

Linear Computation of the Maximum Simultaneous Forward and Backward Bisimulation for Node-Labeled Trees

Nils Grimsmo, Truls Amundsen Bjørklund, and Magnus Lie Hetland

Norwegian University of Science and Technology
{nilsgri,trulsamu,mlh}@idi.ntnu.no

Abstract. The F&B-index is used to speed up pattern matching in tree and graph data, and is based on the maximum F&B-bisimulation, which can be computed in loglinear time for graphs. It has been shown that the maximum F-bisimulation can be computed in linear time for DAGs. We build on this result, and introduce a linear algorithm for computing the maximum F&B-bisimulation for tree data. It first computes the maximum F-bisimulation, and then refines this to a maximal B-bisimulation. We prove that the result equals the maximum F&B-bisimulation.

1 Introduction

Structure indexes are used to reduce the cost of pattern matching in labeled trees and graphs [5,14,9], by capturing structure properties of the data in a *structure summary*, where some or all of the matching can be performed. Efficient construction of such indexes is important for their practical usefulness [14], and in this paper we reduce the construction cost of the *F&B-index* [9] for tree data.

In a structure index, data nodes are typically *partitioned* into *blocks* based on properties of the surrounding nodes. A structure summary typically has one node per block in the partition, and edges between summary nodes where there are edges between data nodes in the related blocks. Matching in structure summaries is usually more efficient than partitioning the data nodes on label and using structural joins to find full query matches [14,9].

In a *path index*, data nodes are classified by the labels on the paths by which they are reachable [5,14]. For tree data this equals partitioning nodes on their label and the block of the parent node. Figures 1c and 1b show path partitioning and the related summary for the example data in Figure 1a. With path indexes, non-branching queries can be evaluated without processing joins [14,18].

A natural extension of a path index is the F&B-index, where nodes are partitioned on both their label, the partitions of the parents, *and* the partitions of the children, as shown in Figure 1d. This gives an index where more of the pattern matching can be performed on the summary, and also branching queries can be evaluated without processing joins [9].

The focus of this paper is efficient computation of the maximum *simultaneous forward and backward bisimulation* (F&B-bisimulation), which is the underlying

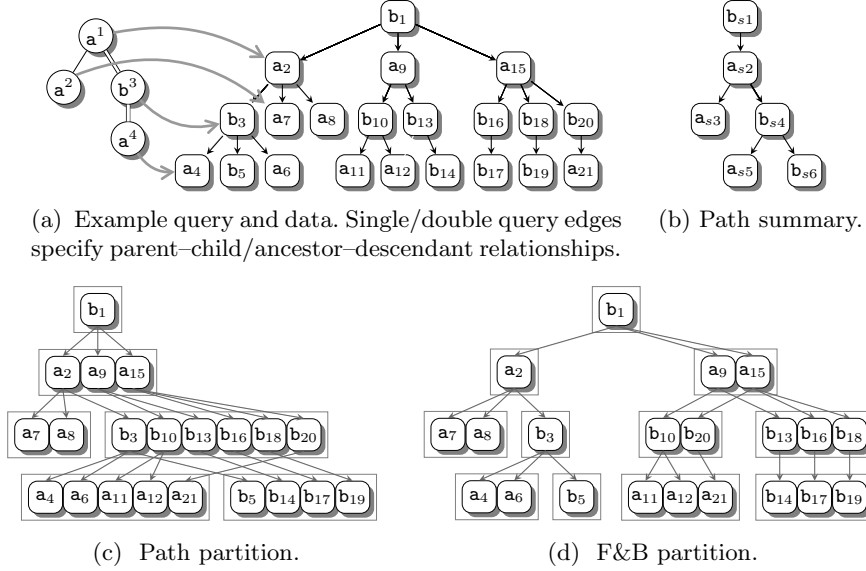


Fig. 1. Partitioning strategies. Superscripts and subscripts give node identifiers.

concept used to partition nodes in the F&B-index. It can be computed in time loglinear in the number of edges in the graph [9]. A linear construction algorithm for directed acyclic graphs (DAGs) has been presented recently [12], but we show that it is incorrect. On the other hand, there exists a correct algorithm which can compute *either* the maximum *forward bisimulation* (F-bisimulation) *or* the maximum *backward bisimulation* (B-bisimulation) in linear time for DAGs [3]. We extend this algorithm to compute the maximum F&B-bisimulation in linear time for tree data. This has relevance for applications where the underlying data is known to be tree shaped, such as in many uses of XML [6].

2 Background

In this section we present different types of bisimulations, and show how they can be computed by first partitioning on label, and then *stabilizing* the graph.

We use the following notation: A directed graph $G = \langle V, E \rangle$ has node set V and edge set $E \subseteq V \times V$. Let $n = |V|$ and $m = |E|$. For $X \subseteq V$, $E(X) = \{w \mid \exists v \in X : vEw\}$ and $E^{-1}(X) = \{u \mid \exists v \in X : uEv\}$. Each node $v \in V$ has label $L(v)$. A *partition* P of V is a set of *blocks*, such that each node $v \in V$ is contained in exactly one block. For an equivalence relation $\sim \subseteq V \times V$, the equivalence class containing $v \in V$ is denoted by $[v]_{\sim}$. The equivalence relation arising from the partition P is denoted $=_P$. A relation R_2 is a *refinement* of R_1 iff $R_2 \subseteq R_1$. A partition P_2 is a refinement of the *coarser* P_1 iff $=_{P_2} \subseteq =_{P_1}$. Let the *contraction graph* of a partition P be a graph with one node for each equivalence class of $=_P$, and an edge $\langle [u]_{=P}, [v]_{=P} \rangle$ whenever $\langle u, v \rangle \in E$.

The structure summary built for a structure index is typically isomorphic with the contraction graph for the data partition. For a partitioning to be useful, it must yield a summary that somehow simulates the data, such that pattern matching in the summary gives the same results as pattern matching in the data, or at least no false negatives. If nodes are partitioned into blocks where nodes in some way simulate each other, then the contraction graph also simulates the data in the same way.

2.1 Bisimulation and Bisimilarity

Broadly speaking, bisimulations relate nodes that have the same label and related neighbors. We use the following properties of a binary relation $R \subseteq V \times V$ to formally define the different types of bisimulation:

$$vRv' \Rightarrow L(v) = L(v') \quad (1)$$

$$vRv' \Rightarrow (uEv \Rightarrow \exists u' : u'Ev' \wedge uRu') \wedge (u'Ev' \Rightarrow \exists u : uEv \wedge uRu') \quad (2)$$

$$vRv' \Rightarrow (vEw \Rightarrow \exists w' : v'Ew' \wedge wRw') \wedge (v'Ew' \Rightarrow \exists w : vEw \wedge wRw') \quad (3)$$

Definition 1 (Bisimulations [13,9]). A relation R is a B-bisimulation iff it satisfies (1) and (2) above, an F-bisimulation iff it satisfies (1) and (3), and an F&B-bisimulation iff it satisfies (1), (2) and (3).

For each type, there exists a unique maximum bisimulation, of which all other bisimulations are refinements [13,9]. We say that two nodes are *bisimilar* if there exists a bisimulation that relates them, i.e., they are related by the maximum bisimulation. Since bisimilarity is an equivalence relation, it can be used to partition the nodes [13,9]. When nodes u and v are backward, forward, and forward and backward bisimilar, we write $u \sim_B v$, $u \sim_F v$ and $u \sim_{F\&B} v$, respectively. Figure 2 illustrates the different types of bisimilarity.

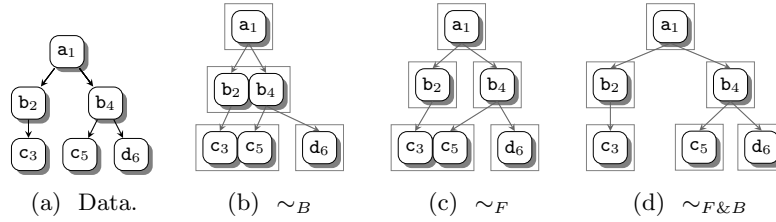


Fig. 2. Partitioning on different types of bisimilarity

Two *graphs* are said to be bisimilar if there exists a total surjective bisimulation from the nodes in one graph to the nodes in the other. For a given graph,

the smallest bisimilar graph is unique, and is exactly the contraction for bisimilarity [3]. The F&B-bisimilarity contraction is the basis for the F&B-index [9].

2.2 Stability

The different types of bisimilarity listed in the previous section can be computed by first partitioning the data nodes on label, and then finding the coarsest *stable* refinements of the initial partition [15,4,9]. A partition is *successor stable* iff all nodes in a block have incoming edges from nodes in the same set of blocks, and is *predecessor stable* iff all nodes in a block have outgoing edges to the same set of blocks [9]. The coarsest successor, predecessor, and successor and predecessor stable refinement of a label partition, equal a partition on B-bisimilarity, F-bisimilarity and F&B-bisimilarity, respectively [4,9].

Definition 2 (Partition stability [15,9]). Given a directed graph $G = \langle V, E \rangle$, then $D \subseteq V$ is *successor stable* with respect to $B \subseteq V$ if either all or none of the nodes in D are pointed to from nodes in B (meaning $D \subseteq E(B)$ or $D \cap E(B) = \emptyset$), and D is *predecessor stable* with respect to B if either none or all of the nodes in D point to nodes in B (meaning $D \subseteq E^{-1}(B)$ or $D \cap E^{-1}(B) = \emptyset$).

For any combination of successor and predecessor stability, a partition P of V is said to be stable with respect to a block B if all blocks in P are stable with respect to B . A partition P is stable with respect to another partition Q if it is stable with respect to all blocks in Q . P is said to be stable if it is stable with respect to itself.

Figure 3 shows cases where a block can be split to achieve different types of stability. The block D is not stable with respect to B , but we can split it into blocks that are: Assume that D is stable with respect to a union of blocks S such that $B \subset S$. We can split D into blocks that are stable with respect to both B and $S \setminus B$, shown as D_B , D_{BS} and D_S in the figure. Stabilizing also with respect to $S \setminus B$ is crucial for obtaining a $\mathcal{O}(m \log n)$ running time in the partition stabilization algorithm explained in the next section.

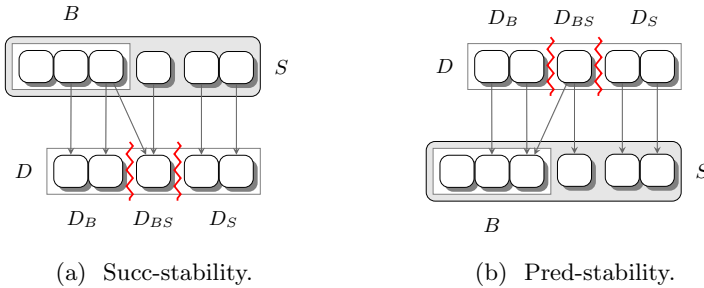


Fig. 3. Refining D into blocks that are stable with respect to B and $S \setminus B$

2.3 Stabilizing Graph Partitions

We now go through Paige and Tarjan’s algorithm for refinement to the coarsest predecessor stable partition [15], extended to simultaneous successor and predecessor stability by Kaushik et al. [9], as shown in Algorithm 1. The input to the algorithm is a partition P and the set of flags $Directions \subseteq \{\text{SUCC}, \text{PRED}\}$, which specifies whether P is to be successor and/or predecessor stabilized.

Algorithm 1. Graph partition stabilization

```

1:  $\triangleright P$  is the initial partition.
2: function StabilizeGraph( $P, Directions$ ):
3:   for  $dir \in Directions$ :
4:     InitialRefinement( $P, dir$ )
5:    $X \leftarrow \{\bigcup_{B \in P} B\}$ 
6:   while  $P \neq X$ :
7:     Extract  $B \in P$  such that  $B \subset S \in X$  and  $|B| \leq |S|/2$ .
8:     Replace  $S$  by  $B$  and  $S \setminus B$  in  $X$ .
9:     for  $dir \in Directions$ :
10:      StabilizeWRT(copy of  $B, P, dir$ )

```

Figure 4 illustrates an example run of the algorithm with $Directions = \{\text{SUCC}, \text{PRED}\}$. In addition to the current partition P , the algorithm maintains a partition X , where the blocks are unions of blocks in P . Initially X contains a single block that is the union of all blocks in P , and the algorithm maintains the loop invariant that P is stable with respect to X by Definition 2. The algorithm terminates when the partitions P and X are equal, which means P must be stable with respect to itself. But the loop invariant may not be true for the given input partition initially: Blocks containing both roots and non-roots are not successor stable with respect to X , because non-roots have incoming edges from the single block $S \in X$, while roots do not. Similarly, blocks containing both leaves and non-leaves are not predecessor stable with respect to X . In Algorithm 1 initial stability is achieved by calls to InitialRefinement(), which splits blocks in a simple linear pass. Initial splitting is illustrated by the step from line (a) to line (b) in Figure 4.

The algorithm repeatedly selects a block $S \in X$ that is a *compound* union of blocks from P , and selects a *splitter block* $B \subset S$ with size at most half of S . Then S is replaced by B and $S \setminus B$ in X , as shown when extracting $B = \{\mathbf{a}_2, \mathbf{a}_9, \mathbf{a}_{15}\}$ between lines (b) and (c) in Figure 4. The call to StabilizeWRT() uses the strategies depicted in Figure 3 to stabilize P with respect to both B and $S \setminus B$, to make sure P is stable with respect to the new X . It is important to use a *copy* of B as splitter, as the stabilization may cause B itself to be split. The step from line (b) to (c) in the figure shows that a block of nodes labeled \mathbf{a} is split when successor stabilizing with respect to $B = \{\mathbf{a}_2, \mathbf{a}_9, \mathbf{a}_{15}\}$.

Efficient implementation of the above requires some attention to detail [15]: The partition X can be realized through a set \mathcal{X} containing sets of pointers to

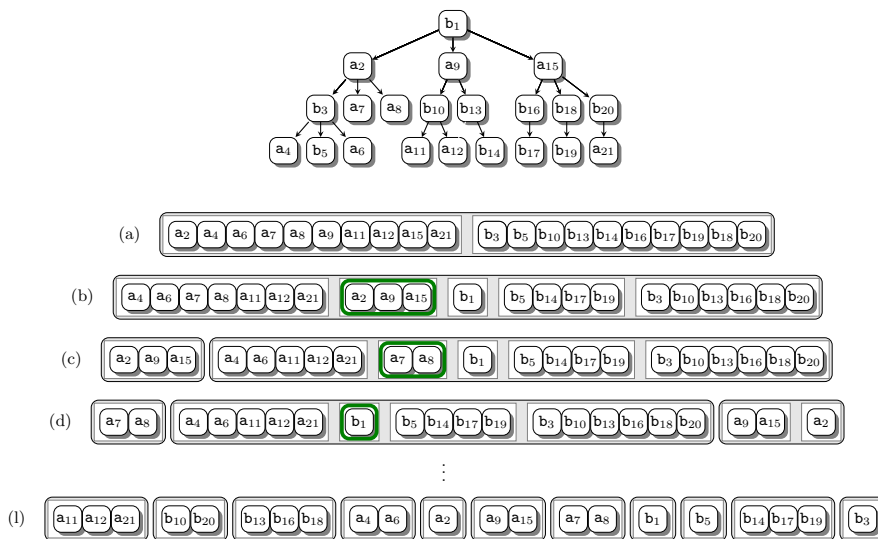


Fig. 4. Algorithm 1 doing successor and predecessor stabilization on a label partition of the data from Figure 1a. The white boxes are the current blocks in P , while the gray boxes are the current blocks in X . Line (a) shows initial label partition. Step (a)–(b) shows refinement separating roots from non-roots and leaves from non-leaves, and steps (b)–(l) show simultaneous predecessor and successor stabilization.

blocks in P , such that for each $S \in X$ we have $S = \bigcup_{B \in \mathcal{S}} B$ for the related $\mathcal{S} \in \mathcal{X}$. To extract a $B \subset S \in X$ in constant time, we also maintain the set of compound unions $\mathcal{C} = \{\mathcal{S} \in \mathcal{X} \mid 1 < |\mathcal{S}|\}$. A block $B \subset S$ such that $|B| \leq |S|/2$ can be found by choosing the smallest of the two first blocks in any $\mathcal{S} \in \mathcal{C}$. Note that we can check if $P = X$ by checking whether \mathcal{C} is empty, and neither P , X nor \mathcal{X} need to be materialized, as they are never used directly. You only need to maintain \mathcal{C} , and a $\mathcal{P} \subseteq P$ containing the blocks in P not in some $\mathcal{S} \in \mathcal{C}$. For inserting and removing elements in constant time, the sets are implemented as doubly linked list. In addition, each $v \in B$ has a pointer to B , and each $B \in \mathcal{S}$ has a pointer to \mathcal{S} . This allows keeping the data structures updated throughout the evaluation.

As all operations in the while-loop excluding the calls to `StabilizeWRT()` are constant time operations on linked lists, the complexity of the loop is bounded by the number of splitter blocks selected, which again is bounded by the number of times a node may be part of such a splitter. Splitter blocks at most half the size of a compound union are always selected, and no node in this block is part of a splitter again before the block itself has become a compound union. This means that the number of times a given node is part of a splitter block is $\mathcal{O}(\log n)$, and that the total number of splitter blocks used is $\mathcal{O}(n \log n)$ [15].

Algorithm 2 shows how all blocks in a partition P can be successor (or predecessor) stabilized with respect to a block $B \in P$ and $S \setminus B$, where $B \subset S \in X$, in

time linear in the number of edges going out from (or into) B [15]. For successor stability, only blocks D pointed to from B are affected, and they are stabilized with respect to both B and $S \setminus B$ without involving $S \setminus B$ directly. This is done by maintaining for each node $v \in V$, the number of times it is pointed to from each set $S \in X$, and storing references to these records from the related edges. We can then differentiate between nodes pointed to from B , pointed to from both B and $S \setminus B$, and pointed to only from $S \setminus B$. Nodes from the first two categories are moved into new blocks, while the rest are untouched.

Algorithm 2. Stabilizing with respect to a block

```

1: function StabilizeWRT( $B, P, d$ ):
2:   Assume  $dir = \text{SUCC}$ . (or PRED)
3:   for  $D \in P$  pointed to from  $B$ : (or pointing into  $B$ )
4:     Initialize  $D_B$  and  $D_{BS}$  and associate with  $D$ .
5:     for  $v \in D \in P$  pointed to from  $B$ : (or pointing into  $B$ )
6:       if  $v$  is pointed to only from  $B$ : (or pointing only into  $B$ )
7:          $D' \leftarrow D_B$ 
8:       else:
9:          $D' \leftarrow D_{BS}$ 
10:      if  $D' \notin P$ : Insert  $D'$  after  $D$  in  $P$ 
11:      Move  $v$  from  $D$  to  $D'$ .
12:      if  $D = \emptyset$ : Remove  $D$  from  $P$ .

```

As the cost of a single call to `StabilizeWRT()` is bounded by the number of nodes in the splitter block and the number of outgoing (or incoming) edges, the total cost for the calls is $\mathcal{O}((m+n)\log n)$, as a given node or edge is used for splitting $\mathcal{O}(\log n)$ times. Assuming $n \in \mathcal{O}(m)$, the cost of Algorithm 1 is $\mathcal{O}(n)$ for the initial refinement, $\mathcal{O}(n \log n)$ for the while-loop excluding `StabilizeWRT()`, and $\mathcal{O}(m \log n)$ for the `StabilizeWRT()` calls, giving a total of $\mathcal{O}(m \log n)$ [15,9].

3 Linear Time Stabilization

Linear time computation of F&B-bisimilarity for DAG data has been attempted earlier. The SAM algorithm [12] partitions the data separately on B-bisimilarity and F-bisimilarity, and then combines the partitions by putting two nodes in the same final block iff they are in the same blocks in both partitions. It builds on the following theorem, which is stated without proof or reference: “*Node n_1 and node n_2 satisfy F&B-bisimulation if and only if they satisfy F-bisimulation and B-bisimulation.*” The *only if* part is of course true, but the *if* part is not, as can be seen from the partitioning of a tree with six nodes in Figure 2. Here $c_3 \sim_B c_5$ and $c_3 \sim_F c_5$, but $c_3 \not\sim_{F\&B} c_5$, because for the parent nodes $b_2 \not\sim_{F\&B} b_4$. Also note that it is assumed for the running time of the SAM algorithm that the number of edges to and from each node can be viewed as a constant.

3.1 Stabilizing DAG Partitions

We now present an algorithm for refining a partition of the nodes in a DAG *either* to successor stability *or* to predecessor stability. It is based on two different previous algorithms: Paige and Tarjan’s loglinear algorithm for stabilizing general graphs [15], and Dovier, Piazza and Policriti’s algorithm for computing F-bisimilarity on unlabeled graphs [3], which has linear complexity for DAG data. A difference between these two algorithms is that the former is given an initial partition as input, which is then *refined*, while the latter starts with the set of singleton blocks, from which the final partition is *constructed*. These are called *negative* and *positive* strategies, respectively [3]. Dovier et al. describe how their algorithm can be extended to compute F-bisimilarity for *labeled* data, but when developing an algorithm for refining to simultaneous successor and predecessor stability in the next section, we use the result of a predecessor stabilization as input to a successor stabilization, and hence cannot use a positive strategy.

Dovier et al.’s algorithm initially computes the *rank* of each node in the DAG, which is the length of the longest path from the node to a leaf. We extend the notion of rank to both directions in the DAG:

Definition 3 (Rank). In a DAG G , the successor and predecessor rank of $v \in V(G)$ is defined as:

$$\begin{aligned}
 \text{rank}_{\text{SUCC}}(v) &= \begin{cases} 0 & \text{if } v \text{ is a root in } G \\ 1 + \max_{\langle u,v \rangle \in E(G)} \text{rank}_{\text{SUCC}}(u) & \text{otherwise} \end{cases} \\
 \text{rank}_{\text{PRED}}(v) &= \begin{cases} 0 & \text{if } v \text{ is a leaf in } G \\ 1 + \max_{\langle v,w \rangle \in E(G)} \text{rank}_{\text{PRED}}(w) & \text{otherwise} \end{cases}
 \end{aligned}$$

Algorithm 3 shows our modification of Paige and Tarjan’s algorithm [15] based on Dovier et al.’s principles [3]. It refines a partition of a DAG *either* to predecessor *or* to successor stability, and runs in linear time, due to the order in which splitter blocks are chosen.

Algorithm 3. DAG partition stabilization

- 1: ▷ Assume sets are ordered.
 - 2: **function** StabilizeDAG(P, dir):
 - 3: RefineAndSortOnRank(P, dir)
 - 4: $X \leftarrow \{\bigcup_{B \in P} B\}$
 - 5: **while** $P \neq X$:
 - 6: Extract first $B \in P$ such that $B \subset S \in X$.
 - 7: Replace S by B and $S \setminus B$ in X .
 - 8: StabilizeWRT(B copy, P, dir)
-

A run of the algorithm with $dir = \text{PRED}$ is illustrated in Figure 5. Instead of only separating between leaves and non-leaves (or roots and non-roots) as in

Algorithm 1, blocks are initially split such that predecessor (or successor) rank is uniform within each block, and sorted, such that the rank is monotonically increasing in the partition. This is done in the function `RefineAndSortOnRank()`, which is described later in this section. An initial refinement and sorting on predecessor rank is shown when going from line (a) to line (b) in Figure 5.

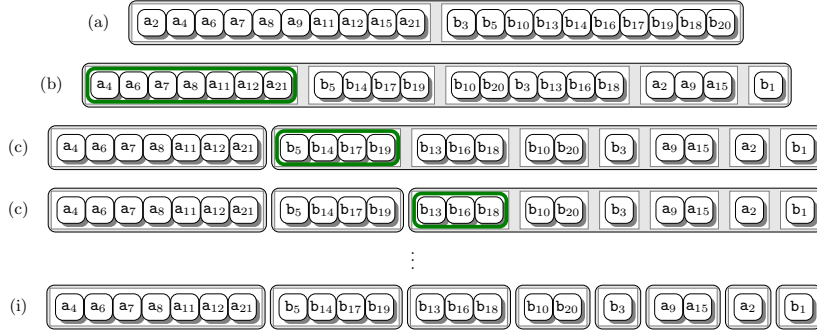


Fig. 5. Using Algorithm 3 to predecessor stabilize a label partition of the data from Figure 1a. The first step shows refinement on predecessor rank.

In a partition that respects a given type of rank, let the rank of a block be equal to the rank of the contained nodes. The following lemma implies that the initial refinement on rank does not split blocks unnecessarily:

Lemma 1. Given nodes $u, v \in V(G)$ and a partition P of G , if P is successor stable then $[u]_{=P} = [v]_{=P} \Rightarrow \text{rank}_{\text{SUCC}}(u) = \text{rank}_{\text{SUCC}}(v)$, and if P is predecessor stable then $[u]_{=P} = [v]_{=P} \Rightarrow \text{rank}_{\text{PRED}}(u) = \text{rank}_{\text{PRED}}(v)$.

Proof. For F-bisimilarity $[u]_{\sim_F} = [v]_{\sim_F} \Rightarrow \text{rank}_{\text{PRED}}(u) = \text{rank}_{\text{PRED}}(v)$ [3]. If P is predecessor stable, then $=_P$ is an F-bisimulation [4], and therefore a refinement of partitioning on \sim_F , such that $[u]_{=P} = [v]_{=P} \Rightarrow [u]_{\sim_F} = [v]_{\sim_F}$. The case for successor stability is symmetric. \square

The next lemma implies that if blocks are chosen as splitters in order of their rank, a node will be part of a block that is used as a splitter at most once. This property is used to achieve a linear complexity in our algorithm.

Lemma 2 ([3]). Given a DAG G , a partition P of G that respects predecessor rank and a block $B \in P$, predecessor stabilization of P with respect to B only splits blocks D where $\text{rank}_{\text{PRED}}(D) > \text{rank}_{\text{PRED}}(B)$.

This is symmetric for successor stabilization and successor rank, as a reversed DAG is also a DAG.

In Algorithm 3 the blocks in the current partition P are maintained ordered on rank. This is implemented through a detail in our method for stabilization with respect to a block in Algorithm 2, which is different from the original description [15]: The new blocks D_B and D_{BS} are inserted into P on the position after the old block D , and not at the end of P . The sets of blocks which make up the unions in X are also ordered such that their concatenation yields an ordered list of blocks. This is maintained by inserting B followed by $S \setminus B$ on the original position of S in X . Notice how blocks never change positions during the stabilization shown in lines (b)–(i) in Figure 5.

In Davier et al.’s algorithm, rank is computed by performing a topological traversal of the DAG depth first [3]. Because we need to *refine* a given partition on rank, as opposed to *constructing* a partition on rank from scratch, the problem is slightly more involved. Algorithm 4 shows how a partitioning can be refined and sorted on successor or predecessor rank in a single pass. The algorithm traverses the DAG with a hybrid between a topological sort and a breadth first search, implemented using edge counters and a queue. Blocks are refined and sorted on the fly.

Algorithm 4. Refining and sorting on rank

```

1: function RefineAndSortOnRank( $P, dir$ ):
2:   Assume  $dir = \text{SUCC}$  (or  $dir = \text{PRED}$ )
3:    $Q$  is a queue.
4:   for  $v \in V$ :
5:      $v.count \leftarrow |\{x \mid \langle x, v \rangle \in E\}|$  (or  $|\{x \mid \langle v, x \rangle \in E\}|$ )
6:     if  $v.count = 0$ :
7:        $v.rank_{dir} \leftarrow 0$ 
8:       PushBack( $Q, v$ )
9:   for  $B \in P$ :
10:     $B.currRank \leftarrow -1$ 
11:   while  $Q$ :
12:     $v \leftarrow \text{PopFront}(Q)$ 
13:    Let  $B \ni v$ .
14:    if  $v.rank_{dir} \neq B.currRank$ :
15:       $B.rankedB \leftarrow \{\}$ 
16:      Append  $B.rankedB$  at the end of  $P$ .
17:       $B.currRank \leftarrow v.rank_{dir}$ 
18:      Move  $v$  from  $B$  to  $B.rankedB$ 
19:      Remove  $B$  from  $P$  if empty.
20:    for  $x$  where  $\langle v, x \rangle \in E$ : (or  $\langle x, v \rangle \in E$ )
21:       $x.count \leftarrow x.count - 1$ 
22:      if  $x.count = 0$ :
23:         $x.rank_{dir} \leftarrow v.rank_{dir} + 1$ 
24:        PushBack( $Q, x$ )
    
```

Lemma 3. Algorithm 4 refines and orders P on successor (or predecessor) rank in $\mathcal{O}(m + n)$ time.

Proof (Sketch). Because the queue is initialized with the roots, and a node is added to the queue when the last parent is popped from the queue, the nodes are queued and popped in order of successor rank, and this distance is calculated from the parent node with the greatest successor rank. As the successor rank of the nodes that are moved to a new block grows monotonically, only one associated block $B.\text{ranked}B$ is created per successor rank found in block B , and all such blocks are appended to P in sorted order. As a node is only queued once, and the cost of processing a node is proportional to the number of outgoing edges, the total running time of the algorithm is $\mathcal{O}(m + n)$. The case for predecessor rank is symmetric. \square

Theorem 1. Algorithm 3 yields the coarsest refinement of a partition of a DAG that is successor (or predecessor) stable.

Proof (Sketch). For predecessor stability, the only differences from Paige and Tarjan’s algorithm are the initial refinement and the order in which splitter blocks are chosen. By Lemma 1 blocks are not refined unnecessarily when refining on rank, and the order of split operations is not used in the correctness proof for the original algorithm [15]. Successor stability is symmetric. \square

To implement Algorithm 3 we use the same underlying data structures as for Algorithm 1: doubly linked lists \mathcal{C} and \mathcal{P} realizing X and P . In Algorithm 3, the extract operation is implemented by removing the first $B \in \mathcal{S}$ from the first $\mathcal{S} \in \mathcal{C}$. The replace operation is implemented by moving B from \mathcal{S} to the end of \mathcal{P} , and if only one block B' is left in \mathcal{S} , this B' is also moved from \mathcal{S} to \mathcal{P} , and \mathcal{S} is removed from \mathcal{C} .

Theorem 2. The running time of Algorithm 3 is $\mathcal{O}(m + n)$.

Proof (Sketch). We analyze the cost of `StabilizeWRT()` separately. Outside the while loop, the call to `RefineAndSortOnRank()` has cost $\mathcal{O}(m + n)$ by Lemma 3, and the construction of \mathcal{C} and \mathcal{P} has cost $\mathcal{O}(|P|) \subseteq \mathcal{O}(n)$. As splitters are chosen in order of their rank, by Lemma 2 a splitter block is not later split itself. This means that each node is only part of a splitter once, and that the while-loop is run $\mathcal{O}(n)$ times. The loop condition is implemented by checking if $\mathcal{C} \neq \emptyset$. As all the operations on linked lists inside the loop have complexity $\mathcal{O}(1)$, the total cost of the while-loop is $\mathcal{O}(n)$. The `StabilizeWRT()` function is called $\mathcal{O}(n)$ times, and the cost of one call is linear in the number of nodes and edges used [15]. As nodes are only part of a splitter block once, edges are also only used for splitting once, and the total cost is $\mathcal{O}(m + n)$. \square

3.2 Stabilizing Trees

We now present an algorithm for finding the coarsest successor *and* predecessor stable refinement of the nodes in a tree. It uses the solution for DAGs from the previous section to refine a partition *first* to the coarsest predecessor stable refinement, and *then* to the coarsest successor stable refinement, as shown in

Algorithm 5. Stabilization for trees

```

1: function StabilizeTree( $P$ ,  $Directions$ ):
2:   if  $PRED \in Directions$ :
3:     StabilizeDAG( $P$ ,  $PRED$ )
4:   if  $SUCC \in Directions$ :
5:     StabilizeDAG( $P$ ,  $SUCC$ )
    
```

Algorithm 5. For trees this yields a partition that is still predecessor stable, as we prove in the following.

Figure 6 shows with a continuation of Figure 5 how Algorithm 5 is used to find a successor and predecessor stable refinement. The starting point in the figure is a predecessor stable refinement found after calling $StabilizeDAG(P, PRED)$. This partition is then successor stabilized by calling $StabilizeDAG(P, SUCC)$, which first refines and sorts on successor rank, shown between lines (i) and (j) in the figure, and then uses the blocks in the current P as splitters in order, shown in lines (j)–(t). Compare this partition with the F&B-bisimilarity partition in Figure 1d.

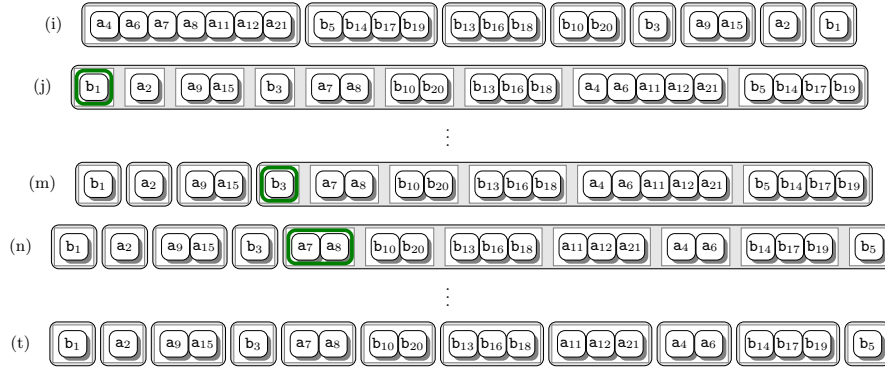


Fig. 6. Continuing Fig. 5: Line (i) shows predecessor stable partition, step (i)–(j) shows successor rank refinement, and steps (j)–(t) show successor stabilization

Theorem 3. If a predecessor stable partition of the nodes in a tree is refined to successor stability, the resulting partition is still predecessor stable.

Proof (Sketch). Blocks are split in three ways: To refine on rank, with respect to a block $B \in P$, or as a side effect with respect to $S \setminus B$ in Algorithm 2. By Lemma 1, the first type of split does not cause any split that would not eventually be caused by an algorithm that iteratively refines P with respect to a random block $B \in P$, and from the correctness proof of Paige and Tarjan’s algorithm [15], neither does splitting with respect to $S \setminus B$.

We now use induction on the refinement steps, and show that the partition P remains predecessor stable. It is true initially by assumption. The induction step is to split a block $D \in P$ on successor stability with respect to a block $B \in P$.

B will split D into two parts D_B and D_S , containing the nodes pointed to and not pointed to from B , respectively. The splitting of D may only affect the predecessor stability of the new blocks D_B and D_S with respect to their descendants, and of the set of blocks \mathcal{B} pointing into D with respect to D_B and D_S . After the split, $B \subseteq E^{-1}(D_B)$ and $B \cap E^{-1}(D_S) = \emptyset$, and for all other $B' \in \mathcal{B}$ we have that $B' \cap E^{-1}(D_B) = \emptyset$ and $B' \subseteq E^{-1}(D_S)$, because these B' by assumption have pointers into D , but by the fact that the data is a tree, do not have pointers into D_B .

For any block G pointed to from some node in D , $D \subseteq E^{-1}(G)$ by the initial assumption of predecessor stability, meaning all nodes in D point into G . This means that all nodes in D_B and D_S also point into G , and thus D_B and D_S are predecessor stable with respect to G . \square

Figure 7a illustrates this theorem. By contrast, assuming the data was not a tree, the blocks $B' \in \mathcal{B}$ would not necessarily be predecessor stable after splitting D , as shown in Figure 7b.

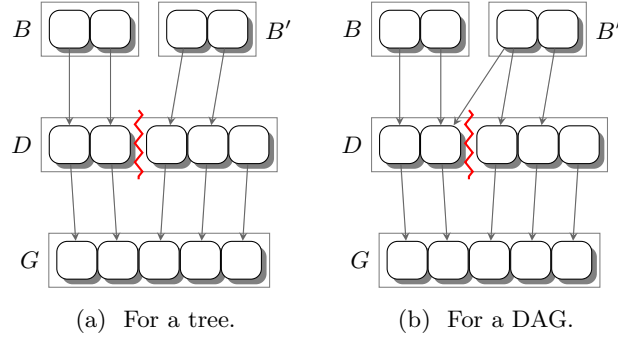


Fig. 7. Splitting D for successor stability w.r.t. B in a partition P . This does not impact predecessor stability for any block given tree data, but may break predecessor stability in a DAG.

Theorem 4. Algorithm 5 finds the coarsest refinement of a partition of the nodes in tree that is successor and predecessor stable in $\mathcal{O}(n)$ time.

Proof. From Theorems 1, 2 and 3, and the fact that $m \in \mathcal{O}(n)$ for tree data. \square

Corollary 1. The F&B-index can be built in $\mathcal{O}(n)$ time for tree data using Algorithm 5.

Proof (Sketch). The coarsest successor and predecessor stable refinement of a partitioning on label gives the maximum F&B-bisimulation [9], and the summary structure can be constructed from the contraction graph [9].

4 Related Work

There are many variations of structure summaries for graph data, such as partitionings on similarity [8,14], the F+B-index [9], the $A(k)$ index [11], and the $D(k)$ -index [2]. Note that an implication of Theorem 3 is that the F+B-index and the F&B-index are identical for tree data. The cost of matching in the summary can be reduced by label-partitioning it and using specialized joins [1], or using multi-level structure indexing [17]. For queries using ancestor–descendant edges on graph data, different graph encodings offer trade-offs between space usage and query time [20]. For a general overview of indexing and search in XML see the survey by Gou and Chirkova [6]. There is previous research on updates of bisimilarity partitions [10,19,16]. Some of these methods trade update time for coarseness, as refinements of bisimilarity may be cheaper to compute after a data update. Single directional bisimulations are used in many fields, such as modal logic, concurrency theory, set theory, formal verification, etc. [3], but to our knowledge, F&B-bisimulation is not frequently used outside XML search.

5 Conclusions and Future Work

In this paper we have improved the running time for refining a partition to the coarsest simultaneous successor and predecessor stability for tree data from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$, and with that the computation of F&B-bisimilarity, and construction of the F&B-index.¹ An incorrect linear algorithm for DAGs has been presented recently [12], and it would be interesting to know whether the problem is actually solvable in linear time for DAGs.

A natural extension of our work would be to reduce the cost of updates in F&B-bisimilarity partitions for trees. A particularly interesting direction would be to improve indexing performance for typical XML document collections, where there is a large number of small independent documents. It may be possible to iteratively add documents to the index with (expected) cost dependent only on the size of the documents.

Acknowledgments. This material is based upon work supported by the iAd Project funded by the Research Council of Norway and the Norwegian University of Science and Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the funding agencies.

References

1. Bača, R., Krátký, M., Snášel, V.: On the efficient search of an XML twig query in large DataGuide trees. In: Proc. IDEAS (2008)
2. Chen, Q., Lim, A., Ong, K.W.: $D(k)$ -index: an adaptive structural summary for graph-structured data. In: Proc. SIGMOD (2003)

¹ See the extended version of this paper for some performance experiments [7].

3. Dovier, A., Piazza, C., Policriti, A.: A fast bisimulation algorithm. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 79. Springer, Heidelberg (2001)
4. Fernandez, J.-C.: An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.* 13(2-3) (1990)
5. Goldman, R., Widom, J.: DataGuides: Enabling query formulation and optimization in semistructured databases. In: *Proc. VLDB* (1997)
6. Gou, G., Chirkova, R.: Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.* (2007)
7. Grimsmo, N., Bjørklund, T.A., Hetland, M.L.: Linear computation of the maximum simultaneous forward and backward bisimulation for node-labeled trees (extended version). Technical Report IDI-TR-2010-10, NTNU, Trondheim, Norway (2010)
8. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: *Proc. FOCS* (1995)
9. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: *Proc. SIGMOD* (2002)
10. Kaushik, R., Bohannon, P., Naughton, J.F., Shenoy, P.: Updates for structure indexes. In: *Proc. VLDB* (2002)
11. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: *Proc. ICDE* (2002)
12. Liu, X., Li, J., Wang, H.: SAM: An efficient algorithm for F&B-index construction. In: *Proc. APWeb/WAIM* (2007)
13. Milner, R.: *Communication and concurrency*. Prentice-Hall, Inc., Englewood Cliffs (1989)
14. Milo, T., Suciu, D.: Index structures for path expressions. In: Beeri, C., Bruneman, P. (eds.) *ICDT 1999*. LNCS, vol. 1540, pp. 277–295. Springer, Heidelberg (1998)
15. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. *SIAM J. Comput.* (1987)
16. Saha, D.: An incremental bisimulation algorithm. In: *Proc. FSTTCS* (2007)
17. Wu, X., Liu, G.: XML twig pattern matching using version tree. *Data & Knowl. Eng.* (2008)
18. Yang, B., Fontoura, M., Shekita, E., Rajagopalan, S., Beyer, K.: Virtual cursors for XML joins. In: *Proc. CIKM* (2004)
19. Yi, K., He, H., Stanoi, I., Yang, J.: Incremental maintenance of XML structural indexes. In: *Proc. SIGMOD* (2004)
20. Yu, J.X., Cheng, J.: Graph reachability queries: A survey. In: *Managing and Mining Graph Data* (2010)