Incremental Maintenance of Minimal Bisimulation of Cyclic Graphs

Jintian Deng

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, as opposed to partition-refinement algorithms. We present a maintenance algorithm for a minimal bisimulation of a cyclic graph in the style of merging algorithm. Third, merging algorithms cannot determine the minimum bisimulation without examining all possible SCCs. We propose a feature-based optimization technique 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 and when compared to previous work, our bisimulation can be 100% smaller, depending on the number of bisimilar SCCs in the data graph.

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., [4,5,7,12,17,19,24], have been derived from the notion of *bisimulation* equivalence. In addition to indexing, bisimulation has been adopted as a notion of schemas for semi-structured data [3]. In recent works on selectivity estimation [14, 20, 21], bisimulation has also been used to construct synopses of graphs for path queries.

Two nodes in a data graph are bisimilar if they have the same set of incoming paths. 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 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 Figure 1.

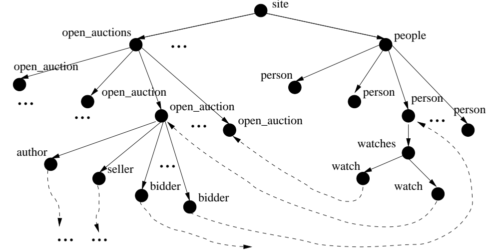


Figure 1: A simplified XMark

Without considering the references, XMark is a tree. It is evident that the bisimulation (see Appendix A) is smaller than the original data tree, which makes it a useful index for path queries. As an index structure, the bisimulation graph needs to be minimized.

In practice, data graphs are often cyclic (e.g., [1]) and subjected to updates. Therefore, when compared to other applications of bisimulation, its maintenance problem is much more important in database applications [13, 22]. 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 have been numerous applications on bisimulation, there has been relatively few work on its maintenance; (ii) Previous works [13,22] on maintenance of bisimulation of graphs mainly focus on directed *acyclic* graphs. There has not been explicit handling of cyclic structures.

There are two key challenges in maintaining minimal bisimulation of cyclic data graphs. Firstly, merging-based bisimulation algorithm as opposed to partition refinement is more natural for incrementally maintaining bisimulation. However, it is known [13] that merging-based algorithms fail to determine the minimum bisimulation of cyclic graphs. This is because the current merging step on a *strongly connected component* (SCC) causes subsequent merging steps to miss some bisimilar SCCs (to be detailed in Sections 3 and 4). Consequently, bisimulation minimization needs to examine many pairs of SCC as in the worst case there are exponentially many SCCs in a cyclic graph. Secondly, the nodes of SCCs must be considered *together* for checking bisimulation between SCCs of a cyclic graph. In particular, a node of an SCC can be bisimilar to a node of another SCC only if the two SCCs are bisimilar. However, merging-based algorithms compare one pair of nodes in each merging step.

In this paper, we carry out a comprehensive investigation on how to incrementally maintain minimal bisimulation of cyclic graphs. The first contribution is a study of some prop-

erties of bisimulation of cyclic graphs. These form the basis and terminologies of our discussions on the maintenance algorithm and influence the design of our algorithms. In particular, we study how a few nestings of cycles may affect bisimulation minimization. Our observation is that (i) determining the minimal bisimulation of cyclic graphs is tedious and subtle; and (ii) while it seems unlikely that there are many bisimilar cycles in a real-world data graph, it may not be reasonable to simply overlook them.

Second, we present a maintenance algorithm for minimal bisimulation of cyclic graphs. Similar to [13], 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. But different from [13], our algorithm has an explicit handling of bisimulation between `SCC`s. As such, our algorithm *always* produces smaller (if not the same) bisimulation graphs when compared to [13].

The third contribution is on our feature-based optimization for determining bisimulation between two `SCC`s. On the one hand, the computation of bisimulation between two `SCC`s can be costly. On the other hand, there may not be many bisimilar `SCC`s. Hence, we aim at deriving structural features from `SCC`s such that two `SCC`s are bisimilar only if they have the same features. Feature-based pruning has been a popular technique in determining subgraph isomorphism in graph-structured databases, among many others. We shall explore label-based, edge-based, path-based, tree-based and circuit-based features, by studying their pruning power, construction and maintenance efficiency.

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 [13,22] for incremental maintenance of bisimulation of cyclic graphs. The algorithm proposed in [13] 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, [13] produces the minimum bisimulation of the updated graph. If the graph is cyclic, [13] returns a minimal bisimulation only. Since [13] considers merging pairs of nodes iteratively, it does not minimize bisimulation between `SCC`s. Thus, to support cyclic graphs, the minimum bisimulation is occasionally re-computed from scratch. [22] can be considered as a follow-up of [13]. [22] proposes a split-merge-split algorithm with a rank flag for `SCC`s. The rank flag is originally proposed in [6] and adopted by [22]. The algorithm also returns a minimal bisimulation in response to an update of a cyclic graph. 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 `SCC`s and propose features to optimize bisimulation maintenance.

A recent partition-refinement algorithm [11] 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 [11] produces a bisimulation that is always within 5% of the minimum bisimulation. It has been shown, through a later experiment, that [13] may produce even smaller bisimulations.

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 [8], in the context of modeling checking. A conclusion of [8] 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.

As discussed in Section 1, bisimulations have been recently used in numerous database applications. Indexes for path queries have been derived from bisimulation [5, 12, 17]. 1-index [17] adopts bisimulation as an index for regular paths. However, in practice, 1-index [17] can be large. A notion of local bisimulation, namely k-bisimulation, has been proposed to reduce index size. During query evaluation, local bisimulations may be combined to determine the complete path information. To balance query performance and index size, [5] proposes to adaptively adjust the k in k-bisimulation of subgraphs. [2, 4] consider bisimulation as a compressed instance of an XML repository for efficient query processing. In addition, bisimulation have been used as a summary structure for path query selectivity estimation, e.g., [20, 21]. It is evident that a study on maintenance of bisimulation benefits all the above mentioned applications.

3. BACKGROUND

Next, we provide the background and the notations used.

Definition 3.1: A *graph-structured database* 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 $\rho : V \to \Sigma$ 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 graph as $G(V,E)$ when r, ρ and Σ are irrelevant to our discussions.

Bisimulation. We recall the definition of bisimulation:

Definition 3.2: Given two graphs $G_1(V_1, E_1, r_1, \rho_1, \Sigma_1)$ and $G_2(V_2, E_2, r_2, \rho_2, \Sigma)$, an *upward bisimulation* $\sim$ is a binary relation between V_1 and V_2:

1. If (i) v_1 (resp. v_2) is the root of G_1 (resp. G_2) and (ii) $\rho(r_1) = \rho(r_2)$, then $v_1 \sim v_2$.
2. If (i) v_1 (resp. v_2) is not the root of G_1 (resp. G_2), (ii) $\rho(v_1) = \rho(v_2)$ and (iii) for each edge $(v_1', v_1) \in E_1$ (resp. $(v_2', v_2) \in E_2$), there is an edge $(v_2', v_2) \in E_2$ (resp. $(v_1', v_1) \in E_1$) such that $v_1' \sim v_2'$, then $v_1 \sim v_2$.

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

Definition 3.2 presents upward bisimulation in the sense that two nodes can be bisimilar only if their parents are bisimilar. This is a recursive definition – two nodes can be bisimilar only if their ancestors (up to the root) are bisimilar. The definition can be paraphrased in terms of paths. That is, two nodes are bisimilar if they have the same set

of incoming paths. This definition is often convenient to simplify our discussions.

Proposition 3.1: *Two nodes are upward bisimilar if and only if the set of incoming paths 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 is also often described as a set of equivalence classes.

We should remark that there have been other notions of bisimulation that have been applied in indexing/selectivity estimation but have not been the focus of this paper. The details of these definitions can be found in Appendix B. Our techniques can be extended to support them with some modifications. For presentation simplicity, we use bisimulation to refer to upward bisimulation, unless otherwise specified.

In this work, we consider the notion of bisimulation minimality presented in [13]. We paraphrase its definition below.

Definition 3.3: Given a bisimulation B of a graph G, B is *minimal* if for any two equivalence 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 equivalence partition $K \in B$ unstable. □

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 strongly connected components of a graph can be determined by classical algorithms, e.g., Gabow's algorithm, in $O(|V|+|E|)$.

Graph contraction has been a popular technique for processing cyclic data graphs. It is often convenient to use graph contraction in our discussions. Intuitively, given a cyclic graph, graph contraction reduces each strongly connected component into a supernode iteratively until an acyclic graph is obtained. Queries are often first processed on the reduced acyclic graph and then the supernodes (the strongly connected components). To present graph contraction, we recall the notion of exits and entries of SCCs.

Definition 3.4: 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 $n \in V'$. □

In general, an SCC can have multiple entry and exit nodes.

Definition 3.5: Given an SCC G' of a graph, a *one-step graph contraction* reduces G' into a supernode n and each incoming (resp. outgoing) edge to G' is modified to be an incoming (resp. outgoing) edge to n. □

The nestings of strongly connected components of a graph affect bisimulation minimization. To illustrate this, we define cycle height to describe the nestings of cycles.

Definition 3.6: The *cycle height* h (or simply *height*) of a graph is h if the height of its SCCs is at most h. The height of an SCC can be inductively defined as follows: A trivial SCC (a graph with a single node and no edge) has a height 0. An SCC s has a height $h + 1$ if the height of the SCCs in s is at most h. □

The cycle height of the nodes in G can be computed in $O(h \times (|E|+|V|))$, where h is the cycle height of the graph. It is straightforward that the cycle height of a node is not unique, which depends on the graph contraction steps.

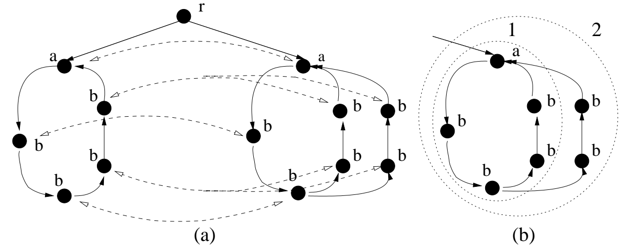


Figure 2: Two bisimilar SCCs with different cycle heights

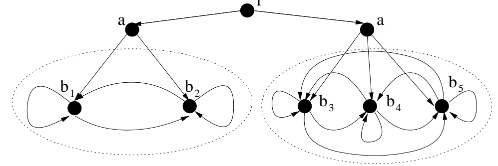


Figure 3: Two bisimilar SCCs with different numbers of entry nodes

4. BISIMULATION OF CYCLIC GRAPHS

In Subsection 4.1, we discuss bisimulation minimization in the presence of cyclic structures . In Subsection 4.2, we present a minimization algorithm for bisimulation of cyclic graphs, which is a major component of the maintenance algorithm presented in Section 5.

4.1 Properties of Bisimulation of Cyclic Graphs

We show a few properties of bisimulation of cyclic graphs. They shred some lights that it seems unlikely that merging algorithms could determine the minimum bisimulation between SCCs without examining many possible sub-SCCs.

Property 4.1: *SCCs with the same cycle height may not be bisimilar. SCCs with different cycle heights can be bisimilar.* □

The first part of this property is straightforward whereas the second part may require some elaborations. Consider a simple example shown in Figure 2(a). Consider the two SCCs in the figure. The cycle height of the SCC on the left is 1 while that on the right is 2 (Figure 2(b)). The dotted arrows in Figure 2(a) show a possible bisimulation between the two SCCs.

Property 4.2: *Two bisimilar SCCs with different numbers of entry nodes can be bisimilar.* □

A simple example is sketched in Figure 3. As discussed in Section 1, a possible bisimulation is $\{(b_1 \sim b_3), (b_2 \sim b_4), (b_2 \sim b_5)\}$. In another words, b_1 and b_3 have the same set of incoming paths and similarly, b_2, b_4 and b_5 have the same set of incoming paths.

Property 4.3: *Two bisimilar SCCs may have the different number of simple cycles. Two SCCs with the same number of simple cycles may not be bisimilar.* □

The first part of this property can be illustrated with Figure 2(a). The second part of the property can be illustrated with Figure 4. Both graphs in Figure 4 consist of 6 simple

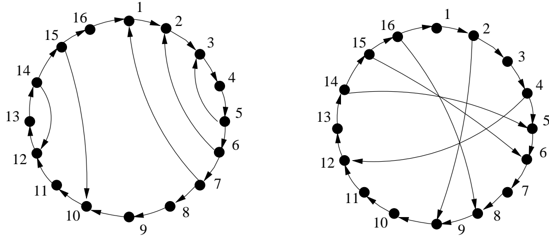


Figure 4: Cases of different overlapping cycles

cycles but are non-bisimilar. In addition, on LHS of Figure 4, the nodes of a cycle may be a subset of the nodes of other cycles. In comparison, on RHS of Figure 4, the cycles are overlapping but not contained in each other. When determining a bisimulation of a cycle, part of other cycles are involved. Hence, in the worst case, one may need to consider all sub-SCCs together, in the style of partition refinement [18].

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 simple cycle may affect other cycles. Therefore, to compute a minimal bisimulation of cyclic graphs, existing merging algorithms need to examine many possible SCCs.

4.2 Minimizing Bisimulation of Cyclic Graphs

Our algorithm `bisimilar_cyclic` for minimizing bisimulation of cyclic graphs is shown in Figure 5, which is a component of our maintenance 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.

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 sytle of a merging algorithm. In this case, we recursively invoke `bisimilar_cyclic`, for handling of cycles reachable from n_2.

Second, if both n_1 and n_2 are in some SCCs, Lines 07-21 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 09-16. The main idea is illustrated with Figure 6. More 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 16-19). If we can construct a possible bisimulation between n'_1 and n'_2 (Line 20), then we declare that S_1 and S_2 are bisimilar.

The main difference between `bisimilar_cyclic` and [13] is that `bisimilar_cyclic` explicitly breaks a cycle when determining bisimulation between SCCs whereas [13] *does not* check bisimulation between SCCs *and* between SCCs' descendants. `bisimilar_cyclic` may be recursively called due to nested SCCs (Line 18). Without breaking a cycle, the feature-based optimization (Line 07) may always derive features of the "topmost" SCC. As verified by experiments (Figures 10(b) and 10(c)), the optimization will be essential for

Procedure `bisimilar_cyclic`
Input: Nodes n_1 and n_2 where $\rho(n_1) = \rho(n_2)$, $n_1 \in G_1$ and $n_2 \in G_2$; B, the current bisimulation relation
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_1 to S'_1; 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'
16   add (n_1, n_2) to B' /* assume $n_1 \sim n_2$ */
17   **for all** c_1 in n_1.`next_nodes_top_order`(S'_1)
18   **for all** c_2 in n_2.`next_nodes_top_order`(S'_2)
19   B' = `bisimilar_cyclic`(c_1, c_2, B')
20   **if** (n'_1, n'_2) in B' **then** $B = B \cup B'$ /* $S_1 \sim S_2$ */
21 **return** B

Figure 5: Bisimulation minimization of cyclic graphs

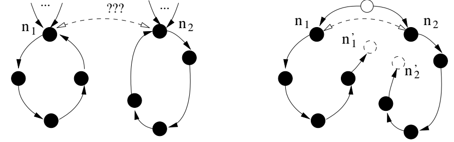


Figure 6: Breaking one cycle in an SCC

early backtracking in determining possible bisimulations between 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`. With classical indexing on nodes, the inner loop can be performed in $O(log(|V|))$ and the overall runtime is $O(|E|log(|V|))$.

5. MAINTENANCE OF BISIMULATION

We present an overall maintenance algorithm in this section. For simplicity, we present an edge insertion algorithm `insert` in Figure 7. Edge deletions are discussed at the end of this section. Our algorithm consists of a split phase and a merge phase, and with an explicit handling of SCCs. The merge phase is essentially bisimulation minimization, which has been detailed in `bisimilar_cyclic` in Section 4. 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.

```
Procedure insert
Input: an insertion of an edge (n1, n2) a data graph G
   and its minimal bisimulation B
Output: An updated graph G' and
   its updated minimal bisimulation B'
01 G' = insert (n1, n2) into G
02 if n2 is new
   then create a new inode I_n2; insert I_n2 into B; mark I_n2
   else if I_n2 is not stable
03      S = {(I_n2, n2) | n2 is in an SCC}
04      Q = {I_n2 | n2 is not in any SCC}
05 while Q ≠ ∅ or S ≠ ∅
        /* split the relevant SCC */
06      if S ≠ ∅ then
07          pick a node (I_n, n) from S; remove (I_n, n) from S
08          while I_n is not stable or a singleton
09            split I_n into I1 = I_n - {n} and I2 = {n}
10            mark I1 and I2
11            S = S ∪ {(I_ns,n_s) | n_s is child of n_i, n_i ∈ I2
                                     and n_s in the SCC of n}
12            Q = Q ∪ {I_nq| n_q is a child of n_i, n_i ∈ I2
                                     and n_q not in any SCCs}
        /* split nodes not related to SCCs */
13      if Q ≠ ∅ then
14          pick a node I_n ∈ Q; remove I_n from Q
15          if I_n is not stable or a singleton
16              split I_n into a stable set I              // [13]
17              for each I in I
18                  mark I
19                  S = S ∪ {(I_ns,n_s) | n_s is a child of n_i, n_i ∈ I
                                         and n_s in the SCC of n}
20                  Q = Q ∪ { I_nq | n_q ∈ child of n_i, n_i ∈ I
                                         and n_q not in any SCCs}
21 Gabow(G')          //update the SCC information in G'
/* merging the marked inodes */
22 (G', B') = bisimilar_cyclic_marked(G, B)
23 return (G', B')
```

Figure 7: Insertion for minimal bisimulation of cyclic graphs

Similar to previous work, we call an equivalence partition, which contains a set of bisimilar nodes, *inodes*, denoted with I. During the split phase we mark the affected inodes which will be examined in the merge phase.

Assume the insertion makes an inode of n_2 not stable. 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 some 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 recursively until $\mathcal{S}$ and $\mathcal{Q}$ 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 that are involved in some SCC(s) into $\mathcal{S}$ and the remaining children into $\mathcal{Q}$.

(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 other works for maintaining bisimulation of acyclic graphs, e.g., [13] (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 11-12.

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 4.2 Figure 5. A simple optimization is that we do not apply merging on all inodes of the bisimulation graph but simply on the inodes that are marked in the split phase.

Example 5.1: We illustrate Algorithm insert with an example. A cyclic data graph is shown in Figure 8(a). For simplicity, we assume the node in the data graph has the same label. We show the node ids next to each node. Its minimal bisimulation is shown in Figure 8(b). We use {} to denote an inode. 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 8(c). We call Gabow'a algorithm to update the SCC information of the data graph. By calling bisimulation minimization, we obtain the bisimulation graph at Figure 8(d).

It should be remarked that while the previous work [13] produces the same split graph (Figure 8(c)). But, it returns Figure 8(c) as the final bisimulation graph. This is because it lacks the handling on SCCs 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. □

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 $O(|E|log(|V|)$. 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

As discussed in the previous section, determining if two SCCs are bisimilar can be computationally costly $O(|E|log(|V|))$. However, in practice, many SCCs may not be bisimilar. This motivates us to optimize bisimulation minimization of cyclic graphs by proposing features to prune computations on non-bisimilar SCCs.

In particular, we exploit the following property of a bisimulation between SCCs.

Proposition 6.1: *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 *s.t. it is not bisimilar to any nodes in* V_2. □

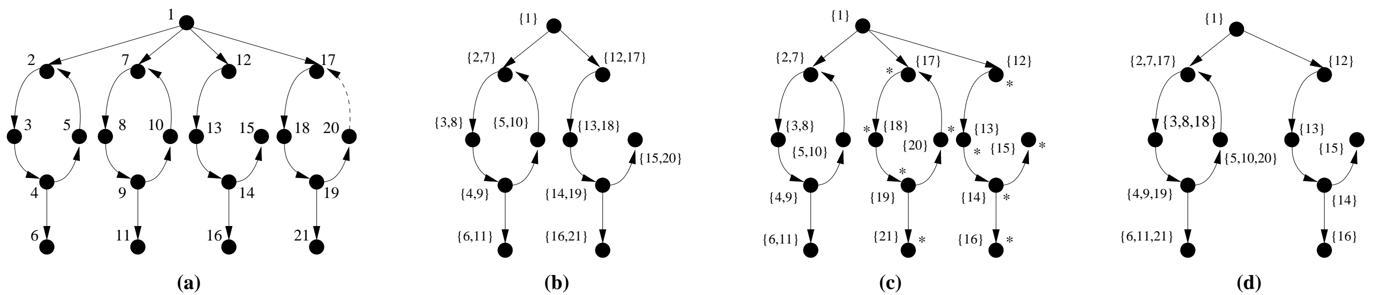


Figure 8: (a) A cyclic data graph; (b) the minimal bisimulation graph; (c) the split bisimulation graph; and (d) an updated minimal bisimulation graph

In this section, we adopt feature-based techniques for filtering computation on two non-bisimilar SCCs. The main idea is to derive a set of features from SCCs such that two SCCs can be bisimilar *only if* their features are the same or bisimilar. The features are ideally discriminative enough to reduce the computation on non-bisimilar graphs. In addition, while the maintenance of bisimulation is non-monotonic in nature, we design features that can be readily incrementally maintained. In the followings, we explore the details of label-based, edge-based, path-based, tree-based and circuit-based features.

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. Obviously, 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 [15] and its maintenance is technically intriguing.

Proposition 6.2: *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, determining the longest paths of a cyclic graph is NP-complete.

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 6.1). 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 $O(|V|^{1/3}log(|V|))$ [10]. 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 direction of the edges in the canonical spanning tree may differ from that of the edges in the original graph.

Proposition 6.3: *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 $O(|V| + |E|)$.

Example 6.1: We illustrate the construction of a canonical spanning discussed above with an example shown in Figure 9. Figure 9(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

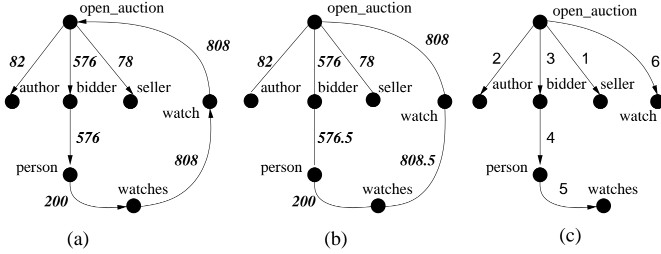


Figure 9: The construction of the canonical spanning tree from a simplified `open_auction`

the edges, shown in Figure 9(b). Then, it is straightforward to compute the spanning tree (shown in Figure 9(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 [9]. Hence, one may be tempted to use circuit bases as a feature to prune computation on non-bisimilar graphs.

Proposition 6.4: *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.

6.1 Offline vs Online Feature Construction

Offline construction. The features of `SCC`s can be computed offline and used and maintained online (in Line 08 Figure 5). It has been known that determining all `SCC`s of a graph runs in exponential time. Therefore, we compute the features for *possible entry nodes* of `SCC`. To determine possible entry nodes, we apply Gabow's algorithm recursively. Gabow's algorithm returns the set of non-overlapping largest `SCC`s (therefore their entry nodes) of a graph. For each `SCC`, we remove an incoming edge of an entry node and apply Gabow's algorithm until no new entry nodes can be determined. The overall runtime of this method is evidently $O(|E| \times (|V| + |E|))$ and it returns all possible entry nodes. Finally, we compute the features for each possible entry node as discussed.

Online construction. Since the proposed features can be constructed efficiently, they may also be constructed during bisimulation computation, i.e., runtime. During runtime, we may incorporate the features with the partial bisimulation constructed so far for constructing features, for a higher pruning power. Specifically, since some nodes have been associated with an inode, we propose to use the id of inodes as opposed to the label alone to build features.

For example, consider the nested cycles on LHS of Figure 4. Assume all nodes have the same label. The cycles of (3,4,5) and (12,13,14) are obviously not bisimilar. However, the label/edge-based and path-based features cannot distinguish these two cycles. In runtime, when computing the bisimulation between (3,4,5) and (12,13,14), we have determined nodes 2 and 11 are not bisimilar. Hence, when we construct the label-based feature in runtime, node 3 has $\{A\{2\},...\}$ and node 12 has $\{A\{11\},...\}$. Thus, determining bisimulation between (3,4,5) and (12,13,14) is not needed.

7. EXPERIMENTAL EVALUATION

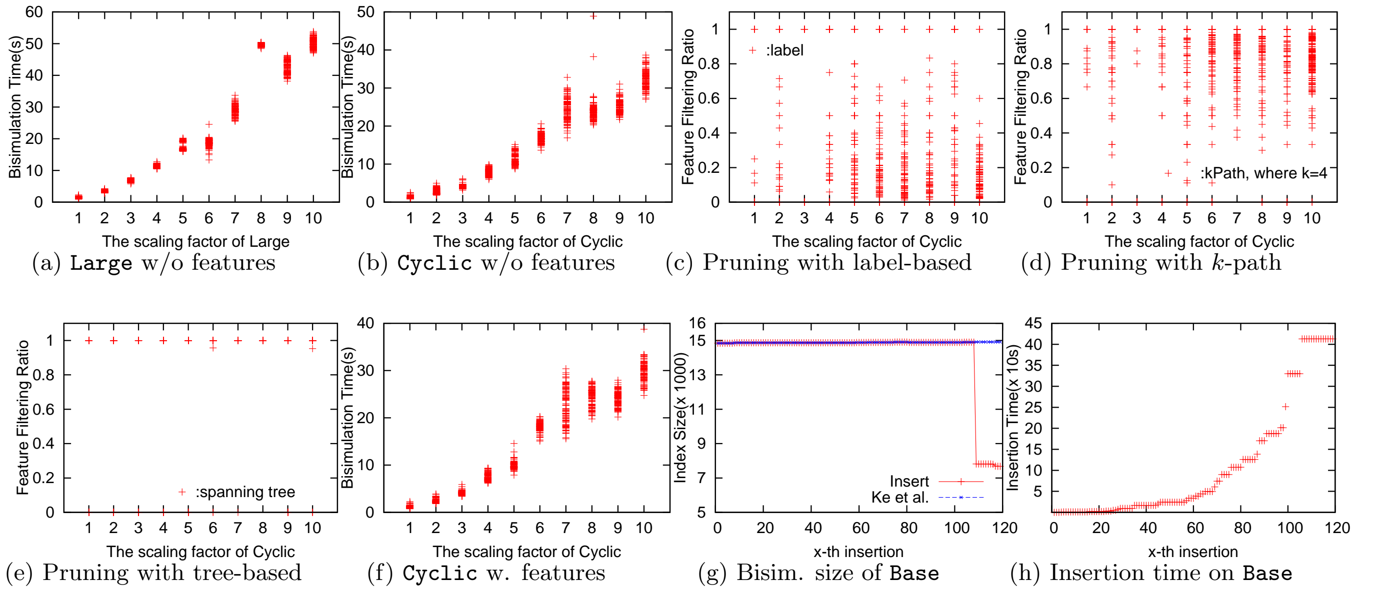
We present an experimental study on our algorithms. We modified the implementation of Ke *et al.* [13] to implement our algorithms. We used `XMark` [23] to obtain a group of graphs with one large `SCC`, denoted as `Large`. We decomposed `Large` to obtain another group of graphs with numerous `SCC`s, denoted as `Cyclic`. Details are in Appendix D. `Large` and `Cyclic` are needed to illustrate different aspects of our techniques.

Figures 10(a) and 10(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 `SCC`s of `Large` and `Cyclic`, we ran 100 graphs for each s.f. Figures 10(a) and 10(b) show that the runtimes are roughly linear to s.f. At the same s.f. (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 10(c), 10(d) and 10(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 10(f) shows the runtime of `bisimilar_cyclic` with features. On average, it is 4% faster than that without features (Figure 10(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 10(g). Figure 10(g) shows the size of the minimal bisimulation produced by `insert` and Ke *et al.* [13]. 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 [13] 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 10(g), `insert` identifies the two bisimilar `SCC`s that lead to a bisimulation graph roughly 100% smaller than the one produced by [13]. We remark that the performance difference (in terms of bisimulation size) between `insert` and [13] depends on how many bisimilar `SCC`s are there in a graph.

The runtime of `insert` is shown in Figure 10(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



(a) `Large` w/o features (b) `Cyclic` w/o features (c) Pruning with label-based (d) Pruning with k-path

(e) Pruning with tree-based (f) `Cyclic` w. features (g) Bisim. size of `Base` (h) Insertion time on `Base`

Figure 10: Scalability test of the minimization algorithm on `XMark`; the effectiveness of features; and the efficiency of the maintenance algorithm

runtime of [13] 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. To tackle the problem, we first presented a few properties about bisimulation on cyclic graphs. Second, we presented a bisimulation minimization algorithm that explicitly handles `SCC`s. Third, we presented a maintenance algorithm for minimal bisimulation of cyclic graphs. Fourth, 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 50% unnecessary bisimulation computation. Our maintenance algorithm can return smaller bisimulation graphs (up to 100%) than previous work, depending the number of bisimilar `SCC`s in the data graph.

9. REFERENCES

[1] V. Batagelj and A. Mrvar. Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[2] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. D. Viglas. Vectorizing and querying large XML repositories. In *ICDE*, 2005.

[3] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, 1997.

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

[5] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[6] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comput. Sci.*, 311(1-3):221–256, 2004.

[7] D. K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. In *ICDE*, 2007.

[8] K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Form. Methods Syst. Des.*, 21(1):39–78, 2002.

[9] P. M. Gleiss, J. Leydold, and P. F. Stadler. Circuit bases of strongly connected digraphs. Working Papers 01-10-056, Santa Fe Institute, 2001.

[10] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *ICALP*, 1997.

[11] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *VLDB*, 2002.

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

[13] Y. Ke, H. Hao, S. Ioana, and Y. Jun. Incremental maintenance of XML structural indexes. In *SIGMOD*, 2004.

[14] H. Li, M. L. Lee, W. Hsu, and G. Cong. An estimation system for XPath expressions. In *ICDE*, 2006.

[15] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, 1989.

[16] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

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

[18] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[19] N. Polyzotis and M. Garofalakis. XCluster synopses for structured XML content. In *ICDE*, 2006.

[20] N. Polyzotis and M. Garofalakis. XSketch synopses for XML data graphs. *ACM Trans. Database Syst.*, 31(3), 2006.

[21] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML query answers. In *SIGMOD*, 2004.

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

[23] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[24] J. Spiegel and N. Polyzotis. Graph-based synopses for relational selectivity estimation. In *SIGMOD*, 2006.

APPENDIX

A. BISIMULATION OF XMARK TREE

A sketch of the bisimulation (graph) of the `XMark` *tree* (shown in Figure 1 with dotted edges ignored) is shown in Figure 11, where bisimilar nodes are placed in an equivalence

partition (enclosed by a rounded rectangle).

B. OTHER NOTIONS OF BISIMULATIONS

There has been a notion of *downward bisimulation*. Since its definition is very similar to Definition 3.2, we only highlight the difference between upward and downward bisimulations. Consider the second condition in Definition 3.2. Downward bisimulation, in contrast, states the following: if for any edge (v_1, v_1') (resp. (v_2, v_2')), there is an edge (v_2, v_2') (resp. (v_1, v_1')) such that $v_1' \sim v_2'$ and $\rho(v_1') = \rho(v_2')$, then $v_1 \sim v_2$. This definition has been adopted for data graph compression [2, 4], among others.

A local notion of bisimulation has been proposed, namely, *k-bisimulation*. Specifically, two nodes are k-bisimilar only if two nodes have the same set of incoming paths up to the length k. In other words, k-bisimulation considers bisimulation up to k steps only. While k-bisimulation may lose some path information, the size of k-bisimulation graph can often be small, in practice.

C. PROOF SKETCHES

Proof (sketch) of Proposition 6.3. The proof can be established by an induction on the run of Prim's algorithm on the graphs. The hypothesis is that the intermediate minimum spanning tree (or simply intermediate tree) of the size k returned by Prim's algorithm of G_1 is bisimilar to an intermediate tree of G_2 if G_1 and G_2 are bisimilar.

The base case is $k = 1$, where the intermediate tree with the root node only. Obviously, the base case is true. Suppose that the hypothesis is true up to the tree size of a size m and the tree for G_1 and G_2 are T_1 and T_2, respectively.

w.l.o.g, suppose that Prim's algorithm adds an edge to T_1 to form T_1' which makes $T_1' \not\sim T_2$. We want to prove that the next intermediate T_2' by Prim's algorithm, where $T_2' \not\sim T_2$, is bisimilar to T_1'. Suppose that the previous edge added to T_1' and T_2' are (a_1, a_1') and (a_2, a_2'). Due to the hypothesis, $a_1 \sim a_2$. Since G_1 and G_2 are bisimilar, there must be a node a_2'' such that $(a_2, a_2'') \in G_2$ and $a_1' \sim a_2''$. If $a_2'' = a_2'$, then $T_1' \sim T_2'$. Otherwise, let $a_2'' \neq a_2'$. If $\texttt{weight}(a_2'') < \texttt{weight}(a_2')$, then Prim's algorithm adds (a_2, a_2''), not (a_2, a_2'), to T_2. If $\texttt{weight}(a_2'') > \texttt{weight}(a_2')$, then Prim's algorithm would not have returned T_1' (contradiction).

Proof (sketch) of Proposition 6.4. Consider that two bisimilar graphs G_1 and G_2. Assume that C_1 and C_2 are the minimum circuit bases of G_1 and G_2, respectively. w.l.o.g, assume $c_1 \in C_1$ is the smallest circuit such that $c_1 \not\sim c_2$ for all $c_2 \in C_2$. Let p_1 be a simple path from the root of G_1 to the "first" node n_1 in c_1. Then, n_1 is not bisimilar to any node in G_2 because we can always find a path $p_1.c_1^i$, where i is an integer representing the repetition of c_1, to distinguish n_1 and any node in G_2.

D. ADDITIONAL INFORMATION FOR EXPERIMENTS

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 [23] to test variou aspects of our algorithms. The cycles in `XMark` is essentially composed by `IDREF`s of `open_auction` to `person` and vice versa. We ran Gabow's algorithm on `XMark`. We note that there are few very large `SCC`s. It is easy to verify that very few, or none, of the `SCC`s 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_auction`s and 12 `person`s.

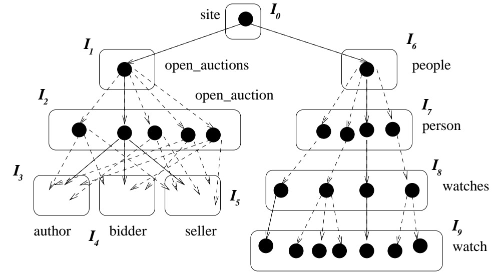


Figure 11: Bisimulation of the tree of `XMark`

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 `SCC`s. 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 `SCC`s. When the edges are inserted by Algorithm `insert`, the bisimilar `SCC`s will be recovered and merged.

Acknowledgements. We are grateful to Yi Ke and Hao He for providing the implementation of [13]. We thank Yun Peng for his comments on the earlier drafts.