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. To illustrate the design of our algorithm, first, we present some challenges of bisimulation minimization of cyclic graphs. Second, in the context of database applications, it is natural to compute minimal bisimulation with merging algorithms. We present a maintenance algorithm for a minimal bisimulation of a cyclic graph in the style of merging algorithm. Third, we propose a feature-based optimization to prune the computation on non-bisimilar SCCs. The features are constructed and maintained more efficiently than bisimulation minimization. Finally, we present an experimental study that verifies the scalability of our algorithm and shows that our features-based optimization pruned 50% unnecessary bisimulation computation on average. Our experiment shows that our algorithm introduces a capability to handle cyclic graphs explicitly.

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, 6, 11, 16, 18, 23], have been derived from the notion of *bisimulation* equivalence. In addition to indexing, bisimulation has been adopted for selectivity estimation [13, 19, 20] and schemas for semi-structured data [2].

To illustrate the application 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

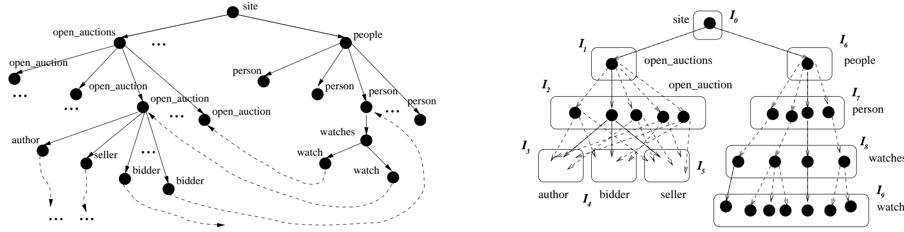


Fig. 1. Illustration – A sketch of `XMark` and its bisimulation

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_n. Consider a query q `/site/open_auction/seller` that selects all `seller`s of `open_auction`s. We can evaluate q on the partitions and retrieve the data nodes in simply I_3. It is crucial to minimize the bisimulation graph for efficient index.

In practice, data graphs are often cyclic (e.g., [1]) and subjected to updates. Therefore, in contrast to other applications of bisimulation, its maintenance problem is much more important in database applications [12, 21]. Our study on the maintenance problem of bisimulation of possibly cyclic graphs contributes to the current state-of-the-art in two aspects: (i) While there are numerous bisimulation applications, there is relatively few work on its maintenance; (ii) Previous works [12,21] on maintenance of bisimulation of graphs mainly focus on directed *acyclic* graphs. These techniques do not explicitly handle cyclic structures.

In this paper, we take the first step to systematically and comprehensively investigate *incremental maintenance of minimal bisimulation in cyclic graphs*. The first contribution is a study of some properties of bisimulation of cyclic graphs, which influence the design of our algorithms (Section 4). There are two key challenges in the maintenance problem. Firstly, merging bisimulation algorithm as opposed to partition refinement is more natural for incrementally maintaining bisimulation. However, it is known [12] that merging algorithms fail to determine the minimum bisimulation of cyclic graphs. The current merging step on a *strongly connected component* (`SCC`) causes subsequent merging steps to miss some bisimilar `SCC`s (to be detailed in Sections 3 and 4). Secondly, the nodes of `SCC`s must be considered *together*. In particular, a node of an `SCC` can be bisimilar to a node of another `SCC` *only if* the two `SCC`s are bisimilar.

Second, we present a maintenance algorithm for minimal bisimulation of cyclic graphs (Section 5). Our algorithm consists of a split and a merge phase. In the split phase, we split and mark the index nodes (i.e., the equivalence partitions) that are affected by an update. In the merge phase, we apply a (partial) bisimulation minimization algorithm on the marked index nodes. We remark that our algorithm introduces an explicit handling of bisimulation between `SCC`s to the current state-of-the-art. As such, our algorithm *always* produces smaller (if not the same) bisimulation graphs when compared to previous works.

The third contribution is on our feature-based optimization for determining bisimulation between two `SCC`s (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 features. Specifically, we explore label- or edge-based, path-based, tree-based and circuit-based features, by studying their pruning power and maintenance efficiency.

The fourth contribution is our extensive experimental evaluation on our proposed technique (Section 7). It shows that (i) our algorithms scales well; (ii) the feature-based optimization prunes 50% computation on non-bisimilar SCCs on average; (iii) the path-based feature is the most effective; and (iv) our algorithm always produces a smaller bisimulation than previous works'.

2 Related Work

Existing works on maintaining bsimulation 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 [5]. [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. In comparison, our algorithm also contains the split and merge phases. A difference between our work and the previous works is that we provide efficient 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 [17] – 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 compare via experiments in Section 7.

Bisimulation (relation) [15] 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 used.

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 either of r, ρ or Σ are irrelevant to our discussions. Since our work focuses on cyclic graphs, we recall some relevant definitions below.

Cyclic graphs. We present the definitions needed to discuss bisimulation of cyclic graphs. A *strongly connected component* (SCC) in a graph $G(V, E)$ is a subgraph $G'(V', E')$ whose nodes is 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 only. In the definition below, we hightlight 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:*

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

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

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:

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. Hence, a bisimulation of a graph can be described as a set of partitions.

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.

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

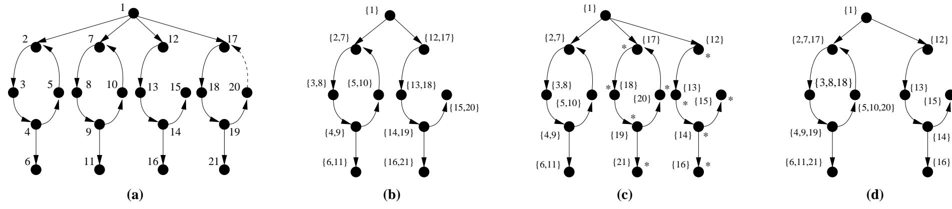


Fig. 2. (a) A cyclic data graph; (b) the minimal bisimulation graph; (c) the split bisimulation graph; and (d) an updated minimal bisimulation graph

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.*

Since the bisimulation B, which can be viewed as a graph, is used as indexes, 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*, or simply *nodes*.

Bisimulation minimization. Next, we illusrate the intuitions of merging algorithm for bisimulation minimization with a brief example shown in Fig. 2. Assume the nodes of the data graph shown in Fig. 2(a) have the same label. We show the node ids 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 paritions top-down, which attemps to merge Nodes 2 and 7. However, the algorithm has not yet determine Nodes 5 and 10. Hence, the algorithm does not return the minimum bisimulation shown in Fig. 2(b), unless it memorizes the `SCC`s 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.

The algorithm can be divided into two parts. First, Lines 01-06, if n_1 and n_2 are not both in some `SCC`s, we compute bisimulation between n_1 and n_2 in the sytle of a 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`.

Second, if both n_1 and n_2 are in some `SCC`s, 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 `SCC`s by using the feature-based optimization presented in Section 6, in Line 08. For

```
Procedure bisimilar_cyclic
Input: Nodes n1 and n2 where ρ(n1) = ρ(n2); B, the current bisimulation
Output: An updated bisimulation relation B'
01 if n1 and n2 are not both in some SCC
02     if ∀p1 ∈ n1.parent ∃p2 ∈ n2.parent s.t. p1 ∼ p2 then
03         add (n1, n2) to B
04         for all c1 in n1.next_nodes_top_order(G1)
05             for all c2 in n2.next_nodes_top_order(G2)
06                 B = bisimilar_cyclic(c1, c2, B)
07 else                /* check bisimulation of the two SCCs */
08     assume n1 and n2 are in SCCs S1 and S2, respectively
       if feature_pruning(S1, S2) return B            /* Sec. 6*/
09     clone S1 to S1';        create an artificial node n1' for n1
10     for all (n, n1) ∈ S1'.E
11         replace (n, n1) with (n, n1') ∈ S1'
12     clone S2 to S2';        create an artificial node n2' for n2
13     for all (n, n2) ∈ S2'.E
14         replace (n, n2) with (n, n2') ∈ S2'
15     clone B to B';     add (n1, n2) to B'             /* assume n1 ∼ n2 */
16     for all c1 in n1.next_nodes_top_order(S1')
17         for all c2 in n2.next_nodes_top_order(S2')
18             B' = bisimilar_cyclic(c1, c2, B')
19     if (n1', n2') in B' then B = B ∪ B'               /* S1 ∼ S2 */
20 return B
```

Fig. 3. Bisimulation minimization of cyclic graphs

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 Fig. 4. Specifically, we redirect the incoming edges of n_1 in n_1' 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 *does not*. bisimilar_cyclic may be recursively called due to nested SCCs (Line 18). Without breaking cycles, the feature-based optimization (Line 07) may always derive features of the "topmost" SCC. As verified by experiments (Figures 7(b) and 7(c)), the features will be essential for pruning computation on non-bisimilar SCCs.

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 17-19, assuming that feature_pruning can be performed more efficiently than bisimilar_cyclic. The inner loop can be performed in $O(log(|V|))$. The overall runtime is $O(|E|log(|V|))$.

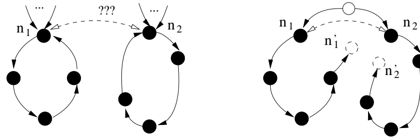


Fig. 4. Breaking one cycle in an SCC

5 Maintenance of Bisimulation

We present an overall maintenance algorithm in this section. 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.

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 SCCs 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 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 SCCs in the data graph to spilt and collect the inodes that are affected by the update. SCCs 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 apply merging on the inodes that are marked in the split phase.

Example 5.1. We illustrate Algorithm `insert` with an example. Reconsider the cyclic data graph 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

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 `SCC`s$\}$
13 **if** $\mathcal{Q} \neq \emptyset$ **then** /* split nodes not related to `SCC`s */
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 `SCC`s$\}$
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

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 `SCC`s as discussed in Section 4. Subsequently, any subgraphs that are connected to the `SCC` (Nodes 17-20), e.g., Node 21, are not merged, as the `SCC`s are not merged.

Analysis. The recursive procedure in Lines 05-20 traverses the graph $O(|E|)$. With optimization in [17], 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 $O(|E|log(|V|)$. Thus, the overall runtime of Algorithm `insert` is $O(|E|log(|V|)$.

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. The non-minimal bisimulation is then minimized by merging. As discussed in the previous section, determining if two SCCs are bisimilar can be computationally costly $O(|E|log(|V|))$. In addition, in practice, SCCs may often be non-bisimilar. This motivates us to optimize bisimulation 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. Futhermore, the features are ideally discriminative enough and can be efficiently derived and maintained.

6.1 Properties of Bisimulation of Cyclic Graphs

This subsection shows 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. We establish these properties with proof by contradictions. (The detaials can be found in the technical report [**?**].)

Property 1. SCCs with the same cycle height may not be bisimilar. 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 the different number of simple cycles.

Property 3. Two bisimilar 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 some nodes 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. Any merging algorithm could not return the minimum bisimulation since the current merging step of a SCC may affect other SCCs. We present some efficient features that gives the merging algoirthm some "lookahead" of SCCs to effeciently conclude that there is some node in a SCC that is not bisimilar to any nodes in other SCC (Proposition 6.2).

Label-based or edge-based features. The label-based and edge-based features are straightforward and have many alternatives. 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.

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 in PSPACE [14] and its maintenance is technically intriguing.

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

There are other notions of paths that do not seem to be appropriate for our problem. For example, the longest paths of a cyclic graph 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 must be non-bisimilar graphs. Hence, k-paths can be used as a feature. It is straightforward that k-paths can be efficiently constructed and maintained.

A remark is that k-paths may not consist of the node(s) that are not bisimilar to any nodes in any other SCC (Proposition 6.2). Another remark is that a node in an SCC may appear in a k-path set multiple times. Next, we propose a spanning tree as a feature of an SCC.

Feature of canonical spanning tree. 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 weight of the edges 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 a couple of tricks on the data graph 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*. (The edge direction is simply needed to check bisimulation between canonical spanning trees.) The

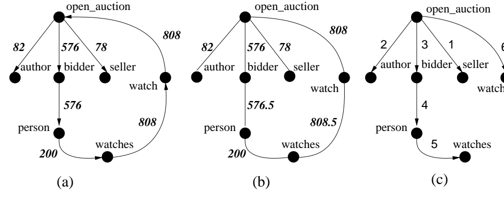


Fig. 6. The construction of the canonical spanning tree from a simplified `open_auction`

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 `SCC`s 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 $O(|V| + |E|)$.

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 is returned by Prim's algorithm). Finally, the direction of the edges are derived from the root of the tree `open_auction`.

Circuit-based features. Finally, we discuss the feature of circuit bases, which contains much more structural information than spanning trees. It has been shown that the minimum circuit bases of directed graphs is unique [8]. Hence, one may be tempted to use circuit bases as a feature to prune computation on non-bisimilar graphs.

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

However, determining the circuit bases is essentially $O(|V|^3)$. It is therefore more efficient to simply compute the bisimulation of two `SCC`s than using the feature of circuit bases.

6.3 Offline versus Online Feature Construction

Since the proposed features can be constructed efficiently, they may be constructed and used during bisimulation computation, i.e., runtime. During runtime, we may incorporate the features with the partial bisimulation constructed

so far for constructing features. Specifically, some nodes in SCCs have been associated with Inode. The id of Inodes together with the label, as opposed to the label alone, to build features.

In comparison, the features may be built the features offline and maintained with each update of the graph. However, given a cyclic graph, there can be exponientially many SCCs to the number of nodes of the graph, in the worst case. To build all possible features offline, we may determine features for each node, in the worst case. This size requirement may sometimes be prohibitive.

7 Experimental Evaluation

We present an experimental study on our algorithms. We modified the implementation of Ke *et al.* [12] to implement our algorithms. The implementation used in the experiment is available at http://code.google.com/p/minimal-bisimulation-cyclic-graphs/. The program is written in JDK 1.5. The implementation is run on a laptop computer with a dual CPU at 2.0 GHz and 2GB RAM running Ubuntu hardy.

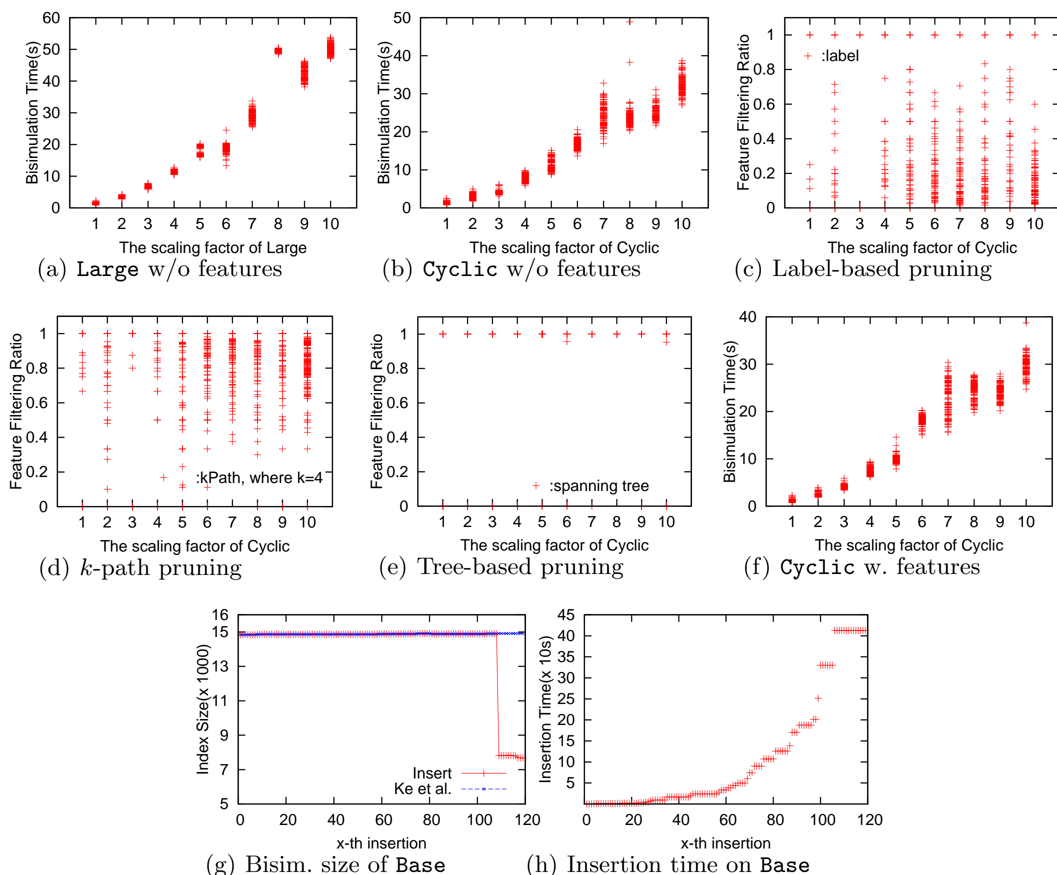
We used the XMark dataset [22] to test variou aspects of our algorithms. The cycles in XMark is essentially composed by IDREFs of open_auction to person and vice versa. We ran Gabow's algorithm on XMark. We note that there are few very large SCCs. It is easy to verify that very few, or none, of the SCCs are bisimilar. Hence, we modify the cycles of XMark in the following way: 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. We set s and r to 10 and 1.2, respectively. 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 may be hardly shown systematically with Large because it only contains one large SCC. Cyclic contains numerous random non-bisimilar SCCs. In both cases, insert and Ke *et al.* return very similar bisimulation graphs. Therefore, we design Base to demonstrate the performance difference between the algorithms.

Base is constructed by connecting to XMark graphs with a s.f. 0.01 and removing 120 edges from the graph. Prior the removal of the edges, the graph has two bisimilar SCCs. When the edges are inserted by Algorithm insert, the bisimilar SCCs will be recovered and merged.

Figures 7(a) and 7(b) show the performance of bisimilar_cyclic *without* feature-based optimization on Large and Cyclic with a scaling factor (s.f.) ranging from 0.01 to 0.1 (i.e., 1MB to 10MB). Since there is some randomness in the SCCs of Large and Cyclic, we ran 100 graphs for each s.f. Figures 7(a) and 7(b) show that the runtimes are roughly linear to s.f. At the same s.f.



(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 on `Base`

Fig. 7. Scalability test of the minimization algorithm on `XMark`; the effectiveness of features; and the efficiency of the maintenance algorithm

(hence same graph size), the runtime for `Large` is longer than that for `Cyclic`. The reason is that in `Cyclic`, there are many smaller random `SCC`s, which are often non-bisimilar, and `bisimilar_cyclic` can identify them relatively earlier. In comparison, `bisimilar_cyclic` in `Large` may spend more time in checking substructures in a large `SCC`.

Next, we verify 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 results are shown in Figures 7(c), 7(d) and 7(e). The y-axis is the percentage of non-bisimilar `SCC`s that were pruned by a feature. 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 features. On average, it is 4% faster than

that without features (Figure 7(b)). However, we remark that on average, 7.7% of the runtime is due to online feature construction.

Lastly, we present an experiment on Algorithm `insert`. We connect two `Large` graphs with a s.f. 0.01 and randomly remove 120 edges from the `SCC`s to form the base graph, denoted as `Base`. We insert the removed edges (randomly) one-by-one to `Base`. The result is shown in Figure 7(g). Figure 7(g) shows the size of the minimal bisimulation produced by `insert` and Ke *et al.* [12]. We did not show the result from Paige and Tarjan (the minimum) as `insert` always produces a bisimulation that is within 2% of the minimum. Initially, both `insert` and [12] are very close to the minimum. After some number of insertions, the two bisimilar `SCC`s in the original `Large` graph are recovered. We ran this experiment multiple times and find that this occurred randomly between 100th and 120th insertion. As shown in Figure 7(g), `insert` identifies the two bisimilar `SCC`s that lead to a bisimulation graph roughly 100% smaller than the one produced by [12]. We remark that the performance difference (in terms of bisimulation size) between `insert` and [12] depends on the size of bisimilar `SCC`s and their bisimilar subgraphs are there in a graph.

The runtime of `insert` is shown in Figure 7(h). The runtime increases as we insert more edges into `Base`. After many insertions, `insert` runs slower because the two `SCC`s in `Base` become very similar. `bisimilar_cyclic` checks many nodes before it declares the `SCC`s are not bisimilar. The runtime of [12] is close to 0s as it does not process `SCC`s.

8 Conclusions

In this paper, we studied the maintenance problem of minimal bisimulation of cyclic graph. First, we presented a bisimulation minimization algorithm that explicitly handles `SCC`s. Second, we presented a maintenance algorithm for minimal bisimulation of cyclic graphs. Third, we propose a feature-based optimization to avoid computation of non-bisimilar `SCC`s. We present an experiment to verify the scalability of our algorithms. In addition, our experiment shows that on average, the features can prune unnecessary bisimulation computation. Our maintenance algorithm can return smaller bisimulation graphs than previous work, depending the sizeof bisimilar `SCC`s and their bisimilar subgraphs in the data graph.

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: ICDT (1997)
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. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theor. Comput. Sci. 311(1-3), 221–256 (2004)
6. Fisher, D.K., Maneth, S.: Structural selectivity estimation for XML documents. In: ICDE (2007)
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: ICALP (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. Li, H., Lee, M.L., Hsu, W., Cong, G.: An estimation system for XPath expressions. In: ICDE (2006)
14. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. In: VLDB (1989)
15. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
16. Milo, T., Suciu, D.: Index structures for path expressions. In: ICDT (1999)
17. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
18. Polyzotis, N., Garofalakis, M.: XCluster synopses for structured XML content. In: ICDE (2006)
19. Polyzotis, N., Garofalakis, M.: XSketch synopses for XML data graphs. ACM Trans. Database Syst. 31(3) (2006)
20. Polyzotis, N., Garofalakis, M., Ioannidis, Y.: Approximate XML query answers. In: SIGMOD (2004)
21. Saha, D.: An incremental bisimulation algorithm. In: FSTTCS (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)