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 $S \in \mathsf{Par}$, $\mathsf{compress_B}$ 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

Input: A graph G, its compressed graph G_r, batch updates ΔG.
Output: New compressed graph $G_r \oplus \Delta G_r$.

1. reduce ΔG;
2. update the topological rank r of the nodes in G_r *w.r.t.* ΔG;
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** $\mathsf{incRCM}^+(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
7.   **else if** e is an edge deletion
8.   **then** $\mathsf{incRCM}^-(e, [u]_{R_e}, [u']_{R_e}, G_r)$;
9.   reduce ΔG;
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

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). 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

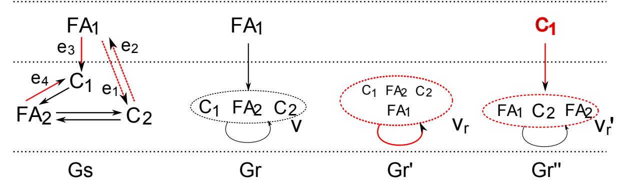


Figure 9: Incremental compression: reachability

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.

For the complexity, one can show that the first two steps of the algorithm (lines 1-3) are in $O(|\mathsf{AFF}||G_r|)$ time. Indeed, (1) it takes $O(|\mathsf{AFF}||G_r|)$ 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 $O(|\mathsf{AFF}||G_r|)$ 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 $O(|\mathsf{AFF}||G_r|)$ time. Thus incRCM is in $O(|\mathsf{AFF}||G_r|)$ time. As will be verified by our experimental study, $|G_r|$ and $|\mathsf{AFF}|$ 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 $O(|\mathsf{AFF}|^2 + |G_r|)$ 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 $V \setminus \mathsf{WF}$. (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 $(v_{\mathsf{scc}}, v'_{\mathsf{scc}})$ and $(v_{\mathsf{scc}}, v''_{\mathsf{scc}})$ are in E_{scc}, for all $v' \in \mathsf{WF}$ and all $v'' \in \mathsf{NWF}$. 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

Input: A graph G, a compressed graph G_r, batch updates ΔG;
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 $\{\mathsf{AFF}_i\}$, 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($\mathsf{AFF}_i, G_r, \Delta G$); update AFF;
8.   **for each** $[u']_{R_b} \in \mathsf{AFF}_i$ 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 := $\mathsf{AFF}_p \cup \{[u]_{R_b}\} \cup P([u']_{R_b})$;
3. **for each** node $[v_p]_{R_b} \in \mathsf{AFF}_p$ 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. $\mathsf{AFF}_{r_b([v_p]_{R_b})}$:= $\mathsf{AFF}_{r_b([v_p]_{R_b})} \cup \{[v_p]_{R_b}, [v_{p2}]_{R_b}\}$;
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

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 affected 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 $\mathsf{AFF}_{r_b(v_p)}$ (line 6). SplitMerge then merges

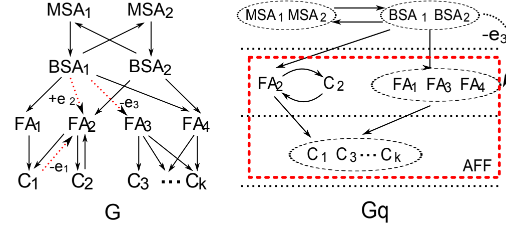


Figure 11: Incremental compression: graph pattern

$[v_{p1}]_{R_b}$ 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.* □

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 $O(|\mathsf{AFF}| \log |\mathsf{AFF}|)$ time. Moreover, procedures minDelta, PT and SplitMerge take $O(|\mathsf{AFF}|)$ time, $O(|\mathsf{AFF}| \log |\mathsf{AFF}| + |G_r|)$, and $O(|\mathsf{AFF}|^2)$ time in total. Hence incPCM is in $O(|\mathsf{AFF}|^2 + |G_r|)$ 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:

dataset	$\|G\|(\|V\|, \|E\|)$	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

(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 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 $\mathsf{compress_R}$ (Section 3) and $\mathsf{compress_B}$ (Section 4); (2) AHO [1] which, as a comparison to $\mathsf{compress_R}$, 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 $\mathsf{compress_R}$ to

[2] *http://netsg.cs.sfu.ca/youtubedata/*
[3] *http://www.cs.cornell.edu/courses/cs685/2002fa/*
[4] *http://www.caida.org/data/overview/*
[5] *http://snap.stanford.edu/data/index.html*

dataset	$\|G\|(\|V\|, \|E\|, \|L\|)$	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

be $RC_r = |G_r|/|G|$, where G is the original graph and G_r is its compressed graph by $\mathsf{compress_R}$. Similarly, we define PC_r of $\mathsf{compress_B}$, 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 $\mathsf{compress_R}$ on SCC graphs.

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).

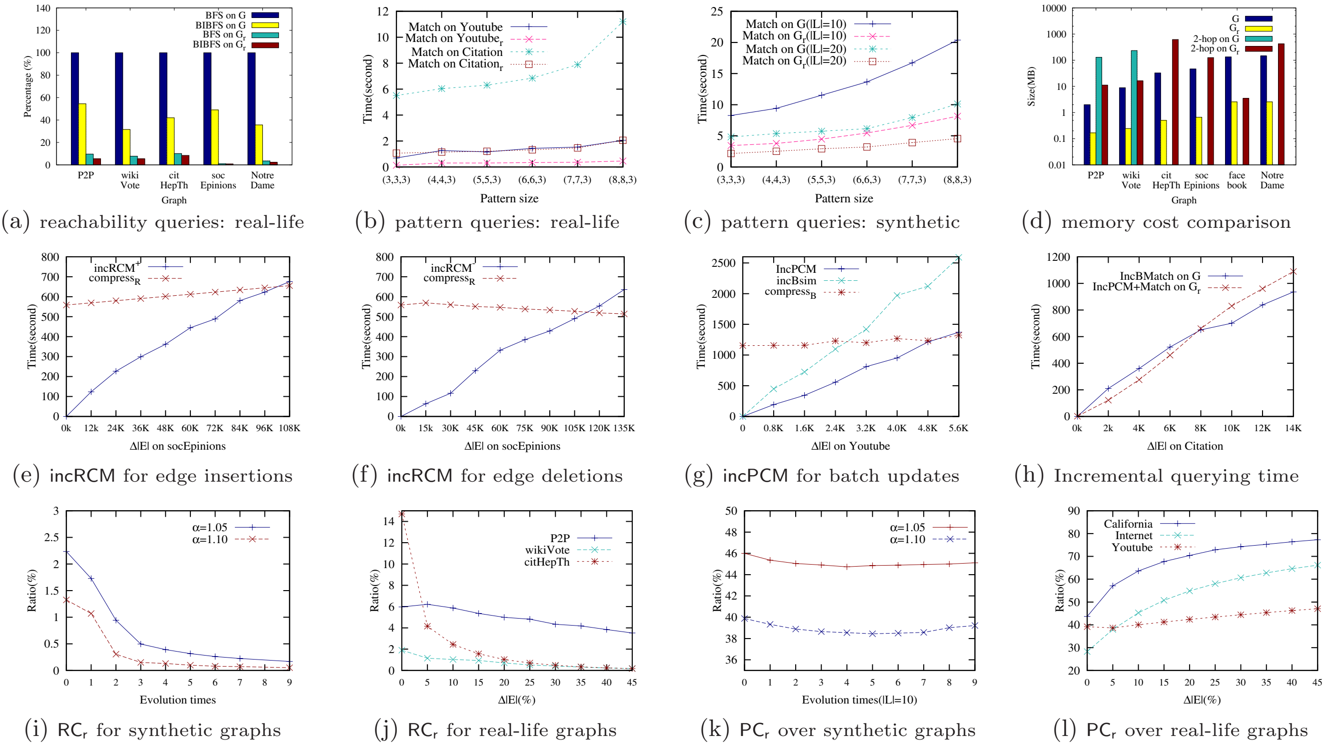
The compression ratios of reachability preserving compression $\mathsf{compress_R}$ are reported in Table 1. We find the following. (1) Real-life graphs can be highly compressed for reachability queries. Indeed, RC_r is in average 5% over these datasets. In other words, it reduces real-life graphs by 95%. (2) Algorithm $\mathsf{compress_R}$ 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 $\mathsf{compress_B}$ 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) $\mathsf{compress_R}$ performs better than $\mathsf{compress_B}$ 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



(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

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 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 $|\Delta E|$ 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.