, D.d.C., Ribeiro-Neto, B.: Efficient search ranking in social networks. In: CIKM (2007)
90. W3C Recommendation: SPARQL query language for RDF. https://www.w3.org/TR/rdf-sparql-query/ (2008)
91. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world'networks. Nature **393**(6684), 440 (1998)
92. Wu, Y., Jin, R., Zhang, X.: Efficient and exact local search for random walk based top-k proximity query in large graphs. TKDE **28**(5), 1160–1174 (2016)
93. Yahia, S.A., Benedikt, M., Lakshmanan, L.V., Stoyanovich, J.: Efficient network aware search in collaborative tagging sites. PVLDB **1**(1), 710–721 (2008)
94. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: ICDM (2012)
95. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: Scalable reachability index for large graphs. PVLDB **3**(1-2) (2010)