THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Query preserving graph compression

Citation for published version:
Fan, W, Li, J, Wang, X & Wu, Y 2012, Query preserving graph compression. in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data.* SIGMOD '12, ACM, New York, NY, USA, pp. 157-168. https://doi.org/10.1145/2213836.2213855

Digital Object Identifier (DOI):
10.1145/2213836.2213855

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 19. Apr. 2024

Query Preserving Graph Compression

Wenfei Fan[1,2] Jianzhong Li[2] Xin Wang[1] Yinghui Wu[1,3]
[1]University of Edinburgh [2]Harbin Institute of Technology [3]UC Santa Barbara
{wenfei@inf, x.wang-36@sms, y.wu-18@sms}.ed.ac.uk lijzh@hit.edu.cn

ABSTRACT

It is common to find graphs with millions of nodes and billions of edges in, *e.g.,* social networks. Queries on such graphs are often prohibitively expensive. These motivate us to propose *query preserving graph compression*, to compress graphs *relative to* a class $\mathcal{Q}$ of queries of users' choice. We compute a small G_r from a graph G such that (a) for *any* query $Q \in \mathcal{Q}$, $Q(G) = Q'(G_r)$, where $Q' \in \mathcal{Q}$ can be efficiently computed from Q; and (b) any algorithm for computing $Q(G)$ can be *directly* applied to evaluating Q' on G_r *as is*. That is, while we cannot lower the complexity of evaluating graph queries, we reduce data graphs while preserving the answers to *all* the queries in $\mathcal{Q}$. To verify the effectiveness of this approach, (1) we develop compression strategies for two classes of queries: reachability and graph pattern queries via (bounded) simulation. We show that graphs can be efficiently compressed via a reachability equivalence relation and graph bisimulation, respectively, while preserving query answers. (2) We provide techniques for maintaining compressed graph G_r in response to changes ΔG to the original graph G. We show that the incremental maintenance problems are *unbounded* for the two classes of queries, *i.e.,* their costs are not a function of the size of ΔG and changes in G_r. Nevertheless, we develop incremental algorithms that depend only on ΔG and G_r, *independent of* G, *i.e.,* we do not have to decompress G_r to propagate the changes. (3) Using real-life data, we experimentally verify that our compression techniques could reduce graphs in average by 95% for reachability and 57% for graph pattern matching, and that our incremental maintenance algorithms are efficient.

1. INTRODUCTION

It is increasingly common to find large graphs in, *e.g.,* social networks [16], Web graphs [29] and recommendation networks [25]. For example, Facebook currently has more than 800 million users with 104 billion links[1]. It is costly to query such large graphs. Indeed, graph pattern matching takes quadratic time (by simulation [12]) or cubic time (via bounded simulation [9]) to determine whether there exists a match in a data graph for a graph pattern. Worse still, it is NP-complete when matching is defined in terms of subgraph isomorphism. Even for reachability queries that are to decide whether there exists a path connecting a pair of nodes in a graph $G = (V, E)$, it takes $O(|V| + |E|)$ time via DFS/BFS search. Although one may use indexes to speed up the evaluation, indexes incur extra cost, *e.g.,* a reachability matrix takes $O(|V|(|V|+|E|))$ time to build and $O(|V|^2)$ space to maintain (see [35] for a survey). Hence it is often *prohibitively expensive* to evaluate queries on graphs with millions of nodes and billions of edges, and it is *unlikely* that we can lower its computational complexity.

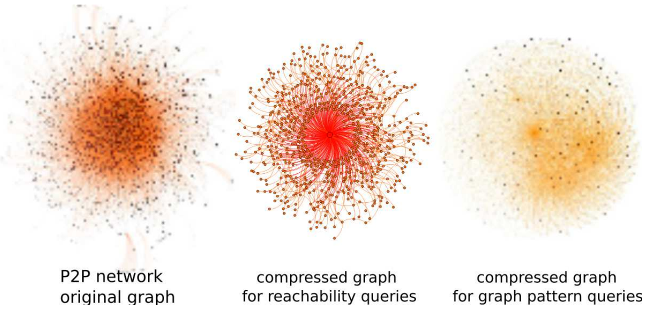


Figure 1: Compressing a real-life P2P network

Not all is lost. Observe that users typically adopt a class $\mathcal{Q}$ of queries when querying data graphs G. We propose *graph compression preserving queries of* $\mathcal{Q}$: given G, we find a smaller graph G_r via an efficient compression function R, such that for *all* queries $Q \in \mathcal{Q}$, $Q(G) = Q'(G_r)$, where Q' is a query in the same class $\mathcal{Q}$, computed from Q via an efficient query rewriting function. In other words, while we may not change the complexity functions of graph queries, we reduce the size of their parameters, *i.e.,* the data graphs.

In contrast to previous lossless compressions (*e.g.,* [3, 5, 11]), query preserving compression is *relative to* a class $\mathcal{Q}$ of queries of users' choice, *i.e.,* it generates small graphs that preserve the information *only relevant* to queries in $\mathcal{Q}$ rather than the entire original graphs, and hence, achieves a better compression ratio. Furthermore, any algorithm available for evaluating $\mathcal{Q}$ can be directly used to query the compressed graphs G_r *as is*, *without decompressing* G_r.

We find that this approach is effective when querying large graphs. For instance, a real-life P2P network can be reduced 94% and 51% for reachability and graph pattern queries, respectively, as depicted in Fig. 1. These reduce query evaluation time by 93% and 77%, respectively.

To illustrate the idea, let us consider an example.

Example 1: Graph G in Fig. 2 is a fraction of a multi-agent recommendation network. Each node denotes a customer (C), a book server agent (BSA), a music shop agent (MSA), or a facilitator agent (FA) assisting customers to find BSAs and MSAs. Each edge indicates a recommendation.

To locate potential buyers, a bookstore owner issues a pattern query $\mathsf{Q_p}$ depicted in Fig. 2. It is to find a set of BSAs such that they can reach a set of customers C who interact with a set of FAs, and moreover, the customers should be

[1] *http://www.facebook.com/press/info.php?statistics*; visited Jan. 2012

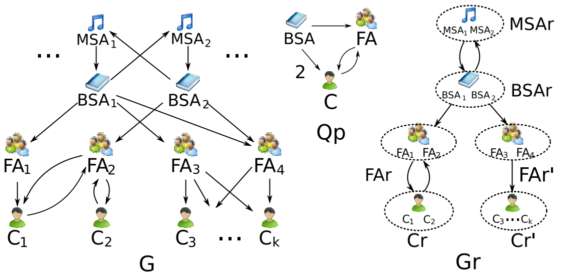


Figure 2: Recommendation Network

within 2 hops from the BSAs. One may verify that the match of $\mathsf{Q_p}$ in G is a relation $S = \{(X, X_i)\}$ for $X \in \{$BSA,FA,C$\}$ and $i \in [1, 2]$. It is expensive to compute S when G is large. Among other things, one has to check the connectivity between all the k customers and all the BSAs in G.

We can do better. Observe that BSA$_1$ and BSA$_2$ are of the same type of nodes (BSA), and both make recommendations to MSA and FA. Since they "simulate" the behavior of each other in the recommendation network G, they could be considered *equivalent* when evaluating $\mathsf{Q_p}$. Similarly, the pairs (FA$_1$, FA$_2$), (C$_1$, C$_2$), and any pair (C$_i$, C$_j$) of nodes for $i, j \in [3, k]$ can also be considered equivalent, among others.

This suggests that we build a compressed graph G_r of G, also shown in Fig. 2. Graph G_r consists of hypernodes X_r for $X \in \{$MSA,BSA,FA,FA',C,C'$\}$, each denoting a class of equivalent nodes. Observe that (1) G_r has less nodes and edges than G, (2) $\mathsf{Q_p}$ can be directly evaluated on G_r; its result $S_r = \{(\mathsf{X}, X_r)\}$ can be converted to the original result S by simply replacing X_r with the set of nodes represented by X_r; and (3) the evaluation of $\mathsf{Q_p}$ in G_r is more efficient than in G since, among other things, it only needs to check C_r and C'_r in G_r to identify matches for the query node C.

One can verify that G_r preserves the result for *all* pattern queries defined in terms of (bounded) simulation, not limited to $\mathsf{Q_p}$. That is, for any such pattern query Q on G, we can directly evaluate Q on the much smaller G_r instead. □

Contributions. Our main contributions are as follows.

(1) We propose *query preserving compression* for querying large real-life graphs (Section 2). As opposed to previous graph compression strategies, it only preserves information needed for answering queries in a particular class $\mathcal{Q}$ of users' interest, and hences, achieves a better compression ratio. It is not yet another algorithm for evaluating graph queries; instead, *any algorithms* for evaluating queries of $\mathcal{Q}$ on the original graphs can be directly applied to computing query answers in the compressed graphs, without decompression.

To verify the effectiveness of this approach, we develop query preserving compression strategies for two classes of queries commonly used in practice, namely, reachability queries and graph pattern queries via (bounded) simulation.

(2) For reachability queries, we introduce *reachability preserving compression* (Section 3). We propose a notion of *reachability equivalence relations*, and based on this, we provide a compression function R that, given a graph G, computes a small graph $G_r = R(G)$ in $O(|V||E|)$ time, where $|V|$ and $|E|$ are the number of the nodes and edges in G, respectively. We show that G_r is reachability preserving: for any reachability query Q, one can find in *constant time* another reachability query Q' such that $Q(G) = Q'(G_r)$.

(3) For graph pattern queries defined in terms of (bounded) simulation [9, 12], we define *graph pattern preserving compression* in terms of a *bisimulation equivalence relation* [8] (Section 4). We show that graphs G can be compressed into a smaller G_r in $O(|E| \log |V|)$ time. We also show that the compression preserves pattern queries: for any graph pattern Q, G matches Q if and only if G_r matches the same Q, and moreover, the match of Q in G can be computed in G_r.

(4) Real-life graphs constantly change [16]. This highlights the need for studying *incremental query preserving compression*. Given a graph G, its compression G_r via function R, and updates ΔG to G, it is to compute changes ΔG_r to G_r such that $R(G \oplus \Delta G) = G_r \oplus \Delta G_r$, where $G \oplus \Delta G$ denotes G updated by ΔG. When ΔG is small as commonly found in practice, ΔG_r tends to be small as well and hence, is more efficient to find than recomputing $R(G \oplus \Delta G)$ starting from scratch. This allows us to compute compression G_r *once*, and *incrementally* maintain it in response to changes to G.

We study this issue for reachability queries and graph pattern queries (Section 5). (a) We provide a complexity analysis for the problem, in terms of the size of the changes in the input (ΔG) and output (ΔG_r) characterized by *affected area* (AFF). We show that the problem is *unbounded* for both classes of queries, *i.e.,* its cost is not a function of |AFF|. (b) Nevertheless, we develop incremental maintenance algorithms: (i) for reachability preserving compression, we show that compressed graphs can be maintained in $O(|\mathsf{AFF}||G_r|)$ time; and (ii) for graph pattern queries, we incrementally compress graphs in $O(|\mathsf{AFF}|^2 + |G_r|)$ time. In both cases the algorithms are independent of the original graph G, and propagate changes *without decompressing* G_r.

(5) We experimentally verify the effectiveness and efficiency of our (incremental) compression techniques using synthetic data and real-life data. We find that query preserving compression reduces the size of real-life graphs by 95% and 57% in average for reachability and pattern queries, respectively, and by 98% and 59%, respectively, for social networks. These lead to a reduction of 94% and 70% in query evaluation time, respectively. In addition, our incremental compression algorithms for reachability queries outperform their batch counterparts when changes are up to 20%.

We contend that query preserving compression yields a promising approach to querying real-life graphs. This work is among the first efforts to provide a complete package for query preserving compression, from complexity bounds to compression algorithms to incremental maintenance.

Related work. We categorize related work as follows.

General graph compression. Graph compression has been studied for *e.g.,* Web graphs and social networks [2, 5, 27]. The idea is to encode a graph or its transitive closure into compact data structures via node ordering determined by, *e.g.,* lexicographic URL and hosts [27], linkage similarity [3], and document similarity [5]. These general methods preserve the information of the entire graph, and highly depend on extrinsic information, coding mechanisms and application domains [2]. To overcome the limitations, [2] proposes a compression-friendly node ordering but stops short of giving a compression strategy. Our work differs from these in the following: (a) our compression techniques rely only on intrinsic graph information that is relevant to a specific class of queries; (b) our *compressed graphs* can be directly queried without decompression; in contrast, even to answer simple queries, previous work requires the original graph to be restored from *compact structures* [5], as observed in [2]; and (c) we provide efficient incremental maintenance algorithms.

Query-friendly compression. Closer to our work are compression methods developed for specific classes of queries.

(1) Neighborhood queries [18,22,27], to find nodes connected to a designated node in a graph. The idea of query-able compression (querying without decompression) for such queries is advocated in [18], which adopts compressed data structures by exploiting Eulerian paths and multi-position linearization. A S-node representation is introduced in [27] for answering neighborhood queries on Web graphs. Graph summarization [22] aims to sketch graphs with small subgraphs and construct hypergraph abstraction. These methods construct compact data structures that have to be (partially) decompressed to answer the queries [2]. Moreover, the query evaluation algorithms on original graphs have to be modified to answer queries in their compact structures.

(2) Reachability queries [1, 10, 21, 32]. To answer such queries, [21] computes the minimum subgraphs with the same transitive closure as the original graphs, and [1] reduces graphs by substituting a simple cycle for each strongly connected component. These methods allow reachability queries to be evaluated on compressed graphs without decompression. We show in Section 3 (and verify in Section 6) that our method achieves a better compression ratio, because (1) our compressed graphs do not have to be subgraphs of the original graphs, and (2) by merging nodes into hypernodes, we can further reduce edges. Bipartite compression [10] reduces graphs by introducing dummy nodes and compressing bicliques. However, (1) its compression is a bijection between graphs and their compressed graphs, such that they can be converted to each other. In contrast, we do not require that the original graphs can be restored; and (2) algorithms for reachability queries have to be modified before they can be applied to their compressed graphs [10]. [32] computes a compressed bit vector to encode the transitive closure of a graph. In contrast, we compute compressed graphs on which reachability algorithms and the compression scheme in [32] can be directly applied. The incremental maintenance of the bit vectors is not addressed in [32].

(3) Path queries [4]. There has also been work on compressing XML trees via bisimulation, to evaluate XPath queries. It is shown there that this may lead to exponential reduction, an observation that carries over to our setting. In contrast to [4], we consider compressing general graphs, to answer graph-structured queries rather than XPath. Moreover, we develop incremental techniques to maintain compressed graphs, which are not studied in [4].

We are not aware of any previous work on compressing graphs for answering graph pattern queries.

Graph indexing. There has been a host of work on building indexes on graphs to improve the query time [6,11,13–15,19, 26,34]. (1) 2-hop [6], PathTree [14], 3-hop [13], GRAIL [34] and HLSS [11] are developed for answering reachability queries. However, (a) these indexes come with high costs. For example, the construction time is biquadratic for 2-hop and 3-hop, cubic for HLSS, and quadratic for GRAIL and PathTree; the space costs of these indexes are all (near) quadratic [11, 32, 34, 34, 35]; and maintenance for 2-hop index easily degrades into recomputation [35]. (b) The algorithms for reachability queries on original graphs often do not run on these indexes. For example, it requires extra search or auxiliary data structures to answer the queries involving nodes that are not covered by PathTree [14, 32]. In contrast, all these algorithms can be directly applied to our compressed graphs. (2) 1-index [19], $A(k)$-index [15] and their generalization $D(k)$-index [26] yield index graphs as structure summarizations based on (parameterized) graph bisimulation. However, (a) only rooted graphs are considered for those indexes; and (b) those indexes are for regular path queries, instead of graph patterns and reachability queries. Indeed, none of these indexes preserves query results for reachability queries (shown in Section 3), and neither $A(k)$-index nor $D(k)$-index preserves query results for graph pattern queries (shown in Section 4); (c) those indexes are only accurate for those queries satisfying certain query load constraints (*e.g.,* query templates [19], path lengths [15,26]); in contrast, we compute compressed graphs that preserve results for *all queries* in a given query class; and (d) Incremental maintenance is not studied for 1-index and $A(k)$-index [15, 19]. The issue is addressed in [26], but the technique there depends on the query load constraints.

Incremental bisimulation. We use graph bisimulation to compress graphs for pattern queries. A bisimulation computation algorithm is given in [8]. Incremental computation of bisimulation for single edge insertions is studied in [7,30]. Our work differs from these in (1) that we give complexity bounds (boundedness and unboundedness results) of incremental pattern preserving compression, of which incremental bisimulation is a subproblem, and (2) that we propose algorithms for batch updates instead of single updates.

2. PRELIMINARY

Below we first review graphs and graph queries. We then introduce the notion of query preserving graph compression.

2.1 Data Graphs and Graph Queries

Graphs. A *labeled (directed) graph* $G = (V, E, L)$ consists of (1) a set V of nodes; (2) a set of edges $E \subseteq V \times V$, where $(v, w) \in E$ denotes a *directed* edge from node v to w; and (3) a function L defined on V such that for each node v in V, $L(v)$ is a label from a set Σ of labels. Intuitively, the node labels may present *e.g.,* keywords, social roles, ratings [16].

We use the following notations. A *path* ρ from node v to w in G is a sequence of nodes $(v = v_0, v_1, \ldots, v_n = w)$ such that for every $i \in [1, n]$, $(v_{i-1}, v_i) \in E$. The *length* of path ρ, denoted by $\mathsf{len}(\rho)$, is n, *i.e.,* the number of edges in ρ. A path ρ is said to be *nonempty* if $\mathsf{len}(\rho) \geq 1$. A node v can *reach* w (or w is reachable from v) if and only if (iff) there exists a path from v to w in G. The *distance* between node v and w is the length of the shortest paths from v to w.

Graph queries. In general, a *graph query* is a computable function from a graph to another object, *e.g.,* a Boolean value, a graph, a relation, etc. It is independent of how the input data graphs are represented and therefore, ask for certain intrinsic properties of the graphs. In this paper, we consider two classes of queries commonly used in practice.

Reachability queries. A reachability query on a graph G, denoted by $\mathsf{Q_R}(v, w)$, is a Boolean query that asks whether node v can reach node w in G. For instance, $\mathsf{Q_R}(\mathsf{BSA}_1, \mathsf{FA}_2)$ is a reachability query on graph G of Fig. 2; the answer to the query is true, as there is a path from BSA_1 to FA_2 in G.

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "Q"'] & Gr \arrow[d, "Q'"] \\
Q(G) & Q'(Gr) \arrow[l, "P"']
\end{tikzcd}$$

(a) General framework

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "Q_R"'] & Gr \arrow[d, "Q'_R"] \\
Q_R(G) \arrow[r, leftrightarrow] & Q'_R(Gr)
\end{tikzcd}$$

(b) Reachability preserving compression

$$\begin{tikzcd}
G \arrow[r, "R"] \arrow[d, "Q_P"'] & Gr \arrow[d, "Q_P"] \\
Q_P(G) & Q_P(Gr) \arrow[l, "P"']
\end{tikzcd}$$

(c) Graph pattern preserving compression

Figure 3: Query preserving compression

Graph patterns. We define graph pattern matching in terms of *bounded simulation* [9]. A graph pattern query is defined as $\mathsf{Q_p} = (V_p, E_p, f_v, f_e)$, where (1) (V_p, E_p, f_v) is a directed graph as defined above; and (2) f_e is a function defined on E_p such that for each edge (u, u'), $f_e(u, u')$ is either a positive integer k or a symbol $*$, called the *bound* of (u, u').

A graph $G = (V, E, L)$ *matches* $\mathsf{Q_p}$, denoted by $\mathsf{Q_p} \trianglelefteq G$, if there exists a binary relation $S \subseteq V_p \times V$ such that: (1) for each $u \in V_p$, there exists $v \in V$ such that $(u, v) \in S$; (2) for each $(u, v) \in S$, (a) $f_v(u) = L(v)$, and (b) for each edge (u, u') in E_p, there exists a nonempty path ρ from v to v' in G such that $(u', v') \in S$, and $\mathsf{len}(\rho) \leq k$ if $f_e(u, u')$ is a constant k. We refer to S as *a match* for P in G.

Intuitively, $(u, v) \in S$ if (1) node v in G satisfies the search condition specified by $f_v(u)$ in $\mathsf{Q_p}$, and (2) each edge (u, u') in $\mathsf{Q_p}$ is mapped to a nonempty path $\rho = (v, \ldots, v')$ in G, such that (u', v') is also in the match S, and moreover, $\mathsf{len}(\rho)$ is bounded by k if $f_e(u, u') = k$. If $f_e(u, u') = *$, $\mathsf{len}(\rho)$ is not bounded. Observe that the child u' of u is mapped to a *descendant* v' of v via S. For instance, relation S given in Example 1 is a match in graph G for pattern P of Fig. 2.

It has been shown [9] that there exists a *unique maximum* match S_M in G for $\mathsf{Q_p}$ if $\mathsf{Q_p} \trianglelefteq G$; *i.e.,* for any match S in G for P, $S \subseteq S_M$. The *answer to* $\mathsf{Q_p}$ *in* G is defined as the maximum match S_M if $\mathsf{Q_p} \trianglelefteq G$, and as $\emptyset$ otherwise.

Lemma 1 [9]: *For any graph G and pattern $\mathsf{Q_p}$, if $\mathsf{Q_p} \trianglelefteq G$, then there is a unique maximum match in G for P.* □

There are two special cases of graph pattern queries. (1) A *Boolean* pattern query $\mathsf{Q_p}$ returns true if $\mathsf{Q_p} \trianglelefteq G$, and false otherwise. (2) A pattern query $\mathsf{Q_p}$ via *graph simulation* [12] is a query in which $f_e(u, u') = 1$ for each edge $(u, u') \in E_p$ of $\mathsf{Q_p}$, *i.e.,* it maps edges in $\mathsf{Q_p}$ to edges in a data graph.

2.2 Query Preserving Graph Compression

For a class $\mathcal{Q}$ of queries, a *query preserving graph compression* is a triple $<R, F, P>$, where R is a *compression function*, $F : \mathcal{Q} \rightarrow \mathcal{Q}$ is a *query rewriting function*, and P is a *post-processing function*. For any graph G, $G_r = R(G)$ is a graph computed from G by R, referred to as the *compressed graph* of G, such that $|G_r| \leq |G|$, and *for any query* $Q \in \mathcal{Q}$,

- ○ $Q(G) = P(Q'(G_r))$, and
- ○ *any algorithm* for evaluating $\mathcal{Q}$ queries can be *directly used to* compute $Q'(G_r)$, *without decompressing* G_r.

Here $Q' = F(Q)$, $Q(G)$ is the answer to Q in G, $Q'(G_r)$ is the answer to Q' in G', and $P(Q'(G_r))$ is the result of post-processing the answer to Q' in the compressed G_r; and

As shown in Fig. 3(a), (1) for any query $Q \in \mathcal{Q}$, the answer $Q(G)$ to Q in G can be computed by evaluating Q' in the (smaller) compressed graph G_r of G; (2) the compression is *generic*: any data structures and indexing techniques for the original graph can be directly applied to G_r (*e.g.,* the 2-hop techniques of [6], see Section 6); (3) the post-processing function finds the answer in the original G by *only* accessing the query answer $Q'(G_r)$ and an index on the inverse of node mappings of R; (4) in contrast to generic lossless compression schemes (*e.g.,* [10]), we do not need to restore the original graph G from G_r, and moreover, the compressed graph G_r is not necessarily a subgraph of G.

For instance, a query preserving compression for graph pattern queries is described in Example 1, where the compression function R groups nodes into hypernodes based on graph bisimulation; the query rewriting function F is the identity mapping: for any pattern query Q, $F(Q) = Q$; and the post-processing function P simply replaces each hypernode with the set of equivalent nodes it represents.

In Sections 3 and 4, we show that there exist query preserving compressions with efficient R, F and P functions.

(1) For reachability queries, R reduces graph G by 95% in average, in $O(|V||E|)$ time; and F is in *O(1) time.* Moreover, as shown in Fig. 3(b), post-processing P is not needed at all.

(2) For pattern queries, R reduces the size of G by 57% in average, in $O(|E|\log|V|)$ time; F is the *identity* mapping, and P is in *linear time* in the size of the query answer, a cost *necessary* for any evaluation algorithm (see Fig. 3(c)). Better still, for Boolean pattern queries, P is no longer needed.

We remark that for each graph G, its compression $G_r = R(G)$ is computed *once* for *all queries* in $\mathcal{Q}$, and is *incrementally maintained* in response to updates to G (Section 5).

3. COMPRESSION FOR REACHABILITY

In this section we study query preserving compression for reachability queries, referred to as *reachability preserving compression.* The main result of the section is as follows.

Theorem 2: *There exists a reachability preserving compression $<R, F>$, where R is in quadratic time, and F is in constant time, while no post-processing P is required.* □

As a proof of the theorem, we first define the compression $<R, F>$ in Section 3.1. We then provide an algorithm for implementing the compression function R in Section 3.2.

3.1 Reachability Equivalence Relations

Our compression is based on the following notion.

Reachability equivalence relations. We first define a *reachability relation* on a graph $G = (V, E, L)$ to be a binary relation $R_e \subseteq V \times V$ such that for each $(u, v) \in R_e$ and any node $x \in V$, (1) x can reach u iff x can reach v; and (2) u can reach x iff v can reach x. Intuitively, $(u, v) \in R_e$ if and only if they have the same set of ancestors and the same set of descendants. One can readily verify the following.

Lemma 3: *For any graph G, (1) there is a unique maximum reachability relation R_e on G, and (2) R_e is an equivalence relation,* i.e., *it is reflexive, symmetric and transitive.* □

The *reachability equivalence relation of* G is the maximum reachability relation of G, denoted by $R_e(G)$ or simply R_e. We denote by $[v]_{R_e}$ the equivalence class containing node v.

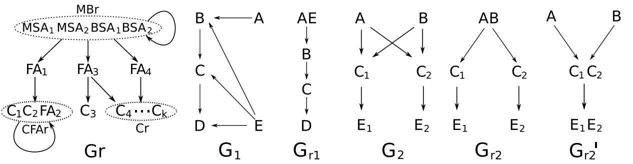


Figure 4: Reachability equivalence

Example 2: Consider graph G given in Fig. 2. One can verify that (BSA_1,BSA_2) $\in R_e(G)$. Indeed, BSA_1 and BSA_2 share the same ancestors and descendants. Similarly, (MSA_1,MSA_2) $\in R_e(G)$. In contrast, (FA_3, FA_4) $\notin R_e(G)$ since FA_3 can reach C_3, while FA_4 cannot. □

Reachability preserving compression. Based on reachability equivalence relations we define $<R, F>$ as follows.

(1) Compression function R. Given $G = (V, E, L)$, we define $R(G) = G_r = (V_r, E_r, L_r)$, where (a) $V_r = \{[v]_{R_e} \mid v \in V\}$; (b) E_r consists of all edges $([v]_{R_e}, [w]_{R_e})$ if there exist nodes $v' \in [v]_{R_e}$ and $w' \in [w]_{R_e}$ such that $(v', w') \in E$, and (c) for each $u \in V_r$, $L_r(u) = \sigma$, where σ is a fixed label in Σ. Here R_e is the reachability equivalence relation of G.

Intuitively, (a) for each node $v \in V$, there exists a node $[v]_{R_e}$ in V_r; abusing R, we use $R(v)$ to denote $[v]_{R_e}$; (b) for each edge $(v, w) \in E$, $(R(v), R(w))$ is an edge in E_r; and (c) all the node labels in G_r are fixed to be a symbol σ in Σ since node labels are irrelevant to reachability queries.

(2) Query rewriting function F. We define F such that for any reachability query $\mathsf{Q_R}(v, w)$ on G, $F(\mathsf{Q_R}(v, w)) = Q'$, where $Q' = \mathsf{Q_R}(R(v), R(w))$ is a reachability query on G_r. It simply asks whether there is a path from $[v]_{R_e}$ to $[w]_{R_e}$ in G_r. Using index structures for the equivalence classes of R_e, Q' can be computed from $\mathsf{Q_R}(v, w)$ in constant time.

Correctness. One can easily verify that $<R, F>$ is a reachability preserving compression. Indeed, $|G_r| \leq |G|$ since $|V_r| \leq |V|$ and $|E_r| \leq |E|$. Moreover, for any reachability query $\mathsf{Q_R}(v, w)$ posed on G, one can show by contradiction that there exists a path from v to w in G if and only if $R(v)$ can reach $R(w)$ in G_r. Hence, given $\mathsf{Q_R}(v, w)$ on G, one can find its answer in G by evaluating $\mathsf{Q_R}(R(v), R(w))$ in the smaller compressed graph G_r of G, as shown in Fig. 3(b).

Example 3: Recall graph G of Fig. 2. Using the reachability preserving compression $<R, F>$ given above, one can get $G_r = R(G)$ shown in Fig. 4, in which, *e.g.,* $R(\mathsf{C}_1) = R(\mathsf{C}_2) = R(\mathsf{FA}_2) = \mathsf{CFA}_r$. Given a reachability query $\mathsf{Q_R}(\mathsf{BSA}_1,\mathsf{C}_2)$ on G, $F(\mathsf{Q_R}) = \mathsf{Q_R}(\mathsf{MB}_r,\mathsf{CFA}_r)$ on the smaller G_r. As another example, G_{r_1} and G_{r_2} in Fig. 4 are the compressed graphs generated by R for G_1 and G_2 of Fig. 4, respectively. □

As remarked earlier, there has been work on index graphs based on bisimulation [15, 19, 26]. However, such indexes do not preserve reachability. To see this, consider the index graph G'_{r_2} of G_2 shown in Fig. 4, where $\{C_1, C_2\}$ and $\{E_1, E_2\}$ are bisimilar and thus merged [19]. However, G'_{r_2} cannot be directly queried to answer *e.g.,* $\mathsf{Q_R}(C_1, E_2)$ posed on G_2, *i.e.,* one cannot find its equivalent reachability query on G'_{r_2}. Indeed, C_2 can reach E_2 in G_2 but C_1 does not, while in G'_{r_2}, C_1 and C_2 are merged into a single node.

3.2 Compression Method for Reachability Queries

We next present an algorithm that, given a graph $G = (V, E, L)$, computes its compressed graph $G_r = R(G)$ based on the compression function R given earlier. The algorithm, denoted as $\mathsf{compress_R}$, is shown in Fig. 5.

```
Input: A graph $G = (V, E, L)$.
Output: A compressed graph $G_r = R(G) = (V_r, E_r, L_r)$.

1. set $V_r := \emptyset$, $E_r := \emptyset$;
2. compute reachability preserving relation $R_e$;
3. compute the partition $\mathsf{Par} := V/R_e$ of $G$;
4. for each $S \in \mathsf{Par}$ do
5.     create a node $v_S$; $L_r(v_S) := \sigma$; $V_r := V_r \cup \{v_S\}$;
6. for each $v_S, v_{S'} \in V_r$ do
7.     if there exist $u \in S$, $v \in S'$ such that
           $(u, v) \in E$ but $v_S$ does not reach $v_{S'}$
8.     then $E_r := E_r \cup \{(v_S, v_{S'})\}$;
9. return $G_r = (V_r, E_r, L_r)$;
```

Figure 5: Algorithm $\mathsf{compress_R}$ for reachability

Given a graph G, the algorithm first computes its reachability equivalence relation R_e and the induced partition Par by R_e over the node set V (lines 2-3). Here R_e is found as follows (details omitted): for each node in V, it computes its ancestors and descendants, via forward (resp. backward) BFS traversals, respectively; it identifies those nodes with the same ancestors and descendants. After this, for each equivalence class $S \in \mathsf{Par}$, it creates a node v_S representing S, assigns a fixed label σ to v_S, and adds v_S to V_r (lines 4-5). It constructs the edge set E_r by connecting nodes $(v_S, v_{S'})$ in V_r if (1) there exists an edge $(v, w) \in E$ of G, where v and w are in the equivalence classes represented by S and S', respectively, and (2) v_S does not reach $v_{S'}$ via E_r (lines 6-8). Condition (2) assures that $\mathsf{compress_R}$ inserts no redundant edges, *e.g.,* if $(v_S, v_{S'})$ and $(v_{S'}, v_{S''})$ are already in E_r, then $(v_S, v_{S''})$ is *not* added to E_r. While it is a departure from the reachability equivalence relation R_e, it is an optimization without losing reachability information, as noted for transitivity equivalent graphs [1] (lines 6-8). The compressed graph G_r is then constructed and returned (line 9).

Correctness & Complexity. One can verify that the algorithm correctly computes G_r by the definition of R given above. In addition, $\mathsf{compress_R}$ is in $O(|V|^2{+}|V||E|)$ time. Indeed, R_e and Par can be computed in $O(|V|(|V|{+}|E|))$ time (lines 2-3). The construction of G_r is in $O(|V_r|(|V_r| + E_r|))$ time (lines 4-8). This completes the proof of Theorem 2.

Optimizations. Instead of compressing G directly, we first compute its SCC graph G_{scc}, which collapses each strongly connected component into a single node without losing reachability information. We then apply $\mathsf{compress_R}$ to G_{scc}, which is often much smaller than $|G|$ (see Section 6).

Note that $|G_r|$ is much smaller than reachability matrices [35], which take $O(|V|^2)$ space. Further, G_r takes substantially less construction time (quadratic) and space (linear) as opposed to 2-hop indexing [6], which is biquadratic.

4. COMPRESSION FOR GRAPH PATTERN

We next present a query preserving compression for graph pattern queries, referred to as *graph pattern preserving compression.* The main result of the section is as follows.

Theorem 4: *There exists a graph pattern preserving compression $<R, F, P>$ in which for any graph $G = (V, E, L)$, R is in $O(|E| \log |V|)$ time, F is the identity mapping, and P is in linear time in the size of the query answer.* □

To show the result above, we first define the compression

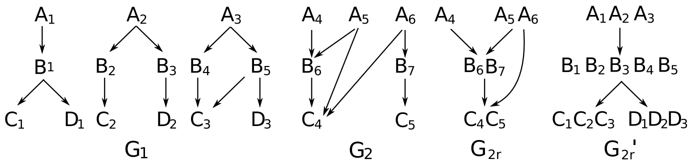


Figure 6: Examples of bisimulation relations

$<R, F, P>$ in Section 4.1. We then provide an algorithm to implement the compression function R in Section 4.2.

4.1 Compressing Graphs via Bisimilarity

We construct a graph pattern preserving compression in terms of *bisimulation* relations, which are defined as follows.

Bisimulation relations [8]. A *bisimulation relation* on a graph $G = (V, E, L)$ is a binary relation $B \subseteq V \times V$, such that for each $(u, v) \in B$, (1) $L(u) = L(v)$; (2) for each edge $(u, u') \in E$, there exists an edge $(v, v') \in E$, such that $(u', v') \in B$; and (3) for each edge $(v, v') \in E$, there exists an edge $(u, u') \in E$ such that $(u', v') \in B$.

Intuitively, $(u, v) \in B$ if and only if for each child u' of u, there exists an child v' of v such that $(u', v') \in B$, and vice versa. Similar to Lemma 3, one can verify the following.

Lemma 5: *For any graph G, (1) there is a unique maximum bisimulation relation R_b on G, and (2) R_b is an equivalence relation,* i.e., *it is reflexive, symmetric and transitive.* □

We define the *bisimulation equivalence relation* of G to be the maximum bisimulation relation of G, denoted by $R_b(G)$ or simply R_b. We denote by $[v]_{R_b}$ *the equivalence class containing node v*. We say that nodes u and v are *bisimilar* if $(u, v) \in R_b$. Since for any nodes v and v' in $[v]_{R_b}$, $L(v) = L(v')$, we simply call $L(v)$ the *label* of $[v]_{R_b}$.

Example 4: Recall the graph G given in Fig. 2. One can verify that FA_3 and FA_4 are bisimilar. In contrast, FA_2 and FA_3 are not bisimilar; indeed, FA_2 has a child C_2, which is not bisimilar to any C child of FA_3.

Consider graphs given in Fig. 6. Note that A_1 and A_2 in G_1 are not bisimilar, as there is no child of A_1 bisimilar to child B_2 or B_3 of A_2. Similarly, A_1 and A_3 in G_1 are not bisimilar. In contrast, A_5 and A_6 in G_2 are bisimilar.

Note that A_4 and A_5 in G_2 are not bisimilar, but they are in the same reachability equivalence class; while A_5 and A_6 are bisimilar, they are not reachability equivalent. This illustrates the *difference* between the reachability equivalence relation and the bisimulation equivalence relation. □

Graph pattern preserving compression. Based on bisimulation equivalence relations, we define $<R, F, P>$.

(1) Compression function R. Given $G = (V, E, L)$, we define $R(G) = G_r = (V_r, E_r, L_r)$, where (a) $V_r = \{[v]_{R_b} \mid v \in V\}$; (b) an edge $([v]_{R_b}, [w]_{R_b})$ is in E_r as long as there exist nodes $v' \in [v]_{R_b}$ and $w' \in [w]_{R_b}$ such that $(v', w') \in E$, and (c) for each $[v]_{R_b} \in V_r$, $L_r([v]_{R_b})$ is its label $L(v)$. Intuitively, (a) for each node $v \in V$, there exists a node $[v]_{R_b}$ in V_r; (b) for each edge $(v, w) \in E$, $([v]_{R_b}, [w]_{R_b})$ is an edge in E_r; and (c) each $[v]_{R_b}$ has the same label as $L(v)$.

(2) Query rewriting function F is simply the identity mapping, *i.e., $F(\mathsf{Q_p}) = \mathsf{Q_p}$*.

(3) Post processing function P. Recall that $\mathsf{Q_p}(G)$ is the maximum match in G for pattern $\mathsf{Q_p}$. We define P such that $P(\mathsf{Q_p}(G_r)) = \mathsf{Q_p}(G)$ as follows. For each $(v_p, [v]_{R_b}) \in \mathsf{Q_p}(G_r)$ and each $v' \in [v]_{R_b}$, $(v_p, v') \in \mathsf{Q_p}(G)$. Intuitively, if $[v]_{R_b}$ simulates v_p in G_r, then so does each $v' \in [v]_{R_b}$ in G. Hence, P expands $\mathsf{Q_p}(G_r)$ by replacing $[v]_{R_b}$ with all the nodes v' in the class $[v]_{R_b}$, in $O(|\mathsf{Q_p}(G)|)$ time via an index structure for the inverse node mapping of R. When $\mathsf{Q_p}$ is a Boolean pattern query, P is not needed.

```
Input: A graph $G = (V, E, L)$.
Output: A compressed graph $G_r = R(G) = (V_r, E_r, L_r)$.

$V_r := \emptyset$; $E_r := \emptyset$;
compute the maximum bisimulation relation $R_b$ of $G$;
compute the partition $\mathsf{Par} := V/R_b$;
for each $S \in \mathsf{Par}$ do
    create a node $v_S$ and set $L_r(v_S) := L(v)$ where $v \in S$;
    $V_r := V_r \cup \{v_S\}$;
for each $v_S, v_{S'} \in V_r$ do
    if there exist $u \in S$ and $v \in S'$ such that $(u, v) \in E$
    then $E_r := E_r \cup \{(v_S, v_{S'})\}$;
return $G_r = (V_r, E_r, L_r)$;
```

Figure 7: Algorithm $\mathsf{compress_B}$ for pattern queries

Example 5: Recall the graph G of Fig. 2. Using the graph pattern preserving compression $<R, F, P>$, one can get the compressed graph G_r of G shown in Fig. 2, in which *e.g.,* $R(\mathsf{FA}_1) = R(\mathsf{FA}_2) = \mathsf{FA}_r$, where FA_r is the equivalence class containing FA_1 and FA_2. For the graph G_2 of Fig. 6, its compressed graph $R(G_2)$ is G_{2_r}, as shown in Fig. 6. □

Correctness. We show that $<R, F, P>$ given above is indeed a graph pattern preserving compression. (1) $|G_r| \leq |G|$, as $|V_r| \leq |V|$ and $|E_r| \leq |E|$. (2) For any pattern query $\mathsf{Q_p}$, $\mathsf{Q_p}(G) = P(\mathsf{Q_p}(G_r))$. To see this, it suffices to verify that $(u, v) \in \mathsf{Q_p}(G)$ if and only if $(u, [v]_{R_b}) \in \mathsf{Q_p}(G_r)$. If $(u, [v]_{R_b}) \in \mathsf{Q_p}(G_r)$, then for any child u' of u, there is a node $[v']_{R_b}$ such that $(u', [v']_{R_b}) \in \mathsf{Q_p}(G_r)$, and there is a bounded path ρ from $[v]_{R_b}$ to $[v']_{R_b}$. By the definition of R, we can show that for each node $w \in [v]_{R_b}$, there is a node $w' \in [v']_{R_b}$, to which there is a path ρ' from w such that $\mathsf{len}(\rho) = \mathsf{len}(\rho')$, $(u, w) \in \mathsf{Q_p}(G)$ and $(u', w') \in \mathsf{Q_p}(G)$. Conversely, if $(u, v) \in \mathsf{Q_p}(G)$, then one can show that for any node w bisimilar to v in G, $(u, w) \in \mathsf{Q_p}(G)$, and moreover, for each query edge (u, u'), $[v]_{R_b}$ has a bounded path to a node $[v']_{R_b}$ in G_r with $(u', v') \in \mathsf{Q_p}(G)$. Hence $(u, [v]_{R_b}) \in \mathsf{Q_p}(G_r)$. From these it also follows that $P(\mathsf{Q_p}(G_r))$ is indeed the unique maximum match in G for $\mathsf{Q_p}$. In light of this, as shown in Fig. 3(c), we can find the match of $\mathsf{Q_p}$ in G by computing $P(\mathsf{Q_p}(G_r))$ via *any* algorithm for answering $\mathsf{Q_p}$.

As remarked earlier, $A(k)$-index and $D(k)$-index [15, 26] may *not* preserve the answers to graph pattern queries. To see this, consider graph G_1 of Fig. 6 and its index graph G'_{2_r} of $A(k)$-index when $k = 1$, also shown in Fig. 6. Although A_1, A_2 and A_3 are not bisimilar, they all have and only have B children; as such, they are 1-bisimilar [26], and are merged into a single node in G'_{2_r}. However, G'_{2_r} cannot be directly queried by *e.g.,* a $\mathsf{Q_p}$ consisting of two query edges $\{(B, C), (B, D)\}$, both with bound 1. Indeed, for $\mathsf{Q_p}$, G'_{2_r} returns all the B nodes in G as matches for query node B in $\mathsf{Q_p}$, while only B_1 and B_5 are the true matches in G_1.

4.2 Compression Algorithm for Graph Pattens

We next present an algorithm that computes the compressed graph $G_r = R(G)$ for a given graph $G = (V, E, L)$, where R is the compression function given earlier.

The algorithm, denoted as $\mathsf{compress_B}$, is shown in Fig. 7. Given a graph $G = (V, E, L)$, $\mathsf{compress_B}$ first computes the maximum bisimulation relation R_b of G, and finds the induced partition Par by R_b over the node set V (lines 2-3). To do this, it follows [8]: it first partitions V into $\{S_1, \ldots, S_k\}$, where each set S_i consists of nodes with the same label; the algorithm then iteratively refines Par by splitting S_i if it does not represents an equivalence class of R_b, until a fixpoint is reached (details omitted). For each class $S \in$ Par, $\mathsf{compress_B}$ then creates a node v_S, assigns the label of a node $v \in S$ to v_S, and adds v_S to V_r (lines 4-6). For each edge $(u, v) \in E$, it adds an edge $(v_S, v_{S'})$, where u and v are in the equivalence classes represented by v_S and $v_{S'}$, respectively (lines 7-9). Finally $G_r = (V_r, E_r, L_r)$ is returned (lines 10).

Correctness & Complexity. Algorithm $\mathsf{compress_B}$ indeed computes the compressed graph G_r by the definition of R (Section 4.1). In addition, $\mathsf{compress_B}$ is in $O(|E| \log |V|)$ time: R_b and Par can be computed in $O(|E| \log |V|)$ time [8] (lines 2-3), and G_r can be constructed in $O(|V_r| + |E|)$ time (lines 4-9). This completes the proof of Theorem 4.

5. INCREMENTAL COMPRESSION

To cope with the dynamic nature of social networks and Web graphs, incremental techniques have to be developed to maintain compressed graphs. Given a query preserving compression $<R, F, P>$ for a class $\mathcal{Q}$ of queries, a graph G, a compressed graph $G_r = R(G)$ of G, and *batch updates* ΔG (a list of edge deletions and insertions) to G, the *incremental query preserving compression* problem is to compute changes ΔG_r to G_r such that $G_r \oplus \Delta G_r = R(G \oplus \Delta G)$, *i.e.,* the updated compressed graph $G_r \oplus \Delta G_r$ is the compressed graph of the updated graph $G \oplus \Delta G$. It is known that while real-life graphs are constantly updated, the changes are typically minor [23]. As remarked earlier, when ΔG is small, ΔG_r is often small as well. It is thus often more efficient to compute ΔG_r than compressing $G \oplus \Delta G$ starting from scratch, by minimizing unnecessary recomputation.

As observed in [28], it is no longer adequate to measure the complexity of incremental algorithms by using the traditional complexity analysis for batch algorithms. Following [28], we characterize the complexity of an incremental compression algorithm in terms of the size of the *affected area (*AFF*)*, which indicates the changes in the input ΔG and the output ΔG_r, *i.e.,* $|\mathsf{AFF}| = |\Delta G| + |\Delta G_r|$. An incremental algorithm is said to be *bounded* if its time complexity can be expressed as a function $f(|\mathsf{AFF}|)$, *i.e.,* it depends only on $|\Delta G| + |\Delta G_r|$ rather than the entire input G. An incremental problem is *bounded* if there exists a bounded incremental algorithm for it, and is *unbounded* otherwise.

5.1 Incremental Maintenance for Reachability

We first study the incremental graph compression problem for reachability queries, referred to as *incremental reachability compression* and denoted as RCM. One may want to develop a bounded algorithm for incremental reachability compression. The problem is, however, nontrivial.

Theorem 6: RCM *is unbounded even for unit update,* i.e., *a single edge insertion or deletion.* □

Proof sketch: We verify this by reduction from the *single source reachability problem* (SSR). Given a graph G_s, a fixed source node s and updates ΔG_s, SSR is to decide whether for all $u \in G_s$, s reaches u in $G_s \oplus \Delta G_s$. It is known that SSR is unbounded [28]. We show that SSR is bounded iff RCM with unit update is bounded. □

Incremental algorithm. Despite the unbounded result, we present an incremental algorithm for RCM that is in $O(|\mathsf{AFF}||G_r|)$ time, *i.e.,* it only depends on $|\mathsf{AFF}|$ and $|G_r|$ instead of $|G|$, and solves RCM *without decompressing* G_r.

To present the algorithm, we need the following notations. (1) A *strongly connected component* (SCC) *graph* $G_{\mathsf{scc}} = (V_{\mathsf{scc}}, E_{\mathsf{scc}})$ merges each strongly connected component into a single node without self cycle. We use v_{scc} to denote an SCC node containing v, and E_{scc} the edges between SCC nodes. (2) The *topological rank* $r(s)$ *of a node* s *in* G is defined as follows: (a) $r(s) = 0$ if s has no child in G, *i.e.,* s_{scc} has no child in G_{scc}, (b) $r(s) = r(s')$ if s and s' are in the same SCC, and otherwise, (c) $r(s) = \max(r(s')) + 1$ when s' ranges over the children of s. We also define $r(e) = r(s)$ for an edge update $e = (s, v)$. One can verify the lemma below, which reveals the connection between topological ranks and the reachability equivalence relation R_e in a graph.

Lemma 7: *In any graph* G, $r(u) = r(v)$ *if* $(u, v) \in R_e$. □

Leveraging Lemma 7, we present the algorithm, denoted as incRCM and shown in Fig. 8. It has three steps.

(1) Preprocessing. The algorithm first preprocesses updates ΔG and compressed graph G_r (lines 1–2). (a) It first removes redundant updates in ΔG that have no impact on reachability (line 1). More specifically, it removes (i) edge insertions (u, u') where $[u]_{R_e} \neq [u']_{R_e}$, and $[u]_{R_e}$ can reach $[u']_{R_e}$ in G_r; and (ii) edge deletions (u, u') if either $[u]_{R_e}$ reaches $[u']_{R_e}$ via a path of length no less than 2 in G_r, or if $[u]_{R_e} = [u']_{R_e}$, and there is a child u'' of u such that $(u, u'') \notin \Delta G$ and $[u]_{R_e} = [u'']_{R_e}$. (b) It then identifies a set of nodes u with $r(u)$ changed in G_r, for each edge update $(u, u') \in \Delta G$; it updates the rank of u in G_r accordingly.

(2) Updating. The algorithm then updates G_r based on r (line 3). It first splits those nodes $[u]_{R_e}$ of G_r in which there exist nodes with different ranks. By Lemma 7, these nodes are not in the same equivalence class, thus $[u]_{R_e}$ must be split. Then it finds all the newly formed SCCs in G, and introduce a new node for each of them in G_r. These two steps identify an initial area affected by updates ΔG.

(3) Propagation. The algorithm then locates ΔG_r by propagating changes from the initial affected area identified in step (2). It processes updates $e = (u, u')$ in the ascending topological rank (line 4). It first finds $[u]_{R_e}$ and $[u']_{R_e}$, the (revised) equivalence classes of u and u' in the current compressed graph G_r. It then invokes procedure incRCM^+ (resp. incRCM^-) to update G_r when e is to be inserted (resp. deleted) (lines 5–8). Updating G_r may make some updates in ΔG redundant, which are removed from ΔG (line 9). After all updates in ΔG are processed, the updated compressed graph G_r is returned (line 10).

Given an edge $e = (u, u')$ to be inserted into G and their corresponding nodes $[u]_{R_e}$ and $[u']_{R_e}$ in G_r, procedure incRCM^+ updates G_r as follows. First, note that since (u, u') is not redundant (by lines 1 and 9 of incRCM), u cannot reach u' in G, but after the insertion of e, u' becomes a child of u. Moreover, no nodes in $[u]_{R_e} \setminus \{u\}$ can reach u' in G. Hence u and nodes in $[u]_{R_e} \setminus \{u\}$ can no longer be in the same equivalence class after the insertion of e. Thus incRCM^+ splits $[u]_{R_e}$ into two nodes representing $\{u\}$ and $[u]_{R_e} \setminus \{u\}$, respectively; similarly for $[u']_{R_e}$ (line 1). This is done by invoking procedure Split (omitted).

In addition, nodes may also have to be merged (lines 2–8).

```
Input: A graph $G$, its compressed graph $G_r$, batch updates $\Delta G$.
Output: New compressed graph $G_r \oplus \Delta G_r$.
1.  reduce $\Delta G$;
2.  update the topological rank $r$ of the nodes in $G_r$ w.r.t. $\Delta G$;
3.  update $G_r$ w.r.t. the updated $r$;
4.  for each update $e = (u, u') \in \Delta G$
    following the ascending topological rank do
5.      if $e$ is an edge insertion
6.        then incRCM$^+(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
7.      else if $e$ is an edge deletion
8.        then incRCM$^-(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
9.      reduce $\Delta G$;
10. return $G_r$;

Procedure incRCM$^+$
Input: Compressed graph $G_r = (V_r, E_r)$, edge insertion $(u, u')$,
       and node $[u]_{R_e}, [u']_{R_e}$ in $G_r$.
Output: An updated $G_r$.
1.  Split $(u, u', [u]_{R_e}, [u']_{R_e})$;
2.  if $r([u]_{R_e}) > r([u']_{R_e})$ then
3.      for each $v \in B([u]_{R_e})$ do Merge $(\{u\}, v)$;
4.      for each $v' \in B([u']_{R_e})$ do Merge $(\{u'\}, v')$;
5.  else if $r([u]_{R_e}) = r([u']_{R_e})$ then
6.      for each $v \in P([u']_{R_e})$ do Merge $(\{u\}, v)$);
7.      for each $v' \in C([u]_{R_e})$ do Merge $(\{u'\}, v')$);
8.  return $G_r$;
```

Figure 8: Algorithm incRCM

We denote the set of children (resp. parents) of a node u as $C(u)$ (resp. $P(u)$), and use $B(u)$ to denote the set of nodes having the same parents as u. By Lemma 7, consider $r(u)$ and $r(u')$ in the updated G. Observe that $r(u) \geq r(u')$ since u' is a child of u after the insertion of e. (1) If $r(u) > r(u')$, *i.e.,* u and u' are not in the same SCC, then $\{u\}$ may only be merged with those nodes $v' \in B([u]_{R_e})$ such that $C(\{u\}) = C(v')$; similarly for u' (lines 2–4). Hence we invoke procedure Merge (omitted) that works on G_r: given nodes w and w', it checks whether $P(w) = P(w')$ and $C(w) = C(w')$; if so, it merges w and w' into one that shares the same parents and children as w and w'. (2) When $r(u) = r(u')$, as e is non-redundant, u and u' may not be in the same SCC. Thus $\{u\}$ (resp. $\{u'\}$) may only be merged with a parent of $[u']_{R_e}$ (resp. a child of $[u]_{R_e}$; lines 5–7).

Similarly, procedure incRCM$^-$ updates G_r by using Split and Merge in response to the deletion of an edge (omitted). Here when a node is split, its parents may need to be split as well, *i.e.,* the changes are propagated upward.

Example 6: Recall graph G of Fig 2. A subgraph G_s (excluding e_1 and e_2) of G and its compressed graph G_r are shown in Fig 9. (1) Suppose that edges e_1 and e_2 are inserted into G_s. Algorithm incRCM first identifies e_1 as a redundant insertion, since FA$_1$ can reach v in G_r (line 1). It then updates the rank r of FA$_1$ to be 0 due to the insertion of e_2 (line 2), by traversing G_r to identify a newly formed SCC. It next invokes procedure incRCM$^+$ (line 6), which merges FA$_1$ to the node v in G_r, and constructs G'_r as the compressed graph, shown in Fig 9. The affected area AFF includes nodes v, v_r and edge (v_r, v_r). (2) Now suppose that edges e_3 and e_4 are removed. The algorithm first identifies e_3 as a redundant update, since FA$_1$ has a child C$_2$ in the nodes V_r. It then processes update e_4 by updating the rank of FA$_2$, and splits the node v_r in G'_r into FA$_2$ and v'_r via incRCM$^-$ (line 8). This yields G''_r by updating G'_r (see Fig 9). The AFF includes nodes v_r, v'_r, C$_1$ and their edges. □

Correctness & Complexity. Algorithm incRCM correctly maintains the compressed graph G_r. Indeed, one can verify that the loop (lines 3-7) guarantees that for any nodes u and u' of G, u can reach u' if and only if $[u]_{R_e}$ reaches $[u']_{R_e}$ in G_r when G_r is updated in response to ΔG. In particular, procedure Merge is justified by the following: nodes can be merged iff they share same parents and children after non-redundant updates. This can be verified by contradiction.

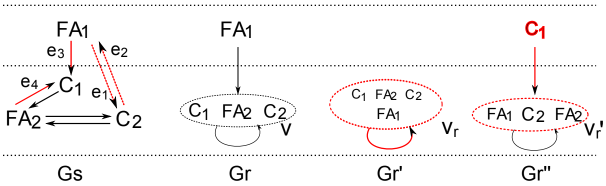


Figure 9: Incremental compression: reachability

For the complexity, one can show that the first two steps of the algorithm (lines 1-3) are in $O(|\text{AFF}||G_r|)$ time. Indeed, (1) it takes $O(|\text{AFF}||G_r|)$ time to identify redundant updates by testing the reachability of the nodes in G_r, which accesses R but does *not* search G; and (2) it takes $O(|\text{AFF}||G_r|)$ time to identify the nodes and their changed rank for each update in ΔG, and updates G_r accordingly. Procedures incRCM$^+$ and incRCM$^-$ are in $O(|\text{AFF}||G_r|)$ time. Thus incRCM is in $O(|\text{AFF}||G_r|)$ time. As will be verified by our experimental study, $|G_r|$ and $|\text{AFF}|$ are typically small in practice.

5.2 Incremental Maintenance for Graph Patterns

We next study the incremental graph compression problem for graph pattern queries, referred to as *incremental graph pattern preserving compression* and denoted as PCM. Like RCM, PCM is also unbounded and hard.

Theorem 8: PCM *is unbounded even for unit update.* □

Proof sketch: We show that SSR is bounded iff PCM with unit update is bounded, also by reduction from SSR. □

Incremental algorithm. Despite this, we develop an incremental algorithm for PCM that is in $O(|\text{AFF}|^2 + |G_r|)$ time. Like incRCM, the complexity of the algorithm is independent of $|G|$. It solves PCM *without decompressing* G.

We first define some notations. (1) A strongly connected component graph G_{scc} is as defined in Section 5.1. (2) Following [8], we define the *well founded* set WF to be the set of nodes that cannot reach any cycle in G, and the *non-well-founded* set NWF to be $V \setminus$ WF. (3) Based on (1) and (2), we define the *rank* $r_b(v)$ of nodes v in G: (a) $r_b(v) = 0$ if v has no child; (b) $r_b(v) = -\infty$ if v_{scc} has no child in G_{scc} but v has children in G; and (c) $r_b(v) = \max(\{r_b(v') + 1\} \cup \{r_b(v'')\})$, where $(v_{\text{scc}}, v'_{\text{scc}})$ and $(v_{\text{scc}}, v''_{\text{scc}})$ are in E_{scc}, for all $v' \in$ WF and all $v'' \in$ NWF. We also define $r_b([u]_{R_b}) = r_b(u)$ for a node $[u]_{R_b}$ in G_r, and $r_b(e) = r_b(v)$ for an update $e = (u, v)$.

Analogous to Lemma 7, we show the lemma below.

Lemma 9: *For any graph G and its compressed graph G_r, (1) $r_b(u) = r_b(v)$ if $(u, v) \in R_b$, and (2) each node u in G_r can only be affected by updates e with $r_b(e) < r_b(u)$.* □

For PCM, the affected area AFF includes (1) the nodes in G with their ranks changed after G is modified, as well as the edges attached to them, and (2) the changes to G_r, including the updated nodes and the edges attached to them.

Our incremental algorithm is based on Lemma 9, denoted as incPCM and shown in Fig. 10. It has two steps.

(1) Preprocessing. The algorithm first finds an initial af-

```
Input: A graph $G$, a compressed graph $G_r$, batch updates $\Delta G$;
Output: An updated $G_r$.
1.  AFF:= $\emptyset$;
2.  incR($G, G_r, \Delta G$); /* update rank and $G_r$ */
3.  for each $i \in \{-\infty\} \cup [0, \max(r_b(v))]$ do
4.      AFF := AFF.add {AFF$_i$}, where AFF$_i$ is
          the set of new nodes $v$ with $r_b(v) = i$;
5.  for each AFF$_i$ of ascending rank order do
6.      PT(AFF$_i$); /*update compressed graph at rank i*/
7.      minDelta(AFF$_i, G_r, \Delta G$); update AFF;
8.      for each $[u']_{R_b} \in$ AFF$_i$ and $e = (u, u') \in \Delta G$ do
9.          SplitMerge($[u']_{R_b}, G_r, e,$ AFF);
10. return $G_r$;

Procedure SplitMerge
Input: Compressed graph $G_r = (V_r, E_r, L_r)$, an update $(u, u')$,
       node $[u']_{R_b}$, AFF;
Output: An updated $G_r$.
1.  Boolean flag := true; AFF$_p$ := $\emptyset$;
2.  AFF$_p$ := AFF$_p \cup \{[u]_{R_b}\} \cup P([u']_{R_b})$;
3.  for each node $[v_p]_{R_b} \in$ AFF$_p$ with $r([v_p]_{R_b}) > r([u']_{R_b})$ do
    /* split $[v_p]_{R_b}$ w.r.t. $v$ into $[v_{p1}]_{R_b}$ and $[v_{p2}]_{R_b}$ */
4.      flag := bSplit $([v_p]_{R_b}, [u']_{R_b})$;
5.      if flag then
6.          AFF$_{r_b([v_p]_{R_b})}$ := AFF$_{r_b([v_p]_{R_b})}$ $\cup\{[v_p]_{R_b}, [v_{p2}]_{R_b}\}$;
7.          for each $v'$ with $r_b(v') = r_b([v_p]_{R_b})$ do
8.              if mergeCon $(v', [v_p]_{R_b})$ then bMerge $(v', [v_p]_{R_b})$;
9.          for each $v''$ with $r_b(v'') = r_b([v_{p2}]_{R_b})$ do
10.             if mergeCon $(v'', [v_{p2}]_{R_b})$ then bMerge $(v'', [v_{p2}]_{R_b})$;
11. update AFF; return $G_r$;
```

Figure 10: Algorithm incPCM

fected area AFF (lines 1-4). It uses procedure incR (omitted) to do the following (line 2) : (a) update the rank of the nodes in the updated G; and (b) split those nodes $[u]_{R_b}$ of G_r in which there exist nodes with different ranks. By Lemma 9, these nodes are not bisimilar. It then initializes AFF, consisting of AFF$_i$ for each rank i of G, where AFF$_i$ is the set of newly formed nodes in G_r with rank i (lines 3–4).

(2) Propagating. It then identifies ΔG_r by processing each AFF$_i$ in the ascending rank order (lines 5-9). At each iteration of the loop (lines 5-9), it first computes the bisimulation equivalence relation R_b of the subgraph induced by the new nodes in AFF$_i$ (line 6), via procedure PT (omitted). Revising the Paige-Tarjan algorithm [24], PT performs a fixpoint computation until each node of rank i in G_r finds its bisimulation equivalence class. The algorithm then reduces those updates that become redundant via procedure minDelta (see optimization below), and reduces AFF accordingly (line 7). It then propagates changes from AFF$_i$ towards the nodes with higher ranks, by invoking procedure SplitMerge.

Given an affected node $[u']_{R_b}$ and an update $e = (u, u')$, procedure SplitMerge identifies other nodes that are affected. It starts with $[u]_{R_b}$ and its parents $P([u]_{R_b})$ (line 2). For each $[v_p]_{R_b}$ of these nodes with a rank higher than $[u']_{R_b}$, SplitMerge splits it into $[v_{p1}]_{R_b}$ and $[v_{p2}]_{R_b}$, denoting node sets $[v_1]_{R_b} \setminus [u']_{R_b}$ and $[v_1]_{R_b} \cap [v_2]_{R_b}$, respectively (line 4). Indeed, no nodes $v_{p1} \in [v_{p1}]_{R_b}$ and $v_{p2} \in [v_{p2}]_{R_b}$ are bisimilar. Here we call $[u']_{R_b}$ a *splitter* of $[v_p]_{R_b}$, and conduct the splitting via procedure bSplit (omitted). The changes are propagated to AFF$_{r_b(v_p)}$ (line 6). SplitMerge then merges $[v_{p1}]_{R_b}$ with nodes having the same rank; similarly for $[v_{p2}]_{R_b}$ (lines 7-10). The merging takes place under condition mergeCon, specified and justified by the lemma below.

Lemma 10: *Nodes v_1 and v_2 can be merged in G_r if and only if (1) they have the same label, and (2) there exists no node v_3 that is a splitter of v_1 but is not a splitter of v_2.* □

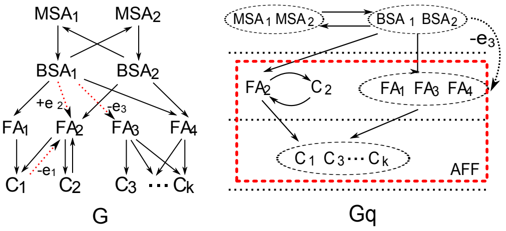


Figure 11: Incremental compression: graph pattern

Optimization. Procedure minDelta reduces redundant updates based on rules. Consider a node $[u']_{R_b}$ in G_r updated by incPCM (line 6). For a node $[u]_{R_b}$ with $r_b([u]_{R_b}) \geq r_b([u']_{R_b})$, we give some example rules used by minDelta (the full set of rules is omitted). (1) *Insertion*: The insertion of (u, w) is redundant if $w \in [u']_{R_b}$ and $([u]_{R_b}, [u']_{R_b}) \in E_r$. (2) *Deletion*: The deletion of (u, w) is redundant if $w \in [u']_{R_b}$, $([u]_{R_b}, [u']_{R_b}) \in E_r$, u has a child w' in $[u']_{R_b}$ and $w \neq w'$. (3) *Cancellation*: An insertion (u, u_1) and a deletion (u, u_2) are both redundant if there is a node u_3 such that $\{u_1, u_2, u_3\} \subseteq [u']_{R_b}$, $(u, u_3) \in E$ and $([u]_{R_b}, [u']_{R_b}) \in E_r$.

Example 7: Recall G and its compressed graph G_r from Fig 2. Consider removing e_1 and e_3 from G, followed by the insertion of e_2, as indicated in Fig 11. When e_1 is removed, the algorithm incPCM first updates the rank of C$_1$ (line 2), and adds C$_1$ to AFF (line 4). Since C$_1$ has a different rank from C$_2$, it is split from (C$_1$,C$_2$) at the same time (line 4). The algorithm then invokes PT to merge C$_1$ and (C$_3$,. . .,C$_k$) (line 6), and uses SplitMerge to (a) remove FA$_1$ from (FA$_1$,FA$_2$), and (b) merges FA$_1$ with (FA$_3$,FA$_4$) (line 9). Observe that the deletion of e_3 becomes redundant, as identified by minDelta (line 7). The updated compressed graph G_q is shown in Fig 11, in which AFF is marked. □

Correctness & Complexity. One can verify that incPCM correctly maintains compressed graphs, by induction on the rank of nodes in G_r processed by the algorithm. For its complexity, note that procedure incR is in $O(|\text{AFF}| \log |\text{AFF}|)$ time. Moreover, procedures minDelta, PT and SplitMerge take $O(|\text{AFF}|)$ time, $O(|\text{AFF}| \log |\text{AFF}| + |G_r|)$, and $O(|\text{AFF}|^2)$ time in total. Hence incPCM is in $O(|\text{AFF}|^2 + |G_r|)$ time. The algorithm accesses R and G_r, *without* searching G.

6. EXPERIMENTAL EVALUATION

We next present an experimental study using both real-life and synthetic data. For reachability and graph pattern queries, we conducted four sets of experiments to evaluate: (1) the effectiveness of the query preserving compressions proposed, measured by compression ratio, *i.e.,* the ratio of the compressed graph size to the original graph size, (2) query evaluation time over original and compressed graphs, (3) the efficiency of the incremental compression algorithms, and (4) the effectiveness of incremental compression.

Experimental setting. We used the following datasets.

(1) Real-life data. For graph pattern queries, we used the following graphs with attributes and labels on the nodes: (a) Youtube[2] where nodes are videos labeled with their category; (b) California[3], a Web graph in which each node is a host labeled with its domain; (c) Citation [31], a citation

[2] *http://netsg.cs.sfu.ca/youtubedata/*
[3] *http://www.cs.cornell.edu/courses/cs685/2002fa/*

dataset	$\lvert G\rvert(\lvert V\rvert, \lvert E\rvert)$	RC$_{aho}$	RC$_{scc}$	RC$_r$
facebook	$1.6M$ ($64K$, $1.5M$)	13.19%	5.89%	0.028%
amazon	$1.5M$ ($262K$, $1.2M$)	35.09%	18.94%	0.18%
Youtube	$931K$($155K$, $796K$)	41.60%	17.02%	1.77%
wikiVote	$111K$($7K$, $104K$)	65.56%	8.33%	1.91%
wikiTalk	$7.4M$ ($2.4M$, $5.0M$)	48.21%	16.82%	3.27%
socEpinions	$585K$ ($76K$, $509K$)	29.53%	19.59%	2.88%
NotreDame	$1.8M$ ($326K$, $1.5M$)	43.27%	10.75%	2.61%
P2P	$27K$ ($6K$, $21K$)	73.24%	17.02%	5.97%
Internet	$155K$ ($52K$, $103K$)	88.32%	28.89%	16.08%
citHepTh	$381K$ ($28K$, $353K$)	71.32%	37.15%	14.70%

Table 1: Reachability preserving: compression ratio

network in which nodes represent papers, labeled with their publishing information; and (d) Internet[4] where a node represents an autonomous system labeled with its location.

For reachability queries, we used (a) six social networks: a Wikipedia voting network wikiVote[5], a Wikipedia communication network wikiTalk[5], an online social network a product co-purchasing network amazon[5], socEpinions[5], a fragment of facebook [33], and Youtube[2];(b) three Web graphs: a peer-to-peer network P2P[5], a Web graph NotreDame[5], and Internet[4]; and (c) a citation network citHepTh[5].

The sizes of these graphs (the number $|V|$ of nodes and the number $|E|$ of edges) are shown in Tables 1 and 2.

(2) Synthetic data. We designed a graph generator to produce synthetic graphs. Graph generation was controlled by three parameters: the number of nodes $|V|$, the number of edges $|E|$, and the size $|L|$ of the node label set L.

(3) Pattern generator. We implemented a generator for graph pattern queries controlled by four parameters: the number of query nodes V_p, the number of edges E_p, label set L_p along the same lines as their counterpart L for data graphs, and an upper bound k for edge constraints.

(4) Implementation. We implemented the following algorithms, in Java. (1) our compression algorithms compress$_R$ (Section 3) and compress$_B$ (Section 4); (2) AHO [1] which, as a comparison to compress$_R$, computes transitive reduced graphs; (3) our incremental compression algorithms incRCM and incPCM for batch updates (Section 5); we also implemented IncBsim, an algorithm that invokes the algorithm of [30] (for a single update) multiple times when processing batch updates; (4) query evaluation algorithms: for reachability queries, the breadth-first (resp. bidirectional) search algorithm BFS (resp. BIBFS); for pattern queries, algorithm Match and its incremental version IncBMatch [9]; and (5) algorithms for building 2-hop indexes [6].

All experiments were run on a machine powered by an Intel Core(TM)2 Duo 3.00GHz CPU with 4GB of memory, using scientific linux. Each experiment was run 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness: Compression ratio. We first evaluate the compression ratios of our methods using real-life data. We define the *compression ratio* of compress$_R$ to be RC$_r$ = $|G_r|/|G|$, where G is the original graph and G_r is its compressed graph by compress$_R$. Similarly, we define PC$_r$ of compress$_B$, and RC$_{aho}$ of AHO [1] in which G_r denotes the transitive reduced graph. We also consider SCC graphs G_{scc} (Section 3), and define RC$_{scc}$ as $|G_r|/|G_{scc}|$ to evaluate the effectiveness of compress$_R$ on SCC graphs.

dataset	$\lvert G\rvert(\lvert V\rvert, \lvert E\rvert, \lvert L\rvert)$	PC$_r$
California	$26K$ ($10K$, $16K$, 95)	45.9%
Internet	$155K$ ($52K$, $103K$, 247)	29.8%
Youtube	$951K$ ($155K$, $796K$, 16)	41.3%
Citation	$1.2M$ ($630K$, $633K$, 67)	48.2%
P2P	$27K$ ($6K$, $21K$, 1)	49.3%

Table 2: Pattern preserving: compression ratio

Observe the following. (1) The *smaller* the compression ratio is, the *more effective* the compressing scheme used is. (2) We treat the compression ratio as a measurement for representation compression, which differs from the ratio measuring the memory cost reduction (to be discussed shortly).

The compression ratios of reachability preserving compression compress$_R$ are reported in Table 1. We find the following. (1) Real-life graphs can be highly compressed for reachability queries. Indeed, RC$_r$ is in average 5% over these datasets. In other words, it reduces real-life graphs by 95%. (2) Algorithm compress$_R$ performs significantly better than AHO. It also reduces SCC graphs by 81% in average. (3) The compression algorithms perform best on social networks *e.g.,*wikiVote, socEpinions, facebook and Youtube. The average RC$_r$ is 2%, 8% and 14.7% for (six) social networks, (three) Web graphs and the citation network, respectively. This is because social networks have higher connectivity.

The effectiveness of compress$_B$ is reported in Table 2. We find that (1) graphs can also be effectively compressed by pattern preserving compression, with PC$_r$ of 43% in average, *i.e.,* it reduces graphs by 57%; (2) Internet can be better compressed for graph pattern queries than social networks (Youtube) and citation networks (Citation), since the latter two have more diverse topological structures than the former, as observed in [22]; and (3) compress$_R$ performs better than compress$_B$ over all the datasets. This is because it is more difficult to merge nodes due to the requirements on topological structures and label equivalence imposed by pattern queries, compared to reachability queries.

Exp-2: Effectiveness: query processing. In this set of experiments, we evaluated the performance of the algorithms for reachability and pattern queries on original and compressed graphs, respectively. We used exactly *the same* algorithms in both settings, *without decompressing* graphs.

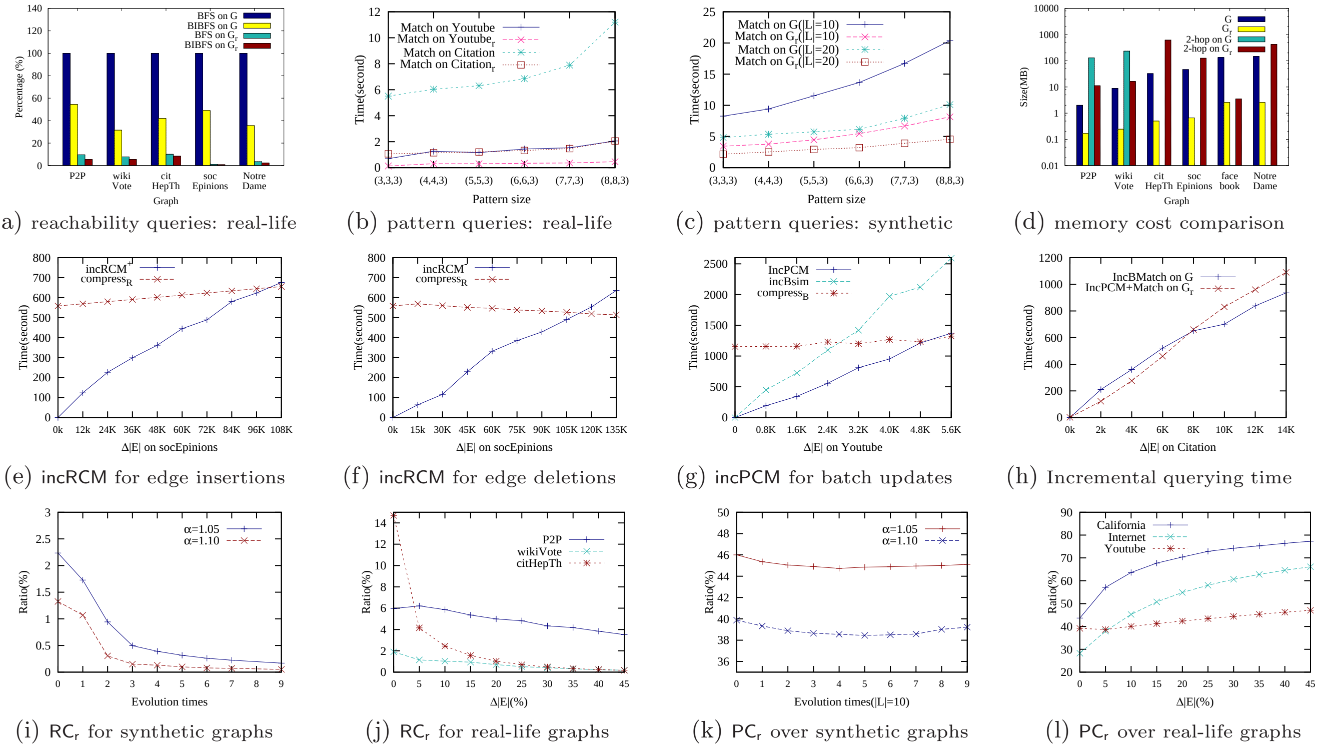
For a pair of randomly selected nodes, we queried their reachability and evaluated the running time of BFS and BIBFS on the original graph G and its compressed graph G_r. As shown in Fig. 12(a), the evaluation time on the compressed G_r is much less than that on G, when either BFS or BIBFS is used. Indeed, for socEpinions the running time of BFS on G_r is only 2% of the cost on G in average.

For graph pattern queries, Figure 12(b) shows the running time of Match on Youtube and Citation, and on their compressed counterparts (L_p is the same as L; see Table 2). In addition, we conducted the same experiments on synthetic graphs with $|V| = 50K$, $|E| = 435K$ while $|L|$=10 or $|L|$=20, and on compressed graphs. Fixing $L_p = 10$, we varied (V_p, E_p, k) of these queries from $(3, 3, 3)$ to $(8, 8, 3)$, as reported in Fig. 12(c). These results tell us the following: (a) the running time of Match on compressed graphs is only 30% of that on their original graphs; and (b) when $|L|$ is changed from 10 to 20 on synthetic data, Match runs faster as the compressed graphs contain more node labels.

As remarked earlier, the compression ratio of Table 1 only measures graph representation. In Fig. 12(d) we compare

[4] *http://www.caida.org/data/overview/*

[5] *http://snap.stanford.edu/data/index.html*



(a) reachability queries: real-life

(b) pattern queries: real-life

(c) pattern queries: synthetic

(d) memory cost comparison

(e) incRCM for edge insertions

(f) incRCM for edge deletions

(g) incPCM for batch updates

(h) Incremental querying time

(i) $\mathsf{RC_r}$ for synthetic graphs

(j) $\mathsf{RC_r}$ for real-life graphs

(k) $\mathsf{PC_r}$ over synthetic graphs

(l) $\mathsf{PC_r}$ over real-life graphs

Figure 12: Performance Evaluation

the memory cost of the original graph G, the compressed graph G_r by reachability preserving compression, and their 2-hop indexes [6], for real-life datasets. The result tells us the following: (a) at least 92% of the memory cost of G is reduced by G_r; (b) the 2-hop indexes have higher space cost than G and G_r; *e.g.,* 2-hop on wikiVote took 234MB memory, while its original graph took 8.9MB and the compressed graph took 0.2MB; and (c) 2-hop indexes can be built over small compressed graphs, but may not be feasible over large original graphs, *e.g.,* facebook, due to its high cost.

The results of the same experiments on other real-life graphs are consistent and hence, are not reported here.

Exp-3: Efficiency of incremental compression. We next evaluate the efficiency of incRCM and incPCM. Fixing the number of nodes in the social network socEpinions, we varied the number of edges from 509K to 617K (resp. from 509K to 374K) by inserting (resp. deleting) edges in 12K increments (resp. 15K decrements). The results in Figures 12(e) and 12(f) tell us that incRCM outperforms $\mathsf{compress_R}$ when insertions are up to 20% and deletions are up to 22% of the original graph.

Figure 12(g) shows the performance of incPCM on Youtube compared with $\mathsf{compress_B}$ and IncBsim in response to mixed updates, where we fixed the node size, and varied the size of the updates $|\varDelta E|$ in 0.8K increments. The result shows that incPCM is more efficient than $\mathsf{compress_B}$ when the total updates are up to 5K, and *consistently* outperforms IncBsim, due to the removal of redundant updates by incPCM.

Figure 12(h) compares the performance of the following two approaches, both for incrementally evaluating pattern queries over Citation: (1) we used IncBMatch to incrementally update the query result, and alternatively, (2) we first used incPCM to update the compressed graph, and then ran Match over the updated compressed graph to get the result. The total running times, reported in Fig. 12(h), tell us that once the updates are more than 8K, it is more efficient to update and query the compressed graphs than to incrementally update the query results.

We also conducted the same experiments on other real-life datasets. The results are consistent and hence not reported.

Exp-4: Effectiveness of incremental compression. We evaluated the effectiveness of incRCM and incPCM, in terms of compression ratios $\mathsf{RC_r}$ and $\mathsf{PC_r}$, respectively. (1) Fixing $|L| = 10$ and starting with $|V_0| = 1M$, we varied the size of synthetic graphs G by simulating the densification law [17]: for a synthetic graph G_i with $|V_i|$ nodes and $|E_i| = |V_i|^\alpha$ edges at iteration i, we increased its nodes to $|V_{i+1}| = \beta|V_i|$, and edges to $|E_{i+1}| = |V_{i+1}|^\alpha$ in the next iteration. (2) We varied the size of real-life graphs following power-law [20], where the edge growth rate was fixed to be 5%, and an edge was attached to the high degree nodes with 80% probability.

Figure 12(i) shows that for reachability queries, $\mathsf{RC_r}$ varies from 2.2% to 0.2% with α =1.05, and decreases from 1.4% to 0.05% with $\alpha = 1.1$, when β is fixed to be 1.2. This shows that the more edges are inserted into dense graph, the better the graph can be compressed for reachability queries. Indeed, when edges are increased, more nodes may become reachability equivalent, as expected (Section 3). The results over real-life graphs in Fig. 12(j) also verify this observation.

The results in Fig. 12(k) tell us that for graph pattern queries, $\mathsf{PC_r}$ is not sensitive to the changes of the size of graphs. On the other hand, Figure 12(l) shows the following. (1) When more edges are inserted into the real-life graphs, $\mathsf{PC_r}$ increases; this is because when new edges are added, the bisimilar nodes may have diverse topological structures and hence are no longer bisimilar; and (2) $\mathsf{PC_r}$ is more sensitive to the changes of the size of Web graphs (*e.g.,* California, Internet) than social networks (*e.g.,* Youtube), because the high connectivity of social networks makes most of the insertions redundant, *i.e.,* having less impact on $\mathsf{PC_r}$.

Summary. From the experimental results we find the following. (1) Real-life graphs can be effectively and efficiently compressed by reachability and graph pattern preserving compressions. (2) Evaluating queries on compressed graphs is far more efficient than on the original graphs, and is less sensitive to the query sizes. Moreover, existing index techniques can be directly applied to compressed graphs, *e.g.,* 2-hop index. (3) Compressed graphs by query preserving compressions can be efficiently maintained in response to batch updates. Better still, it is more efficient to evaluate queries on incrementally updated compressed graphs than incrementally evaluate queries on updated original graphs.

7. CONCLUSION

We have proposed query preserving graph compression for querying large real-life graphs. For queries of users' choice, the compressed graphs can be directly queried without decompression, using any available evaluation algorithms for the queries. As examples, we have developed efficient compression schemes for reachability queries and graph pattern queries. We have also provided incremental techniques for maintaining the compressed graphs, from boundedness results to algorithms. Our experimental results have verified that our methods are able to achieve high compression ratios, and reduce both storage space and query processing time; moreover, our compressed graphs can be efficiently maintained in response to updates to the original graphs.

We are currently experimenting with real-life graphs in various domains. We are also studying compression methods for other queries, *e.g.,* pattern queries with embedded regular expressions. We are also to extend our compression and maintenance techniques to query distributed graphs.

Acknowledgments. Fan and Wu are supported in part by EPSRC EP/J015377/1, the RSE-NSFC Joint Project Scheme and an IBM scalable data analytics for a smarter planet innovation award. Fan and Li are supported in part by the 973 Program 2012CB316200 and NSFC 61133002 of China.

8. REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SICOMP*, 1(2), 1972.

[2] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, 2011.

[3] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.

[4] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, 2003.

[5] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, 2009.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SICOMP*, 32(5), 2003.

[7] J. Deng, B. Choi, J. Xu, and S. S. Bhowmick. Optimizing incremental maintenance of minimal bisimulation of cyclic graphs. In *DASFAA*, 2011.

[8] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *CAV*, 2001.

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

[10] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *JCSS*, 51(2):261–272, 1995.

[11] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *CIKM*, 2005.

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

[13] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.

[14] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, 2008.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, 2002.

[16] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, 2006.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2, 2007.

[18] H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, 2010.

[19] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[20] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Internet Measurement Comference*, 2007.

[21] D. Moyles and G. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *JACM*, 16(3), 1969.

[22] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[23] A. Ntoulas, J. Cho, and C. Olston. What's new on the Web? The evolution of the Web from a search engine perspective. In *WWW*, 2004.

[24] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SICOMP*, 16(6), 1987.

[25] S. Perugini, M. A. Gonçalves, and E. A. Fox. Recommender systems research: A connection-centric survey. *J. Intell. Inf. Syst.*, 23(2):107–143, 2004.

[26] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[27] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *ICDE*, 2003.

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

[29] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The link database: Fast access to graphs of the web. In *DCC*, 2002.

[30] D. Saha. An incremental bisimulation algorithm. In *FSTTCS*, 2007.

[31] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. Arnetminer: extraction and mining of academic social networks. In *KDD*, 2008.

[32] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, 2011.

[33] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *SIGCOMM Workshop on Social Networks (WOSN)*, 2009.

[34] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1), 2010.

[35] J. X. Yu and J. Cheng. *Graph Reachability Queries: A Survey*. 2010.