Optimizing Incremental Maintenance of Minimal Bisimulation of Cyclic Graphs

Jintian Deng[1], Byron Choi[1], Jianliang Xu[1], and Sourav S. Bhowmick[2]

[1] Hong Kong Baptist University, China
{jtdeng,bchoi,xujl}@comp.hkbu.edu.hk
[2] Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Abstract. Graph-structured databases have numerous recent applications including the Semantic Web, biological databases and XML, among many others. In this paper, we study the maintenance problem of a popular structural index, namely *bisimulation*, of a possibly *cyclic* data graph. In comparison, previous work mainly focuses on acyclic graphs. In the context of database applications, it is natural to compute minimal bisimulation with merging algorithms. First, we propose a maintenance algorithm for a minimal bisimulation of a cyclic graph, in the style of merging. Second, to prune the computation on non-bisimilar `SCC`s, we propose a feature-based optimization. The features are designed to be constructed and used more efficiently than bisimulation minimization. Third, we conduct an experimental study that verifies the effectiveness and efficiency of our algorithm. Our features-based optimization pruned 50% (on average) unnecessary bisimulation computation. Our experiment verifies that we extend the current state-of-the-art with a capability on handling cyclic graphs in maintenance of minimal bisimulation.

1 Introduction

Graph-structured databases have a wide range of recent applications, *e.g.*, the Semantic Web, biological databases, `XML` and network topologies. To optimize the query evaluation in graph-structured databases, indexes have been proposed to summarize the paths of a data graph. In particular, many indexing techniques, *e.g.*, [3,4,11,17,19,23], have been derived from the notion of *bisimulation* equivalence. In addition to indexing, bisimulation has been adopted for selectivity estimation [14,20] and schemas for semi-structured data [2].

To illustrate the applications of bisimulation in graph-structured databases, we present a simplified sketch of a popular graph used in `XML` research, shown in the left hand side of Figure 1, namely `XMark`. `XMark` is a synthetic 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. The references are encoded by `IDREFs` and represented by the dotted arrows in the figure. Two nodes in a data graph are bisimilar if they

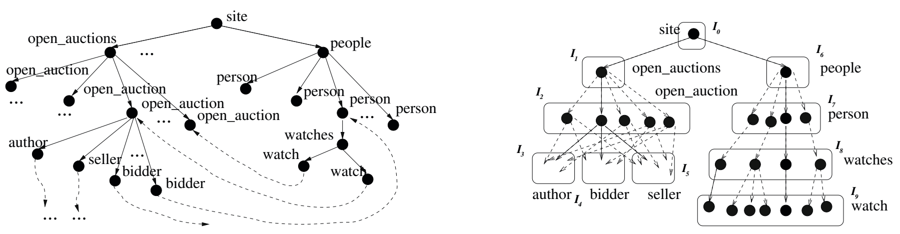


Fig. 1. Illustration – A sketch of XMark and its bisimulation

have the same set of incoming paths. A sketch of the bisimulation graph of XMark is shown in the right hand side of Figure 1. In the bisimulation graph, bisimilar data nodes are placed in a partition, denoted as I_i. Consider a query q /site//open_auction//seller that selects all sellers of open_auctions. We can evaluate q on the partitions and simply retrieve the data nodes in I_3. Therefore, it is crucial to minimize the bisimulation graph for efficient index to reduce I/O.

In practice, many data graphs are cyclic (*e.g.*, [1]) and subject to updates. Therefore, different from other applications of bisimulation, its maintenance problem is much more important in database applications [12,21]. Furthermore, previous work [12,21] on maintenance of bisimulation of graphs mainly focuses on *directed acyclic* graphs. In contrast, this paper focuses on the maintenance problem of bisimulation of possibly cyclic graphs.

In this paper, we take the first step to systematically and comprehensively investigate *incremental maintenance of minimal bisimulation of cyclic graphs*. There are two key challenges in the maintenance problem. Firstly, merging algorithms for bisimulation as opposed to partition refinement are more natural for incremental maintenance of bisimulation. However, it is known [12] that merging algorithms fail to determine the minimum bisimulation of cyclic graphs. The main reason is that nodes of SCCs must be considered *together*, which is not the case in merging algorithms.

The first contribution is a maintenance algorithm for minimal bisimulation of cyclic graphs (Section 5), in the style of merging algorithms. Our algorithm consists of a split and a merge phase. In the split phase, we split and mark the index updated nodes (*i.e.*, the equivalence partitions) into a correct but non-minimal bisimulation. In the merge phase, we apply a (partial) bisimulation minimization algorithm on the marked index nodes. Our algorithm has an explicit handling of bisimulation between SCCs, when compared to previous work. As such, our algorithm *always* produces smaller (if not the same) bisimulation graphs when compared to previous work. In case of acyclic graphs, our algorithm and the previous work will produce the same bisimulation.

The second contribution is on a feature-based optimization for determining bisimulation between two SCCs (Section 6). On one hand, the computation of bisimulation between two SCCs can be costly. On the other hand, there may not be many bisimilar SCCs, in practice. We aim at deriving structural features from SCCs such that two SCCs are bisimilar *only if* they have the same or bisimilar features. Specifically, we explore label- or edge-based, path-based, tree-based and circuit-based features. With these, the merging algorithm has a more global information of SCCs and may prune computation on non-bisimilar SCCs early.

Third, we conduct an experimental study that verifies the effectiveness and efficiency of our algorithm. In particular, our feature-based optimization prunes an average of 50% unnecessary bisimulation computation. The results validate the practical feasibility of extending the current state-of-the-art with a capability of maintaining minimal bisimulation on cyclic graphs.

2 Related Work

Previous work on maintaining bisimulation can be categorized into two: *merging* and *partition-refinement* algorithms. There have been two previous merging algorithms [12,21] for incremental maintenance of bisimulation of cyclic graphs. The algorithm proposed in [12] contains a split and a merge phase. Upon an update on the data graph, the bisimulation graph is split to a correct but non-minimal bisimulation of the updated graph. Next, the bisimulation graph is minimized in the merge phase. For acyclic graphs, [12] produces the minimum bisimulation of the updated graph. If the graph is cyclic, [12] returns a minimal bisimulation only. Thus, to support cyclic graphs, the minimum bisimulation is occasionally re-computed from scratch. [21] proposes a split-merge-split algorithm with a rank flag for SCCs, which is originally proposed in [6]. [21] also returns a minimal bisimulation in response to an update of a cyclic graph. However, there is neither experimental evaluation [21] nor implementation for us to perform comparisons. A difference between our work and the previous work is that we introduce explicit handling of SCCs and propose features to optimize bisimulation maintenance.

A recent partition-refinement algorithm [10] can be considered as a variant of Paige and Tarjan's algorithm [18] – a construction algorithm for the minimum bisimulation. The algorithm proposes its own split to handle edge changes. It has been extended to support maintenance of k-bisimulation. Their experiment shows that [10] produces a bisimulation that is always within 5% of the minimum bisimulation. It is shown, through a later experiment, that [12] may produce even smaller bisimulations, which we compared via experiments in Section 7.

Bisimulation (relation) [16] has its root at symbolic model checking, state transition systems and concurrency theories. In a nutshell, two state transition systems are bisimilar if and only if they *behave* the same from an observer's point of view. Bisimulation minimization has been extensively studied through experiments in [7], in the context of modeling checking. A conclusion of [7] is that minimization may not be worthwhile for model checking as it may easily be more costly than checking invariance properties of systems. In comparison, when bisimulation is used as an index structure for query processing, bisimulation minimization and therefore its maintenance are far more important.

3 Background

This section presents the background and the notations of this paper.

Definition 3.1. *A* graph-structured database *(or* 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 $\rho : V \rightarrow \Sigma$ is a function that maps a vertex to a label, and Σ is a finite set of labels.*

For clarity, we may often denote a data graph as $G(V, E)$ when r, ρ and Σ are irrelevant to our discussions. Since our work focuses on cyclic graphs, we recall some relevant definitions below.

Cyclic graphs. A *strongly connected component* (SCC) in a graph $G(V, E)$ is a subgraph $G'(V', E')$ whose nodes are a subset of nodes $V' \subseteq V$ where the nodes in V' can reach each other. The SCCs of a graph can be determined by classical graph contraction algorithms, *e.g.*, Gabow's algorithm, in $O(|V|+|E|)$, where each SCC is reduced to a supernode. The resulting graph is a *directed acyclic graph* DAG, which is often called the *reduced graph.* In subsequent discussions, we use SCCs to refer to non-trivial SCCs (SCCs with more than one node) only. In the definition below, we highlight two special kinds of nodes in SCCs, namely, exit and entry nodes.

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 $n \in V'$ and $n_1 \notin V'$. Similarly, n is an* entry *node if there exists an edge (n_0, n) where $n_0 \notin V'$ and $n \in V'$.*

Bisimulation. Next, we recall the relevant definitions of bisimulation.

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

$$\begin{aligned}&\forall\, v_1 \in V_1, v_2 \in V_2 \,.\, v_1 \sim v_2 \rightarrow\\ &\quad \forall\, (v_1', v_1) \in E_1\ \exists\ (v_2', v_2) \in E_2 \,.\, v_1' \sim v_2' \wedge \rho_1(v_1') = \rho_2(v_2') \wedge\\ &\quad \forall\, (v_2'', v_2) \in E_2\ \exists\ (v_1'', v_1) \in E_1 \,.\, v_1'' \sim v_2'' \wedge \rho_1(v_1'') = \rho_2(v_2'').\end{aligned}$$

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

Examples of bisimilar nodes can be found in Figure 1, where the bisimilar nodes are placed in the same rounded rectangle. Definition 3.3 presents upward bisimulation in the sense that two nodes can be bisimilar only if their parents are bisimilar. The definition can be paraphrased in terms of paths, which is often convenient to simplify our discussions[1] .

Proposition 3.1: *Two nodes are upward bisimilar if and only if the incoming path set of the two nodes are the same.* □

A set of bisimilar nodes is often referred to as an *equivalence partition* of nodes, or simply partitions. Hence, a bisimulation of a graph can be described as a partition graph. In the context of indexing, the partitions are sometimes referred

[1] We should remark that there have been other notions of bisimulation, such as downward bisimulation and k-bisimulation, that have been applied in indexing/selectivity estimation but have not been the focus of this paper. Our techniques can be extended to support them with minor modifications.

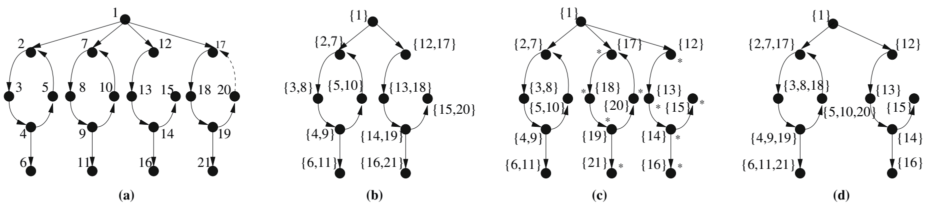


Fig. 2. (a) A cyclic data graph; (b) the minimal bisimulation graph; (c) the split bisimulation graph; and (d) an updated minimal bisimulation graph

to as *index nodes*, or simply *Inodes*, whereas the nodes of the data graph are referred to as *data nodes*, or simply *nodes*.

In this work, we consider the notion of bisimulation minimality defined in Definition 3.5. First, we recall the notion of *stability*.

Definition 3.4. *Given two partitions of nodes X and I, X is* stable *with respect to I if either (i) X is contained in the children of the nodes in the partition I or (ii) X and the children of the nodes in I are disjoint.*

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

Definition 3.6. *A bisimulation B of a graph G is the* minimum *bisimulation if B contains the minimum number of partitions, among all bisimulations of G. According to [12], the minimum bisimulation of a graph is unique.*

Bisimulation minimization. Next, we illustrate the intuitions of merging algorithm for bisimulation minimization with a brief example shown in Figure 2. Assume the nodes of the data graph shown in Figure 2(a) have the same label. The node id is shown next to each node. We use {} to denote an Inode. A merging algorithm initially places each node in a single partition. Assume that the algorithm merges pairs of partitions top-down, which attempts to merge Nodes 2 and 7. However, the algorithm has not yet determined Nodes 5 and 10. Hence, the algorithm terminates and fails to return the minimum bisimulation shown in Figure 2(b), unless it memorizes the SCCs containing Nodes 2 and 7 together.

4 Bisimulation of Cyclic Graphs

This section presents a minimization algorithm for bisimulation of cyclic graphs, shown in Figure 3, which is a component of the maintenance algorithm. We focus on the logic of handling SCCs during the minimization.

The algorithm can be divided into two parts. First, Lines 01-06, if n_1 and n_2 are not both in some SCCs, we compute bisimulation between n_1 and n_2 in the style of any merging algorithm. We assume the existence of a procedure `next_nodes_top_order`(G) of a node n which returns the next n's child in topological order in G. Then, we recursively invoke `bisimilar_cyclic`.

```
Procedure bisimilar_cyclic
Input: Nodes $n_1$ and $n_2$ where $\rho(n_1) = \rho(n_2)$; $B$, the current bisimulation
Output: An updated bisimulation relation $B'$

01 if $n_1$ and $n_2$ are not both in some SCC
02     if $\forall p_1 \in n_1$.parent $\exists p_2 \in n_2$.parent s.t. $p_1 \sim p_2$ then
03         add $(n_1, n_2)$ to $B$
04         for all $c_1$ in $n_1$.next_nodes_top_order($G_1$)
05             for all $c_2$ in $n_2$.next_nodes_top_order($G_2$)
06                 $B$ = bisimilar_cyclic($c_1$, $c_2$, $B$)

07 else                /* check bisimulation of the two SCCs */
08     assume $n_1$ and $n_2$ are in SCCs $S_1$ and $S_2$, respectively
       if feature_pruning($S_1$, $S_2$) return $B$                /* Sec. 6*/

09     clone $S_1$ to $S'_1$;          create an artificial node $n'_1$ for $n_1$
10     for all $(n, n_1) \in S'_1.E$
11         replace $(n, n_1)$ with $(n, n'_1) \in S'_1$
12     clone $S_2$ to $S'_2$;          create an artificial node $n'_2$ for $n_2$
13     for all $(n, n_2) \in S'_2.E$
14         replace $(n, n_2)$ with $(n, n'_2) \in S'_2$

15     clone $B$ to $B'$;     add $(n_1, n_2)$ to $B'$              /* assume $n_1 \sim n_2$ */

16     for all $c_1$ in $n_1$.next_nodes_top_order($S'_1$)
17         for all $c_2$ in $n_2$.next_nodes_top_order($S'_2$)
18             $B'$ = bisimilar_cyclic($c_1$, $c_2$, $B'$)
19     if $(n'_1, n'_2)$ in $B'$ then $B = B \cup B'$              /* $S_1 \sim S_2$ */
20 return $B$
```

Fig. 3. Bisimulation minimization of cyclic graphs

Second, if both n_1 and n_2 are in some SCCs, Lines 07-20 check if S_1 *and* S_2, as opposed to simply n_1 and n_2, can be bisimilar. We prune non-bisimilar SCCs by using the feature-based optimization presented in Section 6, in Line 08. For presentation clarity, we assume that n_1 and n_2 are in two different SCCs. Then, we break the SCCs and check bisimulation recursively, in Lines 09-15. The main idea is illustrated with Figure 4. Specifically, we redirect the incoming edges of n_1 in SCC (Lines 09-11) to an artificial node n'_1. Similarly, we redirect the incoming edges of n_2 to n'_2 (Lines 12-14). We clone the current bisimulation relation determined thus far (Line 15). Assuming that n_1 and n_2 are bisimilar, we check the possible bisimulation between the children of n_1 and n_2 by calling `bisimilar_cyclic` recursively (Lines 16-18). If we can construct a possible bisimulation between n'_1 and n'_2 (Line 19), then S_1 and S_2 are bisimilar.

The main idea of `bisimilar_cyclic` on handling SCCs is that `bisimilar_cyclic` explicitly breaks a cycle, whereas previous work *overlooks* cycles. `bisimilar_cyclic` may be recursively called due to nested SCCs (Line 18). Without breaking a cycle in each call, `bisimilar_cyclic` may not terminate and the feature-based optimization (Line 07) may always derive features of the "topmost" SCC.

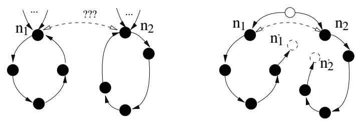


Fig. 4. Breaking one cycle in an `SCC` in `bisimilar_cyclic`

Analysis. For presentation clarity, `bisimilar_cyclic` did not incorporate with classical indexing techniques. `bisimilar_cyclic` runs in $O(|E|^2)$ due to the for loops at Lines 04-06 and Lines 16-18, assuming that `feature_pruning` can be performed more efficiently than $O(|E^2|)$.

5 Maintenance of Bisimulation

In this section, we present the overall maintenance algorithm. For simplicity, we present an edge insertion algorithm `insert` in Figure 5. Edge deletions are discussed at the end of this section. Our algorithm consists of a split phase and a merge phase. In the following, we focus on the split phase, as the merge phase is essentially `bisimilar_cyclic`.

The split phase. The split phase is presented in Lines 05-20. We maintain two variables to record two kinds of nodes that are needed to be split. More specifically, we use $\mathcal{S}$ to record the nodes of *SCCs* needed to be split and $\mathcal{Q}$ to record the *nodes* that are not in any `SCC`s but needed to be split. In the split phase, we mark the affected Inodes, which will be examined in the merge phase.

Suppose the insertion makes the Inode of n_2 unstable. To initialize $\mathcal{S}$ (Line 03), we set $\mathcal{S}$ to the Inode of n_2 and n_2, *i.e.*, $\{(I_{n_2}, n_2)\}$, if n_2 is in an `SCC`. Otherwise, $\mathcal{S}$ is empty. Similarly, we initialize $\mathcal{Q}$ to I_{n_2} if n_2 is not in any `SCC` and $\mathcal{Q}$ is empty otherwise (Line 04). Next, we split the Inodes in $\mathcal{S}$ and $\mathcal{Q}$ recursively until they are empty (Line 05).

(1) We process the nodes in $\mathcal{S}$ as follows (Lines 06-12): We select a node n from $\mathcal{S}$ and retrieve its Inode I_n. We split n from I_n as the `SCC` of n is potentially non-bisimilar to the `SCC` of other nodes in I_n (Line 09). We mark the split Inodes so that they will be checked in the merge phase (Line 10). In Lines 11-12, we insert the children of the split Inode to $\mathcal{S}$ and $\mathcal{Q}$, similar to Lines 03-04.

(2) The handling of $\mathcal{Q}$ is shown in Lines 13-20. We select an Inode I_n from $\mathcal{Q}$ (Line 14). If I_n is not stable, we split I_n into a set of stable Inodes $\mathcal{I}$, as in the pervious work [12] for acyclic graphs (Lines 15-16). We mark Inodes in $\mathcal{I}$ in Line 18. In Lines 19-20, we update the affected nodes $\mathcal{S}$ and $\mathcal{Q}$, similar to Lines 03-04.

The split phase essentially traverses the bisimulation graph B and `SCC`s in the data graph to spilt and collect the Inodes that are affected by the update. `SCC`s themselves may be affected by an update. In Line 21, we call Gabow's algorithm to update `SCC` information of a graph, which is needed in the merge phase.

The merge phase. The merge phase can be done by applying the minimization algorithm presented in Section 4 (Figure 3). An optimization is that we apply merging on only the Inodes that are marked in the split phase.

```
Procedure insert
Input: an insertion of an edge ($n_1$, $n_2$) to a graph $G$; its minimal bisimulation $B$
Output: An updated graph $G'$ and its updated minimal bisimulation $B'$
01 $G'$ = insert ($n_1$, $n_2$) into $G$
02 if $n_2$ is new
     then create a new Inode $I_{n_2}$; insert $I_{n_2}$ into $B$; mark $I_{n_2}$
     else if $I_{n_2}$ is not stable
03         $\mathcal{S} = \{(I_{n_2}, n_2) \mid n_2$ is in an SCC$\}$
04         $\mathcal{Q} = \{I_{n_2} \mid n_2$ is not in any SCC$\}$
05 while $\mathcal{Q} \neq \emptyset$ or $\mathcal{S} \neq \emptyset$
06   if $\mathcal{S} \neq \emptyset$ then                    /* split the relevant SCC */
07     pick a node ($I_n$, $n$) from $\mathcal{S}$; remove ($I_n$, $n$) from $\mathcal{S}$
08     while $I_n$ is not stable or a singleton
09       split $I_n$ into $I_1 = I_n$ - $\{n\}$ and $I_2 = \{n\}$
10       mark $I_1$ and $I_2$
11       $\mathcal{S} = \mathcal{S} \cup \{(I_{n_s}, n_s) \mid n_s$ is $n_i$'s child, $n_i \in I_2$ and $n_s$ in the SCC of $n\}$
12       $\mathcal{Q} = \mathcal{Q} \cup \{I_{n_q} \mid n_q$ is a child of $n_i$, $n_i \in I_2$ and $n_q$ not in any SCCs$\}$
13   if $\mathcal{Q} \neq \emptyset$ then                    /* split nodes not related to SCCs */
14     pick a node $I_n \in \mathcal{Q}$; remove $I_n$ from $\mathcal{Q}$
15     if $I_n$ is not stable or a singleton
16       split $I_n$ into a stable set $\mathcal{I}$                        /* [12] */
17       for each $I$ in $\mathcal{I}$
18         mark $I$
19         $\mathcal{S} = \mathcal{S} \cup \{(I_{n_s}, n_s) \mid n_s$ is $n_i$'s child, $n_i \in I$ and $n_s$ in the SCC of $n\}$
20         $\mathcal{Q} = \mathcal{Q} \cup \{ I_{n_q} \mid n_q \in$ child of $n_i$, $n_i \in I$ and $n_q$ not in any SCCs$\}$
21 Gabow($G'$)                              /* update the SCC information in $G'$ */
22 ($G'$, $B'$) = bisimilar_cyclic_marked($G$, $B$) /* merging the marked Inodes */
23 return ($G'$, $B'$)
```

Fig. 5. Insertion for minimal bisimulation of cyclic graphs

Example 5.1. We illustrate Algorithm insert with an example. Reconsider the cyclic data graph that is shown in Figure 2(a). Its minimal bisimulation is shown in Figure 2(b). Assume that we insert an edge (20,17) into the data graph. Algorithm insert initially puts {12,17} into $\mathcal{Q}$ (Line 04). Then, in Line 16, Node 17 is split from {12,17}. The split Inodes are marked, with a "*" sign in the figure. The split phase proceeds recursively and finally produces the graph in Figure 2(c). Then, we update the SCC information of the data graph. By bisimilar_cyclic_marked, we obtain the bisimulation at Figure 2(d).

While the previous work [12] produces the same split graph (Figure 2(c)), it returns the bisimulation in Figure 2(c), due to the lack of the handling on SCCs. Subsequently, any subgraphs that are connected to the SCC (Nodes 17-20), *e.g.*, Node 21, are not merged, as the SCCs are not merged.

Analysis. The recursive procedure in Lines 05-20 traverses the graph $O(|E|)$. With optimization in [18], stablizing a set can be done in $O(log(|V|))$. Hence.

the split phase runs in $O(|E|log(|V|)$. Gabow's algorithm in Line 21 runs in $O(|V| + |E|)$. The merge phase with optimization runs in $O(|E|^2)$. Thus, the overall runtime of Algorithm `insert` is $O(|E|^2)$.

Edge deletions. While our discussions focused on insertions, our technique can be generalized to support edge deletions with the following modifications. (i) In Line 01, we delete the edge from the data graph. (ii) If n_2 is connected after the deletion, we check the stability of I_{n_2} in Line 02, initialize $\mathcal{S}$ and $\mathcal{Q}$ and then invoke the split phase as before.

6 Feature-Based Optimization

The maintenance algorithm presented in Section 5 involves splitting the updated bisimulation into a non-minimal bisimulation followed by bisimulation minimization. As discussed, determining if two SCCs are bisimilar can be computationally costly, $O(|E|^2)$. In practice, SCCs may often be non-bisimilar. This motivates us to optimize the minimization of cyclic graphs by proposing features to prune computations on non-bisimilar SCCs. The main idea is to derive features of SCCs such that two SCCs can be bisimilar *only if* their features are the same or bisimilar. Ideally, the features are discriminative enough and can be efficiently constructed and used. Furthermore, features may be efficiently maintainable so that they are constructed once and maintained with the bisimulation.

6.1 Properties of Bisimulation of Cyclic Graphs

Prior to the discussions on features, we list some properties of bisimulation of cyclic graphs. These properties show that a number of classic properties of graphs are not suitable for our feature-based optimization. Due to space constraints, we omitted the proofs, which are established by simple proof by contradictions [5].

Property 1. Two SCCs with the same cycle height may not be bisimilar. Two SCCs with different cycle heights can be bisimilar.

Property 2. Two SCCs with the same number of simple cycles may not be bisimilar. Two bisimilar SCCs may have different number of simple cycles.

Property 3. Two SCCs with different numbers of entry nodes can be bisimilar.

The design of features exploits the following proposition on bisimulation of SCCs. The intuition is that as long as we find a node in a SCC that is not bisimilar to any node in another SCC, the two SCCs will not be bisimilar.

Proposition 6.2: *An* SCC $G_1(V_1, E_1)$ *is not bisimilar to another* SCC $G_2(V_2, E_2)$ *if and only if there is a node v in V_1 such that it is not bisimilar to any node in V_2.* □

6.2 Features of SCCs

Merging algorithms for bisimulation minimization are iterative in nature. The current merging step of a SCC may not have sufficient information for determining bisimulation between SCCs. Hence, we propose some features that give merging algorithms some "lookahead" of SCCs to check Proposition 6.2.

1. Label-based or edge-based features. We begin with the label-based and edge-based features, which are straightforward, and have many alterative implementations. For example, we may use all label and edge types that appeared in an SCC as an SCC feature. Two bisimilar graphs must contain the same type of labels and edges. In our experiments, we found that the incoming label or edge sets of an entry node are relatively concise and effective in distinguishing non-bisimilar SCCs. For example, in Figure 1, the incoming label set of the entry node `open_auction` is {`open_auction`, `watch`} and that of the entry node `watches` is {`person`, `bidder`}. The construction and maintenance of such labels can be efficiently supported by hashtables.

2. Path-based features. Regarding path-based features, one may be tempted to use all simple paths in an SCC. However, determining all simple paths of a cyclic graph is a problem in PSPACE [15].

Proposition 6.3: *Two SCCs are bisimilar only if they have the same set of simple path(s) from their entry node(s).* □

Next, the longest paths of a cyclic graph are not appropriate for our problem either, as they cannot be determined in PTIME.

In this work, we propose to use the set of incoming paths with a length at most k (or simply k-paths) as a feature of the entry nodes, where k is a user parameter. The value of k may be increased when maintenance of bisimulation spends substantial time on bisimulation computation. From Proposition 3.1, two bisimilar graphs must have the same set of k-paths. Contrarily, two graphs with different sets of k-paths are non-bisimilar. Hence, k-paths can be used as a feature. It is straightforward that k-paths can be efficiently constructed. However, since k-paths is local, k-paths may not contain a node that is not bisimilar to any nodes in any other SCCs. Another simple remark is that a node in an SCC may appear in a k-path set multiple times.

3. Feature of canonical spanning tree. We further explore more complex structural features of SCCs. First, we define the weight used in determining the canonical spanning tree. The *weight* of an edge (n_1, n_2) is *directly proportional* to the count of $(\rho(n_1), \rho(n_2))$-edges in the graph. We exploit a popular trick to perturb the edge's weight such that each kind of edges has a unique weight.

Given the weight defined above, we can compute a minimum spanning tree, in the style of a greedy breath first traversal in $O(|V|+|E|)$. As the weight is defined to be directly proportional to the edge count, a minimum spanning contains more infrequent edge kinds of a graph. However, minimum spanning trees of a *directed* graph are often difficult to maintain. In comparison, maintenance of spanning trees of an undirected graph is much simpler, *e.g.*, in amortized time $O(|V|^{1/3}log(|V|))$ [9]. Hence, we perform some simple tricks on the data graph

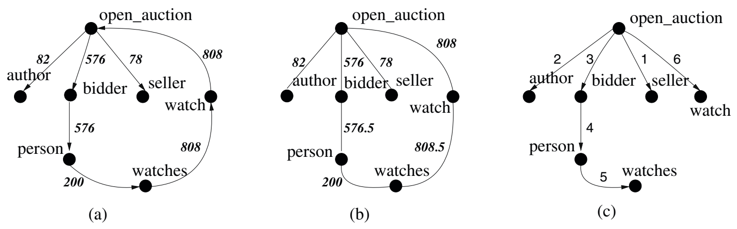


Fig. 6. The construction of the canonical spanning tree from a simplified `open_auction`

when constructing the spanning tree. First, we ignore the direction of the edges. Second, we adopt Prim's algorithm to construct the minimum spanning tree of the undirected graph. From the root of the minimum spanning tree, we derive the edge direction, which gives us the *canonical spanning tree.* Note that the edge direction is simply needed for checking bisimulation between canonical spanning trees and the direction of the edges in the canonical spanning tree may differ from that of the edges in the original graph.

Proposition 6.4: *Two* SCC*s are bisimilar only if their minimum canonical spanning trees returned by Prim's algorithm are bisimilar.* □

It should be remarked that SCCs are often nested. In the worst case, the total size of the spanning trees of all possible entry nodes of an SCC is $O((|V| + |E|)^2)$. In addition, computing bisimulation between large canonical spanning trees can be costly. Therefore, we introduce a termination condition to the Prim's algorithm – we do not expand the spanning tree further from a node n when there is an ancestor of n having the same label as n. The total size of the canonical spanning trees is then $O(|V| \times |E|)$.

Example 6.2. We illustrate the construction of a canonical spanning discussed above with an example shown in Figure 6. Figure 6(a) shows a simplified SCC of `open_auction` from `XMark` with a scaling factor 0.1. The count of each edge type is shown on the edge. We perturb the weight to make each weight in the SCC unique. We ignore the direction of the edges, shown in Figure 6(b). Then, it is straightforward to compute the spanning tree (shown in Figure 6(c), where the number on an edge shows the order of the edge returned by Prim's algorithm). Finally, the direction of the edges are derived from the root of the spanning tree `open_auction`.

4. Circuit-based features. While the time for checking bisimulation between minimum spanning trees is very close to that between SCCs, one may be tempted to explore structural features further. Here, we illustrate that complicated structural features can lead to inefficient maintenance. For example, circuit bases contain much more structural information than spanning trees. It has been shown that the minimum circuit bases of directed graphs is unique [8]. However, determining the circuit bases is $O(|V|^3)$. It is therefore more efficient to simply compute the bisimulation of two SCCs than using the feature of circuit bases.

Proposition 6.5: *Two* SCC*s are bisimilar if their circuit bases are bisimilar.* □

6.3 Offline versus Online Feature Construction

Since the proposed features can be constructed relatively efficiently, they may be constructed and used during bisimulation computation, *i.e.*, runtime. Then, during runtime, we may incorporate the features with not only the labels but also the partial bisimulation constructed so far. Specifically, some nodes in SCCs have been associated with Inode. The ids of Inodes together with the labels, as opposed to the labels alone, are used in online feature construction.

In comparison, the features may be built offline and maintained with each update of the graph. However, given a cyclic graph, we may determine features for each entry node, in the worst case, to build all possible features offline. However, this size requirement may sometimes be prohibitive, in practice.

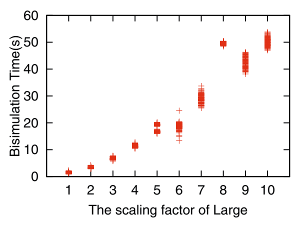
7 Experimental Evaluation

This section presents an experimental study that verifies the efficiency of our algorithms. Our implementation is written in JDK building on top of Ke *et al.* [12]. It is available at http://code.google.com/p/minimal-bisimulation-cyclic-graphs/. The experiments were run on a laptop computer with a dual CPU at 2.0 GHz and 2GB RAM running Ubuntu hardy.

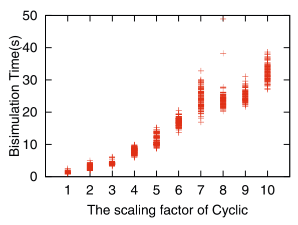
Datasets. We used both synthetic and real-life graph data to test various aspects of our algorithms. (i) XMark is a synthetic XML dataset provided by the XMark Benchmark Projects [22]. The cycles in XMark is essentially composed by IDREFs of open_auction to person and vice versa. We ran Gabow's algorithm on XMark. We note that there are few very large SCCs. It is easy to verify that very few, or none, of the SCCs are bisimilar. Hence, we randomly decompose SCCs into smaller SCCs as follows: We define a parameter s to set the average number of open_auction nodes and another parameter r to define the ratio between open_auction and person nodes in an SCC. For example, when s and r are set to 10 and 1.2, respectively, an SCC contains approximately 10 open_auctions and 12 persons. In our experiment, the dataset generated directly from XMark is referred to Large and the decomposed Large is refered to Cyclic.

In the experiment on Algorithm insert, we generated a dataset Base to test the performance difference between insert and Ke *et al.* The performance difference can hardly be shown because Large only contains few large SCCs and Cyclic contains numerous random non-bisimilar SCCs. Therefore, we constructed Base by connecting two XMark graphs with the same scaling factor (s.f.) and removing a number of edges from the graph. When the edges are inserted by Algorithm insert, the bisimilar SCCs will be recovered and merged.

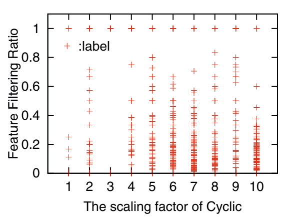
We tested insert over real-life data Cite. Cite is a citation graph extracted from papers on high energy physics [13]. It covers those papers in the period from Jan. 1993 to Apr. 2003 and contains 35k papers in total. Cite represents each paper as a data node and a citation in paper i to paper j as an edge. We removed self-citing edges, for simplicity. Cite is highly cyclic. Similar to Cyclic, we removed citation edges randomly and used them for insertions.



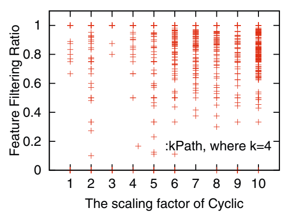
(a) `Large` w/o features



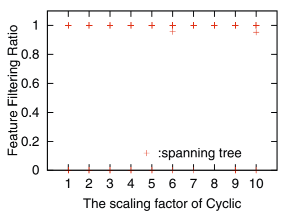
(b) `Cyclic` w/o features



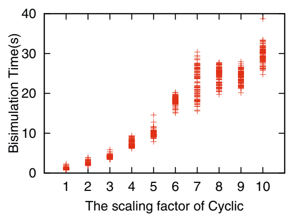
(c) Label-based pruning



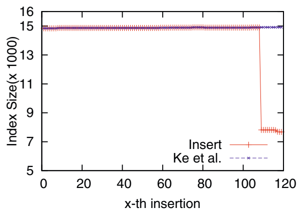
(d) k-path pruning



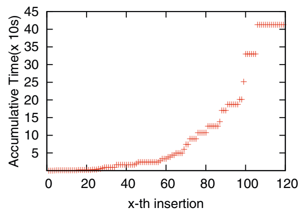
(e) Tree-based pruning



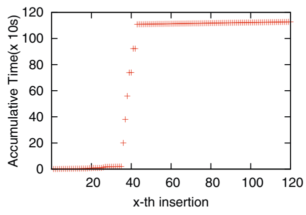
(f) `Cyclic` w. features



(g) Bisim. size of `Base`



(h) Insertion time (`Base`)



(i) Insertion time (`Cite`)

Fig. 7. Performance results on `bisimilar_cyclic` with and without feature optimization on `Large` and `Cyclic` and `insert` performance on `Base` and `Cite`

Performance analysis. To test the runtime of `bisimilar_cyclic` with feature-based optimization, we ran 100 random `Large` and `Cyclic` for each s.f. ranging from 0.01 to 0.1 (*i.e.*, 17k nodes to 168k nodes). Figures 7(a) and 7(b) show that the runtimes are roughly linear to s.f.. At the same s.f., the runtimes for `Large` are longer than those for `Cyclic`. The reason is that `Cyclic` contains are more smaller random `SCC`s, which are often non-bisimilar, and `bisimilar_cyclic` identifies them relatively earlier. In comparison, `bisimilar_cyclic` in `Large` may spend more time in checking sub-`SCC`s inside a large `SCC`.

Next, we verified the effectiveness of the features by using each feature on 100 `Cyclic` graphs for each s.f.. The features were *computed in runtime* and k in the path-based feature is 4. We skipped the edge-based feature as its performance is similar to the label-based feature, in `Cyclic`. The pruning of each feature is plotted in Figures 7(c), 7(d) and 7(e). The y-axis is the percentage of pruned non-bisimilar SCCs. In all, the label-based, path-based and canonical-tree feature pruned (on average) 14%, 62% and 73%, respectively. Figure 7(f) shows the runtime of `bisimilar_cyclic` with all feature optimization. On average, it is 4% faster than the one without optimization (Figure 7(b)). We remark that on average, 7.7% of the runtime was due to *online* feature construction.

Lastly, we conducted an experiment on Algorithm `insert` over `Base` and `Cite`. The results are shown in Figures 7(g),(h) and (i). Figure 7(g) shows the size of the minimal bisimulation produced by `insert` and Ke *et al.* [12]. We did not show the minimum bisimulation as `insert` always produces a bisimulation that is within 2% of the minimum. Initially, both `insert` and [12] are very close. After some number of insertions, the two bisimilar SCCs in the `Base` were recovered. We ran this experiment multiple times and found that the drop occurs randomly between 100th and 120th insertion. As illustrated in Figure 7(g), the difference in the size of bisimulation returned by `insert` and [12] depends on the number and the size of bisimilar SCCs in a graph. In this particular graph, `insert` returns a bisimulation graph that is 100% smaller than that by [12].

The accumulative runtime of `insert` over `Base` is shown in Figure 7(h). The accumulative runtime increases as we insert more edges into `Base`. After some insertions, `insert` ran slower because the two SCCs in `Base` became similar. `bisimilar_cyclic` checked many nodes before it declared the SCCs were not bisimilar. The runtime of [12] is close to 0s as it does not process SCCs, as the minimal bisimulation remains the same.

The accumulative runtime of `insert` over `Cite` is shown in Figure 7(i). As expected, the runtime increases as more edges are inserted. Between the 30th and 40th insertion, the largest SCC in `Cite` was involved and `insert` ran slower. In most of the cases, the runtime of `insert` is close to 0s when it did not process the SCCs. The average runtime for one insertion is around 10s. However, there is no bisimilar SCCs in `Cite` and `insert` and [12] returned the same bisimulation.

8 Conclusions

In this paper, we studied the optimization in maintaining the minimal bisimulation of cyclic graphs. Our first contribution is a bisimulation minimization algorithm that *explicitly* handles SCCs and a maintenance algorithm for minimal bisimulation of cyclic graphs. Second, we propose a feature-based optimization to avoid computing non-bisimilar SCCs. Third, we presented an experiment to verify the effectiveness and efficiency of our algorithms. Our experimental results show that the features can prune unnecessary bisimulation computation and our maintenance algorithm can return smaller bisimulation graphs than previous work, depending on the size and number of bisimilar SCCs in the data graph. As for future work, we plan to refine the selection of discriminative features to further reduce the maintenance time. We are studying on maintenance algorithms that can produce either the minimum bisimulation or one whose size is bounded by a theoretical guarantee.

References

1. Batagelj, V., Mrvar, A.: Pajek datasets, `http://vlado.fmf.uni-lj.si/pub/networks/data/`
2. Buneman, P., Davidson, S.B., Fernandez, M.F., Suciu, D.: Adding structure to unstructured data. In: Afrati, F.N., Kolaitis, P.G. (eds.) ICDT 1997. LNCS, vol. 1186, Springer, Heidelberg (1996)
3. Buneman, P., Grohe, M., Koch, C.: Path queries on compressed XML. In: VLDB (2003)
4. Chen, Q., Lim, A., Ong, K.W.: D(k)-index: an adaptive structural summary for graph-structured data. In: SIGMOD (2003)
5. Deng, J., Choi, B., Xu, J., Bhowmick, S.S.: Optimizing incremental maintenance of minimal bisimulation of cyclic graphs. Technical report, HKBU (2010), `http://www.comp.hkbu.edu/~jtdeng/techreport.pdf`
6. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theor. Comput. Sci. 311(1-3), 221–256 (2004)
7. Fisler, K., Vardi, M.Y.: Bisimulation minimization and symbolic model checking. Form. Methods Syst. Des. 21(1), 39–78 (2002)
8. Gleiss, P.M., Leydold, J., Stadler, P.F.: Circuit bases of strongly connected digraphs. Working Papers 01-10-056, Santa Fe Institute (2001), `http://ideas.repec.org/p/wop/safiwp/01-10-056.html`
9. Henzinger, M.R., King, V.: Maintaining minimum spanning trees in dynamic graphs. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256. Springer, Heidelberg (1997)
10. Kaushik, R., Bohannon, P., Naughton, J.F., Shenoy, P.: Updates for structure indexes. In: VLDB (2002)
11. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: ICDE (2002)
12. Ke, Y., Hao, H., Ioana, S., Jun, Y.: Incremental maintenance of XML structural indexes. In: SIGMOD (2004)
13. Leskovec, J.: Stanford large network dataset collection, `http://snap.stanford.edu/data`
14. Li, H., Lee, M.L., Hsu, W., Cong, G.: An estimation system for XPath expressions. In: ICDE (2006)
15. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. In: VLDB (1989)
16. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
17. Milo, T., Suciu, D.: Index structures for path expressions. In: Beeri, C., Bruneman, P. (eds.) ICDT 1999. LNCS, vol. 1540, pp. 277–295. Springer, Heidelberg (1999)
18. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
19. Polyzotis, N., Garofalakis, M.: XCluster synopses for structured XML content. In: ICDE (2006)
20. Polyzotis, N., Garofalakis, M.: XSketch synopses for XML data graphs. ACM Trans. Database Syst. 31(3) (2006)
21. Saha, D.: An incremental bisimulation algorithm. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 204–215. Springer, Heidelberg (2007)
22. Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: VLDB (2002)
23. Spiegel, J., Polyzotis, N.: Graph-based synopses for relational selectivity estimation. In: SIGMOD (2006)