Updates for Structure Indexes

Raghav Kaushik* Univ. of Wisconsin Madison

Philip Bohannon Lucent Technologies Bell Laboratories

Jeffrey F Naughton Univ. of Wisconsin Madison

Pradeep Shenoy* Univ. of Washington Seattle

Abstract

The problem of indexing path queries in semistructured/XML databases has received considerable attention recently, and several proposals have advocated the use of structure indexes as supporting data structures for this problem. In this paper, we investigate efficient update algorithms for structure indexes. We study two kinds of updates — the addition of a subgraph, intended to represent the addition of a new file to the database, and the addition of an edge, to represent a small incremental change. We focus on three instances of structure indexes that are based on the notion of graph bisimilarity. We propose algorithms to update the bisimulation partition for both kinds of updates and show how they extend to these indexes. Our experiments on two real world data sets show that our update algorithms are an order of magnitude faster than dropping and rebuilding the index. To the best of our knowledge, no previous work has addressed updates for structure indexes based on graph bisimilarity.

1 Introduction

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a labeled-tree or labeled-graph data model. To summarize the structure of such data in the absence of a schema and to support path expression evaluation, novel *structural indexes* [5, 10, 11] have been proposed for semi-structured data. Unlike a schema, structure indexes are not prescriptive and thus may change with any update. Generalizations of these structures have gained increasing attention recently, as flexible index structures for XML [6, 15, 16], and size and performance issues in the original proposals have been addressed [16]. In addition, the ideas behind these structure indexes have been used as statistical synopses for estimating path expression selectivity [2, 12].

Obviously, a significant part of implementing these ideas in practice is keeping these indexes consistent with the underlying data (without having to rebuild them after each update). However, with the exception of the Strong DataGuide (which is, in the worst case, exponential in the size of the base data), we are aware of no discussion of incremental update algorithms for structure indexes. We observe that the update problem for these structures has more in common with path indexes than normal value indexes, since the impact of a small update can be arbitrarily large, in terms of the change in the size of the index.

Several structure indexes are based on some variant of the notion of graph bisimilarity. In this paper, we propose efficient update algorithms for the bisimulation partition. In particular, we propose algorithms for two classes of updates: 1) the addition of a disjoint subgraph of data, intended to represent the addition of a new file to a database, and 2) the addition of an edge, to represent a small incremental change. We present experimental results on two large, real-world data sets that show that these algorithms are 50 to 80 times faster than rebuilding the index.

We then discuss how these algorithms extend to three representatives of structure indexes based on bisimilarity as follows.

*Work done when author was visiting Bell Labs

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

- 1-Index [10]: This is directly defined from the graph bisimulation partition. It is designed to cover simple path expressions.
- F&B-Index [15]: This index is also based on bisimulation, but is extended to work for branching path expressions (e.g. XPath expressions).
- A(k)-index [16]: This is based on a variant of the bisimulation concept and addresses size and performance problems with the 1-Index. While our algorithm to add a subgraph generalizes to this case, the edge addition algorithm does not directly apply.

Our success in extending the basic update algorithms to these cases suggests that it will be broadly applicable to new, but related, structures.

The rest of the paper is organized as follows. Background material is presented in Section 2. Update algorithms for the bisimulation partition are discussed in Sections 3 and 4. This directly applies to the 1-Index. Extensions to the F&B-Index and the A(k)-index are discussed in Sections 5 and 6. Section 7 reviews the results of our experimental evaluation. We conclude in Section 8.

1.1 Related Work

The only known update algorithm in the context of structure indexes is the algorithm proposed in [5] which maintains the Strong DataGuide. The strong DataGuide can be computed by interpreting the data graph as a non-deterministic automaton and obtaining an equivalent deterministic automaton [1]. All indexes we focus on in this paper, based on graph bisimulation, are non-deterministic when thought of as automata. As a result, the update algorithms for the Strong DataGuide do not generalize directly to apply in this context. To the best of our knowledge, no previous work has addressed updates for structure indexes based on graph bisimilarity.

2 Background

For the purposes of this paper, the distinction between tree and non-tree edges, or IDREF edges is not crucial. Hence, we model XML or other semi-structured data as a directed, labeled graph $G = (V_G, E_G, root, \Sigma_G, label, oid, value)$. Each edge in E_G indicates an object-subobject or object-value relationship. "Simple" nodes in V_G have no outgoing edges and are given a value via the *value* function. Each node in V_G is labeled with a string-literal from Σ_G via the *label* function and with a unique identifier via the *oid* function, with simple objects given the distinguished label, VALUE. There is a single *root* element with the distinguished label, ROOT. We note that our model differs little from other semi-structured or XML data models [1, 4, 8]. When we have a database with multiple XML documents, the database consists of a single graph with an artificial root under which lie the graphs corresponding to the individual files.

Example 1: *Figure 1 shows a portion of a hypothetical "metro-guide", represented as a data graph. In the figure, the numeric identifiers in nodes represent oid's. Such a guide could reasonably be built from a collection of XML documents published by businesses, civic groups and other interested parties. Non-tree edges may be implemented with the* ID/IDREF *construct or XLink [3] syntax. Every neighborhood points to the business and cultural objects physically located in it.*

A structural summary for the data takes the form of another labeled, directed graph. The idea is to preserve all the *paths* in the data graph in the summary graph, while having far fewer nodes and edges. In addition to other functions, structural summaries can also be used as *index graphs* to aid in evaluating path expressions. A structural summary can generally aid query answering by associating an *extent* with each node in the summary to produce an *index graph*. If A is a node in an index graph, $I(G)$, then $ext^I(A)$, the extent of A, is a subset of V_G. The *index graph result* of executing a path expression R on $I(G)$ is the union of the extents of the index nodes that match R. We require that the extent mapping be *safe*, that is, that the result of any path expression R on G is *contained* in the result of R on $I(G)$. An index graph is said to be *precise* if the converse holds. For example, an index graph that is safe for simple path expressions has the property that if $l_1.l_2 \ldots l_k$ is a label path which matches a path to node v in G, then there is some node A in $I(G)$ for which $l_1.l_2 \ldots l_k$ matches a path to A and $v \in ext^I(A)$. Similarly, an index graph that is precise for simple path expressions has the property that if $v \in ext^I(A)$ and $l_1.l_2 \ldots l_k$ is a valid label path for A, then $l_1.l_2 \ldots l_k$ is a valid label path for v.

In general, any partition of the data nodes defines an index graph where we (1) associate an index node with every equivalence class, (2) define each index node's extent to be the equivalence class that formed it and (3) add an edge from index node A to index node B if there is an edge from some data node in $ext(A)$ to some data node in $ext(B)$. Henceforth, whenever we refer to an index graph obtained from

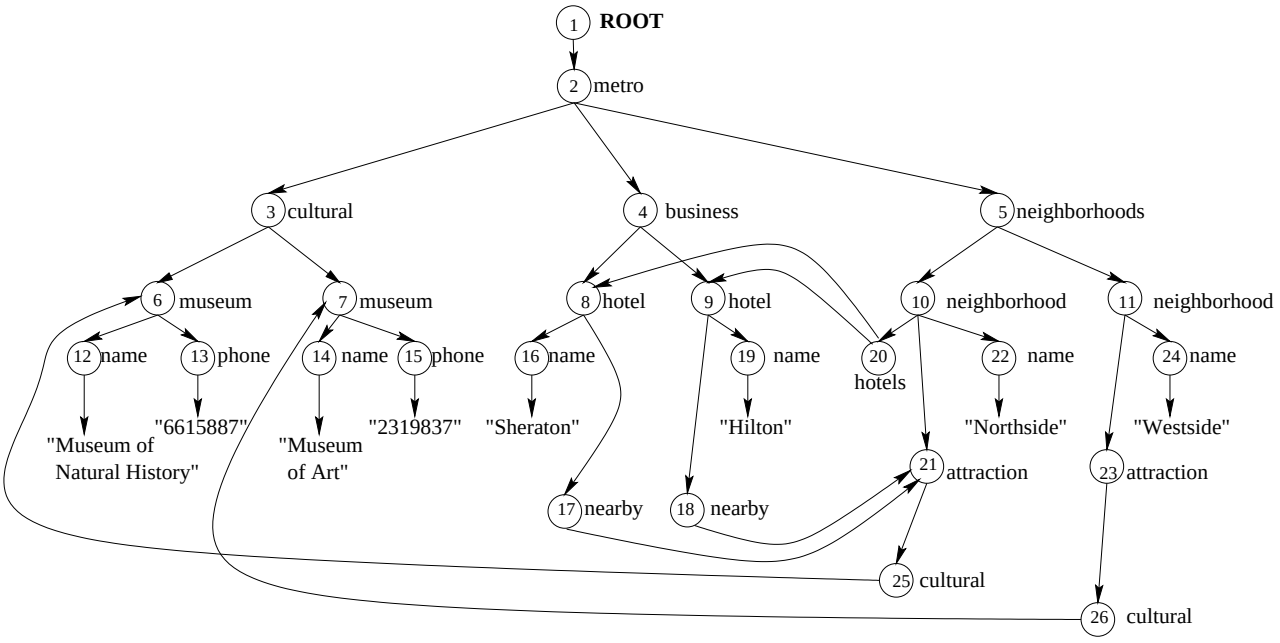


Figure 1: **An example graph-structured database**

a partition of the data nodes, we mean the above construction. Thus, even a simple grouping of the data nodes by label defines an index graph. All the structure indexes we consider in this paper are instances of index graphs.

We now introduce terminology about partitions of data nodes. A partition P_1 of the data nodes is a *refinement* of another partition P_2 if the following condition holds: whenever two nodes are in the same equivalence class in P_1, they are in the same equivalence class in P_2 as well. If P_1 is a refinement of P_2, then P_2 is *coarser* than P_1. We also talk about one index graph being a *refinement* of another — this refers to the corresponding partitions.

2.1 Bisimilarity

As mentioned earlier, all structure indexes we consider are based on the notion of graph bisimilarity which we now define.

Definition 1 *A symmetric, binary relation $\approx$ on V_G is called a* bisimulation *if, for any two data nodes u and v with $u \approx v$, we have that*

1. *u and v have the same label, and*
2. *if u' is a parent of u, then there is a parent v' of v such that $u' \approx v'$, and vice-versa.*

Two nodes u and v in G are said to be bisimilar, *denoted by $u \approx^b v$, if there is some bisimulation $\approx$ such that $u \approx v$.*

For example, in Figure 1, objects 8 and 9 (the hotel nodes) are bisimilar, while objects 21 and 23 (labeled attraction) are not, since 21 has a parent labeled nearby. By extension, objects 25 and 26 (the nodes immediately below them, labeled cultural) are also not bisimilar, since their unique parents are non-bisimilar. An easy induction shows that if two nodes are bisimilar, the set of in-coming paths into them is the same.

The partition of V_G induced by $\approx^b$ defines an index graph, referred to as $Bisim(G)$ or simply "the 1-Index" in this paper[1]. Thus, there is a worst case guarantee on the index size, since the 1-Index can never be bigger than the data graph. Further, it can be computed in time $O(m \lg n)$ where n is the number of nodes and m is the number of edges in the data graph, using an algorithm proposed by Paige and Tarjan [13], which we review in Section 4.3.

We next present the general problem of updating the bisimulation partition, which directly corresponds to the 1-Index, and then extend the ideas to the A(k)-index and the F&B-Index in later sections.

3 Subgraph Addition

Our view of the data is that of a set of XML files. In this case, one natural class of updates is the addition of a new file to the database. In our model, this

[1]The authors of [10] also consider the use of the *similarity* relationship [9] for the 1-Index. We do not consider this alternative due to inefficient construction algorithms for the similarity relation – see [10] for details.

procedure subgraph-add(G,H)
//Graph H added under root of G
begin
1. Let $I \leftarrow$ 1-Index of G
2. Compute the 1-Index of H. Let it be I_H
3. Add I_H as a subgraph under the root of I. Let this graph be I'
4. Treat I' as a data graph and compute its 1-Index. Let it be I_{new}
5. Set the extents of the nodes of I_{new} by "blowing up" the current extents
6. return I_{new}

end

Figure 2: **Addition of a subgraph**

corresponds to the addition of a subgraph under the root.

Suppose we have a database of XML documents on which the 1-Index is already built and a new document is added to the database. Let us assume that there are no inter-document references. We are interested in finding the new 1-index without having to recompute it from scratch. In this section, we state a theorem that enables us to compute the modified index from the old index and the index computed on the new file — without having to look at the whole of the current data.

Let the data graph corresponding to the database before the addition of the new file be G, the 1-index be I_G and let the addition of the new file correspond to the addition of a new subgraph H under the (artificial) root.

If we compute the 1-Index I_H on the new file and add I_H as a subgraph under the root of I_G, what we have, I', is a refinement of the actual 1-Index, I_{new}. The following theorem enables us to compute I_{new} from I'. The proof is attached in the appendix.

Theorem 1: *Let G be a data graph. Let $Bisim(G)$ be the 1-Index constructed from the bisimulation relation and $Bisim^{ref}(G)$ be the index graph constructed from any refinement of $Bisim(G)$. Then, $Bisim(Bisim^{ref}(G)) = Bisim(G)$. Here, graph equality means isomorphism.*

Thus, we have an algorithm to find the new 1-Index without having to recompute it from scratch. We sketch this algorithm in Figure 2. When we compute I_{new} from I', the index extents have nodes of I'. Thus, in order to obtain the original data nodes corresponding to an extent, we need to "blow up" the nodes of I' with their respective extents which consist of data nodes. Figure 3 illustrates how this algorithm works. The dashed edges represent the addition of the new subgraph to the current database.

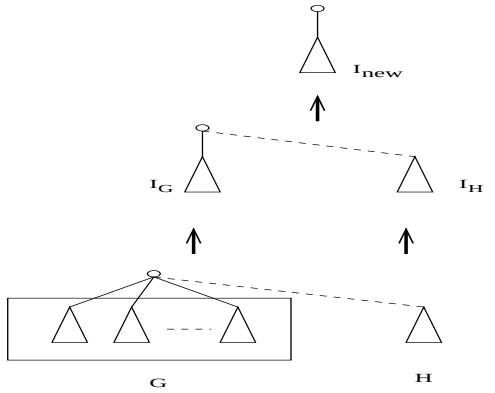


Figure 3: **Addition of subgraph - illustration**

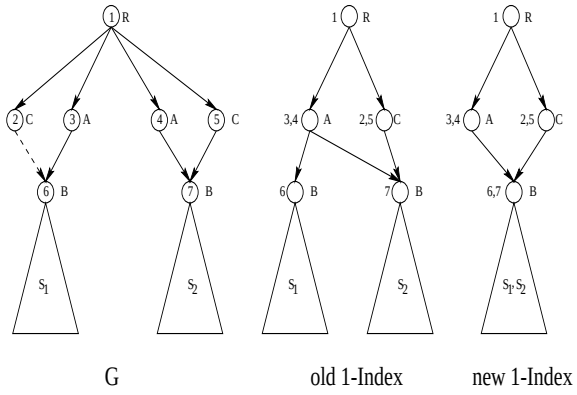


Figure 4: **Example for expensive update**

Let the number of nodes in I_G, H and I_H be n_{IG}, n_H and n_{IH} respectively, and the number of edges be m_{IG}, m_H, m_{IH} respectively. The time taken by subgraph-add is $O(m_H log(n_H) + (m_{IH} + m_{IG}) log(n_{IH} + n_{IG}))$. This is independent of the size of G, i.e, the total size of the database. Instead, it is dependent on the size of I_G which is often much smaller.

4 Edge Addition

We now present the edge insertion algorithm for the bisimulation partition. An edge insertion represents a small incremental change in the data.

4.1 Impact of an Edge Insertion

There is a crucial difference between updates to structure indexes and traditional value indexes — the impact of an update on a structure index can, in general, be arbitrary. A small change in a graph can trigger large changes in an index graph, a fact we now illustrate with an example.

Example 2: *Consider the data graph G in Figure 4. The two sub-trees S_1 and S_2 under the two*

"B" nodes are identical. The 1-Index is shown adjacent to G. The numbers beside the nodes indicate the extents. The 1-Index does not group nodes in S_1 and S_2 since the two B-labeled nodes are not bisimilar (one has a parent labeled C, the other does not), and hence neither are corresponding descendants. Suppose an edge is added between nodes 2(labeled C) and 6(labeled B) (shown with the dotted edge) and G changes to G′. Now, nodes 6 and 7 become bisimilar and so, the two sub-trees get merged in the new bisimulation based 1-Index. Thus, the size of the index graph reduces by about 50%, if the two sub-trees S_1 and S_2 are large.

We now introduce our update algorithm. Let us note here that the 1-Index is a precise index for simple path expressions (i.e, no branches), and so in this section when we talk about precise index graphs, we mean precise for simple path expressions.

Let data graph G be updated by adding an edge from node u to node v. Let the modified data graph be G'. Let the bisimulation based 1-index on G be $Bisim(G)$ and on G' be $Bisim(G')$. The update algorithm we propose, called *propagate*, begins by taking $Bisim(G)$ and outputs some refinement of $Bisim(G')$, $Bisim^{ref}(G')$. The corresponding binary relations on the data nodes are denoted by $\approx^{max}_{G'}$ and $\approx^{ref}$. Notice that for query processing, any refinement of the bisimulation partitioning yields a precise index. Thus, $Bisim^{ref}(G')$ by itself is precise. The actual bisimulation partitioning can then be computed by using the special property of the bisimulation relation stated in Theorem 1.

Further, if the refinement, $Bisim^{ref}(G')$, is not much larger than $Bisim(G')$, a *lazy* approach can be taken to index minimization since the refinement maintains all the properties necessary for correct functioning of the index graph. We will revisit this point later.

While we discuss edge addition, our algorithm generalizes to edge deletion as well.

4.2 Computation of a Refinement

This phase of the algorithm is based on an alternative definition of bisimilarity through the notion of *stability*.

Definition 2 *Given two sets of data graph nodes A and B, A is said to be stable with respect to B if either A is a subset of Succ(B) or A and Succ(B) are disjoint. Here, Succ(B) denotes the successors of the nodes in the set B, i.e,* $\{v : \exists u \in B$ *such that there is an edge from u to v*$\}$.

Thus, if A and B are nodes in an index graph, A is stable with respect to B, and there is an edge from B to A, then every data graph node in the extent of A has a parent in the extent of B. This property is crucial for precision of an index graph. In the above, when talking about stability of index nodes, we actually refer to the stability of their extents. If A is unstable with respect to B, we can *stabilize* it by *splitting* A into two nodes A_1 and A_2 where the extent of A_1 is $ext(A) \cap Succ(ext(B))$ and the extent of A_2 is $ext(A) - ext(A_1)$.

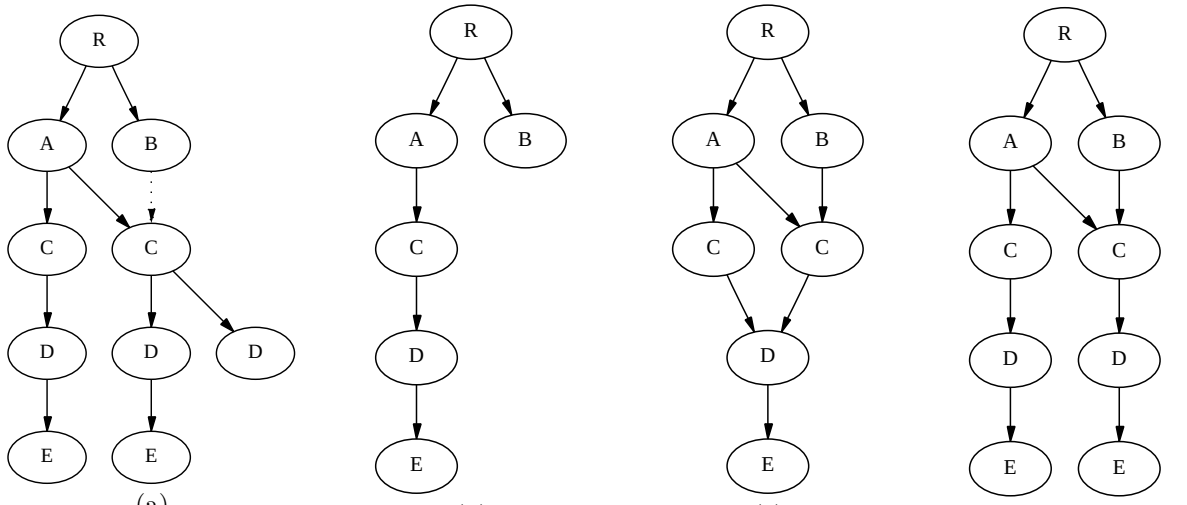
We call a partition of the nodes *stable* if, for every pair of equivalence classes p_1 and p_2, p_1 is stable with respect to p_2. The coarsest partitioning of the nodes of a graph that 1) is constrained not to group together nodes of different labels, and 2) is stable, is the maximal bisimulation. In other words, any stable partitioning of the data nodes is *either equal to or is a refinement of* the maximal bisimulation. This is the key observation used by this phase of the *propagate* algorithm.

Now, the original bisimulation partitioning is stable. By adding an edge from u to v, the index node containing v in its extent, $I[v]$, may no longer be stable with respect to $I[u]$. To resolve this, *propagate* first checks if there is already an edge from $I[u]$ to $I[v]$. If so, the stability condition is ensured and *propagate* returns. If not, it removes v from the extent of $I[v]$ and creates a new index node I' with only v in its extent.

Now, the index nodes containing children of v may not be stable with respect to the rest of the index nodes. If this is indeed the case, the children of v are pulled out of the index nodes containing them and put into new index nodes. Here, if two children of v were earlier bisimilar, *propagate* tries to put them into the same index node. It then proceeds to the grand-children of v and so on. This procedure terminates when the current partition is stable. We illustrate this with an example.

Example 3 *Consider the data graph G shown in Figure 5(a). Suppose an edge is added as shown by the dotted line in the figure. Let the source node be src and the destination node be dst. The old bisimulation based 1-index is shown in Figure 5(b). For clarity, the extents are not shown. The insertion of this edge causes the index node labeled C to become unstable, since only one of the C labeled nodes in G has a parent labeled B.*

Thus, the node labeled C splits and we obtain an intermediate index graph as shown in Figure 5(c). This split means that the index node with label D is now unstable, and so needs to be split. During the split of the index node labeled D, the two children of dst can be kept together since they are still bisimilar.



(a) Data Graph (b) Original 1-index (c) In the middle of *propagate* (d) Final 1-index

Figure 5: **Execution of** *propagate* **on an example**

propagate *groups together data nodes by extent in this manner whenever possible. The splits propagate in this manner till the leaf level is reached. Thus, the final index output by* propagate *is as shown in Figure 5(d). In general, if the graph is strongly connected, the termination condition is stability.*

Note that in this example, the final 1-index obtained after *propagate* happens to be the actual 1-index of the updated data graph. This need not be true in general.

To implement this efficiently, we use a variant of the Paige-Tarjan (PT) algorithm for bisimulation computation [13], which we review next.

4.3 Review of Bisimulation Computation

In order to explain our update algorithm, we need to understand how the bisimilarity partition is computed. The algorithm discussed below was proposed by Paige and Tarjan [13].

Their algorithm maintains two partitions of the data nodes, $\mathcal{X}$ and $\mathcal{Q}$ with the following invariants.

1. $\mathcal{Q}$ is a refinement of $\mathcal{X}$
2. $\mathcal{Q}$ is stable w.r.t $\mathcal{X}$

The details are shown in Figure 6. A *compound_X_block* is an X partition such that it is properly refined in $\mathcal{Q}$. In each iteration, some compound_X_block X is chosen. One of its component Q partitions *that has fewer than half* of the data nodes in X is chosen as splitter. The compound X partition is replaced by its two parts, the splitter and the rest. $\mathcal{Q}$ is stabilized w.r.t this current $\mathcal{X}$ partition. This process is repeated till $\mathcal{X}$ and $\mathcal{Q}$ become the same. Notice that by maintaining the above invariants, this termination condition ensures that the resulting $\mathcal{Q}$ partitioning is stable. In fact, it is the maximal bisimulation partitioning.

procedure compute_bisim(graph G)
begin
1. Initialize $\mathcal{Q}$ by partitioning the data nodes by label
2. Initialize $\mathcal{X}$ to a single partition with all data nodes
3. Initialize *compound_X_blocks* appropriately
4. **while** there is a *compound_X_block* **do**
5. pick partition $X \in \mathcal{X}$ which is compound
6. pick partition $Q \in \mathcal{Q}$ contained in X and smaller than half its size
7. replace X in $\mathcal{X}$ by Q and $X - Q$
8. stabilize $\mathcal{Q}$ w.r.t Q and $X - Q$

end

Figure 6: **Bisimulation computation**

The main performance gain in this algorithm comes by picking a small Q partition to split $\mathcal{Q}$. By maintaining proper data structures, the above algorithm can be implemented in $O(m \log n)$ (where m is the number of edges in G and n is the number of nodes).

4.4 Details of *propagate*

The *propagate* algorithm is summarized in Figure 7, where the update is the addition of an edge between data nodes u and v (in the direction u to v).

The algorithm works with partitions $\mathcal{X}$ and $\mathcal{Q}$ of the data nodes, as described in section 4.3. However, these are initialized differently in the update algorithm, as shown in the figure. The initialization maintains the invariants needed for PT, which we

described in Section 4.3. Thus, when we run PT with this initialization, we get a partitioning of the data nodes that is stable, which is what we desire. However, since the data structures needed by PT are not materialized here, the internals of Step 4.4 are different. However, this algorithm does exploit the "use the smaller half" feature of PT that helps it converge fast to the final partition. Indeed, this is one of the main reasons why we describe our algorithm in terms of PT.

procedure propagate-brief(u,v)
//Edge added from u to v
begin
1. add an edge from u to v in the data graph
2. **if** there is already an edge between $I[u]$ and $I[v]$ **then**
3. return;
4. Initialize $\mathcal{X}$ to be the old partitioning of nodes
5. Create a new index node I'
6. Put v in the extent of I'
7. Add an edge from $I[u]$ to I'
8. Initialize $\mathcal{Q}$ to be this partitioning of nodes
9. Run PT (lines 17 to 65 in Figure 8)

end

Figure 7: **Summary of Propagate**

PT handles how the nodes get repartitioned. However, it does not deal with edge changes. In our algorithm, we take edges also into account. The detailed algorithm involving edges is shown in Figure 8. The notation "dnode" and "dedge" refer to a data node and data edge respectively. Similarly, "inode" and "iedge" refer to an index node and edge respectively. The main thing we need to take care of to maintain the edges is that when an index node I is split by a splitter index node I_{split} to give new nodes I_1 and I_2, the in-coming edges into I_1 and I_2 are the same as those coming into I. In addition, one of I_1 and I_2 will have an incoming edge from I_{split}. The out-edges of I_1 and I_2 will be set when they are used as splitters (except if either of them is empty, in which case, the out edges are the same as that of I).

Theorem 2: *The partition created by the procedure* propagate *is stable, and hence is a refinement of the actual bisimulation index.*

Proof: If we can show that the initialization of $\mathcal{X}$ and $\mathcal{Q}$ maintains the invariant that all $\mathcal{Q}$-partitions are stable with respect to all $\mathcal{X}$ partitions, then the claim follows by the correctness of the original algorithm. By construction, the index nodes in the original index graph are mutually stable. Once the update is performed, the index node containing v

procedure propagate(u,v) //Edge added from u to v
begin
1. **if** there is an edge from u to v **then**
2. return
3. **else**
4. add a dedge from u to v
5. **if** there is an iedge from $I[u]$ to $I[v]$ **then**
6. return
7. **if** $I[v]$ has only v in its extent **then**
8. add an iedge from $I[u]$ to $I[v]$
9. return

//Split $I[v]$ into two by plucking v out of its extent
10. remove all outgoing iedges from $I[v]$
11. create new inode I' - put v in its extent
12. remove v from extent of $I[v]$
13. add in-iedges to I' from all parents of $I[v]$
14. add an in-iedge to I' from $I[u]$
15. initialize compound_X_blocks with a single inode-list consisting of $I[v]$ and I'
16. set $I[v] = I'$

//Begin actual processing
17. **while** (compound_X_blocks is not empty) **do**
18. pick an arbitrary inode-list in compound_X_blocks — call it splitter_old_list
19. pick an arbitrary inode from splitter_old_list that is less than half the size of splitter_old_list — call it splitter
20. make a copy of the dnodes in this inode's extent — call this set S
21. compute $S' = E(S)$ without duplicates

//Split with respect to S'
22. **foreach** node s in S' **do**
23. **if** $I[s]$ is not already split **then**
24. create new inode I'
25. store a mapping from $I[s]$ to I'
26. add in-iedges to I' from all parents of $I[s]$
27. add an in-iedge to I' from splitter
28. **if** $I[s]$ is in some inode-list **then**
29. add I' to it
30. **else**
31. create new inode-list with $I[s]$ and I' and add it to compound_X_blocks
32. insert $I[s]$ into a list of split inodes
33. put s in the extent of I' and set $I[s] = I'$
34. **foreach** inode I that was split **do**
35. remove it from the list of split inodes
36. let I' be its split image
37. **if** the extent of I is empty **then**
38. add out-iedges from I' to each child of I
39. delete I (this includes deleting iedges to and from I)
40. delete I and I' from I's inode-list
41. **if** this inode-list is empty **then** delete it
42. **else**
43. delete out-iedges from I

//Stabilize w.r.t the rest of nodes in splitter_old_list
44. **foreach** dnode s in S' **do**
45. **if** s has a parent in splitter_old_list **then**
46. **if** $I[s]$ is not already split **then**
47. create new inode I'
48. store a mapping from $I[s]$ to I'
49. add in-iedges to I' from all parents of $I[s]$
50. **if** $I[s]$ is in some inode-list **then**
51. add I' to it
52. **else**
53. create new inode-list with $I[s]$ and I' and add it to compound_X_blocks
54. insert $I[s]$ into a list of split inodes
55. put s in the extent of I' and set $I[s] = I'$
56. **foreach** inode I that was split **do**
57. remove it from the list of split inodes
58. let I' be its split image
59. **if** the extent of I is empty **then**
60. add out-iedges from I' to each child of I
61. delete I (this includes deleting iedges to and from I)
62. delete I and I' from I's inode-list
63. **if** this inode-list is empty **then** delete it
64. **else**
65. delete out-iedges from I

end

Figure 8: **Propagate in Detail**

may not be stable w.r.t. the index node containing u. By placing v in a partition by itself, we guarantee that the $\mathcal{Q}$ partitions are stable w.r.t. the $\mathcal{X}$ partitions. □

4.5 Computation of the Final 1-index

To complete the update, we run a new bisimulation computation, treating the nodes of the partition computed by *propagate* as the nodes of a data graph. Finally, these "nodes" are replaced by their extents to create an index graph based on the generated partition. Theorem 1 guarantees that the resulting partition is, in fact, the maximal bisimulation of the updated data graph.

However, in our experiments, as we will see in Section 7, the size of the refined bisimulation after the *propagate* is executed, differs by less than 5% from the correct 1-Index.

Thus, in our implementation, we adopt a *lazy update scheme*, where the recomputation of the accurate bisimulation is done periodically. Note that by doing this, we do not compromise the accuracy of the bisimulation index since any refinement of the bisimulation is also a precise index. This approach also in some sense amortizes over all edge additions and deletions when the change due each individual update is small, which is usually the case as we show later.

4.6 Analysis

We analyze the above algorithm along the following lines — 1) how different is the refined index output by *propagate* from the actual 1-Index (the difference being measured by the number of nodes) and 2) how expensive is *propagate* in the worst case, where the cost is measured in terms of number of nodes and edges touched in the data graph.

The refinement output by *propagate* can be very different from the actual 1-Index. This difference in number of nodes can be as large as $O(n)$ where n is the number of nodes in the data graph.

Consider for example, the data graph G (with n nodes) shown in Figure 9. The two subtrees labeled s are identical. However, each node in s has a distinct label. Consider the addition of the dashed edge in the figure. The initial 1-Index is shown beside G. The node with label A which is numbered 2 is in the extent of the index node j. After the addition of this edge, it moves to the extent of i to yield the final 1-Index. In particular, the subtree under this node is untouched. However, *propagate* does not realize this. Instead, it removes node 2 from its extent and creates a separate index node for it. This

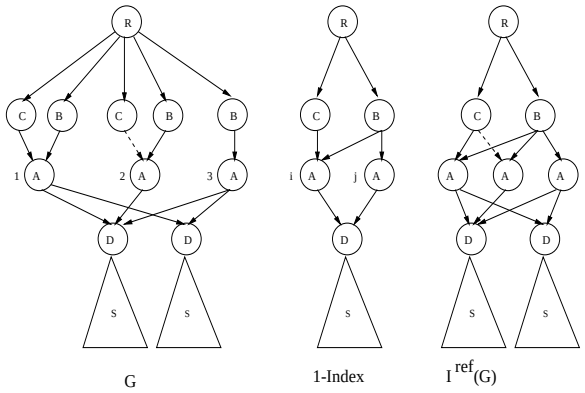


Figure 9: **Worst case scenario for** *propagate*

is propagated to all its descendants. Hence, *propagate* finally yields the index graph marked $I^{ref}(G)$. Thus, by varying s (say by increasing the depth), we obtain a family of instances where $I^{ref}(G)$ has $n-O(1)$ nodes. Thus, $I^{ref}(G)$ has $O(n)$ more nodes than the correct 1-Index.

For the same example, the number of nodes and edges touched by *propagate* is $O(m+n)$ (where m is the number of edges in G). However, in order to change the initial 1-Index to obtain the final 1-Index, $O(1)$ operations are sufficient (move node 2 from the extent of j to that of i). Thus, this instance is also a worst case instance for the performance of *propagate*.

However, in our experiments, such cases do not arise. In particular, the refinement output by *propagate* is always within 5% of the correct 1-Index and, the cost of adding an edge is relatively small, as shown by the speedups we obtain, in Section 7.

5 Updates for Forward and Backward Index

In order to handle branching path expressions, structure indexes are constructed that group nodes based both on in-coming and out-going paths. We pick one representative in this family, the F&B-Indexwhich is constructed directly from ideas described in [1]. Although the F&B-Index faces the problem of size explosion, it has some size-efficient variants discussed in [15] and we believe that the insights gained in an updating theF&B-Index will be useful in updating these variants.

In order to ease exposition, for this section, we assume that the data graph is edge-labeled. We note that there is no loss of generality in this assumption. Suppose there is an edge e from node u to node v with label l in the data graph G. We define its *inverse edge* e^{-1} to be an edge from node v to u with new label l^{-1} corresponding to l. Let us consider the

procedure compute_fb_bisim(graph G)
begin
1. Initialize $\mathcal{Q}$ by partitioning the data nodes by the set of in-coming normal *and inverse edges*
2. Initialize $\mathcal{X}$ to a single partition with all data nodes
3. Initialize *compound_X_blocks* appropriately
4. **while** there is a *compound_X_block* **do**
5. pick partition $X \in \mathcal{X}$ which is compound
6. pick partition $Q \in \mathcal{Q}$ contained in X and smaller than half its size
7. replace X in $\mathcal{X}$ by Q and $X - Q$
8. stabilize $\mathcal{Q}$ w.r.t Q and $X - Q$

end

Figure 10: **F&B-Index computation**

following process.

1. For every (edge) label l, add a new label l^{-1}.
2. For every edge e labeled l, add an inverse edge e^{-1} with label l^{-1}.
3. Compute the 1-Index on this modified graph.

The above is a structural summary that captures information about paths both *entering* and *leaving* nodes in the data graph, i.e. that captures information about paths in both the *forward* and *backward* direction. We obtain a partition of the data nodes which can be used to define an index graph. This is called the Forward and Backward-Index(F&B-Index) [15].

A theorem analogous to Theorem 1 holds for the F&B-Index as well. Since the proof is very similar to the one for Theorem 1, we omit it.

Theorem 3: *Let G be a data graph. Let $FB(G)$ be the F&B-Index and $FB^{ref}(G)$ be the index graph constructed from any refinement of $FB(G)$. Then, $FB(FB^{ref}(G)) = FB(G)$. Here, graph equality means isomorphism.*

Thus, when a subgraph is added to the database, an approach similar to the 1-Index can be adopted — suppose a subgraph H gets added to data graph G, whose initial F&B-Index is I_G. We compute the F&B-Index for H (let it be called I_H), then add I_H as a subgraph to I_G to obtain I', compute the F&B-Index on I' and "blow up" the extents to obtain the final F&B-Index for the modified data graph.

For the case of an edge addition, we again start off with the bisimulation update algorithm. While the algorithm in Section 4 for updating the bisimulation is for node-labeled graphs, essentially the same algorithm applies to the addition of a labeled edge to an edge-labeled graph. Now, the addition of a single edge to the data graph G corresponds to the addition of two edges to the modified graph G_{mod} with inverse edges. Hence, the above algorithm can

procedure propagate-brief-fb(u,v,l)
//Edge added from u to v with label l
begin
1. add an edge from u to v in the data graph with label l
2. **if** there is already an edge between $I[u]$ and $I[v]$ with label l **then**
3. return;
4. Initialize $\mathcal{X}$ to be the old partitioning of nodes
5. Create new index nodes I' and I''
6. Put v in the extent of I' and u in the extent of I''
7. Add an edge from I'' to I' with label l
8. Initialize $\mathcal{Q}$ to be this partitioning of nodes
9. Run PT

end

Figure 11: **Summary of Propagate for F&B-Index**

be extended to the update of the F&B-Index. The key here is that while propagating splits, we propagate them corresponding to the usual edges in the graph and the inverse edges as well.

To understand the details, we show the Paige Tarjan algorithm adapted to handle inverse edges in Figure 10. We note that the definition of stability needs to be modified slightly to account for edge labels.

Definition 3 *Given two sets of data graph nodes A and B, A is said to be stable with respect to B if for every edge label l either A is a subset of $Succ^l(B)$ or A and $Succ^l(B)$ are disjoint. Here, $Succ^l(B)$ refers to the successors of the set B considering only edges of label l.*

We also note that we do not actually need to materialize the inverse edges to run this algorithm.

With this modified PT algorithm, the algorithm in Figure 11 can be used to update the F&B-Index when an edge is added. Although the notion of precision in this case corresponds to branching path expressions instead of simple path expressions, the refinement output by *propagate-brief-fb* yields an index graph that is precise for all branching path expressions.

6 Updates for the A(k)-index

As noted in [16], the 1-Index can be big for some data sets. To overcome this problem, the A(k)-index was introduced. The idea here is to group nodes based on their *local structure* instead of the global path information. In particular, we group nodes based on paths of length up to k. Formally, this is based on the notion of k-bisimilarity.

Definition 4: $\approx^k$ *(k-bisimilarity): This is defined inductively.*

1. *For any two nodes, u and v, $u \approx^0 v$ iff u and v have the same label.*
2. *Node $u \approx^k v$ iff $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice versa.*

Note that k-bisimilarity defines an equivalence relation on the nodes of a graph. We call this the *k-bisimulation*. We can obtain an index graph from the k-bisimulation by creating an index node for each equivalence class and associating the data nodes in the class to the extent of the node. Edges are added by a process similar to the one for the 1-Index, explained in Section 2. This index graph is the A(k)-index.

As k increases, the partition induced by this equivalence relation keeps getting refined until at some point, it reaches a fixed point. The equivalence relation associated with this fixed point can be shown [9] to be the (maximal) bisimilarity relation from which $Bisim(G)$ (the 1-Index) is derived.

While our subgraph addition algorithm extends to the A(k)-index, the edge insertion algorithm does not.

When a subgraph is added, the A(k)-index can be updated in the same way as the 1-Index, through the following theorem.

Theorem 4: *Let G be a data graph. Let k-$Bisim(G)$ be the A(k)-index constructed from the k-bisimulation relation and k-$Bisim^{ref}(G)$ be the index graph constructed from any refinement of k-$Bisim(G)$. Then, k-$Bisim(k$-$Bisim^{ref}(G)) = k$-$Bisim(G)$. Here, graph equality means isomorphism.*

In the case of an edge addition, the story with the A(k)-index is different. First of all, unlike the arbitrary impact of an update on the 1-Index, or the F&B-Index, for the A(k)-index, the effect is only local. More formally, we have the following [16].

Property 5: *Let v, x, y be three nodes such that the shortest path to x from v or to y from v contains more than k edges. If an edge is added or deleted going from a node u to v, this update does not affect the k-bisimilarity relationship between x and y.*

The *propagate* algorithm for the full bisimulation cannot be directly extended for the k-bisimilarity partition.

Suppose an edge is added from node u to node v in the data graph. One algorithm that suggests itself is the local variant of propagate — remove v from its extent and propagate this to the descendants of v who are within distance k. For example, suppose $k = 1$. Consider child v_1 of v. The *propagate* algorithm for the full bisimulation will remove v_1 from its extent if some other node in the same extent is not a child of v. However, for the A(1)-index, this condition is too strong — v_1 needs to removed from its extent only if there is some other node in its extent that does not have a parent *with the same tag name*(i.e. 0-bisimilar) as v. In general, v_1 would need to be removed only if it does not have a parent that is $(k-1)$-bisimilar to v. Since we do not have the $(k-1)$-bisimilarity information with us, we can only check for k-bisimilarity, and so would pessimistically remove v_1 from its extent. While this safe approach is likely to work well for small k, for large k, it would cause a lot of unnecessary splits.

Thus, the *propagate* algorithm does not cleanly generalize to the case of local bisimilarity. It is an interesting area for future work.

7 Experimental Evaluation

Our update algorithms for the subgraph addition case are provably better than recomputing the structure index. However, the *propagate* algorithm for edge insertion could output the data graph in the worst case, as discussed in Section 4.6. The goal of our experiments is to investigate the behavior of *propagate* on real world data.

7.1 Data

The experiments described in this section use XML data drawn from two web sites supporting querying and browsing of that data. The first source we use is the Internet Movie Database (IMDB) [7], and the second is the Open Directory Project (ODP) [14]. We now briefly describe these data sets and the subsets we extracted for our experiments. To create our IMDB dataset, we choose a small subset of movies and all the people associated with these movies. We then sample all movies associated with the current set of people and add these movies and their associated people to the database. This process is repeated until the desired database size is reached, then dangling pointers are removed. This data graph has 190652 nodes.

The second source of semistructured data that we use is the Open Directory Project [14] data. This data is a hierarchical classification of topics and internet sites. We extract subsets of this data by choosing a set of top-level topics, in this case "Shopping", "Home", "Society",and "Regional" forming the "SHSR" data set. This data graph has 143242 nodes.

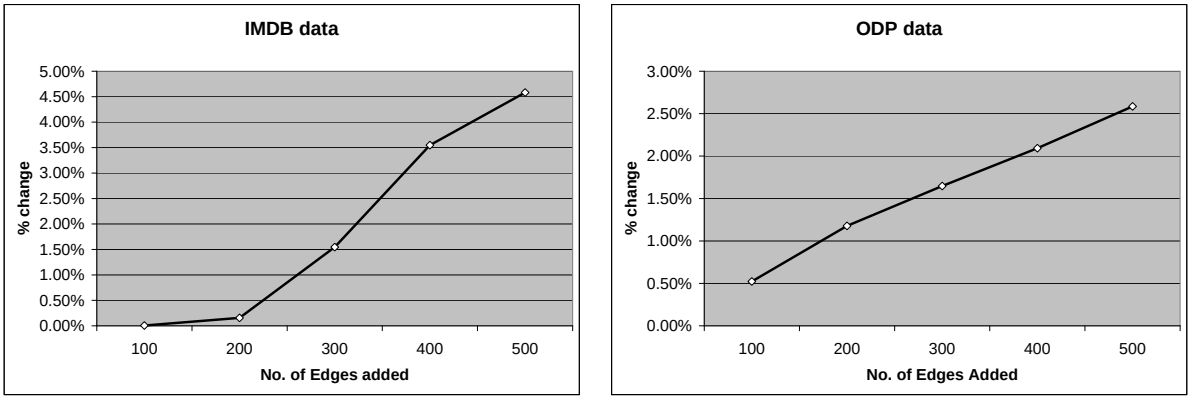


Figure 12: **Lazy Update Effectiveness**

7.2 Effectiveness of Lazy Approach

The updates consist of a sequence of IDREF edge additions. The edges are chosen to be meaningful edges: in fact we randomly select a subset of the already present IDREF edges in the data, and delete them. This resulting graph is used as the initial data graph, and the deleted edges are added one at a time.

The experiment consists of (a) adding an edge to the data graph (b) using the *propagate* procedure to compute the refinement of the updated index (c) using this as the index graph, and continuing to add edges. The goal is to check the cumulative effect of lazy updates on the index size.

Figure 12 shows the result of this experiment on the two data sets. The X-axis shows the number of edges added. The Y-axis shows a comparison between the size of the refinement computed by repeatedly calling *propagate* to handle the IDREF additions, and the size of the accurate bisimulation index after the same IDREF edges. This comparison is performed by measuring the percentage increase. As the graph shows, even a sequence of 500 edge additions makes the refinement larger than the exact index only by about 5% for both the data sets. Further, the refinement can still be used for query processing. Hence a lazy update algorithm is eminently applicable in this scenario.

7.3 Performance

Data	Speedup of *propagate*
IMDB	58.48
ODP	82.64

Table 1: Edge Insertion Performance

We also investigate the performance of the *propagate* algorithm. This is compared to the option of recomputing the bisimulation partition. The cost is measured in terms of the response time. Table 1 shows this comparison. The numbers are averaged over 20 edge insertions. For IMDB, *propagate* performs about 58 times faster, while for ODP, it is 82 times faster. These numbers show that the worst case scenarios described in Section 4.6 are not realized in these data sets.

8 Conclusions

In this paper, we proposed algorithms to update the graph bisimulation partition for two classes of updates: 1) the addition of a disjoint subgraph of data, intended to represent the addition of a new file to a database; and 2) the addition of an edge, to represent a small incremental change. We also discussed how to extend these algorithms to three of these structure indexes. Our experiments showed that our incremental update algorithms are an order of magnitude faster than recomputing the partition. Our success in extending the basic update algorithms to these cases suggests that it will be broadly applicable to new, but related, structures.

There are several interesting directions for future work.

- The *propagate* algorithm exhibits the worst case behavior of producing the data graph. It would be interesting to characterize these worst cases and come up with efficient update algorithms for special cases.

- While we propose update algorithms for the bisimulation partition, as mentioned in Section 6, the edge addition algorithm does not directly generalize to the case of k-bisimilarity. The problem of efficiently updating the k-bisimilarity partition when an edge is added is open.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web: from relations to semistructured data and XML*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[2] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *Proceedings of VLDB*, 2001.

[3] S. DeRose, E. Maler, and D. Orchard. The XLink standard. World Wide Web Consortium, `http://www.w3.org/TR/xquery`, Nov. 1999.

[4] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eighth World-Wide Web Conference*, 1999.

[5] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, 1997.

[6] J.Min, C.Chung, and K.Shim. APEX: An adaptive path index for xml data. In *Proceedings of SIGMOD*, 2002.

[7] The Internet Movie Database Ltd. Internet movie database. `ftp://www.imdb.com`.

[8] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

[9] R. Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[10] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT: 7th International Conference on Database Theory*, 1999.

[11] S. Nestorov, J. Ullman, J. Weiner, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchical data. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 79–90. IEEE, April 1997.

[12] N.Polyzotis and M.Garofalakis. Statistical synopses for graph-structured data. In *Proceedings of SIGMOD*, 2002.

[13] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, December 1987.

[14] Open Directory Project. DMOZ open directory project. `http://www.dmoz.org`.

[15] R.Kaushik, P.Bohannon, J.F.Naughton, and H.F.Korth. Covering indexes for branching path queries. In *Proceedings of SIGMOD*, 2002.

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

A Proofs

Proof of Theorem 1 The graph $Bisim(Bisim^{ref}(G))$ induces a partitioning of the nodes of G if we "blow up" the extents to the graph nodes. It is easy to see that the index graph obtained by this partitioning is the same as $Bisim(Bisim^{ref}(G))$. Thus, all we have to show is that this partitioning is the same as the $Bisim(G)$ partitioning.

For this purpose, we make the following claim. *Claim:* Let u and v be two nodes in G. Let $Bisim^{ref}(u)$ and $Bisim^{ref}(v)$ be the two nodes in $Bisim^{ref}(G)$ that have u and v respectively in their extents. Then,

$$u \approx_G^{max} v \text{ iff } Bisim^{ref}(u) \approx_{Bisim^{ref}(G)}^{max} Bisim^{ref}(v)$$

$\Rightarrow$: Let u and v be bisimilar in G. We show that $Bisim^{ref}(u)$ and $Bisim^{ref}(v)$ are bisimilar in $Bisim^{ref}(G)$ by producing a binary relation $\approx'_{ref}$ on the nodes of $Bisim^{ref}(G)$ that is a bisimulation. $equiv'$ is defined as follows:

$$a \approx_G^{max} b \Rightarrow Bisim^{ref}(a) \approx'_{ref} Bisim^{ref}(b)$$

Note that since $Bisim^{ref}(G)$ is defined from a refinement of the actual bisimulation, if $Bisim^{ref}(a) \approx'_{ref} Bisim^{ref}(b)$, then each node in the extent of $Bisim^{ref}(a)$ is bisimilar to every node in the extent of $Bisim^{ref}(b)$.

To show that $equiv'$ is a bisimulation, consider some parent $Bisim^{ref}(u')$ of $Bisim^{ref}(u)$, where u' is a parent of some u_1 in the extent of $Bisim^{ref}(u)$. Now, u_1 is bisimilar to v and so, there is some parent v' of v such that u' is bisimilar to v'. Thus, $Bisim^{ref}(v')$ is a parent of $Bisim^{ref}(v)$ and is $\approx'_{ref}$ to $Bisim^{ref}(u')$. Now consider any parent $Bisim^{ref}(v'')$ of $Bisim^{ref}(v)$. This part is similar to the above and can be shown in the same way.

$\Leftarrow$: We proceed as in the previous case by defining a binary relation $\approx'$ on the nodes of G that we claim to be a bisimulation.

$$a \approx' b \text{ iff } Bisim^{ref}(a) \approx_{Bisim^{ref}(G)}^{max} Bisim^{ref}(b)$$

Consider some parent u' of u. Hence, $Bisim^{ref}(u')$ is a parent of $Bisim^{ref}(u)$. Since $Bisim^{ref}(u)$ is bisimilar to $Bisim^{ref}(v)$, there is some parent $Bisim^{ref}(v'')$ of $Bisim^{ref}(v)$ (where v'' is a parent of some v_1 in the extent of $Bisim^{ref}(v)$) that is bisimilar to $Bisim^{ref}(u')$. Now, since v and v_1 are bisimilar, there is some parent v' of v that is bisimilar to v''. Hence, by the first part of the proof, $Bisim^{ref}(v'')$ is bisimilar to $Bisim^{ref}(v')$. Hence, since bisimilarity is a transitive relation, $Bisim^{ref}(u')$ is bisimilar to $Bisim^{ref}(v')$. Hence, $u' \approx' v'$. Similarly, for any parent v''' of v, we can produce some parent of u that is related to it by $\approx'$. Thus, $\approx'$ is a bisimulation.□