On Computing k-Bisimilar Partitions of Split Graphs

Vinod Ganapathy Gogul Balakrishnan

{vg, bgogul}@cs.wisc.edu

Abstract

The problem of indexing paths in graph structured data has received considerable attention of late. Techniques have been proposed to use bisimilarity to index data graphs. We focus on computation of an index graph called the $A(k) - index$, which is a k-bisimilar partition of the graph. Traditional techniques to compute the k-bisimilar partition work on the whole graph. In this paper, we propose an algorithm to compute the k-bisimilar partition when the graph has been split into two or more subgraphs. This technique can be used in the computation of the A(k)-index on data graphs of arbitrarily large sizes.

1 Introduction

XML is becoming the language of choice for the representation and exchange of data. XML data is inherently structured as a labeled graph. A lot of interest has been generated in processing queries over such labeled graph data models. Since such data models do not have the notion of a schema, techniques have been proposed in the literature to extract structural summaries from such data.

One important class of techniques proposed to extract such structural summaries exploits local similarity in the graph structure. These techniques were introduced in [3], where an important property called *graph bisimilarity* was used to exploit similarity in the graph structure and construct index graphs called *1-Indexes* for the data graph. These techniques were extended in [1] where a new class of index graphs called *A(k)-indexes* were proposed. They make use of a property called *k-bisimilarity*. This technique is more general, and more flexible than the 1-Index because it is parametrized by k and the precision required in the index can be varied by changing k. Moreover, by increasing the value of k, the A(k)-index converges to the 1-Index.

This paper focuses on a technique to compute the k-bisimilar partition of a data graph. The technique proposed in [1] makes an unstated assumption about the largest size of the data graph the can be processed to obtain the k-bisimilar partition. Specifically, it can process only such data graphs which fit within the address space of the machine. In this paper, we propose a technique to get over this limitation. The way this is achieved is by designing a technique that works on subgraphs of the original data graph, and merges the information across these subgraphs to obtain the k-bisimilar partition for the original data graph.

The rest of this paper is organized as follows: Section 2 covers the necessary background. Section 3 discusses the shortcomings with the existing approach, and lays the foundation for our work. In section 4, we discuss our technique to obtain k-bisimilar partitions by splitting the datagraph into two subgraphs, followed by a proof of correctness of the technique in section 5. Section 6 analyses the running time of our algorithm and briefly compares it with the running time of the technique in [1]. Section 7 generalizes the techniques of section 4 by discussing a method that works on a data graph that has been split into an arbitrary number of partitions. We then give a small example to show the working of our algorithm in section 8 and conclude in section 9.

2 Background

2.1 Notation and Terminology

As mentioned earlier, XML data can be represented as a labeled graph. It is assumed to be represented as a directed, labeled graph $G = (V_G, E_G, root, \Sigma_G, Lab, OId, value)$. Each edge E_G indicates an object-subobject or object-value relationship. "Simple" nodes in the graph have no outgoing edges, and are given a value using the $value$ function. Each node in V_G is given a string literal, also referred to as the *label*, from Σ_G using the *Lab* function. Thus, the string literal at a node v is represented as $Lab(v)$. All "simple" nodes are given the same distinguished label, called VALUE. Each node v is also given a unique identifier using the OId function. A $nodepath$ in G is a sequence of nodes $n_0 \ldots n_p$, such that an edge exists between n_i and n_{i+1} for $0 \leq i \leq p-1$. A labeled path is a sequence of labels $l_0 \ldots l_p$. A labeled path $l_0 \ldots l_p$ *matches* a node path $n_0 \ldots n_p$ if $Lab(n_i) = l_i$ for all $0 \leq i \leq p$. A labeled path matches a node n if it matches some node path ending in n.

Techniques described in [2, 3, 1] exploit local similarity in the graph structure described above to create a structural summary of the graph. The structural summary is another directed, labeled graph which preserves all the paths in the data graph. However, the summary graph, which is also called the *index graph* has far fewer nodes and edges than the original data graph. Such techniques associate an *extent* with each node in the index graph. If A is a node in the index graph $I(G)$, then $ext^I(A)$ which denotes the extent of A with respect to $I(G)$, is a subset of V_G. The extent mapping produced by the techniques in [2, 3, 1] is *safe*, in that if a labeled path to a node v exists in G, then there is a labeled path to a node A in $I(G)$ such that v $\in ext^I(A)$.

2.2 Some Definitions and Theorems

We now introduce the notion of *bisimilarity* [7] and *k-bisimilarity* [8] which are central to the ideas in this paper.

Bisimilarity: A binary relation $\sim$ on V_G called a bisimulation, relates two data nodes u and v iff:

(a) $Lab(u) = Lab(v)$ and,

(b) For each parent u' of u, there is a parent v' of v such that $u' \sim v'$, and vice versa.

Two nodes u and v are said to be bisimilar if they are related by a bisimulation.

k-bisimilarity: A binary relation $\sim^k$ on V_G, parametrized on k, called a k-bisimulation relates two data nodes u and v, and is defined inductively as follows:

(a) $u \sim^0 v$, iff $Lab(u) = Lab(v)$.

(b) $u \sim^k v$, iff $u \sim^{k-1} v$ and for each parent u' of u, there is a parent v' of v such that $u' \sim^{k-1} v'$, and vice-versa.

Two nodes u and v are said to be k-bisimilar if they are related by a k-bisimulation.

We state, without proof, some general properties on bisimilarity and k-bisimilarity, that are exploited by the techniques in [3, 1]:

(a) Bisimulation and k-bisimulation are equivalence relations. Thus both these relations split the data graph into equivalence classes.

(b) If two nodes u and v are bisimilar, then the set of labeled paths to these nodes is the same.

(c) If two nodes u and v are k-bisimilar, then the set of labeled paths of length $\leq k$ into them is the same.

An index graph can be constructed based on the bisimilar partition of the graph. Each equivalence class forms a node of the index graph, and the data nodes in the class form the extent of the node. An edge exists from a node u to a node v in the index graph iff an edge exists from one of the data graph nodes in $ext(u)$ to a data graph node in $ext(v)$. Such an index graph is called a *1-Index*, and was first introduced in [3]. Similarly, an index graph can also be constructed based on the k-bisimlar partition of the graph. Such an index is called the *A(k)-index* [1]. Increasing k refines the partition by splitting some equivalence classes into smaller equivalence classes. This continues until a fixed point $\hat{k}$ is reached, at which the A($\hat{k}$)-index can be shown to be the same as the 1-Index. The A(k)-index, and hence the 1-Index, can be shown to be safe using the properties of k-bisimulation and bisimulation.

2.3 Stability

For a set of nodes A, let $Succ(A)$ denote the set of successors of the nodes in A. Then, given two sets of data nodes A and B, B is said to be *stable* with respect to A if either $B \subseteq Succ(A)$, or $B \cap Succ(A) = \phi$. A node set B can be made stable with respect to another node set A by splitting B into two sets, $B \cap Succ(A)$ and $B - Succ(A)$. It can easily be seen that each of these two sets is stable with respect to A. For a data graph G, partition P_1 of V_G is stable with respect to partition P_2 of V_G if each equivalence class in P_1 is stable with respect to every class in P_2. It can be shown that the $(k+1)$-bisimilar partition is the coarsest refinement of the k-bisimilar partiotion that is stable with respect to it. By a coarsest partition, we mean a partition which has the maximum number of nodes in each equivalence class; thus adding any more nodes into an equivalence class will destroy the stability of the structure. This property suggests a method to compute the $(k+1)$-bisimilar partition given the k-bisimilar partition. If we assume that the k-bisimilar partition is represented as a list of node sets X, where each node set in X corresponds to an equivalence class in the partition, then the $(k+1)$-bisimilar partition is computed as follows: first a copy Q of X is made, and the node sets in X are split till X is stable with respect to Q. Then X is the $(k+1)$-bisimilar partition of the graph.

compute the $(k+1)$-bisimilar partition a copy of this partition is made, and the equivalence classes in it are split till it is stable with respect to the k-bisimilar partition. These ideas form the basis of the algorithm that is suggested in [1] to compute the A(k)-index, and also of the algorithm suggested in this paper.

3 The Need for a new Algorithm

In this paper, we look at techniques to compute the A(k)-index for a data graph. In this section we discuss the existing technique to compute the A(k)-index, and discuss the shortcomings of this algorithm. We then motivate the idea behind our algorithm which overcomes these shortcomings.

3.1 Existing Techniques

In [1] Raghav Kaushik et.al. discuss an algorithm to compute the A(k)-index of a data graph. The technique uses the notion of stability discussed in section 2.3 to compute the A(k)-index of a data graph iteratively. The algorithm starts off with the 0-bisimilar partition, in which the nodes are grouped into equivalence classes based on their labels. The edges of the data graph are not used in the computation of the 0-bisimilar partition. The 1-bisimilar partition can be computed as the coarsest refinement stable with respect to the 0-bisimilar partition using the technique suggested in section 2.3. This is iterated till the required value of k is reached. At this point, the algorithm would have computed the k-bisimilar partition of the graph. The A(k)-index is obtained by adding edges between the equivalence classes so obtained.

3.2 Shortcomings

The technique presented in [1] works well when the data graph fits completely in main memory. Performance is affected when the data graph is big, and some portion of the graph must reside in virtual memory. In this case, a page fault occurs each time a node that is required for computation is not found on main memory; the corresponding page is then loaded from virtual memory. The technique fails to work if the data graph is larger than the address space of the machine, because the algorithm requires the whole graph to be loaded, which is not possible when the graph cannot fit within the address space of the machine.

This paper aims to solve these problems of the algorithm. The techniques that will be discussed in the section 4 use graph splitting to compute the k-bisimilar partition of the graph. We make the following contributions in this paper:

(a) The algorithm that we propose has the ability to compute the k-bisimilar partition on an arbitrarily large data graph, even one that does not fit within the address space of the machine. This is so because unlike [1], we do not require the entire data graph to be loaded in memory.

(b) When the data graph does not fit in main memory, but fits within the address space of the machine, the performance of our algorithm to compute k-bisimilar partitions is comparable with the performance of the algorithm proposed in [1]. While our aim was to propose a faster algorithm, our technique uses some book-keeping information, which unfortunately tends to slow down the algorithm to speeds comparable with [1].

3.3 Idea behind our Algorithm

We observe that the main drawback with the algorithm in [1] is that the whole graph has to be loaded for the algorithm to process it. We get over this problem by splitting the graph, and working on each subgraph at a time. We compute the k-bisimilar partition for the whole graph iteratively by working on each of these subgraphs separately, and "stitching" the partitions together at the end of each iteration. For ease of explanation, we first propose an algorithm that works by splitting the graph into 2 sub-graphs (in section 4), and show how to compute the k-bisimilar partition. The technique is extended to work when the graph is split into an arbitrary number of subgraphs in section 7.

Computing the k-bisimilar partition by splitting the graph will require us to keep some bookkeeping information. When a graph G is split into two or more subgraphs $G_1, \leq, G_n$ for purposes of computing the k-bisimilar partition, some of the nodes that occur in an equivalence class get split between the subgraphs. When constructing the k-bisimilar partition of the graph G, care must be taken to recombine these nodes from the equivalence classes of each of $G_1, \leq, G_n$ into the same equivalence class. The book-keeping information must be able to determine which equivalence classes from $G_1, \leq, G_n$ must be merge together into the same equivalence class. Let us consider two node sets in the k-bisimilar partition of a graph as shown in figure 1 such that an edge exists between node A and node B in the index graph constructed using the k-bisimilar partition. We now look at the problem of obtaining the $(k+1)$-bisimilar partition from the k-bisimilar partition. As mentioned in section 2.3, for every pair of node sets A and B, with an edge from A to B, we make B stable with respect to A. Thus B is broken into two portions the figure 1. Let us call these two parts B^{A^+} and B^{A^-} to represent the $B \cap Succ(A)$ and $B - Succ(A)$ respectilvely.

Now let us suppose that the index graph obtained from the k-bisimilar partition was split into two subgraphs P_1 and P_2. Node set A will get split between these two subgraphs; let the portion of A in P_1 and P_2 be called A_{P_1} and A_{P_2} respectively. B is also split similarly between these two subgraphs into B_{P_1} and B_{P_2}. The edge between A and B in the index graph could get split into four edges as shown in the figure 1. Note that any of these edges could be present or absent depending on the nodes in each of the node sets $A_{P_1}, A_{P_2}, B_{P_1}, B_{P_2}$. We call the edges from A_{P_1} to B_{P_2}, and from A_{P_2} to B_{P_1} *cross-edges*.

Consider the problem of computing stability given the split graph. Each of B_{P_1} and B_{P_2} will be split into two node sets because of the edges into them from A_{P_1} and A_{P_2} respectively. Let us call these node sets $B_{P_1}^{A_{P_1}^+}$, $B_{P_1}^{A_{P_1}^-}$, $B_{P_2}^{A_{P_2}^+}$ and $B_{P_2}^{A_{P_2}^-}$ respectively (following the same convention as was used for B_{A^+} and B_{A^-}). These are shown on either side of the ditted line (which denotes a split) of B_{P_1} and B_{P_2} in the figure. $B_{P_1}^{A_{P_1}^+}$ and $B_{P_1}^{A_{P_1}^-}$ are still to be stabilized with respect to A_{P_2}. On doing so, they get split into two sets each: $B_{P_1}^{A_{P_1}^+}$ gets split into $B_{P_1}^{A_{P_1}^+ A_{P_2}^+}$ and $B_{P_1}^{A_{P_1}^+ A_{P_2}^-}$, and $B_{P_1}^{A_{P_1}^-}$ gets split into $B_{P_1}^{A_{P_1}^- A_{P_2}^+}$ and $B_{P_1}^{A_{P_1}^- A_{P_2}^-}$. Similarly, $B_{P_2}^{A_{P_2}^+}$ gets split into $B_{P_2}^{A_{P_1}^+ A_{P_2}^+}$ and $B_{P_2}^{A_{P_1}^- A_{P_2}^+}$, and $B_{P_2}^{A_{P_2}^-}$ gets split into $B_{P_2}^{A_{P_1}^+ A_{P_2}^-}$ and $B_{P_2}^{A_{P_1}^- A_{P_2}^-}$. To clarify the notation, $B_{P_1}^{A_{P_1}^+ A_{P_2}^+}$ denotes the subset of B_{P_1} that consists of successors of both A_{P_1} and A_{P_2}, while $B_{P_1}^{A_{P_1}^+ A_{P_2}^-}$ denotes the subset of B_{P_1} that consists of successors of A_{P_1}, but not those of A_{P_2}.

Hence, as a result of stabilizing, we are left with 8 buckets which we must keep track of for recombination to produce the $(k+1)$-bisimilar partition of the original data graph. However, as the following theorems show, the book-keeping information that we have to maintain does not have to keep track of all of these 8 buckets.

Theorem 1 $B^{A^-} = B_{P_1}^{A_{P_1}^- A_{P_2}^-} \cup B_{P_2}^{A_{P_1}^- A_{P_2}^-}$

Proof: We know that $A = A_{P_1} \cup A_{P_2}$ and $B =$

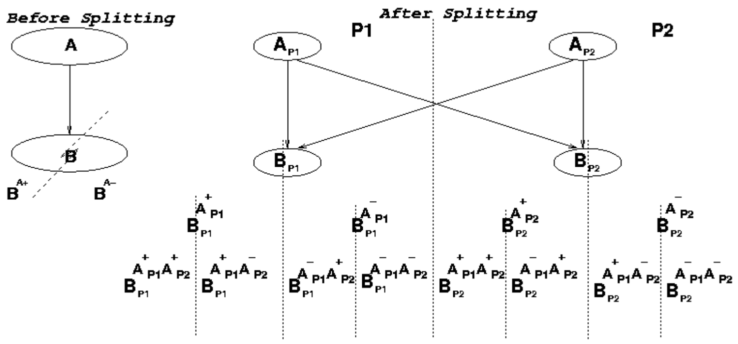


Figure 1: The Effects of Graph Splitting

$B_{P_1} \cup B_{P_2}$. Also, $B^{A^-} = B - Succ(A)$. From the above statements, $B^{A^-} = (B_{P_1} \cup B_{P_2}) - Succ(A_{P_1} \cup A_{P_2}) = (B_{P_1} \cup B_{P_2}) \cap Succ(A_{P_1} \cup A_{P_2})^C$. But $Succ(A_{P_1} \cup A_{P_2}) = Succ(A_{P_1}) \cup Succ(A_{P_2})$. Hence, $B^{A^-} = (B_{P_1} \cap Succ(A_{P_1})^C \cap Succ(A_{P_2})^C) \cup (B_{P_2} \cap Succ(A_{P_1})^C \cap Succ(A_{P_2})^C)$. But $B_{P_1}^{A^-_{P_1} A^-_{P_2}} = B_{P_1} \cap Succ(A_{P_1})^C \cap Succ(A_{P_2})^C$, and $B_{P_2}^{A^-_{P_1} A^-_{P_2}} = B_{P_2} \cap Succ(A_{P_1})^C \cap Succ(A_{P_2})^C$. Hence Proved.

Theorem 2 $B^{A^+} = B_{P_1}^{A^+_{P_1} A^+_{P_2}} \cup B_{P_1}^{A^+_{P_1} A^-_{P_2}} \cup B_{P_1}^{A^-_{P_1} A^+_{P_2}} \cup B_{P_2}^{A^+_{P_1} A^+_{P_2}} \cup B_{P_2}^{A^-_{P_1} A^+_{P_2}} \cup B_{P_2}^{A^-_{P_1} A^+_{P_2}}$

Proof: Similar to Theorem 1

Theorems 1 and 2 show that it is sufficient to maintain two buckets each for B_{P_1} and B_{P_2}. The first bucket for B_{P_1} consists of those nodes which are either successors of A_{P_1} or A_{P_2} or both. In other words $B_{P_1}^{A^+_{P_1} A^+_{P_2}}$, $B_{P_1}^{A^+_{P_1} A^-_{P_2}}$ and $B_{P_1}^{A^-_{P_1} A^+_{P_2}}$ are combined into the first bucket. The second bucket consists of those nodes in B which are neither successors of A_{P_1} nor A_{P_2}, i.e., the second bucket is $B_{P_1}^{A^+_{P_1} A^-_{P_2}}$. Similarly, the first bucket for B_{P_2} consists of nodes from $B_{P_2}^{A^+_{P_1} A^+_{P_2}}$, $B_{P_2}^{A^+_{P_1} A^-_{P_2}}$ and $B_{P_2}^{A^-_{P_1} A^+_{P_2}}$, while the second bucket is $B_{P_2}^{A^-_{P_1} A^-_{P_2}}$.

By merging the first buckets maintained for each of B_{P_1} and B_{P_2} we get B^{A^+}, while merging the second buckets yields B^{A^-}, as has been proved in Theorems 1 and 2. The analysis given here has been simplified to illustrate the idea by only showing the effect of the edge from A to B. B could in general have other parents as well, and similar splitting of equivalence classes happens for each parent of B.

4 Obtaining k-bisimilar partitions of split graphs

In this section, we present the algorithm that produces the k-bisimilar partition of the graph by graph splitting. The data structures used in the algorithm are discussed in section 4.1, the detailed algorithm is in section 4.2.

4.1 Data Structures Used

The algorithm first splits the graph G into two subgraphs G_1 and G_2, the union of which gives G. It then computes the 0-bisimilar partition for G_1 and G_2. Since the 0-bisimilar partition groups nodes by label, the 0-bisimilar partition of G can be reconstructed from the 0-bisimilar partition of G_1 and G_2 by combining node sets whose elements have the same label. The algorithm does this "lazily" by establishing a correspondence between the node sets in the 0-bisimilar partitions of G_1 and G_2 that should be combined to give the 0-bisimilar partition of G.

The procedure to compute the k-bisimilar partition works in k iterations. At the end of the ith iteration, a correspondence is available between the node sets in G_1 and G_2 that must be combined to obtain the node sets in the i-bisimilar partition of G. Observe that a each node can be represented uniquely using its OId, and hence the node set is just a set of integers. To establish the correspondence between these node sets, we maintain some data structures with each node set N in each subgraph, which are explained below:

(a) $Label(N)$: A string associated with node set N, which is the same as the label of any node n in the node set N, i.e., $Label(N) = Lab(n)$ for some $n \in N$.

(b) $SeqNo(N)$: An integer associated with each node set N. The $SeqNo$s for each node set within a subgraph are unique, however this is not true across the subgraphs. The $SeqNo$s are used to establish the correspondence between the node sets in G_1 and G_2 which should be combined to produce the node sets in G: a node set in G_1 is combined with a node set with the same $SeqNo$ as a node set in G_2 to produce a node set of G. The $SeqNo$s are for the node sets in the $(i+1)$th iteration are computed using the $SeqNo$s of the node sets in the ith iteration.

(c) $Tags(N)$: A set of $SeqNo$s. This is an intermediate structure used by the algorithm during the recomputation of the $SeqNo$s. The use of this data structure is explained in Algorithms 4, 5 and 7.

4.2 The Algorithm

Algorithm 1 is the main procedure to compute the k-bisimilar partitions. The first step, `Split`, splits the input graph G into two subgraphs G_1 and G_2, such that union of the set of edges and vertices in G_1 and G_2 is the set of edges and nodes in G. In this paper, we do not give an algorithm to split the graph, any split will work. However, performance is greatly improved with certain kinds of splits. We give an insight into what a good split of the graph is in section 6. `Compute_0_bisim` (H) computes the 0-bisimilar partition of a graph H. `Stitch_0_bisim` (H_1, H_2) assigns $SeqNo$s to the node sets in the lists H_1 and H_2. `Make_SuccStable` (H) makes the node set H stable with respect to a copy of itself. While doing so, only the edges between the node sets in H are considered. However cross-edges exist between the subgraphs as well, and the node sets in one subgraph must be stabilized with respect to the node sets in the other subgraph. This is considered in `Add_CrossEdges`. `Number_Buckets` reassigns the $SeqNo$s to the node sets.

Algorithm 2 computes the 0-bisimilar partition of the graph $Graph$. It does so by grouping the nodes that share the same label into a single node set.

Algorithm 1 `Compute_k_Bisim` (G, k)

```
{G1, G2} = Split (G)
K1 = Compute_0_bisim (G1)
K2 = Compute_0_bisim (G2)
Stitch_0_bisim (K1, K2)
for [p = 1 to k] do
   K1 = Make_SuccStable (K1)
   K2 = Make_SuccStable (K2)
   {K1, K2} = Add_CrossEdges (K1, K2)
   K1 = CleanUp (K1)
   K2 = CleanUp (K2)
   Number_Buckets (K1, K2)
end for
```

Algorithm 2 `Compute_0_bisim` $(Graph)$

Group the nodes of a $Graph$ into a list of set of nodes, so that for every two elements u and v of the same set, $Lab(u) = Lab(v)$.
Return the list of set of nodes.

In Algorithm 3, a correspondence is established between the 0-bisimilar partitions of each of the subgraphs. To do so, $SeqNo$s are first assigned to the node sets of one of the subgraphs. Then the node sets of the second subgraph are labeled based on whether they share the same label as a node set in the first subgraph. If they share a label, then they are given the same $SeqNo$. The algorithm takes both the list of set of nodes H_1 and H_2 as input. However, it is not necessary to load both of them at the same time. The first loop in the program processes H_1 and gives it the $SeqNo$s. While doing so, it also forms the set of labels encountered. During the processing of H_2, it is necessary to establish a correspondence between the node-set in H_2 which shares the same label as a node-set in H_1. One way to do this is to maintain a temporary table which relates the labels to the $SeqNo$s as assigned in H_1. However, we note that it may not always be possible to fit this table and H_2 together in memory (since the number of labels may be very large). The alternative is to load H_2 in memory, and stream H_1 past it, which will require just one input buffer and one output buffer. This technique however has the disadvantage of running slower than the previous approach.

Algorithm 3 `Stitch_0_bisim` (H_1, H_2)

Counter = 0
for all $[h_1 \in H_1]$ **do**
  SeqNo (h_1) = Counter++
  LabelSet = LabelSet $\cup$ $\{$*Label* $(h_1)\}$
end for
for all $[h_2 \in H_2]$ **do**
  if [*Label* $(h_2) \in$ LabelSet] **then**
    SeqNo (h_2) = *SeqNo* (s), such that [(s $\in H_1$) $\wedge$ (*Label* (h_2) == *Label* (s))]
  else
    SeqNo (h_2) = Counter++
  end if
end for

`Make_SuccStable`, described in Algorithm 4 makes the list of node sets H stable with respect to a copy of itself. This is done by using the method in section 2.3 It must be noted however, that the $Succ$ function here returns only the set of successors in the subgraph under consideration i.e., cross-edges are not considered in the computation of the successors, they will be considered as part of Algorithm 5. This algorithm also initializes the $Tags$ for each node set. The idea behind the use of *Tags* stems from Theorems 1 and 2. As mentioned earlier, the $Tags$ are a set of $SeqNo$s. The $SeqNo$ of a node set M is added to the $Tags$ of a node set N if, and only if, there is a node in N which is a successor of a node in M. Hence, absence of a node set M's $SeqNo$ from the $Tags$ field of a node set N means that N does not contain any successors of nodes in M.

The Algorithm 5, `Add_CrossEdges`, stabilizes the node sets in one partition with respect to the node sets in the other partition based on the cross edges between the two subgraphs. Referring to the discussion in 3.3, the effect of algorithms 4 and 5 is to set the $SeqNo$s and $Tags$ such that **if** $SeqNo(A_{P_1}) = SeqNo(A_{P_2})$ and $SeqNo(B_{P_1}) = SeqNo(B_{P_2})$ **then** the $SeqNo$s of $B_{P_1}^{A^+_{P_1} A^+_{P_2}}$, $B_{P_1}^{A^+_{P_1} A^-_{P_2}}$, $B_{P_1}^{A^-_{P_1} A^+_{P_2}}$, $B_{P_1}^{A^-_{P_1} A^-_{P_2}}$, $B_{P_2}^{A^+_{P_1} A^+_{P_2}}$, $B_{P_2}^{A^+_{P_1} A^-_{P_2}}$, $B_{P_2}^{A^-_{P_1} A^+_{P_2}}$ and $B_{P_2}^{A^-_{P_1} A^-_{P_2}}$ will all be equal and the $Tags$ will be set such that,

$SeqNo(A_{P_1}) \in Tags(B_{P_1}^{A^+_{P_1} A^+_{P_2}})$,
$SeqNo(A_{P_1}) \in Tags(B_{P_1}^{A^+_{P_1} A^-_{P_2}})$,
$SeqNo(A_{P_1}) \in Tags(B_{P_1}^{A^-_{P_1} A^+_{P_2}})$,
$SeqNo(A_{P_1}) \in Tags(B_{P_2}^{A^+_{P_1} A^+_{P_2}})$,
$SeqNo(A_{P_1}) \in Tags(B_{P_2}^{A^+_{P_1} A^-_{P_2}})$,
$SeqNo(A_{P_1}) \in Tags(B_{P_2}^{A^-_{P_1} A^+_{P_2}})$,
$SeqNo(A_{P_1}) \notin Tags(B_{P_1}^{A^-_{P_1} A^-_{P_2}})$,
$SeqNo(A_{P_1}) \notin Tags(B_{P_2}^{A^-_{P_1} A^-_{P_2}})$.

Then, by combining the node sets with same $Tags$ and $SeqNo$s, it can be seen that we can reconstruct the k-bisimilar partition of the original graph using the information from the subgraphs. This is precisely the effect acheived by our algorithm.

Algorithm 4 `Make_SuccStable` (H)

X = (a copy of) H
for all $[h \in H]$ **do**
  Tags (h) = ϕ
end for
for all $[x \in X]$ **do**
  for all $[h \in H]$ **do**
    $A = h\cap$ *Succ* (x)
    $B = h$ - *Succ* (x)
    Tags (A) = *Tags* (h) $\cup$ *SeqNo* (x)
    Tags (B) = *Tags* (h)
    SeqNo (A) = *SeqNo* (h)
    SeqNo (B) = *SeqNo* (h)
    Replace h in H by A and B (only the non-empty sets)
  end for
end for
Return H

There is an important point we wish to highlight regarding Algorithm 5. Although the algorithm takes in both the partitions as input and refers to the node sets containing the source and the target of the cross edges, it is not necessary to load both the partitions into memory at the same time. Instead, we maintain a *cross-edges table*. Hence `SeqNoSrc` and `SeqNoTgt` can directly be read off the table given the source and target vertices of a cross edge .This table is linear in the size of the number of cross-edges, and each row of the table consists of (Target Node, Source Node, From subgraph, To subgraph, $SeqNo$ of node set containing Target Node, $SeqNo$ of node set containing Source Node). Using this table, we can process the first subgraph by considering all the cross edges pointing to some node in it, and then process the second subgraph by considering the remaining cross edges. Keeping this table updated requires updating the last two columns, i.e., the $SeqNo$s of the node sets. This is achieved by Algorithm 8.

Algorithm 5 `Add_CrossEdges` (H_1, H_2)

for all [Cross Edges Γ] **do**
  `src` = Source vertex of Γ
  `tgt` = Target vertex of Γ
  Let A be the node set such that (`src` $\in A$)
  Let B be the node set such that (`tgt` $\in B$)
  `SeqNoSrc` = *SeqNo* (A)
  `SeqNoTgt` = *SeqNo* (B)
  if [`SeqNoSrc` $\notin$ *Tags* (B)] **then**
    $M = B$ - *SuccCross* (A)
    $N = B \cap$ *SuccCross* (A)
    SeqNo (M) = *SeqNo* (B)
    SeqNo (N) = *SeqNo* (B)
    Tags (M) = *Tags* (B)
    Tags (N) = *Tags* $(B) \cup$ *SeqNo* (A)
    Replace B (in H_1 if $B \in H_1$, H_2 otherwise) by M and N
  end if
end for
Return $\{H_1, H_2\}$

When a node set has been processed by Algorithms 4 and 5, it may turn out that there are some node sets within a partition which have the same $Tags$ and $SeqNo$s. It would be incorrect to process these as separate sets since the resulting partition will be a refinement of the partition actually desired. A situation where this can arise is shown in figure 2. The graph is split as shown into two parts P_1 and P_2. The node A is replicated in both the subgraphs. Let us consider the first iteration of the loop from Algorithm 1. At the beginning of the loop, we have the 0-bisimilar partition, which is as shown in the figure. For ease of explanation, we have used alphabets to denote $SeqNo$s of node sets. Each partition is shown as a list of node sets, with the extents of each node shown within braces. After the execution of `Make_SuccStable` and `Add_CrossEdges`, the lists are transformed as shown by the state "Before Cleanup". Observe that in each of P_1 and P_2, there are two sets, with $SeqNo$ c having the same $Tags$ (which are shown within parantheses). These are merged by the `Cleanup` procedure to give the lists as shown by "After Cleanup".

Algorithm 6 `CleanUp` (H)

for all node sets $M, N \in H$ with $SeqNo(M) == SeqNo(N)$ **do**
  if [$Tags(M) == Tags(N)$] **then**
    $P = M \cup N$
    $SeqNo(P) = Seqno(M)$
    $Tags(P) = Tags(M)$
    Replace M and N in H by P
  end if
end for
Return H

Algorithm 7, `Number_Buckets`, re-computes the $SeqNo$s on the basis of the $Tags$ that were computed by the previous algorithms. Node sets with the same $SeqNo$s and $Tags$ are given the same new $SeqNo$. The intuition behind this is that the $(i+1)$-bisimulation is a refinement of the i-bisimulation. In the i-bisimulation, node sets with the same $SeqNo$s in both subgraphs would get merged to give the corresponding node set in the index graph. However, in the $(i+1)$-bisimulation, some of these node sets would be split, so that the resulting node sets are stable with respect to one another. The stability aspect is captured by the $Tags$, while the refinement aspect is captured by the $SeqNo$s. By ensuring that only those node sets with the same $SeqNo$s across the subgraphs are compared for $Tags$ equality, we have ensured that the node sets obtained after the $(i+1)$-th iteration are refinements of the node sets obtained after the i-th iteration. This algorithm takes in node sets of both the parts of the graph as input. Just as in `Stitch_0_bisim`, we could load one of the node sets into memory, and stream the other node set past it. This way, we avoid loading both the node sets in memory. Another approach just loads in the appropriately numbered node sets for the each of the list of node sets corresponding to the subgraphs in each iteration, and does the processing. In other words, in the p^{th} iteration of the first `for` loop of Algorithm 7, the all the node sets numbered p from both the subgraphs are loaded into memory for processing.

In the algorithm `SNS1` and `SNS2` denote a set of node sets. They are used to collect up the node sets with the same $SeqNo$ values from the list of set of nodes. The elements of these sets (which are node sets) are then compared for equality of $Tags$. If the $Tags$ are the

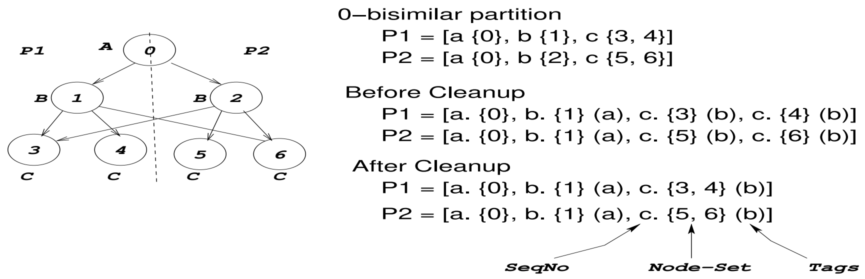


Figure 2: The need for CleanUp

same, it means that the node sets can be merged, and so we give these node sets the same new $SeqNo$s.

`Update_CrossEdges_Info`, discussed in Algorithm 8 updates the *cross-edges table* to reflect the new $SeqNo$s as obtained by `Number_Buckets`.

To obtain the k-bisimilar partition from the entire graph, one has to run the algorithm with the parameter k passed to Algorithm 1. When algorithm 1 terminates, we will be left with the node sets of two subgraphs, with $SeqNo$s assigned to them. $SeqNo$s within a subgraph will be unique; by combining the node sets with the same $SeqNo$s for the two subgraphs, we obtain the node sets for the k-bisimilar partition of the original graph. This is shown in algorithm 9. Once this is done, the A(k)-index can be computed using the algorithm `compute_A(k)_index` described in [1]

5 Proof of Correctness

In this section we prove the correctness of the approach that we described. Theorem 3 is the main result:

Theorem 3 *Two nodes a and b are k-bisimilar if and only if at the end of k iterations of the* `for` *loop in* `Compute_k_Bisim`*, the node sets to which they belong have the same $SeqNo$s.*

Proof: There are two parts to the proof, we prove each separately.

($\Rightarrow$): Given that at the end of k iterations of the `for` loop, the node sets containing a and b have the same $SeqNo$, we are required to prove that they are k-bisimilar. There are two possible scenarios in which this could happen: either a and b belong to the same subgraph, or they belong to different subgraphs.

Suppose a and b belong to the same subgraph, they will occur as part of the same node set because `Number_Buckets` ensures that each node set in the list of node sets corresponding to the subgraph has a unique $SeqNo$. Since the algorithm never merges node sets (except in `CleanUp`, where nodes sets with the same $SeqNo$ and $Tags$ are merged), a and b must have occured in the same node set after $(k-1)$ iterations as well. Moreover, since they did not get split into different node sets in the kth iteration, it means that the node sets containing their parents u and v must have had the same $SeqNo$ after $(k-1)$ iterations. These facts imply that a and b are $(k-1)$ bisimilar. The same holds for u and v as well. Hence, by an inductive argument, a and b are k-bisimilar.

Now consider the case when a and b belong to different subgraphs. Since the node sets containing a and b have the same $SeqNo$ after k iterations, and `Number_Buckets` gives the same $SeqNo$ to two node sets only if they had the same $SeqNo$ at the end of the previous iteration, the node sets with a and b must have had the same $SeqNo$ at the end of $(k-1)$ iterations. Moreover, the $Tags$ associated with the node sets containing a and b must also have been the same, which implies that the parents u and v of a and b also occured in node sets with the same $SeqNo$ after $(k-1)$ iterations. Thus a and b are k-bisimilar.

Algorithm 7 `Number_Buckets` (H_1, H_2)

Counter = 0
if [H_1 has a lower highest *SeqNo* than H_2] **then**
  $J_1 = H_1$, $J_2 = H_2$
else
  $J_1 = H_2$, $J_2 = H_1$
end if
for [i = 0 to highest *SeqNo* in J_1] **do**
  Let `SNS1` be the set of node sets from J_1 such that, $\forall S \in$ `SNS1`, (*SeqNo* (S) == i)
  Let `SNS2` be the set of node sets from J_2 such that, $\forall S \in$ `SNS2`, (*SeqNo*(S) == i)
  for all [$S \in$ `SNS1`] **do**
    Let $P \in$ `SNS2` such that *Tags* (P) = *Tags* (S)
    `Update_CrossEdges_Info` ($SeqNo(S)$, J_1, Counter)
    SeqNo (S) = Counter
    `Update_CrossEdges_Info` ($SeqNo(P)$, J_2, Counter)
    SeqNo (P) = Counter++
  end for
  for all [$S \in$ `SNS2`] **do**
    `Update_CrossEdges_Info` ($SeqNo(S)$, J_2, Counter)
    SeqNo (S) = Counter++
  end for
end for
for [i = (Highest *SeqNo* in J_1) + 1 to (Highest *SeqNo* in J_2)] **do**
  Let `SNS2` be the set of node sets from J_2 such that, $\forall S \in$ `SNS2`, (*SeqNo* (S) == i)
  `Update_CrossEdges_Info` ($SeqNo(S)$, J_2, Counter)
  SeqNo (S) = Counter++
end for

Algorithm 8 `Update_CrossEdges_Info` ($OrigSeqNo$, $Subgraph$, $NewSeqNo$)

for [each cross-edge Γ in *cross-edges* table] **do**
  if [$SeqNo$ of node set containing source node of Γ == $OrigSeqNo$ $\wedge$ source node is in $Subgraph$] **then**
    Replace ($SeqNo$ of node set containing source node) by $NewSeqNo$
  end if
  if [$SeqNo$ of node set containing target node of Γ == $OrigSeqNo$ $\wedge$ target node is in $Subgraph$] **then**
    Replace ($SeqNo$ of node set containing target node) by $NewSeqNo$
  end if
end for

Algorithm 9 `Combine_NodeSets` (K_1, K_2)

for [i = 1 to (Highest $SeqNo$ in K_1)] **do**
  Merge the node set k_1 in K_1 with $SeqNo(k_1)$ = i and the node set k_2 in K_2 with $SeqNo(k_2)$ = i to get node set in the k-bisimilar partition of the original graph.
end for
for [i = (Highest $SeqNo$ in K_1 + 1) to (Highest $SeqNo$ in K_2)] **do**
  Set the node set k_2 in K_2 with $SeqNo(k_2)$ = i as a node set in the k-bisimilar partition of the original graph.
end for

($\Leftarrow$): Given that two nodes a and b are k-bisimilar, we are required to show that they will belong to node sets with the same $SeqNo$ at the end of k iterations of the `for` loop. Since a and b are also $(k-1)$-bisimilar, by an inductive argument, they will occur in node sets with the same $SeqNo$ at the end of $(k-1)$ iterations as well. This holds for the parents u and v of a and b as well, since u and v are also $(k-1)$-bisimilar. Again, we look at two situations: where a and b belong to the same subgraph, and where they belong to different subgraphs.

Suppose a and b belong to the same subgraph, then by an inductive argument they belong to the same node set after $(k-1)$ iterations. Since the parents u and v of a and b occur in node sets with the same label, the $Tags$ of the node sets containing a and b will be the same. Thus the kth iteration does not split a and b into different node sets.

Now suppose that a and b belong to different subgraphs. The parents u and v of a and b occur in sets with the same $SeqNo$ after $(k-1)$ iterations (by induction). Since the sets containing a and b also have the same $SeqNo$ (again, after $(k-1)$ iterations), `Number_Buckets` will give the sets containing a and b the same $SeqNo$. Hence Proved.

6 Analysis of the Algorithm

In this section, we calcuate the time taken by our algorithm. Assume that the input graph has $O(m)$ edges, and each of the subgraphs has $O(m_1)$ and $O(m_2)$ edges respectively, where $m = m1 + m2$. The technique presented in [1] takes $O(km)$ time to compute the k-bisimilar partitions. We note that when the graph fits completely in main memory, our algorithm works slower than the algorithm in [1] due to the extra bookkeeping (such as table lookups on the *cross-edges* table) that has to be done when dealing with a split graph. Thus, we do not recommend using our algorithm when the graph fits completely in main memory; the technique presented in [1] is better suited for such data.

Let us consider the case when the graph does not fit in main memory, but fits in the address space of the machine; we make this assumption so that we can compare the working time of our algorithm with the algorithm presented in [1]. While our algorithm works even for data graphs that do not fit within the address space, the algorithm in [1] does not. We will only consider the cost of disk reads and writes in the analysis, since we expect these costs to dominate.

The algorithms as presented in 4 if used directly may lead to slightly higher I/O costs than if an optimized version were used. Since the same operations are carried out on both subgraphs, by keeping track of which subgraph is currently loaded, and changing slightly the order of operations on these subgraphs, we can reduce the number of I/O operations that should be carried out. In this section, as we analyze each algorithm, we also present the correct order in which they should be executed to minimize I/O costs. Moreover, we shall assume that only one subgraph can exist in memory at a given point in time, thus the analysis given here is the worst case time. Let us call the subgraphs G_1 and G_2.

`Compute_0_bisim` (G_1) requires loading G_1 into memory, which is an $O(m_1)$ operation. `Compute_0_bisim`(G_2) requires G_1 to be flushed to disk and G_2 to be read from disk: it is an $O(m_1) + O(m_2)$ operation. At this point G_2 is in memory, and the `Stitch_0_bisim` can be modified so as to first work on G_2, and then work on G_1. Hence this is an $O(m_1) + O(m_2)$ operation. If we assume (again the worst case) that for the "stitching" operation G_2 has to be streamed past G_1, it will require another $O(m_2)$ operations.

At the end of the `Stitch_0_bisim` procedure as discussed above G_1 is loaded in memory. The `Make_SuccStable` operation can be first performed on the partitions of G_1 followed by the second `Make_SuccStable` operation on the partitions of G_2. Together they are an $O(m_1) + O(m_2)$ operation, with G_2 in memory at the end of the operation. `Add_CrossEdges` now can work on G_2 by first considering those edges from the *cross-edges* table that have their target in G_2, followed by those edges that have their target in G_1. Hence this requires $O(m_1) + O(m_2)$ I/O operations, at the end of which G_1 is loaded in memory.

The `CleanUp` procedure works on G_1, and then on G_2, and both the `CleanUp` operations together cost $O(m_1)+O(m_2)$. `Number_Buckets` is tricky to analyse; at a first glance it looks as though both G_1 and G_2 must be loaded in memory. However this is not the case; only those node sets from G_1 and G_2 with equal $SeqNo$s need to be loaded into memory together. Since there are $O(m_1)$ distinct $SeqNo$s in G_1 and $O(m_2)$ distinct $SeqNo$s in G_2, the entire operation is an $O(m_1) + O(m_2)$ operation.

The I/O operations required are summarized below:

`Compute_0_bisim` (together)	$2O(m_1) + O(m_2)$
`Stitch_0_bisim`	$O(m_1) + 2O(m_2)$
`Make_SuccStable` (together)	$O(m_1) + O(m_2)$
`Add_CrossEdges`	$O(m_1) + O(m_2)$
`CleanUp` (together)	$O(m_1) + O(m_2)$
`Number_Buckets`	$O(m_1) + O(m_2)$

Hence the total I/O cost of the algorithm is $(4K + 3)(O(m_1) + O(m_2))$, which is $(4K + 3)(O(m))$. We note again that this is a worst case time.

The method suggested in [1] also costs roughly $O(Km)$ in I/O operations if we assume that each operation entails a disk access.

7 Working with an arbitrary number of subgraphs

The algorithm to compute k-bisimilar partition for a graph that is split into n subgraphs (n $>$ 2) is very similar to the algorithm with 2 subgraphs except for a few changes.

The algorithm starts off by splitting the graph into n subgraphs. Then it uses the `Compute_0_bisim` to compute the 0-bisimilar partitions for each of the subgraphs. All these partitions have the nodes grouped into sets based on their labels. Now $SeqNo$s are assigned to the node sets in each subgraph to establish the correspondence among node sets having the same $Label$ just as in `Stitch_0_bisim`. When considering a node set S in a partition, all previously processed partitions should be examined to see if one of the previously processed node sets had the same label as S. If so, S should be given the same $SeqNo$ as the other node set. This can be done in two ways as described in the discussion on `Stitch_0_bisim` in section 4.2. When using the streaming approach, all the previously processed partitions should be streamed against the partition that is currently being processed. This approach will obviously be slower than the approach of associating $Label$s with $SeqNo$s and storing this association in a temporary table. As discussed, this approach has the disadvantage that the number of distinct $Label$s may be very large. However, this approach can be made to work by splitting the subgraphs further so that temporary table and the partition currently being processed fit in memory.

The procedures `Make_SuccStable`, `Add_CrossEdges` and `CleanUp` remain the same, but are repeated k times, once for each subgraph. In `Number_Buckets`, when giving new $SeqNo$s to the node sets, all previous processed partitions will have to be examined to see if any of them had the same old $SeqNo$ and $Tags$ as the node set being examined. This might seem like we need to load all the partitions into memory. Again this is avoided by using the same approach as was mentioned for the `Number_Buckets` algorithm for the 2 partition case.

After this the k-bisimilar partition for the original graph is obtained as explained in section 4.2 by merging node sets which are in different partitions but have the same $SeqNo$.

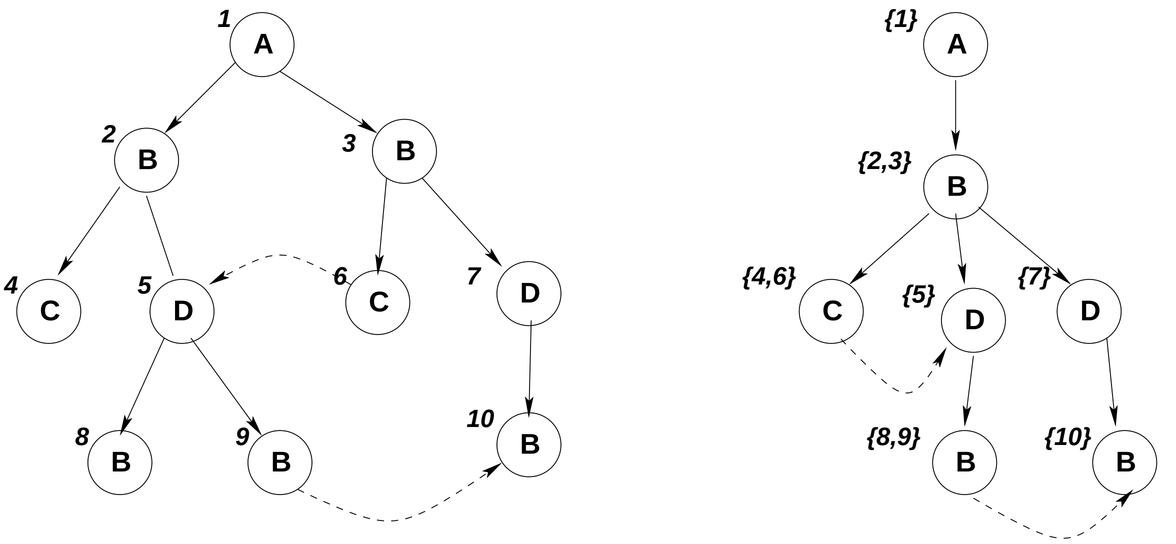
8 An Example

In this section we present an example to show the working of our algorithm. Figure 3 shows a graph and its 1-index obtained by using the algorithm in [1].

For our example, we will consider that the graph has been split into two subgraphs as shown in Figure 4. We will be using the same notation for a node set as the one we used when explaining the need for cleanup in section 4.2, i.e., SeqNo1.{Extent1}(Tags1) represents a node set with $SeqNo$ SeqNo1, extent Extent1 and $Tags$ Tags1. We will be using alphabets for $SeqNo$s.

First the 0-bisimilar partition of each subgraph is obtained by grouping the nodes with the same label together in a node set. The 0-bisimilar partition of each subgraph is listed in Figure 4.

After getting the 0-bisimilar partition of each subgraph, the correspondence required between the node sets in different subgraphs is established. This is done by giving the same $SeqNo$ for the node sets which contain nodes having the same label $Lab(N)$. For example,



(a) The original graph **(b) The index graph (A(1) index here)**

A(0) = { {1}, {2, 3, 8, 9, 10}, {4, 6}, {5, 7} }
A(1) = { {1}, {2,3}, {8,9}, {10}, {4, 6}, {5}, {7} }

Figure 3: Graph and A(1) index for the algorithm which does not split graphs

node sets $\{2, 8, 9\}$ in the subgraph 1 and $\{3, 10\}$ in subgraph 2 have nodes which have the same label 'b'. So they ought be given the same $SeqNo$. They are given the $SeqNo$ 'b' here. Note that the $SeqNo$ and the label of the nodes in the set need not be the same. The fact that they are same here is a mere coincidence.

At this point, we can obtain the 0-bisimilar partition for the whole graph by merging the node sets that have the same $SeqNo$. Note that this is true for any step p in the algorithm i.e., after p steps, we can get the p-bisimilar partition of the original graph just by merging the node sets that have the same $SeqNo$.

In the next step, we make the 0-bisimilar partition of each of the subgraphs to be succ-stable with respect to a copy of itself. The $Tags$ are also updated as explained in the algorithm 4. For example in subgraph 1, the successors of a.$\{1\}$ is $\{2\}$. In the 0-bisimilar partition of subgraph 1, the node set b.$\{$2,8,9$\}$ contains nodes that are not successors of a.$\{1\}$, which means it is not succ-stable with respect to a.$\{1\}$. In order to make it succ-stable, b.$\{$2,8,9$\}$ is split into b.$\{2\}$(a) and b.$\{$8,9$\}$(). Note that $SeqNo$ 'a' is added only to the $Tags$ of b.$\{2\}$. Now proceeding like this for all the node sets within the subgraph we get the list of node sets as shown in the second row of the table in the figure.

Now the $cross - edges$ are added to each partition. Consider the $cross - edge$ from 6 to 5. Since 5 is a the target of the cross edge, the node set d.$\{5\}$(b) in subgraph 1, in which 5 is contained has to be split into two sets, one containing only $SuccCross(6)$ and the other containing all the remaining nodes and $SeqNo$ of the set in which 6 is present should be added to the $Tags$ of the set which contains only $SuccCross(6)$. Here $SuccCross(6)$ is $\{5\}$ and so there is no need to split d.$\{5\}$(b). But the $SeqNo$ 'c' is added to its $Tags$ set making it d.$\{5\}$(b,c) as shown in row four of the table in the figure. Similarly by considering the $cross-edge$ from 9 to 10, the node set b.$\{10\}$(d) in the 0-bisimilar partition of subgraph 2 is changed to b.$\{10\}$(b,d).

After considering the edges of the subgraph and the crossedges, the node sets having the same $SeqNo$ and the same $Tags$ are matched as shown in the fourth row of the table in the figure. Then the matched node sets are given the same new $SeqNo$s.

Now the 1-bisimilar partition for the whole graph can be obtained by merging node sets having the same $SeqNo$. After this no further splits of node sets occur, so the algorithm stops.

9 Conclusions

We have demonstrated a technique that can be used to obtain the k-bisimilar partition of a graph by working on its subgraphs. The technique finds its practical application in being able to compute the A(k)-index of arbitrarily large data graphs.

Acknowledgements

We would like to thank Raghav Kaushik for useful discussions and suggestions. He also made available to us the code used to obtain the results in [1].

References

[1] R. Kaushik, P. Shenoy, P. Bohannon and E. Gudes. *Exploiting Local Similarity to Index Paths in Graph Structured Data*, ICDE 2002.

[2] R. Goldman and J. Widom. *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*, VLDB 1997.

[3] T. Milo and D. Suciu. *Index Structures for Path Expressions*, ICDT 1999.

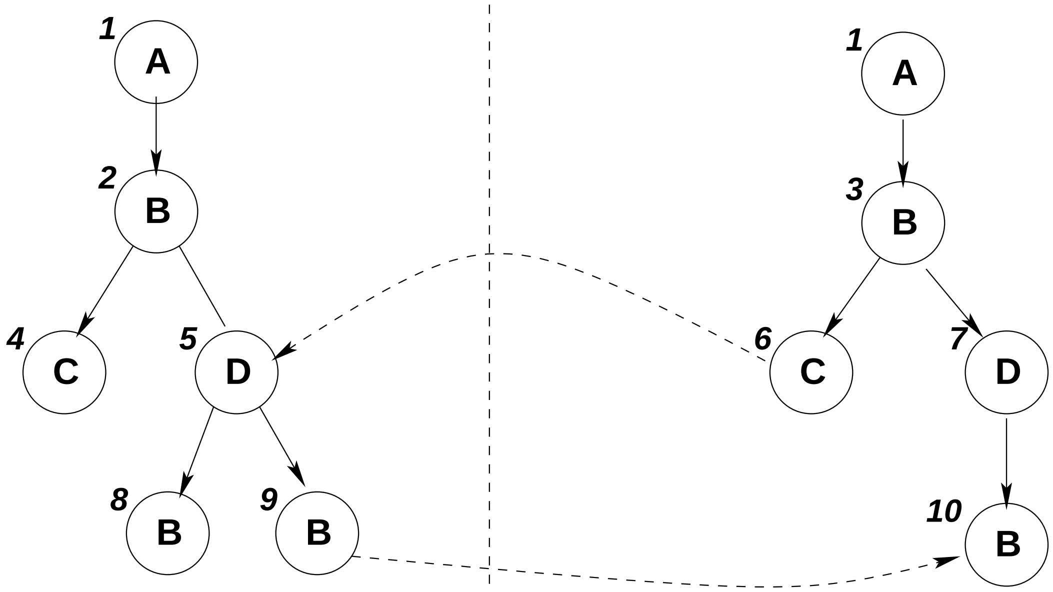
[4] R. Paige and R. E. Tarjan. *Three Partition Refinement Algorithms*, SIAM Journal on Computing 1987.

[5] R. Kaushik, P. Bohannon, J. F. Naughton, H. F. Korth. *Covering Indexes for Branching Path Queries*, SIGMOD 2002.

[6] R. Kaushik, P. Bohannon, J. F. Naughton, P. Shenoy. *Updates for Structure Indexes*, VLDB 2002.

[7] D. Park. *Concurrency and Automata on Infinite Sequences*, TCS 1981.

[8] R. Milner. *A Calculus for Communicating Processes*, LNCS 1980.



Subgraph 1	Subgraph 2	
a.{1} b.{2,8,9} c.{4} d.{5}	a.{1} b.{3, 10} c.{6} d.{7}	**A(0)**
a.{1} b.{2}(a),b.{8,9}(d) c.{4}(b) d.{5}(b)	a.{1} b.{3}(a), b.{10}(d) c.{6}(b) d.{7}(b)	**Computing A(1) from A(0) (Before adding cross edges)**
a.{1} b.{2}(a),b.{8,9}(d) c.{4}(b) d.{5}(b,c)	a.{1} b.{3}(a), b.{10}(b,d) c.{6}(b) d.{7}(b)	**Computing A(1) from A(0)**
a.{1}() b.{2}(a) c.{8,9} (d) d.{}() e.{4}(b) f.{5}(b) g.{}()	a.{1}() b.{3}(a) c.{}() d.{10}(b.d) e.{6}(b) f.{}() g.{7}(b)	**A(1)**

Figure 4: Splitting the graph and computing 1-bisimilarity