An Incremental Bisimulation Algorithm

Diptikalyan Saha

Motorola India Research Lab, Bangalore, India
diptikalyan@motorola.com

Abstract. The notion of bisimulation has been used in various fields including Modal Logic, Set theory, Formal Verification, and XML indexing. In this paper we present the first algorithm for incremental maintenance of maximum bisimulation relation of a graph with respect to changes in the graph. Given a graph, its maximum bisimulation relation, and the changes in the graph, we determine the maximum bisimulation relation with respect to the changed graph by computing the changes in the given bisimulation relation. When the change in the graph induces small changes in the maximum bisimulation relation, our incremental algorithm is able to update the bisimulation relation on average an order of magnitude faster than the fastest available non-incremental algorithm. Preliminary experiments demonstrate the effectiveness of our algorithm. Our algorithm finds extensive use in verification where the specification changes over time, and XML indexing in database where the index structure, obtained by bisimulation on XML graph structure, needs to be maintained with respect to changes in the XML documents.

1 Introduction

The notion of bisimulation equivalence is important in many fields such as Modal Logic, Concurrency Theory, Set Theory, Formal Verification, XML Indexing, and Game Theory. Informally, a pair of automata M, M' are said to be bisimilar if for every transition in M there exists a corresponding transition in M' and vice versa. Milner and Park [15] introduced this notion in Concurrency theory for testing observational equivalence in CCS. Van Benthem [3] used it as an equivalence principal between Kripke Structures. Bisimulation in its various forms like strong or weak has also been used for checking equivalence between finite and infinite transition systems [9]. Verification systems such as the Spin [11], Concurrency Workbench of the New Century (CWB-NC) [5] and CADP [4] incorporate bisimulation checkers in their tool sets. In the area of formal verification, the notion of bisimulation has been primarily used to minimize the state space of the system's description which serves as an important factor in compositional and non-compositional model checking.

Many systems where bisimulation is used are *dynamic* in nature. For example, XML documents are indexed by its minimum bisimilar equivalent graph. As XML documents are updated in the database, their indices need to be updated too. In the area of verification, software systems undergoing verification evolve as a result of bug fixes and requirement changes. Similarly, specifications of security protocols and hardware designs required for verification are also changed over time. However, most of the verification systems use their techniques as a whole on the changed input. They do not consider the changes in the input, although in many cases the changes in the specification or software have small effect to the output. In these cases, incremental algorithms are a way to efficiently recompute the output with respect to the changes in the input.

In this paper, we present an incremental bisimulation algorithm which, given a graph G, its maximum bisimulation relation P, and the changes (ΔG) in the graph, updates the old bisimulation relation to compute the maximum bisimulation relation with respect to graph ($G \cup \Delta G$). To the best of our knowledge, this is the first algorithm which incrementally recomputes the maximum bisimulation relation.

Our algorithm is based on two algorithms for finding maximum bisimulation relation of a graph, viz. Paige Tarjan's algorithm [16] (abbreviated as PTA) and its recent improvement by Dovier et. al. [6] known as fast bisimulation algorithm or FBA. PTA and FBA solve relational coarsest partition problem which is equivalent to finding maximum bisimulation relation of a graph. We assume that the initial bisimulation relation (P) is computed by FBA. After the changes to the graph G, our algorithm tries to confine the over-approximation that can occur while recomputing P.

The rest of the paper is organized as follows. We formally define the notion of bisimulation and present an overview of PTA and FBA in Section 2. We present our incremental bisimulation algorithm in Section 3. Section 4 demonstrates the effectiveness of the incremental algorithms. We compare the various strategies used by our algorithm with other incremental algorithms in Section 5. We conclude with some direction of future work in Section 6.

2 Preliminaries

In this section we formally describe the notion of bisimulation equivalence and its relation to the *relational coarsest partition problem* (abbreviated as RCPP). We also discuss an algorithm which is closest to our algorithm. Below we define the notion of bisimulation using a graph theoretic view.

Definition 1. *Given two graphs $G_1 = \langle N_1, E_1 \rangle$ and $G_2 = \langle N_2, E_2 \rangle$, a bisimulation between G_1 and G_2 is a relation $b \subseteq N_1 \times N_2$ such that:*

(1) $u_1 \, b \, u_2 \wedge \langle u_1, v_1 \rangle \in E_1 \Rightarrow \exists v_2 \in N_2 (v_1 \, b \, v_2 \wedge \langle u_2, v_2 \rangle \in E_2)$
(2) $u_1 \, b \, u_2 \wedge \langle u_2, v_2 \rangle \in E_2 \Rightarrow \exists v_1 \in N_2 (v_1 \, b \, v_2 \wedge \langle u_1, v_1 \rangle \in E_1)$

Given a graph G there can be many bisimulation relations between G and G. However, we are interested in the maximum bisimulation relation which is unique and always exists. Also the problem of recognizing if two graphs are bisimilar and the problem of determining the maximal bisimulation on a graph are equivalent.

The problem of our interest is that of finding minimum graph bisimilar to a given graph $G(N, E)$. This problem was studied by Kanellakis and Smolka [12] in connection with testing congruence of finite state processes in the calculus of communicating systems (CCS) [14]. They presented an algorithm requiring $O(|E|.|N|)$ time and $O(|E| + |N|)$ space. In [16] Paige and Tarjan solved the relational coarsest partition problem which is equivalent to the maximum bisimulation equivalence problem.

RCPP is described in terms of set theory. Let U be a finite set. A partition P of U is a set of pairwise disjoint subsets of U whose union is all of U. The elements of P are called its blocks. If P and Q are partitions of U, Q is a refinement of P if every block of Q is contained in a block of P. The RCPP is defined as follows: given a partition P of U and a binary relation E on U, find the coarsest partition refinement Q of P such that for each pair of blocks B_1, B_2 of Q, either $B_1 \subseteq E^{-1}B_2$ or $B_1 \cap E^{-1}B_2 = \phi$ (in this case B_1 is called stable with respect to B_2).

Given a graph $G = \langle N, E \rangle$, if Q is a partition of its nodes N, we can obtain a bisimulation relation b as $u\ b\ v$ iff $\exists B \in Q, \{u, v\} \subseteq B$. Also given a bisimulation relation (an equivalence relation) of G, the blocks of the stable partition Q are the equivalence classes. Finding maximum bisimulation of a graph thus corresponds to the finding coarsest partition of the set of nodes in the graph with respect to its edge relation [13].

Our incremental bisimulation algorithm is based on Paige-Tarjan's algorithm and its subsequent improvement by Dovier et. al in [6]. Below we give a brief overview of the algorithms presented in [16] and [6].

Paige Tarjan's Algorithm (PTA). PTA is motivated from the algorithm presented by Hopcroft [10] for solving the problem of minimization of the number of states in a given finite automaton which is equivalent to that of determining the coarsest partition problem stable with respect to a set of functions. Hopcroft's solution is based on *negative* strategy where in each step the blocks of the partition are split if they are not stable. Following this negative strategy which is normal in greatest fixed-point computation, PTA uses a primitive refinement operation called *split* which generalizes the split operation used in Hopcroft's algorithm. For any partition Q and subset $S \subseteq U$, the $split(S, Q)$ is refinement of Q obtained by replacing each block $B \in Q$ such that $B \cap E^{-1}S \neq \phi$ and $B - E^{-1}S \neq \phi$ by the two blocks $B \cap E^{-1}S$ and $B - E^{-1}S$.

However, a straightforward use of splitting strategy where in each step union of some of the blocks of the current partition is used as splitter, yields an algorithm whose time complexity is $O(|E|.|N|)$. Thus the refined algorithm exploits the idea of Hopcroft's "process of smaller half" for better way to find splitters to attain worst-case time complexity $O(|E|log(|N|))$.

Algorithm. Given an initial partition P of U, the algorithm finds a coarsest stable partition Q of P. In addition to the current partition Q, another partition X is maintained such that Q is a partition of X, and Q is stable with respect to every block of X. Initially $Q = P$, and X is the partition containing U as its single block. The algorithm consists of repeating the following steps until $Q = X$.

Step 1: Find a block $S \in X$ that is not a block of Q.
Step 2: Find a block $B \in Q$ such that $B \subseteq S$ and $|B| \leq |S|/2$. Replace S within X by the two sets B and $S - B$; replace Q by $split(S - B, split(B, Q))$.

Fast Bisimulation Algorithm. In [6] Dovier et. al. showed improvement over PTA. Their algorithm, known as FBA, reaches a linear worst case complexity for acyclic graph. They also showed the effectiveness of the algorithm for model checking packages. In the paper the authors proposed a strategy which uses positive ([17]) and negative strategies ([16]) to obtain algorithmic solution to RCPP. The algorithm has the same worst-case complexity as PTA. The initial partition is generated based on a notion of rank where *if two nodes are bisimilar, their ranks must be the same* (converse is not true). Thus using rank, the algorithm divides the graph to an over-approximate of the desired coarsest partition.

In the general case when the graph is not well-founded the ranking is done by SCC decomposition ([6]) of the graph using Kosaraju and Sharir's SCC computation algorithm [22]. To find SCCs in a graph G, the algorithm first traverses G^{-1}, the transpose of G, and gives post-order numbers to the vertices in G. Then it traverses G, starting from the vertex with the highest post-order number; this traversal builds a spanning tree for one SCC of G. Whenever the traversal ends, the algorithm begins a new traversal from the unvisited vertex with the highest post-order number, thereby building a spanning tree for another SCC. This process continues until all vertices have been visited, enumerating all SCCs of G.

For each node n, let $c(n)$ denote the SCC containing node n. The idea is to separate the graph into well-founded and non well-founded parts. The boolean flag $\mathit{WFlag}(u)$ denotes whether the node u is well-founded. The well-founded part ($\mathit{WF}(G)$) is defined as the collection of nodes in G whose transitive closure is acyclic. The other nodes in graph form the non-well-founded part of the graph. Then ranking of each node is defined below.

Definition 2. *Let $G = (N, E)$ and its SCC decomposition graph is given by $G^{scc} = (N^{scc}, E^{scc})$. The rank for each node is defined as follows:*

$$\begin{aligned}
&r(n) = 0 \text{ when } n \text{ is a leaf in } G \quad \textit{[Case 1]}\\
&r(n) = -1 \text{ when } c(n) \text{ is a leaf in } G^{scc} \text{ and } n \text{ is not a leaf of } G \quad \textit{[Case 2]}\\
&r(n) = max(\{1 + r(m) : \langle c(n), c(m)\rangle \in E^{scc}, m \in WF(G) \quad \textit{[Case 3.1]}\}\\
&\qquad\qquad \cup \{r(m) : \langle c(n), c(m)\rangle \in E^{scc}, m \notin \mathit{WF}(G) \quad \textit{[Case 3.2]}\})
\end{aligned}$$

At each stratum defined by the ranking strategy, the algorithm uses PTA or Paige-Tarjan-Bonic algorithm ([17]) to refine the stratum. Then it uses the blocks of this stratum to refine the blocks of higher ranked strata using split operation.

We now present an existing work in incremental bisimulation where an incremental algorithm for maintaining XML structural indices is presented ([25]). The initial index graph is computed using PTA applied to the data graph (XML structure). When addition/deletion of edges are done in the data graph an incremental algorithm which consist of a Split phase followed by a Merge phase is applied to update the index graph. Our incremental algorithm has similar Split and Merge phases. However, one of disadvantage of their incremental algorithm is that it does not compute the coarsest partition when the data graph is cyclic. Thus in a general sense the algorithm is not an incremental bisimulation algorithm as it does not maintain maximum bisimulation. Instead it maintains a partition called maximal bisimulation which coincides with the maximum bisimulation when the data graph is acyclic. The authors have mentioned that in case of maintaining XML structural indices, where most XML structures are acyclic, their algorithm produces the minimum index. Another drawback of their algorithm is that they do not take advantage of FBA when the graph is acyclic which is almost the case for XML data graph.

3 Incremental Bisimulation Algorithm

In this section we present our Split-Merge-Split (SMS) algorithms for incremental maintenance of relational coarsest partition. A non-incremental strategy can incorporate any changes in the graph by recomputing its coarsest partition again using the FBA [6] (from-scratch algorithm). However, such re-computation is often wasteful as small changes to the graph can potentially result into small changes to its coarsest partition. As a result, the entire coarsest partition need not be recomputed. The aim of our incremental algorithms is to identify the parts of the existing coarsest partition that need to be changed, and recompute them.

As the name suggests, the SMS algorithms have three phases, although in some cases the last split phase is not required. Let G be the initial input graph and G' be the new graph after the changes and their corresponding relational coarsest partitions are given as X and X'. Also after Split, Merge, and Split phases of the SMS algorithm, the corresponding partitions obtained be X_1, X_2, and X_3. We use small letters to denote nodes and capital letters to denote blocks, $block(u)$ to denote the block which has the node u, $\rightarrow$ to denote the edge relation among nodes, and $\Rightarrow$ is the edge relation among blocks where an $U \Rightarrow V$ iff $\exists u \rightarrow v$, $block(u) = U$, and $block(v) = V$. We use the notation $\neg U \Rightarrow V$ to denote that no block edge exists from U to V.

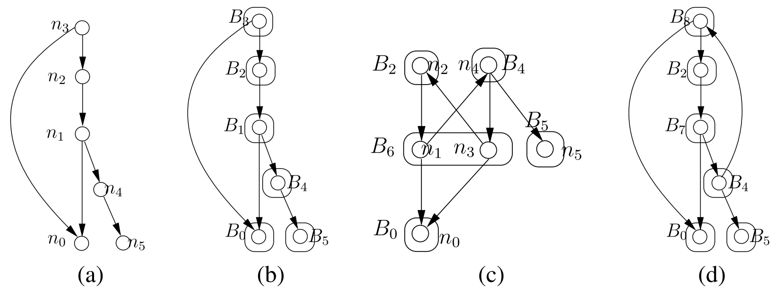
Our single edge addition algorithm SMS-ADD is shown in Figure 1(e). Initially the algorithm checks whether there already exists a block edge between $block(u)$ and $block(v)$ in which case the addition of $u \rightarrow v$ has no effect. The first Split phase of our algorithm is realized using the function `RankedSplit` (Lines 6, 66-74). The algorithm is same as the iterative split strategy of PTA, the only difference being the blocks for splits are chosen in increasing order of ranks. The partition X_1 obtained after the split phase is characterized using the following Lemma.

Lemma 1. *If two nodes are not bisimilar in G' i.e. they belong to different blocks in X', then they belong to the different blocks of X_1.*

The Merge phase performs two operations. Firstly, it incrementally recomputes the ranks and well-founded flags of the nodes. Secondly, it merges the blocks of partition X_1 to obtain the partition X_2 which is characterized using the following Lemma.

Lemma 2. *If two nodes are bisimilar in G' i.e. they belong to same block in X', then they are in the same block of X_2.*

The $r'(u)$ and $WFlag'(u)$ represent the new values of the rank and well-founded flag of node u, respectively. When an edge is added between a non well-founded node and a well-founded node (Lines 9-11), the new rank of u is given by the expression in Line 9. Any changes in the rank of the non well-founded node is propagated to the non well-founded parts of the graph by the function `propagate_nwf(u)` which uses Sharir's SCC decomposition algorithm starting from node u. In contrast, the function `propagate_wf(u)` propagates the change in rank of a well-founded node u to the well-founded parts of the graph in a bottom-up fashion (using topological order of non-updated ranks) and if necessary propagates any changes to the ranks of non-well founded nodes by calling function `propagate_nwf`. The details of these two functions are not provided in this paper.



```
SMS-ADD(node $u$, node $v$)
if ($block(u) \Rightarrow block(v)$)
    return;
Add the edge $u \rightarrow v$ to $G$

RankedSplit($block(v)$)

if $\neg WFlag(u)$ and $WFlag(v)$
  $r'(u) =$ max $\{r(u), r(v)+1\}$
  if $r'(u) \neq r(u)$
    propagate_nwf($u$)
  MergePhase($block(u)$, $block(v)$)
else
  if $r(u) > r(v)$
    MergePhase($block(u)$, $block(v)$)
  else
    $B_u = block(u)$, $B_v = block(v)$
    Visit blocks starting from $u$ in
    $G^{-1}$ between blocks of ranks
    $r(u)$ and $r(v)$. Mark each block
    $B$ as $visited(B)$. Note whether
    it reaches $B_v$ to form a cycle.
    if cycle formed
  $WFlag'(u) = false$
  $r'(u) = re - compute\,rank$
  propagate_nwf($u$)
  MergeAndSplitPhase()
    else
      if $WFlag(u) = true$
        if $WFlag(v) = true$
      $r'(u) = r(v)+1$
      propagate_wf($u$)
    else
          $r'(u) = max\{r(u), r(v)\}$
      propagate_nwf($u$)
  else
    $r'(u) = r(v)$
    if $r'(u) \neq r(u)$
      propagate_nwf($u$)
  MergePhase($B_u$, $B_v$)

propagate_wf($u$)
  Recompute ranks of successor
  of $u$ based on priority of their
  old ranks [in bottom-up order]
  and propagate recursively

MergePhase(block $U$, block $V$)
  $\forall U1, U1 \Rightarrow V$
     if MergeCond($U1$,$U$)
        rec_merge($U1$,$U$)

rec_merge($B1$, $B2$)
   merge the blocks $B1$ and $B2$
   $\forall C1, C1 \Rightarrow B1$, $\forall C2, C2 \Rightarrow B2$
      if (MergeCond($C1$,$C2$))
   rec_merge($C1$,$C2$)

MergeCond($B1$,$B2$)
  $B1$ and $B2$ are not mergeable
  if $label(B1) \neq label(B2)$
  $\vee$ $B1 = B2$
  $\vee$ $r(B1) \neq r(B2)$
  $\vee$ $\exists$ a causal-splitter of $B1$ and $B2$

RankedSplit(block $B$)
   X=P, Q=P
   % P is the current partition
   split(B,Q);
   Until Q=X
     Perform two steps of PTA
     with Step 1: choosing S
     with minimum rank from X that
     is not a block of Q

MergeAndSplitPhase()
  % Merge phase
  Perform DFS on $G$ in order of
  decreasing finishing times of
  the last DFS.
  During the DFS Merge the blocks
  visited using the non-merging
  condition as MergeCond and
  recursively propagate merge as
  shown in function rec_merge
  All the blocks in traversed
  are put in one X partition
  % Split Phase
  Perform PTA in X partition
  and propagate any split using
  RankedSplit

propagate_nwf($u$)
  Perform SCC finding algorithm from
  node $u$ to re-compute non
  Well-Founded ranks
```

(e)

Fig. 1. Example 1 (a, b, c, d); (e) Incremental Addition Algorithm

Note that, the case in Lines 11-13 is the only case where only well-founded flags determine that a new SCC creation is not possible due to addition of the edge, which is also true when $r(u) > r(v)$ (Line 15). In all other cases, the algorithm performs a DFS traversal (Lines 17-22) on G^{-1} to know whether an SCC is formed due to addition of the edge. If the SCC is formed, the algorithm recomputes the rank of the node u based on well-founded flags and ranks of its predecessors using Definition 2. Otherwise, the ranks of the nodes are updated as shown in Lines 29-40. For example, if u and v are both well-founded and $r(u) \leq r(v)$, then $u \rightarrow v$ addition increases the $r(u)$ to $r(v) + 1$ (follows from Case 3.1 of Definition 2). The change is propagated using function `propagate_wf(u)`. The other two cases follow from the Case 3.2 of Definition 2.

The aim of finding new SCC is based on two important reasons, (i) two different merge algorithms are needed based on whether a new SCC is created, and (ii) the last split phase is not required when no new SCC is formed.

When no new SCC is formed, the Merge phase (Function `MergePhase`) of the algorithm considers each of the predecessor blocks of $block(v)$ to merge with $block(u)$. The intuition of this merge is as follows: due to the absence of $u \rightarrow v$, a predecessor block of $block(v)$, say $U1$, which contained u got split into $V1 = U1 \cap E^{-1}\{block(v)\}$ and $block(u)$ using $block(v)$ as splitter. Thus after addition of $u \rightarrow v$, the algorithm needs to reform $U1$ by merging $V1$ and $block(u)$. Due to this merge, their predecessor blocks may also get merged. However, it is not always possible to merge two blocks B and B' as the blocks need to have the same labels and ranks (in the updated graph). Also if there exists a block C which has a predecessor block same as exactly one of blocks B and B' then blocks B and B' should not be merged (see Function `MergeCond`). The block C is called causal-splitter of the blocks B and B', and is formally defined below.

Definition 3 (Candidates for causal-splitter). *A block C is called a causal-splitter of block B and B', if*

- *$B \Rightarrow C$ and $\neg B' \Rightarrow C$, or $B' \Rightarrow C$ and $\neg B \Rightarrow C$.*
- *C is a block in the partition X'.*

When no new SCC is created due to the addition of the edge, the second condition of the causal-splitter trivially holds as the blocks are merged from lower ranked strata to higher ranked strata, and causal splitters are chosen from already stabled lower ranked strata. However, in general, the causal-splitter block may get affected due to the transitive effect of merging blocks B and B'. If due to the propagation of merging of B and B', C gets merged with C', then the condition of having predecessor block edge to exactly one B and B' may no longer hold. This is only possible when addition of edge creates a new SCC in the updated graph, in which case judicious selection of causal-splitters is required, a case explained in more detail with the following example.

Consider the graph in Figure 1(a) with labeling set $\{\{n_0\}, \{n_1, n_3, n_5\}, \{n_2, n_4\}\}$, initial partition in Figure 1(b), and addition of a new edge $n_4 \rightarrow n_3$. As the rank of n_4 is 1 and that of n_3 is 4, an SCC can be potentially formed because of the addition.

In the first split phase, as block B_4 only contains a single node, it is not split. The split phase ends here as no further splits are possible. The first DFS traversal of blocks from $block(u)$ in G^{-1} till the ranked stratum containing $block(v)$ (in this case blocks B_1, B_2, B_3, B_4) confirms creation of a new SCC. Next, the ranks of the nodes n_1, n_2, n_3, n_4 are updated to 1. Then the function `MergeAndSplitPhase` determines the new SCC in the second DFS on G. At the finish time of second DFS of each block, it is tried to merge it with other visited blocks of the SCC. For each label a list of blocks with that label is maintained where the blocks cannot be merged with each other. Firstly, B_1 is put to label-1 list. Then, B_2 is put to the label-2 list. Next, B_3 is considered for merging with B_1 as it has the same label as B_1. Note that, $B_1 \Rightarrow B_2$ and $\neg B_3 \Rightarrow B_2$, and $B_1 \Rightarrow B_4$ and $\neg B_3 \Rightarrow B_4$. But as B_2 and B_4 are marked during the first DFS visit, each of them can be potentially merged to some other visited blocks and thus can be potentially changed. For example, blocks B_2 and B_4 can be potentially merged and in that case none of them should be used as a causal-splitter. Thus B_3 and B_1 are merged to obtain a block B_6. However, as there exist blocks that can be potential causal-splitters, we are introducing over-approximation in the merge phase.

The above discussion hints at a strategy for selecting a causal splitter which preserves the second condition of causal-splitter. A block is selected as causal-splitter if it is *not visited* in the first DFS as it is not going to be affected because of the addition. This is the case when the next block B_4 is considered for merging. Although it has same label as B_2, due to the existence of the causal-splitter B_5 it is not merged with B_2. The resultant partition is shown in Figure 1(c). Although not shown in this example, the effect of merging two blocks may lead to merging of their predecessors blocks in the unvisited region of first DFS.

It can be proved that in case where an addition of an edge to a graph does not create a cycle, we do not require the last split phase of SMS algorithm. The reason is that the merging done in merge phase is not an over-approximation. In general, the merge phase can cause over-approximation of merging which is rectified in the last split phase. The PTA is run on those visited blocks and propagate the splits strata-by-strata. The final partition is shown in Figure 1(d). The below theorem expresses the correctness of the algorithm.

Theorem 1. *The partitions X_3 and X' are equal.*

Single Edge Deletion: The single edge deletion algorithm (SMS-DEL) has the similar Split, Merge, Split phases like the SMS-ADD algorithm. They differ only in the rank re-computation part and in the merging phase where after recomputing ranks if a block's rank is changed to 0, it is merged with the other block of rank 0.

Consider deletion of the edge $n_4 \rightarrow n_5$ after addition of edge $n_4 \rightarrow n_3$ in example in Figure 1(a). The first split phase is ineffective. In the merge phase, Sharir's SCC computation algorithm is performed to update any rank, and merge all the blocks in the same rank as u and reachable to u. Note that unlike in the case of addition, the blocks of nodes n_1 and n_3, n_2 and n_4 are merged as the connection to the causal-splitter block is deleted. This also serves as an example where the resultant partition of the merging phase is the final partition.

Our SMS algorithm can be adapted to multiple edge addition and deletion, subgraph addition and deletion, and update. These algorithms have the same three phases and DFS traversal where each phase and DFS traversal need to be done for *all changes* before starting processing of other phases. The main difference lies in the computation of ranks. Due to want of space we do not discuss these algorithms here.

Complexity. The complexity of the first split phase, rank re-computation, merge phase, last split phase are $O(|E_1|log(|N_1|)$, $O(|\Delta_{WF}|log(|\Delta_{WF}|)+(|E_{nwf}|+|N_{nwf}|))$, and

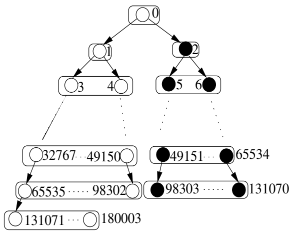
$O(|E'||N'|)$, and $O(|E'|log(|N'|))$ respectively. In the above expressions Δ_{WF} is the set of well-founded nodes whose ranks got changed, (N_1, E_1) and (N', E') are the subgraph of the initial graph $G = (N, E)$ whose blocks got split and merged respectively, and (N_{nwf}, E_{nwf}) is the non-well founded subgraph of G.

4 Experimental Results

We measured the performance of our algorithms by implementing those on top of the source code available from one of the author's website of [6]. The data structures used in their implementation was not changed. We ran our algorithms on benchmarks mentioned in various works for measuring effectiveness of bisimulation problems. Performance measurements were taken on a PC with 1.4Ghz Intel Core Duo processor with 512MB of physical memory running Windows XP.

We present the performance result of our insertion and deletion algorithm on the synthetic benchmarks described below. Each benchmark has different characteristics which have different effects on our algorithm. In these two benchmarks, we noted the average (over all edges) incremental deletion and insertion time as percentage of from-scratch time to be 10%. The results below will highlight the range of these timing results and reason for such distribution. We used an extra priority queue data structure apart from the data structure of FBA implementation, but it uses the memory of FBA. So our algorithm does not incur any extra memory overhead compared to FBA.

Benchmark 1. *Simple Binary Tree.* This benchmark (Benchmark 2 of [6]) consists of a binary tree with 262143 nodes and has two different labels for left and right subtree as shown in Figure 2(a) with node numbers and initial blocks. The initial FBA time is 0.3s. The height of each node gives the ranks. We added one edge and took the incremental time, and compared it with the time taken by FBA for the changed graph. We also show the time for SMS-DEL to delete the added edge. Thus SMS-DEL was not tried on Benchmark 1 but on a graph that results after an added edge to the benchmark. We provide the edges which showed the minimum and maximum time taken by SMS-ADD for three different cases based on relation of ranks and whether the added edge produces



(a)

Edge	$r(u) > r(v)$		$r(u) \leq r(v)$[no cycle]		$r(u) \leq r(v)$[cycle]	
Addition	Min	Max	Min	Max	Min	Max
$u \rightarrow v$	0.01	7.87	0.01	8.22	1.00	20.13
(u,v)	(1,5)	(98302,196607)	(1,2)	(196606, 98303)	(4,1)	(196606,1)
Deletion	1.52	7.10	1.48	2.96	1.44	4.69

(b)

Edge	$r(u) > r(v)$		$r(u) \leq r(v)$[no cycle]		$r(u) \leq r(v)$[cycle]	
Addition	Min	Max	Min	Max	Min	Max
$u \rightarrow v$	0.54	0.54	0.25	12.66	1.00	20.00
(u,v)	Any	Any	(4,2)	(32767, 2)	(6,2)	(65534,2)
Deletion	0.54	0.54	16.00	1.07	27.00	28.00

(c)

Fig. 2. (a) Benchmark 1. Tree; (b) & (c) Incremental times as % or From-scratch times for Benchmark 1 and 2 respectively a new cycle or not. The result is shown in Figure 2(b). As expected SMS-ADD takes maximum time in case the addition of edge creates cycles. Most of time in this case is attributed to the Merge phase. Note that localized addition yields lesser time than the non-local changes.

Benchmark 2. This benchmark is a downward closed tree (Test 2 of [7]) of 65535 nodes obtained by closing downward a binary tree using the rule: if $\langle m, n\rangle$ and $\langle n, p\rangle$ are edges then add a new edge $\langle n, p\rangle$ and two different labels are put to the alternate nodes in each ranked strata of the tree. The initial FBA time is 0.5s. The result for this benchmark is shown in the Figure 2(c). Note that addition of edge $\langle 65534 \rightarrow 2\rangle$ takes 20% of from-scratch time and this time is spent on `MergeCond` function which checks for causal splitter which in turn is due to large number of out-degree of each node. When deletion occurs for the same edge, it takes 28% times of the from-scratch time. Deletion of edge (65534,2) will first merge the block of node 65534 with the blocks which consists of rest of even numbered nodes in rank 0. To propagate the effect of this merging the `rec_merge` function checks all nodes which are predecessors of the nodes in the block of rank 0. As there are large number of such edges to be considered the Merge phase takes large amount of time. This high overhead of Merge phase is attributed to the data structure selection in our implementation. If we keep block edges in our implementation then merge time is reduced; however, in that case Split phase time is increased. We use memoization technique to reduce some overhead for not having the block edges.

The above benchmark in-fact serves as an extreme case of overhead of the merge phase for single change. In most of VLTS benchmarks ([4]) the in-degree and out-degree of nodes are comparably less than this benchmark. On average the SMS algorithm took 3.94% of from-scratch time for VLTS benchmark vasy_386_1171 on 400 random deletion of edges. For 400 random insertion of edges (for each case one edge was not loaded initially and has been incrementally added), the SMS algorithm took 6.93% of from-scratch time for the same benchmark.

We note that for multiple changes in the graph which affect independent parts of the initial partition, the overhead of the merge phase can accumulate to exceed the from-scratch time. Thus it is not possible for our incremental algorithms to perform always better than the from-scratch algorithm when multiple changes are present.

5 Related Work

An important characteristic of incremental bisimulation problem is that adding or deleting an edge in the input graph can potentially result in splitting and merging of blocks in the partition. Thus incremental bisimulation problem is non-monotonic in nature. This is in contrast to the incremental algorithms in many works in view maintenance ([8]), logic programming ([18]), model checking ([23]), where the effect of addition and deletion is monotonic in nature. The problem is also different in nature to incremental functional programming ([1]) where changes can be propagated using in-place updates. Also incremental bisimulation problem cannot be reduced to incremental evaluation of logic programs with stratified negation as the nature of non-monotonism in incremental bisimulation resembles to non-stratified negation in logic programming. The only work we are aware of incremental evaluation of logic programs with non-stratified negation is in [20]. The logic program encoding ([2]) of bisimulation involves a builtin *findall* and with our earlier experience showed that the incremental algorithms do not produce great efficiency when builtins exist.

The idea of having different phases to overapproximate or underapproximate fixpoint before converging to the new fixpoint is not new. Generally in incremental least fix-point (positive strategy) computation, the first phase is a deletion phase (or negative strategy) which is used to bring the incrementally computed fix-point equal or below the final fixpoint, and second phase is used to converge to the final fixpoint ([8,18,23]). For incremental greatest fixpoint computation (negative strategy) the first phase uses the positive phase which is used to bring the current fixpoint above the final fixpoint point ([24]) in the fix-point lattice. In our case, as the from-scratch algorithm (FBA) uses split which is a negative strategy; a positive (merge) followed by a negative strategy (split) will suffice. However, we have incorporated a split phase before the merge-split phase to reduce the size of the blocks that are merged as merge operation is expensive.

We have used several strategies like labels, ranks, and causal splitter to reduce the overapproximation done in the merge phase. The ranks define regions such that blocks can only be merged within each region. The idea of regions is used in other incremental algorithms ([21]) where it is typically used to nullify effect of additions and deletions in each region before propagating the effect to other regions. The idea of finding causal-splitter which is not cyclically dependent on the blocks to be merged to restrict merge propagation is similar to the idea of primary and acyclic support used for restricting deletion propagation in incremental pointer analysis ([19]).

6 Conclusion

In this paper we presented an incremental algorithm to recompute maximum bisimulation relation. We demonstrated the effectiveness of the algorithm on several graph examples. In future we will incorporate our implementation to model checkers and XML database management system. The SMS algorithm presented here *globally* recomputes bisimulation relation. We plan to extend our solution to local bisimulation computation, and to infinite and symbolic graph structure.

Acknowledgment

We thank anonymous reviewers, Prof. Ranjit Jhala, Anu Singh, and C. Manjari for their comments to improve the quality of this paper. We are also grateful to Prof. C. R. Ramakrishnan and Dr. Subir Saha for encouraging this work.

References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: ACM POPL, New York, NY, USA, vol. 37, pp. 247–259. ACM Press, New York (2002)
2. Basu, S., Mukund, M., Ramakrishnan, C.R., Ramakrishnan, I.V., Verma, R.M.: Local and symbolic bisimulation using tabled constraint logic programming. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 166–180. Springer, Heidelberg (2001)

3. Benthem, J.V.: Modal Correspondence Theory. PhD thesis, University van Amsterdam (1976)
4. CADP. Caesar/aldebran developement package c1.112, Available at (2001), `http://www.inrialpes.fr/vasy/cadp.html`
5. CWB-NC. The concurrency workbench of new century v1.1.1, Available at (2001), `http://www.cs.sunysb.edu/~cwb`
6. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theor. Comput. Sci. 311(1-3), 221–256 (2004)
7. Dovier, A., Piazza, C., Policriti, A., Ugel, N.: A fast bisimulation algorithm: Test, `http://www.dimi.uniud.it/∼piazza/bisim/web.ps`
8. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: ACM SIGMOD, pp. 157–166 (1993)
9. Hennessy, M., Lin, H.: Symbolic bisimulations. Theor. Comput. Sci. 138(2), 353–389 (1995)
10. Hopcroft, J.E.: An $nlogn$ algorithm for minimizing states in a finite automaton. In: Theory of Machines and Computations, pp. 189–196. Academic Press, London (1971)
11. Hudson, S.E.: Incremental attribute evaluation: a flexible algorithm for lazy update. ACM Transaction of Programming Languages and Systems 13(3), 315–341 (1991)
12. Kanellakis, P.C., Smolka, S.A.: CCS expressions, finite state processes, and three problems of equivalence. In: PODS, pp. 228–240. ACM Press, New York (1983)
13. Kanellakis, P.C., Smolka, S.A.: CCS expressions finite state processes, and three problems of equivalence. Inf. Comput. 86(1), 43–68 (1990)
14. Milner, R.: A Calculus of Communicating Systems, Secaucus, NJ, USA. Springer, Heidelberg (1982)
15. Milner, R.: Operational and algebraic semantics of concurrent processes, 1201–1242 (1990)
16. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
17. Paige, R., Tarjan, R.E., Bonic, R.: A linear time solution to the single function coarsest partition problem. Theor. Comput. Sci. 40, 67–84 (1985)
18. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled logic programs. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 389–406. Springer, Heidelberg (2003)
19. Saha, D., Ramakrishnan, C.R.: Incremental and demand-driven points-to analysis using logic programming. In: ACM Conference on Principles and Practice of Declarative Programming, ACM Press, New York (2005)
20. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled prolog: Beyond pure logic programs. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 215–229. Springer, Heidelberg (2005)
21. Saha, D., Ramakrishnan, C.R.: A local algorithm for incremental evaluation of logic programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 56–71. Springer, Heidelberg (2006)
22. Sharir, M.: A strong connectivity algorithm and its application in data flow analysis. Computer and Mathematics with Applications 7(1), 67–72 (1981)
23. Sokolsky, O.V., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 351–363. Springer, Heidelberg (1994)
24. Swamy, G.: Incremental Methods for Formal Verification and Logic Synthesis. PhD thesis, University of California at Berkeley (1996)
25. Yi, K., He, H., Stanoi, I., Yang, J.: Incremental maintenance of XML structural indexes. In: SIGMOD, pp. 491–502. ACM Press, New York (2004)