# Incremental Maintenance of the Minimum Bisimulation of Cyclic Graphs

Jintian Deng, Byron Choi, Jianliang Xu, Haibo Hu and Sourav S. Bhowmick
*Members, IEEE*

**Abstract**—There have been numerous recent applications of graph databases (*e.g.*, the Semantic Web, ontology representation, social networks, XML, chemical databases and biological databases). A fundamental structural index for data graphs, namely *minimum bisimulation*, has been reported useful for efficient path query processing and optimization including selectivity estimation, among many others. Data graphs are subject to change and their indexes are updated accordingly. This paper studies the incremental maintenance problem of the minimum bisimulation of a possibly *cyclic data graph*. While cyclic graphs are ubiquitous among the data on the Web, previous work on the maintenance problem has mostly focused on acyclic graphs. To study the problem with cyclic graphs, we first show that the two existing classes of minimization algorithms – merging algorithm and partition refinement – have their strengths and weaknesses. Second, we propose a novel hybrid algorithm and its analytical model. This algorithm supports an edge insertion or deletion and two forms of batch insertions or deletions. To the best of our knowledge, this is the first maintenance algorithm that *guarantees* minimum bisimulation of cyclic graphs. Third, we propose to partially reuse the minimum bisimulation before an update in order to optimize maintenance performance. We present an experimental study on both synthetic and real data graphs that verified the efficiency and effectiveness of our algorithms.

**Index Terms**—Cyclic graphs, minimum bisimulation, incremental maintenance, graph indexing, evolving graphs and graph algorithms.

## 1 Introduction

Graph databases have a wide range of recent applications, *e.g.*, the Semantic Web, ontology representation, network topologies, XML, chemical databases and biological databases. To optimize query processing on large graphs, indexes have been proposed to summarize the paths of data graphs. In particular, many indexing schemes, *e.g.*, [1]–[9], have been derived from a notion of *bisimulation* equivalence of nodes [10]. In addition to indexing, bisimulation has recently been used to support selectivity estimation of structural queries [11]–[13].

To illustrate the applications of bisimulation, we present a simplified sketch of a popular benchmark data graph, namely XMark, in Fig. 1(a). XMark is a synthetic electronic auction dataset: `open_auction` contains an author, a seller and a list of bidders, whose information is stored in `persons`; `person` in turn watches a few `open_auctions`. To model the bidding and watching relationships, `open_auctions` reference `persons` and vice versa. From the perspective of indexing of path queries, two nodes of a data graph are bisimilar only if they have the same set of incoming paths. A sketch of the bisimulation graph of XMark is shown in Fig. 1(b). (More formal examples are given after the relevant definitions are presented.) In the sketched bisimulation graph, bisimilar nodes are placed in a partition, denoted as $I_i$. The bisimulation graph is often smaller than the original data graph, which makes it an efficient index. Consider a query

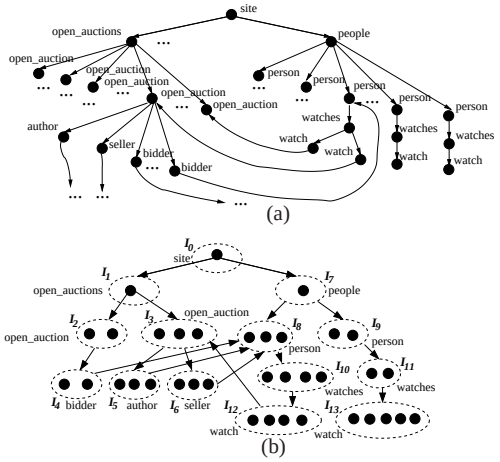



Fig. 1. (a) A simplified XMark data graph; and (b) A sketch of a bisimulation graph of XMark

$q$ `/site//open_auction/seller` that selects all `sellers` of `open_auctions`. We can evaluate $q$ on the bisimulation graph (Fig. 1(b)), as opposed to the data graph, and retrieve the data nodes in $I_6$. For query efficiency, it makes sense to employ the *minimum* bisimulation, *i.e.*, the smallest index.

A pressing issue of graph databases is that data graphs are very often subject to small and/or rapid updates [14], [15]. For example, in the context of social networks, users/pages and friendships are modeled as nodes and edges, respectively. In 2010, Facebook [16] has roughly an increase of eight new users per second, on average. Top growing pages have, on average, one new fan every two seconds [17]. In PubChem [18], a public database of chemical structures, the increase in chemically unique compounds during 2006 and 2011 was, on average, 12k compounds per day. In a popular

- *B. Choi, J. Deng, J. Xu and H. Hu are with the Department of Computer Science, Hong Kong Baptist University, Hong Kong.*
  *S. S. Bhowmick is with the School of Computer Engineering, Nanyang Technological University, Singapore.*

bibliograph XML dataset (DBLP [19]), the increase in nodes has been, on average, 6.7k per day, during the last 9 years. These real-world examples indicate large graphs may be subject to frequent small updates.

Efficient incremental maintenance algorithms for propagating frequent small updates are desirable for three reasons. (1) Incremental algorithms upon small updates have known to be more efficient than full recomputation in many applications. Classical survey on incremental algorithms can be found in [20]. (2) The quality (*a.k.a* efficacy) of bisimulation degrades over updates if it is not minimized [21]. When bisimulation is used as an index, the query performances degrade accordingly. (3) During reconstruction, the bisimulation is taken offline, which can be unacceptable to applications that require high availability, such as social networks.

Recently, the maintenance problem of bisimulation has received a renewed interest from database research [21]–[24]. For example, recent works by Fan et al. [22], [23] consider efficient bisimulation maintenance in the presence of queries. Few previous works [21], [24] have addressed the maintenance problem of the minimum bisimulation of *acyclic* graphs. While one may argue to simply apply previous techniques to cyclic graphs, they may return only the minimal bisimulations for cyclic graphs [21]. In practice, data graphs are often cyclic, *e.g.,* social networks and citation networks [25]. As shown in our experiments, the minimum bisimulation of a real citation network (DBLP) [19] is 9% smaller than the minimal one. In this paper, we present the first study on *bisimulation maintenance algorithms of cyclic graphs that guarantee minimum bisimulation, regardless of queries*.

The existing work on maintaining bisimulation can be divided into two classes: *merging algorithm* and *partition refinement*. However, there are some weaknesses in applying such work to cyclic graphs. First, merging algorithms fail to determine the minimum bisimulation of cyclic graphs. Each merging step often processes nodes by pairs, which is not sufficient to determine the bisimulation within and between *strongly connected components* (SCCs). Contrarily, we can easily establish that a node of an SCC can be bisimilar to a node of another SCC only if the two SCCs are bisimilar (see Section 4). Intuitively, the bisimulation of the nodes of SCCs are determined *together*.

Second, while it is known that partition refinement produces the minimum bisimulation of cyclic graphs, it is more suitable for full construction. Its application to the maintenance problem is rather limited. The reason for this is that it rebuilds the entire updated minimum bisimulation from scratch. Although in the worst case, an edge update may affect the minimum bisimulation arbitrarily, in practice, an update may often affect the bisimulation locally. Therefore, partition refinement can be relatively inefficient in maintaining bisimulation.

**Contributions.** In this paper, we propose *a novel hybrid algorithm of the merging algorithm and partition refinement* to maintain the minimum bisimulation of cyclic graphs. The hybrid algorithm supports the *insertions/deletions of edges and subgraphs*. The hybrid algorithm takes advantage of the two algorithms and overcomes some of the weaknesses of the individual algorithm. The main idea of the hybrid algorithm is that given an update, we use partition refinement to handle SCCs and the merging algorithm to handle the remaining nodes and optimize the nodes on the border of SCCs.

*(1)* First, we show that regarding cyclic graphs, merging algorithms require a space of $\Omega$ of the graphs' SCC in computing the minimum bisimulation. We then present an efficient hybrid algorithm that guarantees minimum bisimulations (Section 4); such a guarantee is absent in previous algorithms. The algorithm is presented with edge insertion (Section 5.1) and its extension on edge deletion (Section 5.2). The support of subgraph and batch updates are also presented (Section 5.3).

*(2)* A unique issue of the hybrid algorithm is that the partitions returned by partition refinement and merging algorithm (respectively) may be merged in order to yield the minimum bisimulation, which requires additional computation. Therefore, our second contribution is to provide a generalization of the cyclic graph representation. This generalization gives us a way to specify the subgraphs to be handled by the two algorithms. To determine the optimal performance from the two algorithms, we propose an analytical model for determining the optimal hybrid algorithm (Section 4.2).

*(3)* Furthermore, we propose three optimizations on the hybrid algorithm by utilizing the minimum bisimulation prior to an update. Through our initial experiments, we observed that the topologies of the minimum bisimulation before and after some random edge insertions/deletions are sometimes very similar, if not identical. This motivated us to reuse some existing minimum bisimulation to optimize the maintenance of the bisimulations of SCCs. Specifically, we propose (i) partial partition refinement, (ii) a new splitter for partition refinement and (iii) a reduced computation between nodes outside and inside SCCs (Section 6). The first two optimizations are applicable to previous partition refinement algorithms while the last optimization is specific to the hybrid algorithm.

*(4)* We conducted a comprehensive experimental study with both synthetic and real datasets, presented in Section 7. It verified that our hybrid algorithm is efficient; the analytical model is accurate, and the optimizations are effective. Since our algorithm maintains the minimum bisimulation, our algorithm always returns smaller (if not the same) bisimulations compared to previous work.

The rest of the paper is organized as follows. In Section 2, we discuss related work. The preliminaries and notations are then presented in Section 3. Section 4 details the hybrid algorithm. We present the overall incremental maintenance algorithm and the support of subgraph/batch insertions or deletions in Section 5. Three optimization techniques are proposed in Section 6, and an evaluation of the experiment is reported in Section 7. Section 8 concludes the paper.

## 2 Related Work

Existing bisimulation minimization and maintenance algorithms can be roughly categorized into two classes, namely *merging algorithms* and *partition refinement algorithms*. Two previous merging algorithms [21], [24] have been proposed for

incremental maintenance of the *minimal* bisimulation of cyclic graphs. The algorithm proposed by Yi et al. [21] contains a split and a merge phase. Upon an update to the data graph, the bisimulation graph is split to a correct but non-minimal bisimulation of the updated graph. The bisimulation graph is then minimized in the merge phase. For acyclic graphs, this algorithm produces the minimum bisimulation of the updated graph. If the graph is cyclic, it only returns a minimal bisimulation. Since Yi et al. [21] merges pairs of nodes iteratively in the merge phase, it is not sufficient to determine the minimum bisimulation of SCCs. When their algorithm encounters SCCs, the current merging step simply terminates. Thus, to support cyclic graphs, the minimum bisimulation is occasionally recomputed from scratch. Saha's work [24] can be considered as a follow-up to Yi et al.'s work. Saha proposed a split-merge-split algorithm with a rank flag for SCCs, which was originally proposed by Dovier et al. [26]. Similar to the other previous work, Saha's algorithm also maintains a minimal bisimulation. However, there is neither an experiment report nor an implementation for comparisons.

The claim on the support of cyclic graphs from the work [21], [24] is based on a direct application of algorithms for acyclic graphs. Thus, minimum bisimulations are not guaranteed. A recent work by Deng et al. [27] on minimal bisimulation maintenance only focused on optimization issues.

The partition-refinement algorithm proposed by Kaushik et al. [28] may be seen as a variant of Paige and Tarjan's algorithm [29], *i.e.,* a *construction* algorithm. Kaushik *et al.* [28] proposed their own split to handle edge changes, and this has been extended to maintain local bisimulation [2]. Their experiment showed that they produced a bisimulation that is always within 5% of the minimum bisimulation. Later experiments showed that smaller bisimulations can be produced by Yi et al. [21], which we compare through experiments.

Recently, Fan et al. [22], [23] proposed incremental graph pattern matching and query preserving bisimulation-based compression. The compressed graph is lossless only relative to some class of queries, in particular, reachability queries and graph patterns. In our work, we do not take queries as input. Hellings et al. [9] focuses on external algorithm for bisimulation of DAG. Cyclic graphs are not considered. The maintenance problem has been left as a future work.

Bisimulations [10] have recently been revisited due to their extensive applications on databases. There has been a host of work on indexing for path queries, which in a nutshell, summarizes a data graph using bisimulation, *e.g.,* [1]–[3]. The seminal work is 1-index [1], which adopts bisimulation as an index for regular paths. In practice, 1-index [1] can often be large. A notion of local bisimulation, namely $k$-bisimulation [2], has been proposed to reduce index size, although some path information may be lost. Local bisimulations are combined/joined to "recover" such information or certain statistical properties (*e.g.,* uniform distribution) are assumed. To balance query performance and index size, a follow-up work [3] proposed to adaptively adjust the $k$ in $k$-bisimulation. Our techniques can be extended to support local bisimulation with straightforward but tedious modifications. Due to space constraints, we opt to focus on bisimulation alone. Some previous studies [7], [8] considered bisimulation as a compression of XML repositories for efficient query processing. Bisimulation has also been adopted for path query selectivity estimation (*e.g.,* [11], [12]). A study on bisimulation maintenance benefits *all of the above-mentioned applications.*

## 3 Preliminaries and Notations

This section presents the preliminaries, notations and definitions required by this work.

### 3.1 Data Graphs

**Definition 3.1:** A *data graph* is a rooted directed labeled graph $G(V, E, r, \rho, \Sigma)$, where $V$ is a set of nodes and $E$: $V \times V$ is a set of edges, $r \in V$ is a root node and $\rho : V \rightarrow \Sigma$ maps a vertex to a label, and $\Sigma$ is a finite set of labels. □

In the case of unrooted graphs, we may simply create an artificial root called "root" to connect to every node of the graph. For succinctness, we may simply denote a data graph as $G(V, E)$ when $r$, $\rho$ or $\Sigma$ are irrelevant to our discussions. To facilitate the discussions on algorithms, we assume some auxiliary functions of nodes. Given a data node $v$, $v$.parent() and $v$.sibling() return the parents and the siblings of $v$, respectively. For simplicity, we often use $|G|$ to denote the number of vertices $|V|$ of the graph, which is also known as the order of the graph, unless otherwise specified.

**Cyclic graphs.** Since our work focuses on cyclic graphs, we review some relevant definitions. A *strongly connected component* (SCC) in a graph $G(V, E)$ is a subgraph $G'(V', E')$, $V' \subseteq V$, $E' \subseteq E$ and $(v_1, v_2) \in E' \Rightarrow v_1 \in V' \wedge v_2 \in V'$, where the nodes in $V'$ can reach each other in $G'$. The SCCs of a graph can be determined by classical graph contraction algorithms (*e.g.,* Gabow's algorithm [30]) with a runtime $O(|V|+|E|)$.

Graph contraction has been a classical preprocessing technique for cyclic data graphs, which we exploit to optimize our proposed hybrid algorithm. Intuitively, given a cyclic graph, graph contraction reduces an SCC into a *supernode* iteratively until a directed acyclic graph (DAG) is obtained. The graph returned by graph contraction is often called the *reduced graph*. The nodes that are not supernodes are sometimes called *simple* nodes. We present some relevant definitions below.

**Definition 3.2:** A node $n$ of an SCC $G'(V',E')$ of a graph $G(V,E)$ is an *exit* node if there exists an edge $(n,n_1)$ where $n \in V'$, $n_1 \in V$ and $n_1 \notin V'$. $n$ is an *entry* node if there exists an edge $(n_0,n)$ where $n_0 \in V$, $n_0 \notin V'$ and $n \in V'$. □

### 3.2 Bisimulation

Next, we present the relevant definitions of bisimulation.

**Definition 3.3:** Given two subgraphs $G_1(V_1, E_1, r_1, \rho_1)$ and $G_2(V_2, E_2, r_2, \rho_2)$, an *upward bisimulation* $\sim$ is a relation of $V_1$ and $V_2$:

$$\forall\ v_1 \in V_1, v_2 \in V_2, v_1 \sim v_2 \rightarrow \\ \forall\ (v_1', v_1) \in E_1, \exists\ (v_2', v_2) \in E_2, v_1' \sim v_2' \wedge \\ \forall\ (v_2'', v_2) \in E_2, \exists\ (v_1'', v_1) \in E_1, v_1'' \sim v_2'' \wedge \rho_1(v_1) = \rho_2(v_2).$$

Two subgraphs $G_1$ and $G_2$ are *upward bisimilar* if an upward bisimulation $\sim$ can be established between $G_1$ and $G_2$. □

We make two remarks before proceeding further. First, with some classic optimization, the minimum bisimulation can be computed in $O(|E|log(|V|))$ time [29]. Second, Definition 3.3 presents *upward* bisimulation in the sense that two nodes can be bisimilar if they have the same label and all of their *parents* are bisimilar. In this paper, we present our techniques with upward bisimulation only and skip other notions of bisimulation (*e.g.,* downward bisimulation used in [7]) which can be supported by some minor extensions of this work. Therefore, we may use the terms bisimulation and upward bisimulation *interchangeably*. Definition 3.3 can be paraphrased in terms of paths (Proposition 3.1), which often simplifies our discussions.

**Proposition 3.1:** *Two nodes are upward bisimilar if the sets of the incoming labeled paths of the nodes are the same.* □

A set of bisimilar nodes is often referred to as an *equivalence partition* of nodes or simply as *partition*. Hence, a bisimulation graph can be described as a set of partitions. Since the bisimulation is often used as an index graph, the partitions are sometimes referred to as *index nodes*, or simply *Inodes*, whereas the nodes of the data graph are referred to as *data nodes*, *Dnodes* or simply *nodes*.

To distinguish between the notions of minimal and minimum bisimulations, we present the following definitions.

**Definition 3.4:** Given two partitions of Dnodes $X$ and $I$, $X$ is *stable w.r.t.* $I$ if either (i) $X$ is contained in the children of the Dnodes in the partition $I$ or (ii) $X$ and the children of the Dnodes in $I$ are disjoint. Otherwise, $X$ is unstable. □

**Example 3.1:** We illustrate the definition of stability with the example shown in Fig. 2. The data graph and a partitioning of its nodes are shown in Figs. 2(a) and 2(d), respectively, where the dotted ovals denote partitions. (i) Let $I_1$ be the partition with the $b$ nodes and $X_1$ and $X_2$ are the partitions of the $c$ nodes. $X_1$ is not contained in the children of the Dnodes in $I$, and thus, $X_1$ is unstable. Similarly, $X_2$ is unstable; (ii) Let $I_2$ be the parition with the $a$ nodes. $I_1$ is contained in the children of the Dnodes of $I_2$. Thus, $I_1$ is stable *w.r.t.* $I_2$. □

**Definition 3.5:** Given a bisimulation $B$ of a graph $G$, $B$ is *minimal* if for any two equivalence partitions $I, J \in B$, either (i) the Dnodes in $I$ and $J$ have different labels, or (ii) merging $I$ and $J$ results in some partition $K$ where $K$ is unstable. □

It is worth noting that the notion of minimality is defined with respect to the algorithm that computes the minimal bisimulation, as stated in Condition (ii) of Definition 3.5. In comparison, minimum bisimulations do not assume any specific minimization algorithms. It is known that the minimum bisimulation of a graph is unique [21].

**Definition 3.6:** A bisimulation $B$ of a graph $G$ is the *minimum* bisimulation if $B$ contains the minimum number of partitions, among all possible minimal bisimulations of $G$. □

**Example 3.2:** With respect to the data graph in Fig. 2(a), Figs. 2(c) and 2(f) show its two possible minimal bisimulations. Merging of any pairs of partitions with the same label results in unstable partitions. For example, merging the partitions of $b$ nodes of Fig. 2(c) results in unstable partitions, shown in Fig. 2(d). Fig. 2(f) is the minimum bisimulation of Fig. 2(a). □

### 3.3 Bisimulation Minimization Algorithms

Next, we shall provide a brief review of the main ideas of the two classes of bisimulation minimization algorithms using the example shown in Fig. 2, which illustrates the strengths and weaknesses of these algorithms in the context of updates.

Fig. 2(a) shows a cyclic data graph. The $b$ and $c$ nodes form an SCC. A merging algorithm (*e.g.,* [21]) initially places each node in a single partition, *i.e.,* an Inode (Fig. 2(b)). Assume that a merging algorithm merges pairs of partitions top-down. It merges the two $a$ nodes under $r$ (Fig. 2(c)). The merging algorithm does not merge the $b$ nodes, because their $c$ parents have not been determined bisimilar. Fig. 2(c) is returned, which is a minimal bisimulation. As discussed in Example 3.2, merging more partitions would result in unstable partitions (Fig. 2(d)). Intuitively, given any fixed merging algorithm with a limited space budget, at a particular merging step on two Inodes, *e.g.,* $I_{b_1}$ and $I_{b_2}$ in Fig. 2(c), it is possible that some bisimilar parents of $I_{b_1}$ and $I_{b_2}$ are not determined or stored. This makes it inefficient to determine the minimum bisimulation using merging algorithms.

In contrast, a partition refinement algorithm (*e.g.,* [29]) initially places all nodes in a single partition (Fig. 2(e)). It then refines (splits) existing partitions until the partitions are stable (Figs. 2(f)). Fig. 2(f) is the minimum bisimulation, which is 38% smaller than the minimal bisimulation in Fig. 2(c). One may further observe that partition refinement is, by its nature, non-incremental. It builds the minimum bisimulation from scratch, which is costly for a single update.

Finally, we note that the relative performance of partition refinement and merging algorithms depends on how bisimilar the graph is: Suppose most subgraphs are non-bisimilar. Partition refinement would take many splits to converge; on the other hand, suppose most subgraphs are bisimilar. Merging algorithms would require many merging iterations.

## 4 Bisimulation of Cyclic Graphs

To begin our investigation, we show that it is often inefficient to extend existing merging algorithms to maintain the minimum bisimulation.

Merging algorithms for bisimulation minimization are iterative in nature. The current merging step of Inodes in some SCCs may affect and depend on the merging of some Inodes in other SCCs. We encapsulate the iterative logic of a merging algorithm in a function $A$. In each iteration, $A(I_1, I_2)$ computes bisimulation between a pair of partitions $I_1$ and $I_2$. For clarity, we use $A(G)$ to denote the application of the algorithm on the graph.

To compute the minimum bisimulation, $A(G)$ requires much information about the to-be-merged Inodes to be maintained and this is often infeasible for maintaining real-world graphs. Hence, existing merging algorithms opt not to return the minimum bisimulation for cyclic graphs. Specifically, we formalize the memory requirement of $A$ in Theorem 4.1.

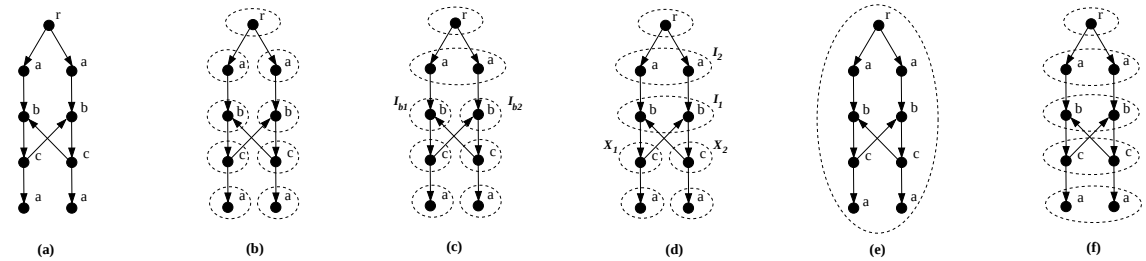



Fig. 2. Illustration of merging algorithm and partition-refinement algorithm

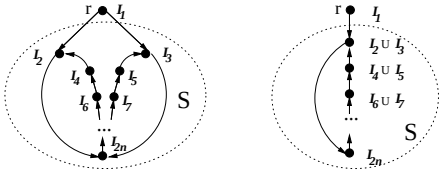



Fig. 3. The cyclic graph used in the proof of Theorem 4.1

**Theorem 4.1:** *Given an* SCC $S$ *and a merging algorithm* $A$ *for determining minimum bisimulation,* $A(S)$ *requires* $\Omega(|S|)$ *space.* □

*Proof:* By definition of $\Omega$, we need to show that there exists constants $c$ and $k$ such that for all $|S| > k$, $A(S)$ requires $c \times |S|$ space.

We establish the theorem with the example graph shown in Fig. 3(a). The graph contains a single special SCC $S$. Specifically, the graph has a root $r$; $r$ reaches the partitions $I_2$ and $I_3$ via the same kind of edge. In addition, $S$ contains two (overlapping) cycles: ($I_{2n}$, $I_{2n-2}$, ..., $I_6$, $I_4$, $I_2$, $I_{2n}$) and ($I_{2n}$, $I_{2n-1}$, ..., $I_7$, $I_5$, $I_3$, $I_{2n}$). For simplicity, we assume that each partition $I$ contains one node initially. Furthermore, we assume that the minimum bisimulation of Fig. 3(a) is the one shown in Fig. 3(b).

To compute the minimum bisimulation, in runtime, $A$ must determine if Inodes $I_2$ and $I_3$ are bisimilar, denoted as $A(I_2, I_3)$ (*i.e.,* whether $I_2$ and $I_3$ are contained in a single partition in the minimum bisimulaton). By Definition 3.3, the bisimulation relationships of the parents of $I_2$ and $I_3$ are needed to determine $A(I_2, I_3)$. Therefore, $A(I_4, I_5)$ is necessary for $A(I_2, I_3)$. Applying this argument recursively, we need $A(I_{2i}, I_{2i+1})$, for all $i \in [2, n\text{-}1]$, to determine $A(I_2, I_3)$. Importantly, since the partitions $I$'s are in an SCC, $A(I_2, I_3)$ is, in turn, *necessary* to compute $A(I_{2i}, I_{2i+1})$ *for all* $i$'s as well. The same argument can be applied to $A(I_{2i}, I_{2i+1})$ for all $i$'s.

Let $c$ = 1. That is, $A(S)$ can use $|S|$ space. $A(I_{2i}, I_{2i+1})$ for all $i$'s can be determined together and the minimum bisimulation of $S$ can be obtained. However, if $c < 1$, then at any merging step of any pair of partitions, $A$ does not have the necessary information to conclude whether they belong to the same partition in the minimum bisimulation. □

It may be worth-noting that using merging algorithms with a $O(|G|)$ space does not make any sense; as one can simply use partition refinement, which can reconstruct the bisimulation from scratch. The hybrid algorithm proposed in this section is specially designed to apply partition refinement on SCCs.

We also note that Theorem 4.1 cannot be extended to acyclic graphs. In fact, a previous work [21] has proposed a merging algorithm that efficiently returns the minimum bisimulation. The intuition that determining the minimum bisimulation of acyclic graphs is strictly simpler is because there is a topological order of nodes. Merging algorithms can exploit this ordering, which determines when the bisimulation between nodes is no longer needed for the computation of other nodes and can be safely removed from the algorithms' runtime stack/memory. Such ordering is absent in cyclic graphs.

In comparison, while partition refinement can determine the minimum bisimulation, it is not incremental. By definition, partition refinement starts with a single partition and applies refinement recursively, where existing bisimulation is not used.

In response to this, we propose a hybrid minimization algorithm for bisimulation in Section 4.1. An analytical model of this algorithm is detailed in Section 4.2.

### 4.1 Hybrid Minimization Algorithm

This section presents our hybrid algorithm, `bisimilar_cyclic`, which is a crucial ingredient of our incremental maintenance algorithm. The algorithm interleaves partition refinement for SCCs and the merging algorithm for acyclic subgraphs. The strength of partition refinement is that it can determine the minimum bisimulation even for cyclic graphs. Therefore, we apply partition refinement on SCCs. On the other hand, merging algorithms are more natural for incremental computation and are therefore applied to the subgraphs that are not part of any SCCs.

The pseudo-code is presented in Fig. 4, which serves as the basis for our analysis and optimization techniques. The input of `bisimilar_cyclic` is a reduced graph $G$ of a data graph $G_0$ and the output is the minimum bisimulation of $G_0$. Its application on updates is detailed in Section 5.

The pseudo-code can be described as follows. It traverses the reduced graph top-down (in topological order, as in Lines 01 and 08). We skip the pseudo-code for traversal as it is straightforward. The nodes that are ready for processing are denoted as $Q$. We process the supernodes $S$ (*i.e.,* SCCs) and then the simple nodes $Q' = Q - S$, where the simple nodes are simply the Inodes that are not involved in any SCCs. We use partition refinement `partition_refinement` to compute the minimum bisimulation between and among the Inodes *inside* supernodes $S$ (Lines 04 and 10-18). Specifically, we create an artificial root node (Line 10) to connect all the parents of SCCs in $S$ (Lines 11-14). Then, we simply apply existing partition refinement `Paige_Tarjan` (*e.g.,* [29]) to determine the minimum bisimulation. We remove the artificial root after the minimum bisimulation is obtained (Line 16).

The simple Inodes in $Q'$ are processed by a merging algorithm `merge_bisimilar` (Lines 07 and 19-25). Lines 20-21 check a simple node $q$ in $Q'$ and its siblings with the

```
Procedure bisimilar_cyclic
Input: a reduced data graph G
Output: the minimum bisimulation of G, B
01 Q = get_next_top_order(G)
02 while Q ≠ ∅
03     S = {s | s is a supernode and s ∈ Q}
04     partition_refinement(S)
05     Q' = Q − S
06     foreach q in Q'
07         merge_bisimilar(q, B)
08     Q = get_next_top_order(G)
09 return B

Procedure partition_refinement(S)
10 G(V, E) = ({r}, ∅)
11 foreach s in S
12     G = G ∪ s
13     foreach p in s.parent()
14         G.E = G.E ∪ (r, p)
15 Paige_Tarjan(G)
16 remove r
17 foreach s in S
18     s.processed = true

Procedure merge_bisimilar(q, B)
19 q.processed = true
20 foreach q' in q.sibling()
21     if ∀p ∈ q.parent() ∃p' ∈ q'.parent() s.t. p ∼ p' ∧
           ∀p' ∈ q'.parent() ∃p ∈ q.parent() s.t. p ∼ p'
22         if q' is inside a supernode s
23             update the parents of q ∉ s to
                   (i) the entries of s and (ii) the parents of q'
24             update the children of q ∉ s to
                   (i) the exits of s and (ii) the children of q'
25         merge(q, q')
```

Fig. 4. Bisimulation minimization of cyclic graphs

bisimulation definition (Definition 3.3). It should be noted that the sibling $q'$ of $q$ can be an Inode *inside* a supernode (*i.e.*, an SCC). Since we process the supernodes (Line 04) prior to the simple nodes (Lines 07) and the simple nodes in topological order (Lines 01 and 08), the bisimulation between the parents of $q$ and $q'$ has already been determined prior to determining bisimulation between $q$ and $q'$. Therefore, if $q' \sim q$, $q'$ can be located from the siblings of $q$ in the bisimulation graph computed so far (Line 20). A correctness argument is provided in the proof of Theorem 4.2. If $q$ and $q'$ are bisimilar, we merge $q$ and $q'$ (Line 25). We omit the pseudo-code of `merge` as it is straightforward. In addition, if $q'$ is in a supernode $s$, the parents and children of $q$ that are not in $s$ are set to be the entry and exit nodes of $s$, respectively (Lines 23-24).

**Example 4.1:** Figs. 5(a)-(d) show a run of `bisimilar_cyclic` on the synthetic graph shown in Fig. 5(a). We assume that the SCCs have been identified by Gabow's algorithm, where the supernodes are highlighted in Fig. 5(a). Also, we assume that each Dnode has been initially placed in an Inode. For clarity, we omit such simple Inodes from the figures. `bisimilar_cyclic` traverses the graph in Fig. 5(a) top-down. It does not encounter any supernode in the first two levels. Hence, only `merge_bisimilar` is invoked and produces Fig. 5(b). In the third iteration, supernodes 1 and 2 are encountered. We note that previous work [21] terminates and returns Fig. 5(b). In contrast, the hybrid algorithm invokes `partition_refinement` and produces the minimum bisimulation between supernodes 1 and 2. The result is shown in Fig. 5(c). The traversal proceeds and encounters a $c$ node connecting to supernode 1. Its only Inode parent is the same as the Inode of other $c$ nodes. Hence, `merge_bisimilar` identifies this case (Line 20) and merges the $c$ node *into the Inode of other $c$ nodes*. Similarly, `bisimilar_cyclic` identifies the $b$ node that connects to supernode 1 but is outside the supernode. Since the $b$ node does not have an $a$ Inode parent as the other $b$ nodes do, the $b$ node is not merged due to Line 21. The traversal proceeds and returns Fig. 5(d) as the final result. We include supernode 3 simply to illustrate that an arbitrary subgraph may be connected to supernodes and previous merging algorithms may return a bisimulation far from the minimum one. □

**Theorem 4.2:** `bisimilar_cyclic`*($G$) returns the minimum bisimulation of $G$.* □

*Proof:* The correctness of `bisimilar_cyclic` can be established by analyzing all possible bisimilar nodes in $G$.

*Case 1: The bisimilar nodes $n_1$ and $n_2$ are not in any SCC.* The reduced graph $G$ is a DAG and $n_1$ and $n_2$ are simple nodes in $G$. The merging part (Lines 01-02 and 06-09) of `bisimilar_cyclic` is a non-optimized version of the merge phase of [21]. By Theorem 1 in [21], `bisimilar_cyclic` returns the minimum bisimulation of acyclic graphs.

*Case 2: The bisimilar nodes $n_1$ and $n_2$ are in some SCCs.* $n_1$ and $n_2$ can be bisimilar only if the supernodes of $n_1$ and $n_2$ are either the same or bisimilar. The bisimulation between supernodes is determined correctly, by Case 1, as they are nodes of the directed acyclic graph $G$. When supernodes are encountered, the classical partition refinement algorithm in `partition_refinement` (Line 04) of `bisimilar_cyclic` produces the minimum bisimulation of the SCCs of and between the supernodes.

*Case 3: $n_1$ is in an SCC; $n_2$ is not in any SCC; and $n_1$ and $n_2$ are bisimilar.* This remaining case shows that `bisimilar_cyclic` can identify the bisimulation between (i) the Inodes that are not *inside* any supernode and (ii) some Inodes inside a supernode (SCC).[1]

We can prove this case by a simple induction on the depth of the reduced graph. Define the depth of a node $n$ of a reduced graph (*directed acyclic graph*) as the maximum distance between the root node and $n$; and the depth of a reduced graph as the maximum depth among all nodes. The *induction hypothesis* is stated as follows:

$\Phi$: "`bisimilar_cyclic` *returns the minimum bisimulation of a reduced graph $G_m$, where $G_m$ has a depth $m$.*"

The *base case*, where $m = 1$, is trivially true.

The *induction case*: Assume that the induction hypothesis is true up to $k$. We now consider a reduced graph with a depth $k+1$. By definition, the nodes with the depth $k$ and those with the depth $k+1$ are not bisimilar. `bisimilar_cyclic` processes all of the nodes with the depth $k$ prior to those nodes with $k+1$ due to a traversal in the topological order `get_next_top_order`($G$). By the induction hypothesis,

1. It is straightforward that (i) a supernode itself and (ii) any simple Inode, that is not in any SCC, are not bisimilar.

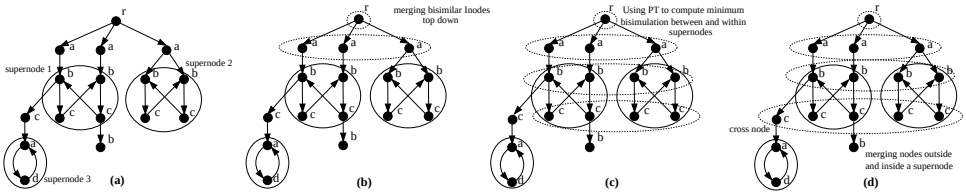



Fig. 5. Illustration of the hybrid algorithm `bisimilar_cyclic`

`bisimilar_cyclic` returns the minimum bisimulation of the subgraph $G_k$ of $G$, where $G_k$ is the subgraph of $G$ with a depth $k$. Suppose a node $n_{k+1}$ is bisimilar to another node $n'$ inside a supernode $s$, where the depth of $n_{k+1}$ is $k+1$. Since $n_{k+1}$ is bisimilar to $n'$, the depth of $s$ must be smaller or equal to $k+1$. In addition, by Definition 3.3, for each parent $p$ of $n_{k+1}$, there is a parent $p'$ of $n'$ such that $p \sim p'$. By the induction hypothesis, $p$ and $p'$ are declared bisimilar and placed in an *Inode* (Lines 22-25) in an earlier run of `bisimilar_cyclic`. That is, $n_{k+1}$ and $n'$ are siblings, after the minimum bisimulation of $G_k$ has been computed. Thus, $n_{k+1}$ and $n'$ are checked (Line 21) and merged (Line 25). □

### 4.2 The Analytical Model

An asymptotic analysis on `bisimilar_cyclic` is fairly simple. `bisimilar_cyclic` runs in Big-theta of the slower of the merging algorithm and partition refinement used. However, the actual performance bottlenecks of an implementation of the algorithm may be better analyzed using an analytical model. This section presents such a model of the hybrid algorithm.

We first distinguish the *cross nodes* and the *non-cross nodes* that are both processed by the merging part of the hybrid algorithm. More specifically, the cross nodes and non-cross nodes are processed by Lines 19-25, and Lines 19-22 and 25, respectively. An example of the only cross node in Fig. 5(a) – the $c$ connected to supernode 1 – is annotated in Fig. 5(d).

**Definition 4.1:** *Cross nodes* $\mathcal{C}$ are the *Inode*s that are not in any `SCC` but are bisimilar to an *Inode* inside an `SCC`. The distance $dist$ of a cross node $c \in \mathcal{C}$ ($c.dist$) is the length of the shortest path from an exit node of an `SCC` to $c$. □

Next, we denote the cost of `bisimilar_cyclic` as *Cost*. *Cost* consists of three costs: the unit costs for (i) partition refinement $P$ of supernodes; (ii) the merging algorithm $M$ of non-cross nodes; and (iii) the merging of cross nodes $C$. $S_p$, $S_m$ and $S_{cr}$ are the subgraphs applied to partition refinement, merging of non-cross nodes and merging of cross nodes, respectively.

$$Cost = |S_p| \times P + |S_m| \times M + |S_{cr}| \times C \qquad (1)$$

$P$, $M$ and $C$ can be measured as the minimum bisimulation is constructed, for example, by a run of `bisimilar_cyclic`. During the run, we can easily determine $|S_p|$, $|S_m|$ and $|S_{cr}|$. $P$ is estimated as the average runtime of Line 04. $M$ represents the average runtime of Lines 19-22 and 25 for non-cross nodes. Finally, $C$ denotes $M$ plus the average overhead due to Lines 23-24 for cross nodes.

Suppose the updates are relatively small. $P$, $M$ and $C$ can be considered as constants before and after updates. In addition, we have $|S_p| + |S_m| + |S_{cr}| \approx |G|$. Also, it is often the case that $P \ll C$, because the maintenance of intermediate bisimulations is far more costly than iterative refinements. In view of this, we present an optimization in Section 6 that relaxes the correctness of intermediate bisimulations.

**Example 4.2:** To illustrate the formula, we show some numbers obtained from a run of `bisimilar_cyclic` of our implementation on an `XMark` graph. $|S_p|$, $|S_m|$ and $|S_{cr}|$ are 137.6k, 1,529k and 62.8k nodes, respectively. $P$, $M$ and $C$ are 0.036ms/Dnode, 0.007ms/Dnode and 0.493ms/Dnode, respectively. The total cost is 46.6 seconds. At first glance, the unit cost of merging algorithm $M$ is very small. We note that $M$ is small since $|S_m|$ includes both bisimilar and non-bisimilar nodes outside the `SCC`s (checked by Line 21), where the time for processing the latter is very small. $C$ shows more precisely the performance of merging as all of the cross nodes are merged (Lines 23-24). In our implementation, $C$ is more than an order of magnitude (13.7 times) slower than $P$. We remark that $|S_p|$ is two times larger than $|S_{cr}|$. □

### 4.3 Generalization of Reduced Graphs

From Example 4.2 and Formula (1), we can observe that to optimize runtime, we may reduce the number of cross nodes. In addition, Theorem 4.2 states that it is sufficient to apply partition refinement on `SCC`s to obtain the minimum bisimulation. In response to the analysis, we present a generalization of reduced graphs that: (i) the generalized reduced graphs contain fewer cross nodes; (ii) partition refinement may not operate on excessively large subgraphs; and (iii) the hybrid algorithm (without any modification) can operate on.

The main idea of the generalization is to reduce `SCC`s together with the subgraphs connecting to them to supernodes. More specifically, given a supernode $s$, we denote $G_{s,k}$ to be the subgraph that contains (i) $s$ and (ii) the nodes connected to $s$ via a path with at most $k$ steps. Next, we extend the supernode of $s$ to represent $s$ as well as $G_{s,k}$. We use $G_k$ to represent $\bigcup_{s \in S} G_{s,k}$, where $S$ is the set of `SCC`s in the graph $G$. As a result, the cost of the hybrid algorithm can be rewritten as follows.

$$Cost_k = |G_k| \times P + (|S_m \setminus G_k|) \times M + (|S_c \setminus G_k|) \times C \qquad (2)$$

As $k$ increases, more subgraphs (*i.e.*, $G_k$) are applied to partition refinement. Meanwhile, smaller subgraphs (*i.e.*, $S_m \setminus G_k$ and $S_c \setminus G_k$) are handled by merging algorithms.

It is worth-noting that the largest possible $k$ is the diameter of the reduced graph. Our preliminary experiments show that if the generalization of the reduced graphs improves the overall performance, a small $k$ is sufficient. Since the sizes of $G_k$, for *all* $k$'s, may be recorded and determined in a run of `bisimilar_cyclic`, the $k$ for the optimal performance, denoted as $k_{min}$, can be easily computed.

Given the value of $k_{min}$, we recompute the supernodes with $k_{min}$, which also results in a reduced graph. Then, the reduced

graph and its minimum bisimulation will be maintained with the hybrid algorithm, which will be presented in Section 5.

**Example 4.3:** Following up Example 4.2, we determine the numbers needed for computing $Cost_k$, as presented in Formula (2), in a run of `bisimilar_cyclic`. We note that $G_1$ to $G_5$ are 47k, 88.5k, 102k, 105.7k and 106k, respectively. Then, we can compute that $k_{min}$ = 2, where $Cost_{k_2}$ = (137.6k + 88.5k) × 0.036ms + ( 1,529k - 88.5k) × 0.007ms + 266 × 0.493ms = 18.4s. As a reference, the actual runtime was 20s. □

Finally, we remark that the model may be used to make a coarse prediction on the largest number of updates (*i.e.*, the size of a batch of updates) to be handled by the hybrid algorithm, as opposed to a full recomputation using partition refinement. $Cost_k$ models the worst case scenario of an update: an update affects all SCCs in the data graph. To simplify our analysis, we assume that an update, on average, affects $f$% of the SCCs. Therefore, the cost of the hybrid algorithm is approximately $f$%× $Cost_k$. For example, the time for a full recomputation of the minimum bisimulation of XMark is approximately 0.036ms × 1.68M = 60.48 seconds. Suppose $f$ is 20%. The hybrid algorithm is efficient when the number of updates is smaller than 60.48/(18.4 × 20%) ≈ 16. From our experiments, we observe that this number is 12.

## 5 Incremental Maintenance

This section proposes the overall maintenance algorithm and its support on various forms of updates. First, in Section 5.1, we present the insertion of an edge ($n_1$, $n_2$) (*i.e.*, `insert` in Fig. 6), where $n_2$ can be either existing or new node. Then, single edge deletion is described in Section 5.2. Subgraph and batch updates are presented in Section 5.3.

### 5.1 Single edge insertion

In a nutshell, the overall algorithm consists of a `stabilize` phase with an explicit SCC handling, which is absent in previous algorithms, and a `minimize` phase which is essentially `bisimilar_cyclic` detailed in Section 4. Since an update may make a bisimulation unstable, the `stabilize` phase computes a stable but non-minimum bisimulation. Then, the `minimize` phase uses the hybrid algorithm for minimization. Below, we give the details of the `stabilize` phase.

**The stabilize phase.** The `stabilize` phase is presented in Lines 01-06 and 08-27 of Fig. 6. We maintain two priority queues, ranked by their topological order, to record two kinds of Inodes that need to be stabilized. Specifically, we use $\mathcal{S}$ to record the Inodes inside some *supernodes* (Line 02) and $\mathcal{Q}$ (Line 03) to record the *Inodes* that are not in any SCCs maintained in topological order. In the `stabilize` phase, we mark the modified Inodes (Lines 13 and 21) which will be examined in the `minimize` phase in topological order.

The pseudo-code in Fig. 6 can be described as follows. First, we simply insert a new data edge into the reduced data graph and a new edge between Inodes into the bisimulation graph, in Line 24 and Line 25, respectively. Assume the insertion makes the Inode of $n_2$ unstable (Line 26). To initialize $\mathcal{S}$, if $n_2$ is in a supernode, we add $n_2$ and its Inode to $\mathcal{S}$ (*i.e.*, $\{(I_{n_2}, n_2)\}$). Similarly, we add to $\mathcal{Q}$ the Inode $I_{n_2}$ if $n_2$ is not in any supernode (Line 27).

```
Procedure insert
Input: an edge to be inserted (n1, n2), a reduced graph G,
       and its minimum bisimulation B
Output: An updated graph G' and its updated minimum
       bisimulation B'

/* 1. Initialization */
01 if n1 ∉ G then raise exception      /* invalid insert */
02 S = ∅          // a priority queue of affected SCCs in top. order
03 Q = ∅          // a priority queue of affected non-SCCs in top. order
04 (G, B, Q, S) = insert_init((n1, n2), G, B, Q, S)

/* 2. Stabilize the updated B into a stable bisimulation */
05 istabilize(Q, S, G, B)
06 G' = Gabow(G)          /* update the SCC info. in G' */

/* 3. Merging the marked inodes by the hybrid algorithm */
07 return bisimilar_cyclic_marked(G', B)

Procedure istabilize(Q, S, G, B)
08 while Q ≠ ∅ or S ≠ ∅
     /* 1. stabilize the all Inodes in supernodes first */
09   while S ≠ ∅ then
10     pick a node (I_n, n) from S;
       remove (I_n, n) from S
11     if I_n is neither stable nor a singleton
12     split I_n into I_1 = I_n - {n} and I_2 = {n}
       for all (I_n, n') in S
           replace it with (I_1, n') and (I_2, n') in S
13     mark I_1 and I_2
14     add {(I_{n_s}, n_s) | n_s is child of n_i, n_i ∈ I_2
           and n_s in some supernodes} to S
15     add {I_{n_q} | n_q is a child of n_i, n_i ∈ I_2
           and n_q is not in any supernodes} to Q
     /* 2. stabilize Inodes that are not inside any supernode */
16   while Q ≠ ∅ then
17     pick a node I_n ∈ Q; remove I_n from Q
18     if I_n is neither stable nor a singleton
19       split I_n into a stable set T
         for all I_n in Q   replace it with T in Q
20       for each I in T
21         mark I
22         add {(I_{n_s}, n_s) | n_s is n_i's child,
               n_i ∈ I and n_s in the supernode of n} to S
23         add { I_{n_q} | n_q ∈ child of n_i,
               n_i ∈ I and n_q not in any supernodes} to Q

Procedure insert_init((n1, n2), G, B, Q, S)
24 G = insert (n1, n2) into G and the data graph of G
25 if n2 is new
   then create a new Inode I_{n2};
        insert I_{n2} into B;
        mark I_{n2}
26 if I_{n2} is not stable
27   add {(I_{n2}, n2) | n2 is inside a supernode} to S
     add {I_{n2} | n2 is not inside any supernode} to Q
```

Fig. 6. A single edge insertion for the minimum bisimulation of cyclic graphs

Next, we stabilize the Inodes in $\mathcal{S}$ and $\mathcal{Q}$ recursively until a stable bisimulation is obtained (*i.e.*, $\mathcal{S}$ and $\mathcal{Q}$ are empty): *(1)* We process the Inodes in $\mathcal{S}$ as follows (Lines 09-15): We select a node $n$ and its Inode $I_n$ from $\mathcal{S}$. We split $n$ from $I_n$ as the SCC of $n$ is potentially non-bisimilar to the SCC of other nodes in $I_n$ (Lines 10-12). (The maintenance of $\mathcal{S}$ will be needed by batch updates and explained in Section 5.3.) We mark the split Inodes (Line 13) so that they will be checked in the `minimize` phase. In Lines 14-15, we insert the children of the split Inode that are involved in some supernodes into $\mathcal{S}$ and the remaining children into $\mathcal{Q}$.

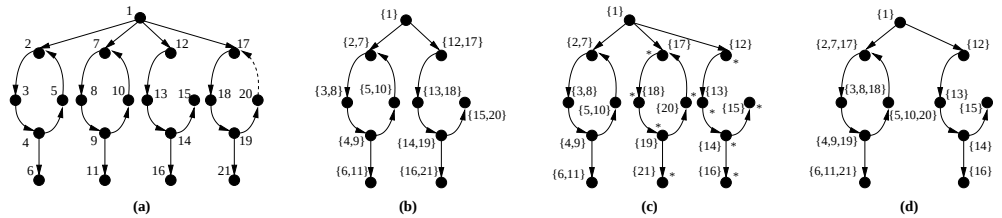



Fig. 7. (a) a data graph; (b) the minimum bisimulation; (c) a stabilized bisimulation; (d) the updated minimum bisimulation

*(2)* The handling of $\mathcal{Q}$ is shown in Lines 16-23. We select an Inode $I_n$ from $\mathcal{Q}$ (Line 17). As in other work (*e.g.,* [21]) for acyclic graphs (Line 19), if $I_n$ is not stable, we split $I_n$ into a set of stable Inodes $\mathcal{I}$. (Similarly, the maintenance of $\mathcal{Q}$ will be elaborated in batch update.) For succinctness of presentation, we postpone the discussion on our splitter optimization until Section 6. We mark Inodes in $\mathcal{I}$ in Line 21. In Lines 22-23, we update the affected nodes $\mathcal{S}$ and $\mathcal{Q}$, similar to Lines 14-15.

**Lemma 5.1:** *Given a graph $G_0$ and its reduced graph $G$, an edge insertion ($n_1$, $n_2$) and the minimum bisimulation $B$ of $G_0$, Procedure* `istabilize` *returns a stable bisimulation graph $B'$ of the updated $G_0$.* □

*Proof:* We prove the lemma by contradiction. First, we make the following assumption:

$\Psi$: "`istabilize` *yields an unstable bisimulation graph.*"

By this assumption $\Psi$, there is at least one unstable Inode $X$ in $B'$. Note that $X$ was stable before the insertion ($n_1$, $n_2$). The stability of some ancestors of $X$ must be altered by ($n_1$, $n_2$). As we are considering a *single* insertion, $X$ must be (directly or indirectly) connected to $n_2$. Then, we consider two (exhaustive) cases of $X$:

*Case 1*. If $X$ is directly connected to $n_2$, $X$ is made stable by Lines 04-05 and either of Lines 14-15 or Lines 22-23. This contradicts with $\Psi$. $\Rightarrow\Leftarrow$

*Case 2*. Suppose $X$ is indirectly connected to $n_2$. Then, we show a contradiction to $\Psi$ by an induction.

*Hypothesis*: `istabilize` stabilizes $\mathcal{I}_k$, where for all $I_k$ in $\mathcal{I}_k$, there is a path of the length $k$ from $n_2$ to $I_k$.

The *base case* is $k$ = 1, which is Case 1. Then, let us consider the *inductive step* $\mathcal{I}_{k+1}$. Consider an Inode $I_{k+1} \in \mathcal{I}_{k+1}$, where $I_{k+1}$ is a child node of an Inode $I_k$ in $\mathcal{I}_k$. Suppose $I_{k+1}$ is unstable *w.r.t.* $I_k$. Since the whole bisimulation graph was stable before the insertion, $I_k$ must have been modified, specifically, *split by* `istabilize`, in an earlier iteration of `istabilize`. `istabilize` is invoked in Line 12 and Line 19 only. Thus, we examine the following two (exhaustive) sub-cases:

*Case 2.1*. Suppose $I_k$ is the result of the split logic at Line 12. The reduced graph is a `DAG` and the while loop in `istabilize` encodes a topological traversal. Therefore, when $I_k$ is split, its child Inodes, including $I_{k+1}$, are not marked. Suppose $I_k$ is either $I_1$ or $I_2$ in Line 12. Since $I_{k+1}$ is a child of $I_k$, $I_{k+1}$ is added to either $\mathcal{S}$ (Line 14) or $\mathcal{Q}$ (Line 15). Therefore, $I_{k+1}$ will be stabilized in later iterations.

*Case 2.2*. Suppose $I_k$ is the result of the split logic at Line 19. Due to the topological traversal of `istabilize`, $I_{k+1}$ is not marked. Its child Inodes, including $I_{k+1}$, are added to either $\mathcal{S}$ or $\mathcal{Q}$ (Lines 22-23) and stabilized in later iterations.

Therefore, all unstable Inodes $I_{k+1}$'s in $\mathcal{I}_{k+1}$ must be stabilized in some iterations in `istabilize`, as they are all placed in either $\mathcal{S}$ or $\mathcal{Q}$. The hypothesis is true, which is a contradiction of $\Psi$. $\Rightarrow\Leftarrow$ □

The `stabilize` phase essentially traverses the bisimulation graph $B$ and `SCC`s in the reduced data graph to stabilize and mark the Inodes that are affected by the update. In addition, since `SCC`s themselves may be affected by an update, we use Gabow's algorithm to compute the updated `SCC` information of the reduced graph, which is needed by the hybrid algorithm (in Line 06).

**The minimize phase.** The `minimize` phase is an application of `bisimilar_cyclic` detailed in Section 4.1, with a minor modification. The modification is that we do not apply the hybrid algorithm to all of the Inodes returned by the `stabilize` phase, but only to the Inodes that have been marked.

**Example 5.1:** We illustrate Algorithm `insert` (mainly `istabilize`) with an example. A cyclic data graph is shown in Fig. 7(a). For simplicity, we assume that the nodes in the data graph have the same label and skip the drawings of trivial supernodes. To facilitate discussion, the node id of each data node is shown. We use $\{\}$ to denote an Inode. The minimum bisimulation of Fig. 7(a) is shown in Fig. 7(b). Assume that we insert an edge (20,17) into the data graph. Algorithm `insert` initially puts $\{12,17\}$ into $\mathcal{Q}$ (Line 04). $\{12,17\}$ is unstable because node 20 has a child in $\{12,17\}$ but node 15 does not. Then, in Line 19, node 17 is split from $\{12,17\}$. The modified Inodes are marked, with a "*" sign in the figure. The `stabilize` phase proceeds recursively and finally produces the graph in Fig. 7(c). We invoke Gabow's algorithm to update the `SCC` information of the reduced data graph. By calling the hybrid algorithm, we obtain the updated minimum bisimulation, shown in Fig. 7(d).

It should be remarked that while a previous work [21] produces the same stabilized graph (Fig. 7(c)), it returns Fig. 7(c) as the final bisimulation. This is because it lacks the handling on `SCC`s as discussed in Section 4. Subsequently, any subgraph that is connected to the `SCC` with nodes 17, 18, 19 and 20 (*e.g.,* node 21) will not be merged, as the `SCC` is not merged with other bisimilar `SCC`s. □

**Analysis.** The time complexity of Procedure `istabilize` can be established as follows. The Inodes are processed by either of the while loops (Lines 09 and 16) at most once, which takes $O(|E|)$. Without any optimization, the split takes $O(|V|)$ in the worst case. With proper optimization (*e.g.,* [29]), stablizing an Inode can be done in $O(log(|V|))$. Hence. the `stabilize` phase runs in $O(|E|log(|V|))$.

**Theorem 5.2:** *The bisimulation graph returned by* `insert` *is minimum.* □

*Proof:* This can be established by putting Theorem 4.2 and Lemma 5.1 together. Specifically, `istabilize` returns a correct bisimulation $B$ of the updated graph and the hybrid algorithm determines the minimum bisimulation of $B$. □

### 5.2 Single edge deletion

Our technique on edge insertion can be adopted to support an edge deletion ($n_1$, $n_2$) with minor modifications. Here, we discuss the modifications in relation to the pseudo-code of the algorithm `insert` presented in Fig. 6: (i) In Line 01, we delete the edge from the data graph. (ii) After the deletion, we check the stability of $I_{n_2}$ in Line 02, initialize $\mathcal{S}$ and $\mathcal{Q}$, and finally invoke `stabilize` and `bisimilar_cyclic_marked` as in `insert`.

**Example 5.2:** Following up the insertion example of Fig. 7, suppose that we delete the edge between nodes 20 and 17 from the data graph of the bisimulation shown in Fig. 7(d). Since node 17 is in a supernode, ($\{$2,7,17$\}$, 17) is placed in $\mathcal{S}$. Then, by Lines 09-15 of `istabilize`, we split 17, 18, 19 and 20 from the Inodes of the supernode iteratively. Since Inode $\{$6,11,21$\}$ is connected to the supernode, node 21 is also split from its Inode by Lines 16-23. This results in the graph in Fig. 7(c) *without the edge ($\{$20$\}$, $\{$17$\}$)*. By applying the hybrid algorithm on such graph, we obtain the minimum bisimulation shown in Fig. 7(b). □

### 5.3 Batch Updates

This subsection shows how updates of a subgraph and batch updates of edges are supported. Similar to the previous sections, we present our techniques with insertions. The details of deletions are similar and are omitted due to space constraints.

**Subgraph insertions.** A subgraph insertion can be supported by a combination of the techniques proposed in earlier sections. The pseudo-code is presented in Procedure `insert_subgraph`, in Fig. 8. We assume that the to-be-inserted subgraph $g$ will be connected to an existing graph $G$ via an edge. `insert_subgraph` computes the minimum bisimulation of $g$, denoted as $b$, and invokes `insert` to insert the edge that connects $G$ and $g$, that returns the minimum bisimulation of the updated graph. The correctness of `insert_subgraph` is a direct extension of Theorem 5.2.

As presented in Section 4, the performance bottleneck of `insert` is on determining bisimulation between `SCC`s and non-`SCC`s (*i.e.*, the cross nodes of `bisimilar_cyclic`). In `insert_subgraph`, when a subgraph is inserted, `bisimilar_cyclic` is invoked once.

**Batch insertions.** It may sometimes be inconvenient to only allow updates that localized in a subgraph. In addition, if a to-be-inserted subgraph $g$ is connected to an exsiting graph $G$ with multiple edges, we first insert the subgraph using one edge and then insert the remaining edges in a batch. To address these updates efficiently, the technique above is generalized to other forms of batch insertions, as shown in Procedure `insert_batch` of Fig. 8. More specifically,

**Procedure** `insert_subgraph`
**Input**: a subgraph to be inserted ($n_1$, $g$), a reduced graph $G$, and its minimum bisimulation $B$
**Output**: An updated graph $G'$ and its updated minimum bisimulation $B'$

01 $b$ = `partition_refinement`($g$)
02 `insert`(($n_1$, $n_2$), $G \cup g$, $B \cup b$)

**Procedure** `insert_batch`
**Input**: a set of edges to-be-inserted $E_\delta$, a reduced graph $G$, and its minimum bisimulation $B$
**Output**: An updated graph $G'$ and its updated minimum bisimulation $B'$

01 **for each** ($n_1$, $n_2$) **in** $E_\delta$
  **if** $n_1 \notin G$ **then** raise exception  /* invalid insert */
02 $\mathcal{S} = \emptyset$
03 $\mathcal{Q} = \emptyset$
04 **for each** $e$ **in** $E_\delta$
05  ($G$, $B$, $\mathcal{S}$, $\mathcal{Q}$) = `insert_init`($e$, $G$, $B$, $\mathcal{Q}$, $\mathcal{S}$)
06 `istabilize`($\mathcal{Q}$, $\mathcal{S}$, $G$, $B$)
07 $G'$ = `Gabow`($G$)
08 **return** `bisimilar_cyclic_marked`($G'$, $B$)

Fig. 8. Insertion of a subgraph

`insert_batch` takes a set of edge insertions $E_\delta$, a reduced graph $G$ and its minimum bisimulation $B$ as input and outputs the minimum bisimulation. `insert_batch` calls `insert_init` to initialize the two priority queues ($\mathcal{S}$ and $\mathcal{Q}$) used in `insert` with each edge in $E_\delta$. If inserting $e$, where $e$ = ($n_1$, $n_2$), makes $B$ unstable, we record $n_2$ in $\mathcal{S}$ and $\mathcal{Q}$ and stabilize the relevant Inodes from $I_{n_2}$ iteratively by using `istabilize` (Line 06). In `stabilize`, the Inodes to-be-split due to the edges in $|E_\delta|$ may be overlapping, which may appear multiple times in $\mathcal{S}$ and $\mathcal{Q}$. Thus, after each split, `istabilize` replaces the existing Inode in $\mathcal{S}$ and $\mathcal{Q}$ with the Inode after the split (Lines 12 and 19 of `istabilize`).

We remark that in `insert_batch`, the initialization of `insert` is invoked $|E_\delta|$ times, which takes $O(|E_\delta|)$ time. In the `stabilize` phase, the Gabow's algorithm and the `minimize` phase are invoked once only. Therefore, `insert_batch` and `insert` have the same time complexity when $|E_\delta| \ll |G|$. In other words, `insert_batch` extends the usability of `insert` by paying a negligible cost.

Due to space limitations, we prove the correctness of `insert_batch` in Appendix B.

## 6 Optimization on the Hybrid Algorithm

On top of the algorithms proposed earlier, we present three optimizations, that arise from bisimulation maintenance. The first two are applicable to partition refinement in general (*i.e.*, for acyclic and cyclic graphs). The third is specific to cross nodes which affect the performance of our hybrid algorithm.

**1. Optimizing partition refinement.** Given a reduced graph $G$ of a data graph $G_0$ (after update), Lemma 5.1 states that the bisimulation $B$ returned by `istabilize` is a stable bisimulation of $G_0$. By Theorem 1 of Kaushik *et al.* [28], $B$ is known to be a refinement of the minimum bisimulation. This implies that it is not necessary to further refine the partitions in $B$. Thus, only partial refinement is invoked in maintaining the minimum bisimulation.

Specifically, in computing the minimum bisimulation of $B$ using partial refinement, we consider the input data graph as $B$

as opposed to the nodes of SCCs of $G_0$. That is, the partitions in $B$ are *non-partitionable*. Then, partition refinement is applied on the SCCs of $B$, whose size is often much smaller than those of $G_0$. Assuming that $B_{min}$ is the minimum bisimulation obtained, the nodes in a partition $I$ in $B_{min}$ are simply all of the nodes in $I_i$ for all $i$: $\forall_{I_i \in I}\ \{n \mid n \in I_i\}$. A similar optimization (in the absence of reduced graphs) has been adopted in previous work (*e.g.*, [21], [28]).

**2. Optimizing refinement splitter.** A classical optimization on partition refinement is to recursively split smaller partitions first [29]. It has been shown that this optimization leads to a lower time complexity and to a faster partition convergence. It is evident that such an optimization assumes that the size (the number of nodes) of a partition is generally directly proportional to the splitting required to stabilize the partition. If the topology of the bisimulation graph does not alter dramatically by an update, the existing topology can be a direct indication of the number of splits that partition refinement requires for convergence. Hence, we propose an efficient optimized splitter that exploits the existing bisimulation.

The details of the new splitter can be presented as follows: Given a partition $I$ during the process of partition refinement, the number of old (overlapping) partitions in $I$ is assumed to be approximately the number of splits required to converge to the final partition(s). The optimized splitter of partition refinement (in Fig. 4, `partition_refinement`, Line 15) will first split the partition that overlaps with the smallest number of old partitions.

**3. Optimization on cross nodes.** A performance issue of the hybrid algorithm is the minimization of cross nodes $\mathcal{C}$ (*i.e.*, its cost $C$ in Formulas (1) and (2)). Given an update, Procedure `istabilize` may split the cross nodes from the Inodes of some SCCs but subsequently, most cross nodes may be merged with their Inodes before, if the update does not change the minimum bisimulation much. Our next optimization aims at reducing some unnecessary splits and merges of cross nodes.

The main idea is to "mask out" the cross nodes in SCCs when applying Procedure `istabilize`. The bisimulation of non-cross nodes are maintained by `insert` without the cross nodes. At the end of `insert`, an (unstable) intermediate bisimulation is yielded. Then, we unmask the cross nodes and minimize the intermediate bisimulation top-down again and verify whether the Inodes of the unmasked cross nodes may require splitting and merging. If the minimum bisimulation does not change much, most cross nodes pass the verification and no splitting and merging are ever required.

## 7 Experimental Evaluation

This section presents a detailed experimental evaluation that verifies the efficiency of the proposed hybrid algorithm and the effectiveness of its optimizations.

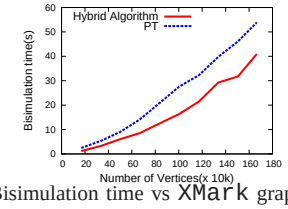
**Hardware and software.** We ran our experiments on a server with a Dual 4-core 2.93GHz CPU and 30 GB RAM running Solaris OS. The hybrid algorithm was developed on top of a previous algorithm [21], which was written in JDK 1.5. The library for graphs is `openjgraph` [31], which is also used in previous work [21], to ensure fair comparison.

**Data graphs.** We used both synthetic and real-world datasets in our experiments: (i) As in the previous work [21], we used the XMark generator [32] to derive synthetic graphs to illustrate various aspects of our algorithms. The SCCs in XMark are mainly composed of many IDREFs of `open_auction`s to `person`s and vice versa. In the test with various graph sizes, the number of vertices/edges ranged from 168k/199k to 1.68M/1.97M and the number of IDREFs ranged from 31k to 307k. Unless specified otherwise, we used XMark with 168k vertices to study various characteristics of our algorithms. We remark that XMark graphs often contain one large SCC (that contains many smaller SCCs). This is the reason why [21] can maintain minimal bisimulations, whose sizes are similar to the minimum ones. To test the algorithm on various cyclic graphs, we will specify some preprocessing on XMark to generate update workloads. (ii) A real-world graph on bibliography data up to Year 2002, denoted as `dblp`, was used. It contains 510k vertices, 784k edges and *167* SCCs. The vertices of `dblp` can be authors and publications. The reference edges in `dblp` represent citations between publications. In some experiments, we used some `dblp` graphs of various sizes by extracting the last $5{\times}x$-th years of `dblp`, by varying $x$. (iii) Another real-world graph on citation data [25], denoted as `citation`, was used. It contains 45k vertices, 374k edges and 11 SCCs. The number of the SCCs in `citation` is somewhere between the number in the XMark and `dblp` datasets. (iv) The last real dataset we used was a social network dataset called `slashdot` [33]. It contains 82k vertices and 948k edges. Each vertex is a user, labeled with `user`. An edge represents friendship between users. `slashdot` has one large SCC containing many smaller non-trivial SCCs. The SCC has 71k vertices and 912k edges (*i.e.*, 87% and 96% of all vertices and edges, respectively).

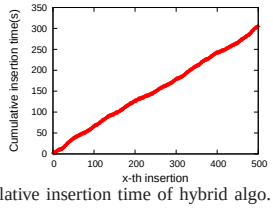
**Experiment 1: Reconstruction performance.** To study the runtimes of the hybrid algorithm, we invoked `bisimilar_cyclic` on XMark graphs of various sizes and compared them with partition refinement [29]. The result is shown in Fig. 9(a). The figure shows that the hybrid algorithm outperformed partition refinement, even in full reconstructions. In particular, for the data graph with 1.68M vertices, the hybrid algorithm was 24% faster. The reason is that the subgraphs in XMark are not "very bisimilar" and partition refinement takes many steps to converge.

**Experiment 2a: Maintenance under single insertion.** We then performed an experiment on insertions. Given the XMark with 168k/199k vertices/edges, we removed 500 edges randomly and then inserted them. The cumulative insertion time is reported in Fig. 9(b). The $x$-axis is the number of insertions executed and the $y$-axis is the cumulative time of $x$ insertions. Fig. 9(b) shows that the cumulative insertion time was linear to the number of insertions. Moreover, the hybrid algorithm always returned the minimum bisimulations (Fig. 9(c)).

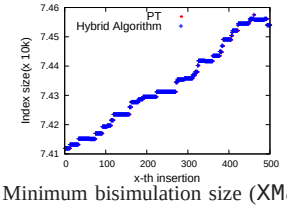
We conducted a similar experiment on `dblp` and `citation`, *i.e.*, real graphs with more SCCs. First, we focused on `dblp`. The hybrid algorithm took 11s to compute the minimum bisimulation of `dblp` from scratch. Similar to the previous experiment, we removed 500 edges randomly from the data



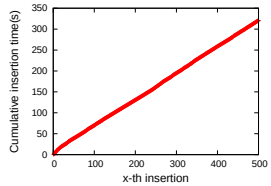

(a) Bisimulation time vs XMark graph sizes



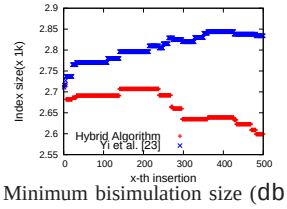

(b) Cumulative insertion time of hybrid algo. (XMark)



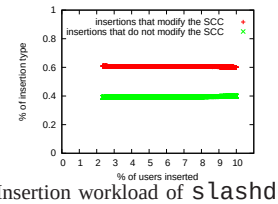

(c) Minimum bisimulation size (XMark)



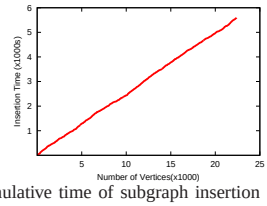

(d) Cumulative insertion time of hybrid algo. (dblp)



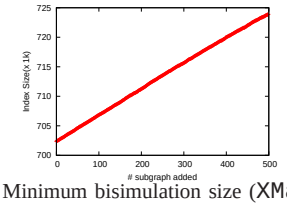

(e) Minimum bisimulation size (dblp)



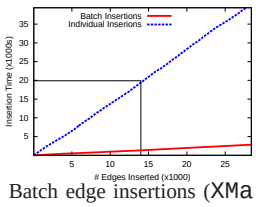

(f) Insertion workload of slashdot




(g) Cumulative time of subgraph insertion (XMark)

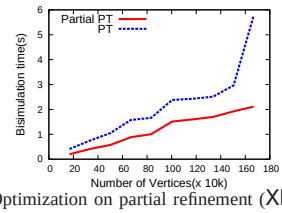



(h) Minimum bisimulation size (XMark)

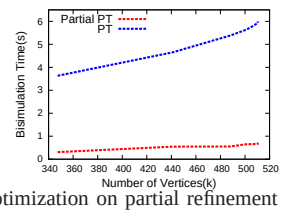



(i) Batch edge insertions (XMark)

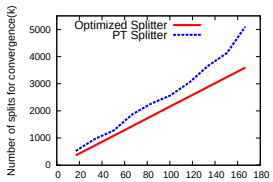
Fig. 9. Performance evaluation of the hybrid algorithm and the minimum bisimulations of data graphs



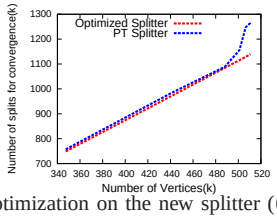

(a) Optimization on partial refinement (XMark)



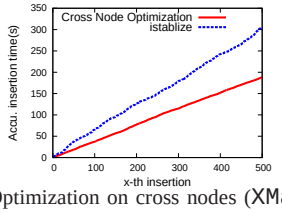

(b) Optimization on partial refinement (dblp)



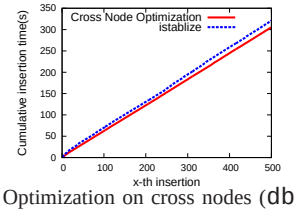

(c) Optimization on the new splitter (XMark)




(d) Optimization on the new splitter (dblp)




(e) Optimization on cross nodes (XMark)




(f) Optimization on cross nodes (dblp)

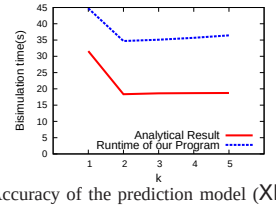
Fig. 10. Effectiveness of the optimization techniques of the hybrid algorithm

graph and inserted them back. On average, each insertion took only 0.63s. The cumulative insertion time is reported in Fig. 9(d). The figure shows that the cumulative insertion time was also linear to the number of insertions. Unlike the XMark datasets, the sizes of the minimum bisimulations of dblp *decreased* as the edges were inserted (Fig. 9(e)). This was because dblp was relatively more bisimilar than XMark and bisimilar nodes were recovered as edges were inserted. We note that the minimal bisimulation of dblp returned by the previous work [21] was 10% larger than the minimum one.

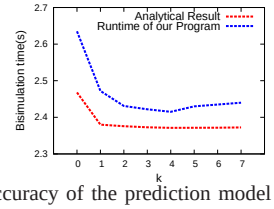
We then applied the hybrid algorithm on citation. The algorithm took 1.59s to compute the minimum bisimulation from scratch. Similar to previous insertion experiments, we removed 500 edges from the data graph and then inserted them. Each insertion took 0.37s on average and the cumulative insertion time is omitted as it also roughly linear to the number of insertions. Unlike the XMark and dblp datasets, the sizes of the minimum bisimulations of citation (roughly 10.4k Inodes) did not change much throughout the 500 insertions.

We tested our algorithm on slashdot. The vertex has an id and it can be considered as user id [33]. We assumed the larger the id, the newer the user. We removed $x$% of vertices of the largest ids and their edges, and inserted their edges back (as new users registered and added friends) in the ascending order of ids. (For each new vertex, we inserted their edges randomly.) We ranged $x$ from 1% to 10% and consistently obtained the following workload (shown in Fig. 9(f)) and results: The update workload contained 40% of new edges that did not modify the SCC and the hybrid algorithm finished each of such insertion in 0.3s, as the non-SCC part (in terms of edges) is only 4% of the entire graph. The remaining 60% of the insertions led to a performance that was identical to that of recomputation (*i.e.*, 8s), since the SCC was large.

**Experiment 2b: Maintenance under batch insertions.** We tested the performance of subgraph insertions and batch edge insertions using XMark, since we could have a better control on generating batch insertions. First, we selected 500 open_auction subgraphs randomly and removed them from the XMark graph. The open_auction subgraphs were often connected to the XMark graph via many IDREF edges. These edges, denoted as $S$, were used for batch edge insertions. On average, each subgraph had 45 vertices and 44 edges. The




(a) Accuracy of the prediction model (XMark)




(b) Accuracy of the prediction model (dblp)

Fig. 11. Accuracy of the analytical model on the relative performance of $k$'s in the generalization of the reduced graphs

cumulative insertion time is reported in Fig. 9(g).

The $x$-axis is the total number of edges of those subgraphs to-be-inserted and the $y$-axis is the cumulative insertion time. The result shows that the insertion time was linear to the total number of vertices. The slope was 11.2s per subgraph (4 vertices per second). In contrast, a full reconstruction took 39s.

For batch edge insertions, we used the IDREF edges $S$, as described above: We extracted 500 sets of edges from XMark; Each set contained 56 edges on average. We compared batch insertion (*i.e.*, insert_batch) and individual insertion (*i.e.*, insert). The result is shown in Fig. 9(i). The throughputs of insert_batch and insert were 8.97 and 0.72 edges per second, respectively. insert_batch is about 12.5 times faster than insert. The time for the insertions by reconstructions was about 19,500s and insert_batch is more than 800 times faster. While insert_batch is slower than the maintenance of minimal bisimulation [21], [21] requires occasional reconstruction and insert_batch does not.

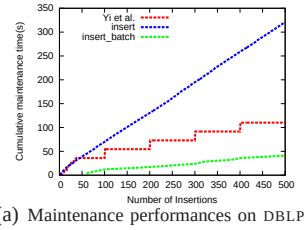
Similarly, we applied insert_batch on slashdot with 500 edge insertions containing 347 new users. We obtained a throughput 34.3 edges (23.8 nodes) per second. insert_batch is 372 times faster than full reconstructions. This verified that batch insertions with the hybrid algorithm can handle rapid updates in real time.

**Experiment 3: Optimization effectiveness.** We next present an experiment on the synthetic dataset XMark and the largest real dataset dblp that verified the effectiveness of each optimization. Due to space limitations, the experiments on citation and slashdot are placed in Appendix A.

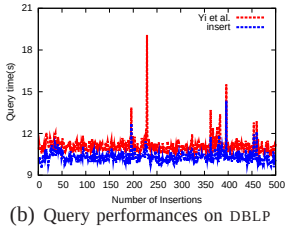
(1) First, we applied partition refinement in bisimilar_cyclic with and without partial refinement. The result is reported in Figs. 10(a)-(b). Fig. 10(a) shows that the performance improvement from the optimization on XMark increased as the graph size increased. The result from dblp (Fig. 10(b)) shows that the optimization only required roughly 11% of splits required by the classical splitter.

(2) Second, we verified the effectiveness of our splitter; the results are shown in Figs. 10(c)-(d). The figures show that our splitter always converged earlier than the classical splitter. In XMark, out splitter required up to 30% fewer steps to converge (Fig. 10(c)). In full dblp (*i.e.*, with 510k vertices), our splitter offered up to 10% fewer splits (Fig. 10(d)).

(3) Third, we tested the optimization on cross nodes, where the optimal $k$'s had been accurately determined by the next experiment on the analytical model (Section 4.2). We performed the insertions on XMark as specified before. As shown in Fig. 10(e), the total time for 500 insertions with the optimization was approximately 62% of the time taken using the non-optimized version. Fig. 10(f) shows that the optimization on cross nodes offered roughly 5% improvement on dblp.




(a) Maintenance performances on DBLP




(b) Query performances on DBLP

Fig. 12. Maintenance and query performances of minimum and minimal bisimulations on DBLP

**Experiment 4: Accuracy of the cost model.** We present the experimental results on the accuracy of the analytical model presented in Section 4.2. We compared the actual runtimes of our implementation and those determined by the analytical model on XMark and dblp. The results are presented in Figs. 11(a)-(b). The figures show that the $k$ with the optimal performance of the hybrid algorithm on the three data graphs were correctly predicted. That is, the trend of the actual runtimes and the predicted ones were consistent. Our experimental results reveal that when we used the generalized reduced graphs, small $k$'s would be sufficient. However, there were clear gaps between the runtimes and the predicted times. The reason is that the implementation involved manipulations of many data structures. These implementation specifics were not modeled, and they introduced some non-trivial runtimes.

**Experiment 5: Maintenance and query performances.** Finally, we illustrate the effects of maintenance and query evaluation with bisimulation with the DBLP dataset. We generated 500 random insertions that involve SCC. That is, even one edge was inserted, the minimal and minimum bisimulation may differ. We assume that the previous method reconstructed the bisimulation after every 100 insertions. Moreover, we generated 1000 reachability queries randomly between nodes involving SCCs. Queries were evaluated via a depth-first traversal of the bisimulation. Fig. 12(a) shows the maintenance times. The reconstruction of previous work led to clear increases in the maintenance time. At the first glance, individual maintenance did not appear efficient. When the batch maintenance was used, for example, a batch of insertions containing roughly 50 insertions on average, it was already clearly more efficient than the previous work. One may be tempted to use the minimal method with less frequent reconstructions. However, queries would be evaluated on larger bisimulations. Fig. 12(b) shows the total query times between time of the minimal and minimum bisimulations. The minimum one consistently produced smaller query times than the minimal one. In addition, the minimal bisimulation sometimes led to significantly longer query times.

## 8 Conclusions

We have proposed a novel incremental maintenance of the minimum bisimulation of cyclic graphs with respect to an edge insertion/deletion or a batch insertion/deletion. We have proposed a hybrid algorithm that takes advantage of the two existing classes of bisimulation minimization algorithms, namely merging algorithms and partition refinement. To our knowledge, the hybrid algorithm is the first incremental maintenance algorithm that guarantees minimum bisimulation for cyclic graphs. We have proposed a generalization of reduced graphs and an analytical model to facilitate an optimal performance from the hybrid algorithm. We have complemented the hybrid algorithm with three optimizations. We have presented a detailed experiment on both synthetic and real graphs that verified the efficiency of the hybrid algorithm and the effectiveness of our optimizations. We are investigating to incorporate the recent external algorithm for bisimulation [9] into our algorithms.

**Acknowledgements.** We are grateful to Ke Yi for providing the implementation of [21]. We are thankful to anonymous reviewers whose comments greatly improved the quality of this work. This work is supported by research grants GRFs HKBU210510 and HKBU210409, and FRG/07-08/I-59.

## References

[1] T. Milo et al, "Index structures for path expressions," in *ICDT*, 1999.
[2] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *ICDE*, 2002.
[3] Q. Chen, A. Lim, and K. W. Ong, "D(k)-index: an adaptive structural summary for graph-structured data," in *SIGMOD*, 2003.
[4] J. Spiegel and N. Polyzotis, "Graph-based synopses for relational selectivity estimation," in *SIGMOD*, 2006.
[5] N. Polyzotis and M. Garofalakis, "XCluster synopses for structured XML content," in *ICDE*, 2006.
[6] D. K. Fisher and S. Maneth, "Structural selectivity estimation for XML documents," in *ICDE*, 2007.
[7] P. Buneman, M. Grohe, and C. Koch, "Path queries on compressed XML," in *VLDB*, 2003.
[8] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas, "Vectorizing and querying large XML repositories," in *ICDE*, 2005.
[9] J. Hellings, G. H. Fletcher, and H. Haverkort, "Efficient external-memory bisimulation on DAGs," in *SIGMOD*, 2012, pp. 553–564.
[10] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.
[11] N. Polyzotis and M. Garofalakis, "XSketch synopses for XML data graphs," *ACM Trans. Database Syst.*, vol. 31, no. 3, 2006.
[12] N. Polyzotis, M. Garofalakis, and Y. Ioannidis, "Approximate XML query answers," in *SIGMOD*, 2004.
[13] H. Li, M. L. Lee, W. Hsu, and G. Cong, "An estimation system for XPath expressions," in *ICDE*, 2006.
[14] R. Kumar, J. Novak, and A. Tomkins, "Structure and evolution of online social networks," in *KDD*, 2006, pp. 611–617.
[15] Ntoulas et al, "What's new on the web?: the evolution of the web from a search engine perspective," in *WWW*, 2004, pp. 1–12.
[16] R. Wauters, "Facebook averaged almost 8 new registrations per second in 2010," http://techcrunch.com/2011/02/01/facebook-averaged-almost-8-new-registrations-per-second-in-2010/, 2011.
[17] Socialbakers, "Facebook pages statistics," http://www.socialbakers.com/facebook-pages/, 2011.
[18] E. Bolton, Y. Wang, P. Thiessen, and S. Bryant, "Pubchem: Integrated platform of small molecules and biological activities," *Annual Reports in Computational Chemistry*, vol. 4, 2008.
[19] M. Ley, "The DBLP computer science bibliography: Evolution, research issues, perspectives," in *SPIRE'02*, 2002, pp. 1–10.
[20] G. Ramalingam and T. Reps, "A categorized bibliography on incremental computation," in *POPL*, 1993, pp. 502–510.
[21] K. Yi, H. He, I. Stanoi, and J. Yang, "Incremental maintenance of XML structural indexes," in *SIGMOD*, 2004.
[22] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *SIGMOD*, 2011, pp. 925–936.
[23] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, 2012, pp. 157–168.
[24] D. Saha, "An incremental bisimulation algorithm," in *FSTTCS*, 2007.
[25] J. Leskovec, "Stanford large network dataset collection," http://snap.stanford.edu/data.
[26] A. Dovier, C. Piazza, and A. Policriti, "An efficient algorithm for computing bisimulation equivalence," *Theor. Comput. Sci.*, vol. 311, no. 1-3, pp. 221–256, 2004.
[27] J. Deng et al., "Optimizing incremental maintenance of minimal bisimulation of cyclic graphs," in *DASFAA*, 2011.
[28] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy, "Updates for structure indexes," in *VLDB*, 2002.
[29] R. Paige and R. E. Tarjan, "Three partition refinement algorithms," *SIAM J. Comput.*, vol. 16, no. 6, pp. 973–989, 1987.
[30] H. Gabow, "Searching," in *Discrete Math. and its Applications: Handbook of Graph Theory*, J. Gross and J. Yellen, Eds. CRC Press, 2003, pp. 953–984.
[31] J. M. Salvo, "Openjgraph – java graph and graph drawing project," http://openjgraph.sourceforge.net/latest.html.
[32] A. Schmidt et al., "XMark: A benchmark for XML data management," in *VLDB*, 2002.
[33] J. Leskovec et al., "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.


**Jintian Deng** is a post-graduate research student in the Department of Computer Science, Hong Kong Baptist University. He received his BSc degree in Computer Science from Nanjing University in 2009. His research interests include graph-structured databases and XML. He is a member of the Database Group at Hong Kong Baptist University (http://www.comp.hkbu.edu.hk/∼db/).


**Byron Choi** received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is now an assistant professor in the Department of Computer Science at the Hong Kong Baptist University.


**Jianliang Xu** is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and his PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He held visiting positions at Pennsylvania State University and Fudan University.


**Haibo Hu** is a research assistant professor in the Department of Computer Science, Hong Kong Baptist University. Prior to this, he held several research and teaching posts at HKUST and HKBU. He received his PhD degree in Computer Science from the Hong Kong University of Science and Technology in 2005. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing.


**Sourav S Bhowmick** is an Associate Professor in the School of Computer Engineering, Nanyang Technological University and the Director of Data-Intensive Scalable Computing (DISCO) Lab. He is a visiting associate professor at the Biological Engineering Division, Massachusetts Institute of Technology. He also holds the position of Singapore-MIT Alliance Fellow in Computation and Systems Biology. Sourav received his Ph.D. in Computer Engineering 2001.

## APPENDIX A
## SUPPLEMENTARY EXPERIMENT

Due to the space constraint, we present the experiment on the optimization techniques on the two real datasets `citation` and `slashdot` in this appendix.

We applied partition refinement in `bisimilar_cyclic` with and without partial refinement on `citation` and `slashdot` and reported the results in Figs. 13(a)-(b). Fig. 13(a) shows that the optimiation on `citation` reduced the runtime by approximately 41%. Fig. 13(b) shows that this optimization offered a marginal improvement on `slashdot`.

Regarding the optimization on the splitter, in `citation`, the new splitter required 16% fewer splits compared to the classical splitter (Fig. 14(a)). Regarding `slashdot`, the new splitter often reduced the number of splits by 8% (Fig. 14(b)).

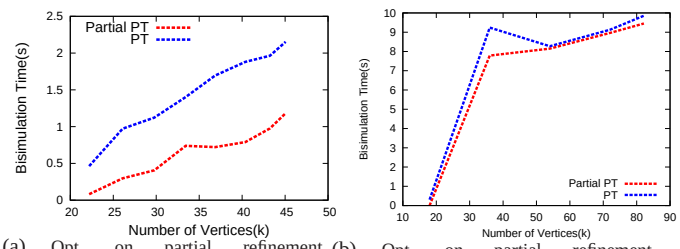
Finally, we tested the optimization on cross nodes. Fig. 15(a) shows that the optimization on cross nodes offered roughly 24% performance improvement on `citation`. When tested with `slashdot`, the performance of the optimized hybrid algorithm was 2.25 times better than that of non-optimized version (shown in Fig. 15(b)). This is because the updates did not often change the bisimulation of the `SCC` and the `SCC` of `slashdot` and therefore the number of cross nodes were large.

## APPENDIX B
## CORRECTNESS OF BATCH UPDATES

**Proposition 2.1:** *Given an insertion of a subgraph $g$ and the existing minimum bisimulation $B$ of a graph $G$, where $g$ is connected to $G$ via an edge $(n_1, n_2)$, the bisimulation graph returned by* `insert_subgraph` *is minimum.* □

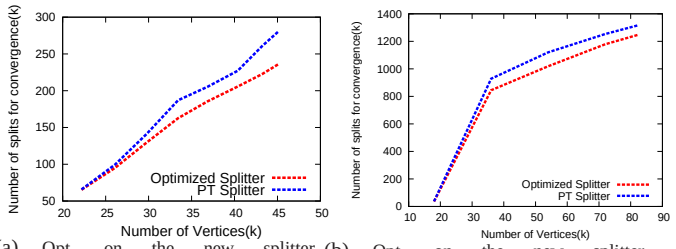
*Proof:* First, `partition_refinement`($g$) returns the minimum bisimulation $b$ of $g$. Without ($n_1$, $n_2$), $g$ and $G$ are disconnected and $b$ and $B$ are their minimum bisimulation, respectively. By Defintion 3.6, $b \cup B$ is the minimum bisimulation of $g \cup G$. Next, consider the insertion of ($n_1$, $n_2$). By Theorem 5.2, `insert` is correct and the maintenance of $b \cup B$ in response to the insertion of a single edge $(n_1, n_2)$ into $g \cup G$ is minimum. □

**Proposition 2.2:** *Given a batch of edges of $E_\delta$ of a graph $G$ and its minimum bisimulation $B$, the bisimulation graph returned by* `insert_batch` *is minimum.* □



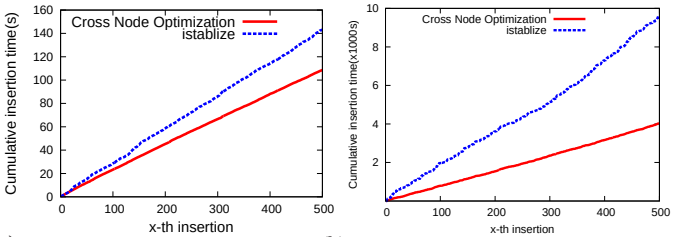

(a) Opt. on partial refinement (`citation`) (b) Opt. on partial refinement (`slashdot`)

Fig. 13. Effectiveness of partial refinement of the hybrid algorithm on `citation` and `slashdot`




(a) Opt. on the new splitter (`citation`) (b) Opt. on the new splitter (`slashdot`)

Fig. 14. Effectiveness of the new splitter on `citation` and `slashdot`




(a) Optimization on cross nodes (`citation`) (b) Optimization on cross nodes (`slashdot`)

Fig. 15. Effectiveness of optimization on cross nodes on `citation` and `slashdot`

*Proof:* The proof can be established by applying Lemma 5.1 and Theorem 5.2 in a simple case analysis and mathematical induction on the size of $|E_\delta|$, where $|E_\delta|$ is the batch of edges to-be-inserted.

*Hypothesis:* The `istabilize` of `insert_batch` returns a stable bisimulation when $|E_\delta| \leq n$.

*Base case:* When $|E_\delta|$ is 1, the hypothesis is true due to Lemma 5.1.

*Inductive step: Suppose the hypothesis is true up to* $|E_\delta| = m$. Next, consider $|E_\delta| = m + 1$. Denote that $E_\delta = \{e_1, e_2, ..., e_m, e_{m+1}\}$.

*Case 1: $e_{m+1}$:$(n_1, n_2)$ does not directly cause $I_{n_2}$ unstable.* Then, $I_{n_2}$ is not added to the queues $\mathcal{S}$ and $\mathcal{Q}$ by `insert_init`. If $I_{n_2}$ causes the bisimulation unstable, it must be only due to the insertions of $e_1, e_2, ...,$ or $e_m$. Due to the hypothesis assumption, $I_{n_2}$ is stabilized by `istabilize`.

*Case 2: $e_{m+1}$:$(n_1, n_2)$ directly causes $I_{n_2}$ unstable.* $I_{n_2}$ is added to the priority queues $\mathcal{S}$ and/or $\mathcal{Q}$ by `insert_init`. We exploit the following two facts in our arguments.

- *Fact (1)*: `istabilize` stabilizes (unstable) Inodes by splitting. It does not merge Inodes. We note that stable Inodes will not be split into unstable Inodes.
- *Fact (2)*: In each split of an Inode $I$, we denote $I$ is split into an Inode set $\mathcal{I}$. $\mathcal{I}$ replaces $I$ and is maintained in $\mathcal{S}$ and/or $\mathcal{Q}$ (Lines 12 and/or 19 of `istabilize`).

The Inodes in the priority queue $\mathcal{Q}$ are maintained in topological order. The unstable ancestors of $I_{n_2}$ in $\mathcal{Q}$ due to insertions of $e_1, e_2, ...,$ or $e_m$ are stabilized, by induction hypothesis.

(i) Suppose $I_{n_2}$ has been split due to insertions of $e_1, e_2, ...,$ or $e_m$. By Fact (2), $I_{n_2}$ is replaced by $\mathcal{I}_{n_2}$ and maintained in

$\mathcal{S}$ and/or $\mathcal{Q}$. If some of the Inodes in $\mathcal{I}_{n_2}$ are not stabilized, they are detected and stabilized. By Lines 09-15 and 19, $\mathcal{I}_{n_2}$ and its descendants are stabilized.
(ii) Otherwise, suppose that $I_{n_2}$ is not split due to other insertions. $I_{n_2}$ and its descendants are stabilized as in `insert`, by Lemma 5.1.

The above two cases are exhaustive. Therefore, `istabilize` returns a stable bisimulation for $E_\delta$ = $m+1$. The hypothesis is true for $m+1$. Thus, by induction, `istabilize` of `insert_batch` returns a stable bisimulation for all $|E_\delta| \geq 1$.

By Theorem 4.2, the stabilized bisimulation returned by `istabilize` is minimized. Therefore, `insert_batch` returns the minimum bisimulation of $G$ after the insertions of $E_\delta$. □

## APPENDIX C
## COST MODEL FOR INCREMENTAL MAINTENANCE ALGORITHM

To further show the performance differences between incremental methods, we derive a model for the incremental maintenance algorithms over a sequence of $n$ insertions. For presentation simplicity, we use $M$'s to denote the *method*s being discussed. Specifically, we refer the maintenance of minimal bisimulation as $M_Y$ [21], our maintenance of minimum bisimulation as $M_{min}$ and the batch update version of $M_{min}$ as $M_B$. Moreover, our model does not consider recomputation per update as it is inefficient by far when compared to the mothods $M_Y$, $M_{min}$ and $M_B$.

We assume that the size of minimal bisimulation gradually deviates from that of minimum bisimulation, in practice. We assume the average query time differences between minimal and minimum bisimulations are $\Delta$'s. The query time on the initial minimum bisimulation is $t$. The query time for the minimum bisimulation after the $i$-th insertion $T(i)$ is $t_i$ + $\Sigma_{j=1}^{i}\ \Delta_j$. Based on the assumption that the average query times between minimal and minimum ones deviate modestly, we assume $\Delta_1 = \Delta$ and $\Delta_i = i \times \Delta$. Finally, we remark that if the minimal bisimulation deviated from the minimum one drastically because of some updates, the advantages on average query time of $M_{min}$ or $M_B$ would be more apparent than those of $M_Y$.

Denote the number of queries during the $n$ updates is $c \times n$, where $c$ can be viewed as the query rate relative to the update rate. Assume further the queries arrive evenly over the period of the $n$ updates for the ease of analysis. The average maintenance time for minimal bisimulation is $Y$ whereas that for minimum bisimulation is $I$.

The time for processing the $n$ insertions and $c \times n$ queries of $M_Y$ is: $n \times Y$ + $c \times \Sigma_{i=1}^{n} T(i)$. The time for the $M_{min}$ is: $n \times I$ + $c \times \Sigma_{i=1}^{n} t_i$. Hence, the following condition holds when $M_{min}$ is more efficient than $M_Y$:

$$n \times Y + c \times \Sigma_{i=1}^{n} T(i) \geq n \times I + c \times \Sigma_{i=1}^{n} t_i \tag{3}$$

By applying some simple arithmetics, we have

$$\frac{c}{n} \times \Sigma_{i=1}^{n} \Delta_i \geq I - Y$$

Finally, we have the following:

$$\frac{c \times (n-1)}{2} \times \Delta \geq I - Y \tag{4}$$

We may further generalize Formula (4) to the algorithm for batch insertions. Denote $B$ to be the size of batch updates. For analysis simplicity, we assume $n$ is divisible by $B$. Denote the time for batch updates is $I(B)$. Then, the time for processing the $n$ insertions and $c \times n$ queries with $M_B$ is:

$$n/B \times I(B) + (n - n/B) \times Y + c \times n/B \times \Sigma_{i=1}^{B} T(i) \tag{5}$$

Similarly, we can derive an inequality for $M_B$ to be more efficient than $M_Y$:

$$c \times B(n-B)/2 \times \Delta \geq I(B) - Y \tag{6}$$

**Discussions.** We can make two observations from the models of Formulas (4) and (6). Firstly, $M_{min}$ or $M_B$ is more efficient than $M_Y$ when the query performances gained from using the minimum bisimulation (LHS) are larger than the additional time needed to maintain the minimum bisimulation (RHS). Therefore, the faster the queries $c$ are, the more likely the inequalities of Formulas (4) and (6) are true. Secondly, $I(B)$ increases slightly with $B$. When $B$ is large (but smaller than $n/2$), Formula (6) is more likely to be true. $M_B$ is relatively more efficient. Thirdly, we remark that $M_Y$ requires reconstructions occasionally and the minimal bisimulation is not available during reconstructions. Such reconstructions are eliminated by using either $M_{min}$ or $M_B$.