Efficient External-Memory Bisimulation on DAGs

Jelle Hellings*
Hasselt University and
Transnational University of
Limburg
Belgium
jelle.hellings@uhasselt.be

George H.L. Fletcher
Eindhoven University of
Technology
The Netherlands
g.h.l.fletcher@tue.nl

Herman Haverkort
Eindhoven University of
Technology
The Netherlands
cs.herman@haverkort.net

ABSTRACT

In this paper we introduce the first efficient external-memory algorithm to compute the bisimilarity equivalence classes of a directed acyclic graph (DAG). DAGs are commonly used to model data in a wide variety of practical applications, ranging from XML documents and data provenance models, to web taxonomies and scientific workflows. In the study of efficient reasoning over massive graphs, the notion of node bisimilarity plays a central role. For example, grouping together bisimilar nodes in an XML data set is the first step in many sophisticated approaches to building indexing data structures for efficient XPath query evaluation. To date, however, only internal-memory bisimulation algorithms have been investigated. As the size of real-world DAG data sets often exceeds available main memory, storage in external memory becomes necessary. Hence, there is a practical need for an efficient approach to computing bisimulation in external memory.

Our general algorithm has a worst-case IO-complexity of $O(\textsc{Sort}(|N| + |E|))$, where $|N|$ and $|E|$ are the numbers of nodes and edges, resp., in the data graph and $\textsc{Sort}(n)$ is the number of accesses to external memory needed to sort an input of size n. We also study specializations of this algorithm to common variations of bisimulation for tree-structured XML data sets. We empirically verify efficient performance of the algorithms on graphs and XML documents having billions of nodes and edges, and find that the algorithms can process such graphs efficiently even when very limited internal memory is available. The proposed algorithms are simple enough for practical implementation and use, and open the door for further study of external-memory bisimulation algorithms. To this end, the full open-source C++ implementation has been made freely available.

*Research done while at Eindhoven University of Technology.

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

Categories and Subject Descriptors

H.2.0 [**Database Management**]: General; F.2.2 [**Analysis Of Algorithms And Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Theory

Keywords

bisimulation, graphs, external memory, I/O

1. INTRODUCTION

Data modeled as directed acyclic graphs (DAGs) arise in a diversity of practical applications such as biological and biomedical ontologies [28], web folksonomies [25], scientific workflows [30], semantic web schemas [9], business process modeling [6, 12], data provenance modeling [22, 23], and the widely adopted XML standard [1]. It is anticipated that the variety, uses, and quantity of DAG-structured data sets will only continue to grow in the future.

In each of these application areas, efficient searching and querying on the data is a basic challenge. In reasoning over massive data sets, typically index data structures are computed and maintained to accelerate processing. These indexes are essentially a reduction or summary of the underlying data. Efficiency is achieved by performing reasoning over this reduction to the extent possible, rather than directly over the original data.

Many approaches to indexing have been investigated in preceding decades. Reductions of data sets typically group together data elements based on their shared values or substructures in the data. In graphs, the notion of *bisimulation equivalence* of nodes has proven to be an effective means for indexing and compression (e.g., [1, 7, 14, 15, 17, 19, 21, 29]). Bisimulation, which is a fundamental notion arising in a surprising range of contexts [27], is based on the structural similarity of subgraphs. Intuitively, two nodes are bisimilar to each other if they cannot be distinguished from each other by the sequences of node labels that may appear on the paths that start from these nodes, as well as from each of the nodes on those paths. Grouping bisimilar nodes is known as *bisimulation partitioning*. Blocks of bisimilar nodes are then used as the basis for constructing indexing data structures supporting efficient search and querying over the data.

Efficient *internal*-memory solutions for computing bisimulation partitions have been investigated (e.g., [13, 15, 24]). To scale to real-world data sets such as those discussed

above, it becomes necessary to consider DAGs resident in *external* memory. In considering algorithms for such data, the primary concern is to minimize disk IO operations due to the high cost involved, relative to main-memory operations, in performing reads and writes to disk.

Due to the random access nature of internal-memory algorithms, the design of external-memory algorithms which minimize disk IO typically requires a significant departure from approaches taken for internal memory solutions [20]. In particular, state-of-the-art internal-memory bisimulation algorithms can not be directly adapted to IO-efficient external-memory algorithms due to their inherent random access behaviour. While a study has been made on storing and querying bisimulation partitions on disk [29], there has been to our knowledge no approach developed to date for efficiently computing bisimulation partitioning in external memory.

Motivated by these observations, in this paper we give the first IO-efficient external-memory bisimulation algorithm for DAGs. Our algorithm has a worst-case IO-complexity of $O(\text{SORT}(|N| + |E|))$, where $|N|$ and $|E|$ are the number of nodes and edges, resp., in the data graph and $\text{SORT}(n)$ is the number of accesses to external memory needed to sort an input of size n. Efficiency is achieved by intelligent organization of the graph on disk, and by sophisticated processing of the graph using global and local reorganization and careful staging and use of local bisimulation information. We establish the theoretical efficiency of the algorithm, and demonstrate its practicality via a thorough empirical evaluation on data sets having billions of nodes and edges.

Our algorithm is simple enough for practical implementation and use, and to serve as the basis for further study and design of external-memory bisimulation algorithms. For example, we also develop in this paper specializations of our algorithm for computing common variations of bisimulation for tree-structured graphs in the form of XML documents. Furthermore, the complete implementation is open-source and available for download.

We proceed in the paper as follows. In the next section, we present basic definitions concerning our data model, bisimulation equivalence, and the standard external-memory computational model. In Section 3, we then present and theoretically analyze our external-memory bisimulation algorithm. In Section 4, we show how to specialize our general algorithm for various bisimulation notions proposed for XML data. In Section 5, we then present a thorough empirical analysis of our approach, and conclude in Section 6 with a discussion of future directions for research.

2. PRELIMINARIES

2.1 Graphs and bisimilarity

In the context of this paper, a graph G is a triple $G = \langle N, E, l\rangle$, where N is a finite set of nodes, $E \subseteq N \times N$ is a directed edge relation, and l is a function with domain N that assigns a label $l(n)$ to every node $n \in N$. With a slight abuse of terminology, we call n a child of m, and m a parent of n, if and only if G contains an edge (m, n). Let children(m) be the set of all children of m, and let parents(n) be the set of all parents of n. Note that in our work we only consider acyclic graphs. Furthermore, we assume that the node set N is ordered in reverse topological order, that is, children always precede their parents in the order. Assuming a topological ordering is standard in the design of external memory DAG algorithms [20]. Indeed, real world data is often already ordered (e.g., XML documents), and, furthermore, practical approaches to topological sorting of massive data sets are available [3].

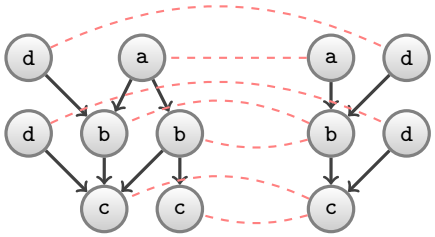


Figure 1: Two bisimilar directed acyclic graphs.

Definition 1. Let $G_1 = \langle N_1, E_1, l_1\rangle$ and $G_2 = \langle N_2, E_2, l_2\rangle$ be two, possibly the same, graphs. Nodes $n_1 \in N_1$ and $n_2 \in N_2$ are *bisimilar* to each other, denoted $n_1 \approx n_2$, if and only if:

1. the nodes have the same label: $l_1(n_1) = l_2(n_2)$;
2. for every node $n_1' \in$ children(n_1) there is a node $n_2' \in$ children(n_2) such that $n_1' \approx n_2'$; and,
3. for every node $n_2' \in$ children(n_2) there is a node $n_1' \in$ children(n_1) such that $n_1' \approx n_2'$.

We can extend this notion to complete graphs as follows:

Definition 2. Let $G_1 = \langle N_1, E_1, l_1\rangle$ and $G_2 = \langle N_2, E_2, l_2\rangle$ be graphs. Graph G_1 and G_2 are bisimilar to each other, denoted as $G_1 \approx G_2$, if and only if:

1. for every node $n_1 \in N_1$ there is a node $n_2 \in N_2$ such that $n_1 \approx n_2$; and,
2. for every node $n_2 \in N_2$ there is a node $n_1 \in N_1$ such that $n_1 \approx n_2$.

Figure 1 shows two graphs that are bisimilar to each other. The figure also shows with dotted lines how the nodes of one graph are bisimilar to nodes of the other graph. Note that in this figure all nodes with label `a` are bisimilar to each other, all nodes with label `b` are bisimilar to each other, and all nodes with label `c` are bisimilar to each other. Note, however, that this does not hold for nodes with label `d`.

For each graph G there is a unique (up to isomorphism) smallest graph (having the fewest nodes) that is bisimilar to G; we call this smallest graph the *maximum bisimulation graph* of G and denote it by $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle$. In Figure 1, the graph on the right is the maximum bisimulation graph of itself. It is also the maximum bisimulation graph of the graph on the left.

In the context of this paper, the *bisimilarity index* of a graph G is a data structure that stores the maximum bisimilarity graph $G_\downarrow$ of G, and stores, for each node $n_\downarrow \in N_\downarrow$, the set of nodes $\{n \in N : n_\downarrow \approx n\}$, i.e., the bisimulation equivalence class of $n_\downarrow$.

A *partition* $\mathcal{P}$ of a graph $G = \langle N, E, l\rangle$ is a subdivision of its nodes N into a set of *blocks* $\mathcal{P} = \{N_1, N_2, ...\}$ such that each block $N_i \in \mathcal{P}$ is a non-empty subset of N, the blocks are mutually disjoint, and their union is N. A *bisimulation*

partition of a graph G is a partition $\mathcal{P}$ of G such that the blocks of $\mathcal{P}$ are exactly the bisimilarity equivalence classes of G.

A partition $\mathcal{P}_1$ is a *refinement* of a partition $\mathcal{P}_2$ if and only if for every $P_1 \in \mathcal{P}_1$ there is exactly one $P_2 \in \mathcal{P}_2$ such that $P_1 \subseteq P_2$.

The rank $rank(n)$ of a node n is defined as the maximum number of edges on any path that starts at n. It is easily proved by induction that $m \approx n$ implies $rank(m) = rank(n)$, and thus the bisimulation partition is a refinement of the partition by rank.

2.2 Analysis of external-memory algorithms

In this paper, we investigate algorithms operating on data that does not fit in main memory. Therefore we need to use external memory, such as disks. In general external memory is slow. In particular, there is a high latency: it takes a lot of time to start reading or writing a random data item in external memory, but after that a large block that is consecutive in external memory can be read or written relatively fast. Thus the performance of algorithms using external memory is often dominated by the external-memory access patterns, where algorithms that read from and write to disk sparingly and in large blocks are at an advantage over algorithms that access the disk often for small amounts of data.

We shall use the following standard computer model to analyze the efficiency of our algorithms [2]. Our computer has a fast memory with a limited size of M units of data, and a slow, external memory of practically unlimited size. The computer has a fast processing unit that can operate on data in fast memory, but not on data in external memory. Therefore, during operation of any algorithm on this computer, data needs to be transferred between the two memories. This is done by moving data in blocks of size B; such a transfer is called an *IO*. The block size B is assumed to be large enough that the latency is dominated by the actual transfer times, and thus, the time spent on external-memory access is roughly proportional to the number of IOs.

The complexity of an external-memory algorithm can now be expressed as the (asymptotic) number of IOs performed by an algorithm, as a function of the input size and, possibly, other parameters. Clearly, reading or writing n units of data that are (to be) stored consecutively in external memory takes $\Theta(\textsc{Scan}(n)) = \Theta(\frac{n}{B})$ IOs. Sorting n units of data that are consecutive in external memory takes $\Theta(\textsc{Sort}(n)) = \Theta(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ IOs [2].

3. BISIMULATION PARTITIONING

State of the art internal memory bisimulation algorithms are based on a process of refinement introduced in the work of Paige and Tarjan (e.g., [13, 15, 24]). An initial partition of the nodes is picked (for example: a partition based on label equivalence). A step-by-step refinement of this initial partition is calculated by picking a single block S of nodes from the partition and stabilize all other blocks B with respect to this group (by splitting B into a block of nodes that have children in S and into a block of nodes that do not have children in S).

These refinement steps require unstructured random access to nodes and their children. In an external memory setting, these accesses translate to high IO costs. Therefore, it is not clear that state-of-the-art internal memory bisimulation algorithms can be effectively adapted to an external-memory setting. Hence, we have chosen to investigate an alternative approach, inspired by the recent use of node rank to accelerate internal memory refinement computations [13, 15].

3.1 Outline of our approach

Suppose each bisimilarity equivalence class is identified by a unique number, the *bisimilarity identifier*. Let now the *bisimilarity family* of a node be the set of bisimilarity identifiers of its children, and let the *bisimilarity decision value* of a node be the combination of its rank, its label, and its bisimilarity family. Then, by Definition 1, all nodes in the same bisimilarity equivalence class have the same bisimilarity decision value, and each bisimilarity equivalence class is uniquely identified by the bisimilarity decision value of its nodes.

The main idea behind our algorithmic approach is now to match bisimilarity identifiers to nodes and their bisimilarity decision values, by processing the nodes in order of increasing rank. Thus, when processing the nodes of any rank r, the bisimilarity identifiers of the children of these nodes are already known and can be used to determine the bisimilarity decision values of the nodes of rank r, which can then be sorted in order to assign a unique identifier to each different bisimilarity decision value.

To implement this approach, we use an algorithm in two phases. In the first phase, we compute the ranks of all nodes and we sort the nodes by rank and label; in the second phase, we obtain the bisimilarity family for each node and sort nodes of equal rank and label by their families. Below we will explain how these phases can be implemented to run in $O(\textsc{Sort}(|N| + |E|))$ IOs. After that, we will present an enhanced version of the algorithm where, in the first phase, nodes are sorted not only by rank and label, but also by a recursively defined hash value. This enhancement leads to a small increase in cost of the first phase, but may result in a substantial reduction in the size of the sets of nodes that need to be sorted in the second phase. Thus the enhanced algorithm still takes $O(\textsc{Sort}(|N|+|E|))$ IOs, but with better constant factors in practice for certain types of inputs.

3.2 Time-forward processing

Our bisimulation partitioning algorithm has two phases in which information about nodes must be computed from information about their children: in the first phase, we need to compute a node's rank (which is one plus the maximum rank of its children); in the second phase, we need to assign bisimilarity identifiers to nodes based on the bisimilarity identifiers of their children. This would be relatively easy if we could access the children of any node n when we process n, but in an external-memory setting, this could cause many IOs.

We can remove explicit access to the children of a node by introducing a supporting data structure that can be used to send information from children to parents. This technique is called *time-forward processing* [8, 20]. Time-forward processing can be used when nodes have unique ordered node identifiers such that children have smaller identifiers than their parents, and the nodes are stored in order of their identifiers, each node being stored with its own identifier and those of its parents. The supporting data structure should support two operations: (i) inserting a message addressed to a given node, identified by its node identifier, and

Algorithm 1 Phase 1: sort by rank and label

Input: file of nodes N as records $(id, label)$, sorted by id;
file of edges E as records $(parent, child)$, sorted by $child$;
$(id(n), id(m)) \in E$ implies $id(m) < id(n)$.
Output: file of nodes N' as records $(id, origId, rank, label)$,
sorted by id and, simultaneously, by $(rank, label)$;
file of edges E' as records $(parent, child)$, sorted by $child$;
$rank(n) > rank(m)$ implies $id(m) < id(n)$.

1: create empty file *Ranks* of records $(id, rank, label)$
2: create empty priority queue Q of records $(id, childsrank)$, ordered by id
3: **for all** $(n, label) \in N$, in order **do**
4: $rank \leftarrow 0$
5: **while** record at head of Q has $id = n$ **do**
6: extract $(n, childsrank)$ from Q
7: $rank \leftarrow \max(rank, childsrank + 1)$
8: append $(n, rank, label)$ to *Ranks*
9: **while** next edge from E has $child = n$ **do**
10: read $(parent, n)$ from E
11: insert $(parent, rank)$ in Q
12: sort *Ranks* lexicographically by $rank, label$
13: copy *Ranks* to N' while assigning new node identifiers
14: copy E to E' while updating node identifiers in E'
15: sort E' by $child$
16: **return** (N', E')

Algorithm 2 Details of line 13 and 14 of Algorithm 1

1: $newId \leftarrow 0$
2: create empty file R of records $(origId, newId)$
3: create empty file N' of records $(newId, origId, rank, label)$
4: **for all** $(origId, rank, label) \in Ranks$, in order **do**
5: $newId \leftarrow newId + 1$
6: append $(origId, newId)$ to R
7: append $(newId, origId, rank, label)$ to N'
8: sort R by $origId$
9: create empty file E' of records $(parent, child)$
10: move read pointer of E to beginning
11: **for all** $(origId, newId) \in R$, in order **do**
12: **while** next edge from E has $child = origId$ **do**
13: read $(parent, origId)$ from E
14: append $(parent, newId)$ to E'
15: sort E' by $parent$
16: move pointers of R and E' to beginning
17: **for all** $(origId, newId) \in R$, in order **do**
18: **while** next edge from E has $parent = origId$ **do**
19: read record $(origId, child)$ from E' and
20: overwrite with record $(newId, child)$

(ii) inspecting and removing all messages addressed to the smallest node identifier that is currently present in the data structure.

An algorithm that computes a value for each node depending on the values of its children can now be implemented as follows. We compute values for all nodes in order of their identifier, and whenever we compute a node's value, we insert messages with that value in the supporting data structure, addressing these messages to each of the node's parents. Thus, before we process each node n, we can obtain the values computed for its children by extracting all messages addressed to n from the data structure. Each node removes all messages addressed to it from the data structure, nodes with lower identifiers are processed before nodes with higher identifiers, and no messages are ever addressed to nodes that have already been processed; thus, when we want to extract the messages addressed to n, these messages will be the messages with the smallest node identifier currently in the data structure and they can be extracted by an operation of type (ii).

The supporting data structure can be implemented as a priority queue. There are external-memory priority queues that, amortized over their life-time, perform k operations of type (i) and (ii) in $\Theta(\textsc{Sort}(k))$ IOs [4].

3.3 The bisimulation partitioning algorithm

Assume the input to our problem consists of a list of nodes N, storing a unique node identifier and a label for every node, and a list of edges E, specified by the node identifiers of their tails (parents) and their heads (children). The list N is sorted by node identifier, and the list E is sorted by head (child). Recall that the node identifiers are assumed to be such that children always have smaller node identifiers than their parents.

Our basic bisimulation partitioning algorithm is now as follows. We use time-forward processing to compute the rank of each node, and make a copy of the list of nodes in which each node is annotated with its rank. Then we sort the nodes lexicographically, with their ranks as primary keys and their labels as secondary keys. We give each node a new identifier which is simply the position of the node in the resulting sorted list, and we replace the identifiers in E accordingly, producing lists N' and E'. We sort these new lists by the (new) node identifiers and by the (new) node identifiers of the heads, respectively. This completes the first phase of the algorithm. Pseudocode for this phase is given in Algorithm 1.

Some additional implementation details on the last lines of Algorithm 1 are in order. We can copy *Ranks* to N' while assigning new node identifiers, going through *Ranks* in order. During this process we construct a list R of (old node identifier, new node identifier)-pairs. To obtain a list E' with updated child node identifiers, we scan E and R in parallel from beginning to end, copying the entries of E to E' while replacing the child node identifiers by the new identifiers as read from R. To update the parent node identifiers in E' we sort E' on parent node identifier; we then scan E' and R in parallel from beginning to end while replacing the parent node identifiers in E' by the new identifiers as read from R. Pseudocode is given in Algorithm 2.

The rank-label combinations of the nodes define a partitioning of the graph. In the second phase of the algorithm, we use time-forward processing to go through the blocks of this partitioning one by one. Each rank-label combination c is processed as follows. Let N_c be the set of nodes that have rank-label combination c. For each node of N_c, we extract the bisimilarity identifiers of its children from the priority queue (assuming that they have been placed there) and sort them, while removing doubles. Thus we get the bisimilarity families for all nodes of N_c. Then we sort the

Algorithm 3 Phase 2: sort by bisimilarity equivalence class

Input: file of nodes N' as records $(id, origId, rank, label)$, sorted by id and, simultaneously, by $(rank, label)$;
file of edges E' as records $(parent, child)$, sorted by $child$;
$rank(n) > rank(m)$ implies $id(m) < id(n)$.
Output: file of nodes B as records $(origId, bisimId)$

```
1: create empty file B of records (origId, bisimId)
2: create priority queue Q of records (id, childsBisimId), ordered by id
3: lastBisimId ← 0
4: create empty file Group of records (bisimFamily, origId, parents)
5: for all (n, origId, r, l) ∈ N', in order do
6:    create an empty list bisimFamily
7:    while record at head of Q has id = n do
8:       extract (n, childsBisimId) from Q
9:       append childsBisimId to bisimFamily
10:   sort bisimFamily, removing doubles
11:   read all parents of n from E' and put them in a list parents
12:   append (bisimFamily, origId, parents) to Group
13:   if N' has no more records with rank = r, label = l then
14:      sort Group by bisimFamily, while marking the first occurrence of each family
15:      for all (bisimFamily, origId, parents) ∈ Group, in order do
16:         if bisimFamily is marked then
17:            lastBisimId ← lastBisimId + 1
18:         append (origId, lastBisimId) to B
19:         for all parentId ∈ parents do
20:            insert (parentId, lastBisimId) in Q
21:      erase Group
22: return B
```

nodes of N_c by bisimilarity family. Finally we go through the nodes of N_c in order, assigning a unique bisimilarity identifier $bisimId(f)$ to each maximal group of nodes N_f within N_c that have the same bisimilarity family f, and putting a message $bisimId(f)$ in the priority queue for all parents of the nodes of N_f. Pseudocode for the second phase of the algorithm is given in Algorithm 3.

THEOREM 1. *Given a labeled directed acyclic graph $G = \langle N, E, l\rangle$ with its nodes numbered in (reverse) topological order, we can compute the bisimilarity equivalence classes of G in $O(\text{SORT}(|N| + |E|))$ IOs.*

PROOF. We use Algorithm 1, followed by Algorithm 3. As observed in Section 2, bisimilar nodes must have the same rank and the same label. As a result, any nodes that are bisimilar to each other are processed in the same execution of lines 14–21 of Algorithm 3. Based on the induction hypothesis that nodes of rank $r-1$ get the same bisimilarity identifier if and only if they are bisimilar to each other, it is now easy to show that in lines 14–21, nodes of rank r get the same bisimilarity identifier if and only if they are bisimilar to each other.

As for the efficiency of the algorithm, the first phase scans and sorts files of at most $|N| + |E|$ records, for a total of $O(\text{SORT}(|N| + |E|))$ IOs. One record is inserted into and extracted from the priority queue for each child-parent relation; thus the total number of IOs required by the priority queue is $O(\text{SORT}(|E|))$.

The second phase is slightly more involved, as it sorts the lists *bisimFamily* and the files *Group*. For each node, the list *bisimFamily* contains one entry for each edge originating from that node; thus the total size of the lists *bisimFamily* is $O(|E|)$ and they are sorted in $O(\text{SORT}(|E|))$ IOs in total. For each node n, one record is added to the file *Group*: this records contains an identifier for n and each of its children and parents. Thus the amount of data inserted into *Group* over the course of the entire algorithm is $O(|N| + |E|)$. On line 14, the variable-size records in *Group* can be sorted and marked with the string sorting algorithm by Arge et al. [5] in $O(\text{SORT}(|N| + |E|))$ IOs. Thus, the complete algorithm takes $O(\text{SORT}(|N| + |E|))$ IOs. □

3.4 Enhanced algorithm

To reduce the amount of sorting needed in the second phase of the algorithm, we propose the following enhanced algorithm. In the first phase, we not only compute a rank for each node, but also a *hash* value, which is computed from the node's label and from the hash values of its children. Thus, the first phase of the algorithm is as in Algorithm 4.

The second phase of the algorithm is exactly as before, except that $rank, label$ is replaced by $rank, label, hash$; in particular, lines 14–21 are executed each time N' has no more records with the same rank, label, *and hash value* as the records seen so far.

By induction on increasing rank one can prove that bisimilar nodes get the same hash value, and therefore, any pair of nodes that are bisimilar to each other will still be processed in the same execution of lines 14–21 in Algorithm 3. Thus, the algorithm is still correct. Note that if the hash value from a label and any given set of k hash values can be computed in $O(\text{SORT}(k))$ IOs, the complete algorithm also still runs in $O(\text{SORT}(|N| + |E|))$ IOs in the worst case.

In many practical settings the priority queues in the first phase may be small and fit in main memory. For example,

Algorithm 4 Phase 1 (enhanced with hash values)

Input: file of nodes N as records $(id, label)$, sorted by id; file of edges E as records $(parent, child)$, sorted by $child$; $(id(n), id(m)) \in E$ implies $id(m) < id(n)$.

Output: nodes N' as records $(id, origId, rank, label, hash)$, sorted by id and, simultaneously, by $(rank, label, hash)$; file of edges E' as records $(parent, child)$, sorted by $child$; $rank(n) > rank(m)$ implies $id(m) < id(n)$.

```
1: create empty file Ranks of records (id, rank, label, hash)
2: create empty priority queue Q of records
   (id, childsRank, childsHash), ordered by id
3: for all (n, label) ∈ N, in order do
4:    rank ← 0
5:    initialize empty list childrensHashes
6:    while record at head of Q has id = n do
7:       extract (n, childsRank, childsHash) from Q
8:       rank ← max(rank, childsRank + 1)
9:       add childsHash to childrensHashes
10:   sort childrensHashes, removing doubles
11:   hash ← hash value from label and childrensHashes
12:   append (n, rank, label, hash) to Ranks
13:   while next edge from E has child = n do
14:      read (parent, n) from E
15:      insert (parent, rank) in Q
16: sort Ranks lexicographically by rank, label, hash
17: copy Ranks to N' while assigning new node identifiers
18: copy E to E' while updating node identifiers in E'
19: sort E' by child
20: return (N', E')
```

if the input graph is a tree in reverse depth-first order, then at any time during phase one, the queues will only contain messages to/from the nodes on a single path between the root and a leaf. As long as the children of a single node always fit in memory, the hash values can be computed in memory as well. Thus, the cost of computing the hash values in the first phase is small, and in practice, each hash value will be read or written at most eight times when writing, sorting, and reading *Ranks* and N'. In return, the grouping by rank, label, *and* hash value induces a much finer partitioning of G than the grouping by rank and label only. As a result, the sorting on line 14 of the second phase will be less likely to require the use of external memory. Note that each node's record in the *Group* file contains one number for the node's original identifier, plus one number for each neighbour of the node (a bisimilarity identifier for every child, and a node identifier for every parent). Therefore, even if on average, nodes have only two neighbours, a record in the group file has an average size of three numbers. Since sorting out-of-memory would take at least two read passes and two write passes, this would amount to the transfer of $3 \cdot 4 = 12$ numbers per node to or from disk. Thus, even in this setting with few edges, the optimization with hashing may already lead to IO-savings in the second phase of twelve numbers per node.

3.5 Implementation using STXXL

We have implemented the enhanced bisimulation partitioning algorithm of Section 3.4 using the building blocks available in the STXXL library, a mature open-source C++ library which provides basic external memory data structures and algorithms [10].[1] Since STXXL does not include algorithms to sort sets of variable-length records in external memory, we used the following adaptation of Algorithm 3.

Instead of storing, for each node n, a record of the form $(bisimFamily, origId, parents)$ in the file *Group*, we store the following fixed-size records: (i) one record of the type $(secondHash, origId)$, where *secondHash* is a secondary hash value computed from the bisimilarity family of n; (ii) for each child of n, a record of the type $(secondHash, origId, childsBisimId)$ (these records collectively store the bisimilarity family of n); (iii) for each parent of n, a record of the type $(secondHash, origId, parentId)$ (these records collectively store the parents of n). The secondary hash values computed from the bisimilarity families are such that bisimilarity families of different size always have different secondary hash values.

In line 14, we sort the above mentioned records lexicographically, thus obtaining a list of nodes with their bisimilarity families and parents, ordered by secondary hash value. Although unlikely, collisions on the secondary hash values may occur: nodes with equivalent secondary hash values (which appear consecutively in the sorted list) may have different bisimilarity families. Therefore we have to be a bit more careful when assigning bisimilarity identifiers in line 16–20: when processing nodes that have the same secondary hash value, we record their bisimilarity families with their bisimilarity identifiers in a dictionary; before assigning a new identifier to a particular bisimilarity family, we first check the dictionary to see if an identifier for this bisimilarity family had already been assigned. Considering that in practice (and, under the assumption of perfect hashing, also in theory) the dictionary is unlikely to ever be large, we used a simple sequential file implementation for this dictionary. Bisimilarity families are only stored in the dictionary as long as nodes with the same secondary hash value are being processed; the dictionary is always erased before proceeding to nodes with another rank, label, hash value, or secondary hash value.

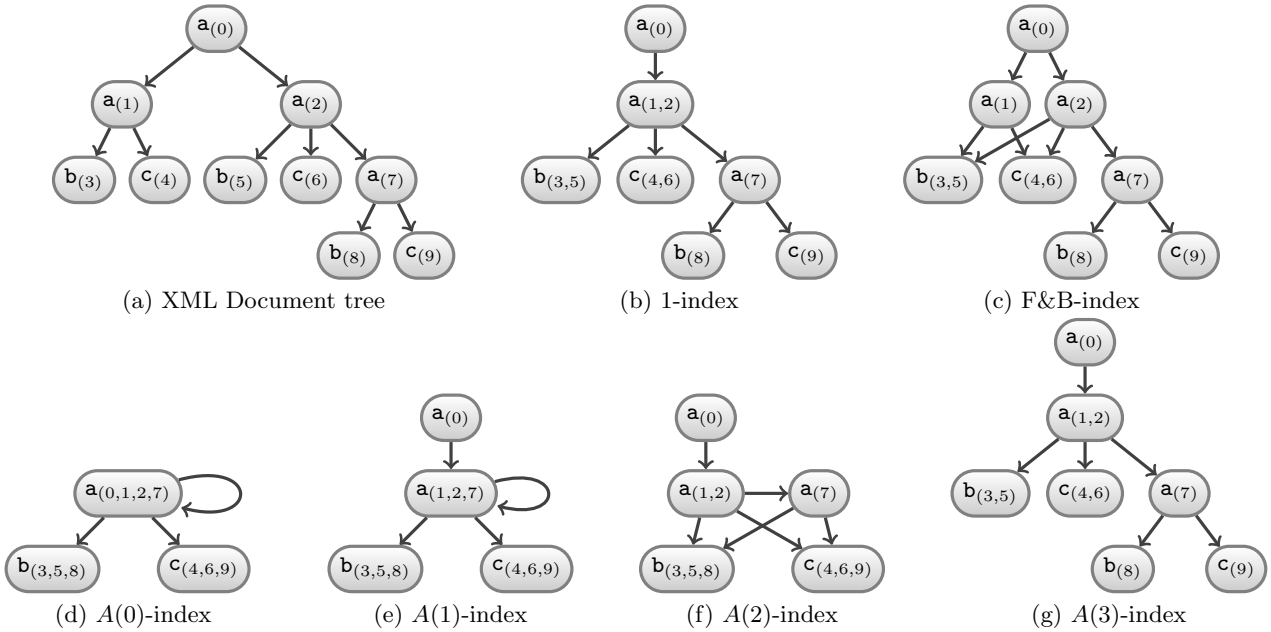
A full analysis of the IO complexity of this approach can be found in [16].

4. INDEXING XML DOCUMENTS

XML documents are widely used to exchange and store tree-structured data [1]. In this section we investigate specializations of the general algorithm from the previous section to efficiently calculate in external memory variations of bisimulation which have been proposed in the design of indexing data structures for XML and semi-structured databases. We shall discuss two well-known variations, namely the 1-index [21] and the $A(k)$-index [19]. We also briefly discuss how our approach can be specialized to efficiently compute the well known F&B-index [1].

In Figure 2 we have given an example of a simple XML document tree and indices built from this tree. In the index figures, nodes represent partition blocks, and there exists an edge from block A to block B if (and only if) there exists an edge in the original document from a node in A to a node in B.

[1]For more information we refer to the STXXL project page at `http://stxxl.sourceforge.net/`.



(a) XML Document tree (b) 1-index (c) F&B-index

(d) $A(0)$-index (e) $A(1)$-index (f) $A(2)$-index (g) $A(3)$-index

Figure 2: An XML document tree and five different index types; namely the 1-index, the F&B-index, and the $A(k)$-index (for $0 \leq k \leq 3$). We have annotated each node in the XML document tree with a unique identifier. This identifier is used in the indices to indicate the nodes represented by each index node.

4.1 The 1-index

The 1-index utilizes "backward" bisimulation for relating nodes with the same structure with respect to path-queries [21]. Backward bisimulation is equivalent to normal bisimulation on a graph wherein all edges are reversed in direction. Figure 2(b) illustrates the 1-index of our example XML document.

Backward bisimulation combined with the nested tree-structure of XML documents gives us several properties that we can utilize to optimize bisimulation partitioning. Recall that the basic algorithm from Section 3.3 consists of two phases: in the first phase nodes are sorted by rank and label; in the second phase nodes of the same rank and label are sorted by bisimilarity family. Alternatively we could take the following approach: in the first phase we sort by rank only; in the second phase we sort nodes of the same rank by label and bisimilarity family. Obviously, this would not affect the correctness and the asymptotic I/O-complexity of the algorithm. However, in the case of 1-indexes for XML-documents it brings the following advantage: we can avoid the use of a priority queue in the first phase, and we can avoid use of several sorting passes to assign identifiers to nodes. We achieve this as follows.

Instead of assigning identifiers to nodes by first sorting the nodes by rank and then using the positions in the sorted list as identifiers, we can use composite node identifiers of the form $(rank, idOnLevel)$, where *rank* is the backward rank of the node (that is, the node's depth in the tree), and *idOnLevel* is a unique identifier with respect to all nodes with backward rank *rank*. We can now use the structure of XML documents to compute these identifiers efficiently—in particular we will exploit the fact that an XML document essentially stores a so-called *Euler tour* [20] of the tree, in order.

Our algorithm will traverse the tree while maintaining a counter *depth* that holds the depth of the current position in the tree, and an array *count*, in which the i-th number (denoted $count[i]$) holds the number of nodes at depth i encountered so far. Initially, $depth = 0$ and the array *count* is empty; whenever we try to read an element of *count* that does not exist yet, this element will be created and initialized to zero.

Now, when we read a start-tag representing a node n during the processing of an XML document, this node is assigned backward rank $rank = depth$ and $idOnLevel = count[rank]$; we increment both $count[rank]$ and *depth* by one, and we construct an edge to n from its parent: this must be the last node encountered on the previous level, with composite identifier $(rank - 1, count[rank - 1] - 1)$. When we read an end-tag we simply decrement *depth* by one. After reading the complete tree, we simply sort the nodes by composite identifier, and the edges by the composite identifiers of the parents. Pseudocode for the complete first phase of the algorithm is given in Algorithm 5.

The second phase of the algorithm is now simple to implement. Note that we are computing *backward* bisimilarity equivalence classes, and therefore parents and children have switched roles. Thus, the bisimilarity family of a node is simply the bisimilarity identifier of the parent of a node, and no implementations of string sorting or secondary hash functions and dictionaries (as in Section 3.5) are needed. Pseudocode is given in Algorithm 6.

THEOREM 2. *Given an XML-document of N nodes, we can compute its 1-index in $O(\text{SORT}(|N|))$ IOs.*

PROOF. We use Algorithm 5, followed by Algorithm 6. The correctness of the algorithm follows from the same arguments as for Algorithm 1 and Algorithm 3 in Theorem 1.

As for the IO-complexity, observe that the accesses to the file *count* follow a very well-structured pattern: effectively

Algorithm 5 XML 1-index, phase 1: sort by depth

Input: XML document D.
Output: file N' of XML nodes as records
$(rank, idOnLevel, origId, label)$,
sorted by $(rank, idOnLevel)$;
file E' of edges as records
$(parentRank, parentIdOnLevel, childIdOnLevel)$,
sorted by $(parentRank, parentIdOnLevel)$;

```
1: create empty file N'
2: create empty file E'
3: create empty array of counters count
4: depth ← 0
5: for all tags tag of D, in order do
6:   if tag is a start tag then
7:     if count[depth] does not exist then
8:       add an entry count[depth] = 0 to count
9:     if depth ≠ 0 then
10:      append
           (depth − 1, count[depth − 1] − 1, count[depth])
         to E'
11:    determine node identifier origId and label label
12:    append (depth, count[depth], origId, label) to N'
13:    increment count[depth]
14:    increment depth
15:  else if tag is an end tag then
16:    decrement depth
17: sort N' by (rank, idOnLevel)
18: sort E' by (parentRank, parentIdOnLevel)
19: return (N', E')
```

we move ahead in the file by one step whenever we encounter a start tag, and we move back in the file by one step whenever we encounter an end tag. Thus, if we keep the two most recently accessed blocks of the file in memory, at least B tags must be read between successive IOs on the *count* file. Otherwise, the algorithm runs in $O(\text{SORT}(|N| + |E|)$ IOs by the same arguments as for Theorem 1; since $|E| = |N| - 1$, this simplifies to $O(\text{SORT}(|N|))$ IOs. □

4.1.1 The F&B-index

The 1-index summarizes the structure of graphs by only looking in one direction, from parent to child. The F&B-index groups nodes based on a summary of their structure with respect to both ancestors and descendants [1, 17]. Figure 2(c) illustrates the F&B-index of our example XML document.

For trees, Grimsmo et al. have shown that the F&B-index partitioning can be obtained by first computing forward bisimulation and then refining the obtained partition by computing backward bisimulation, i.e., by applying the algorithm from Section 3 twice (once with edges reversed) [15]. It is possible to significantly reduce the cost of this computation, by a straightforward adaptation of the algorithm from Section 4.1 for the backwards bisimulation step [16].

4.2 The $A(k)$-index

The $A(k)$-index utilizes backward node k-bisimulation, a localized variant of backward node bisimulation. The $A(k)$-index groups nodes n based on the structure of ancestor nodes at most k steps away from n.

Definition 3. Let $G = \langle N, E, l\rangle$ be a graph, $m, n \in N$, and $k \geq 0$. We say m and n are backward k-bisimilar, denoted $n \approx^k m$, if and only if $k = 0$ and $l(n) = l(m)$, or $k > 0$ and:

1. the nodes n and m are backward $(k-1)$-bisimilar, that is, $n \approx^{k-1} m$;
2. for each $n' \in \text{parents}(n)$, there is an $m' \in \text{parents}(m)$ with $n' \approx^{k-1} m'$; and,
3. for each $m' \in \text{parents}(m)$, there is an $n' \in \text{parents}(n)$ with $n' \approx^{k-1} m'$.

Figures 2(d)-(g) illustrate the $A(0)$-, $A(1)$-, $A(2)$-, and $A(3)$-index, resp., of our example XML document.

The $A(k)$-index seems similar to the 1-index. However, there is a critical difference between the two. Whereas all backward bisimilar nodes have the same rank, this does not necessarily hold for backward k-bisimilar nodes. We thus cannot use backward rank to localize the partitioning computations. We can, however, express backward node k-bisimilarity on trees in another way; namely, in terms of k-traces.

Definition 4. Let $G = \langle N, E, l\rangle$ be a tree, $r \in N$ be the root of G, $n \in N$, and $L(r, n) = \langle l(r), \ldots, l(n)\rangle$ be the sequence of labels of the nodes on the path from r to n in E. For $k \geq 0$, the k-trace of $L(r, n)$, denoted T_n^k, is the sequence containing the last k elements in $L(r, n)$. If $k > |L(r, n)|$, the length of $L(r, n)$, then the k-trace is constructed by prefixing $L(r, n)$ with $k - |L(r, n)|$ occurrences of some reserved label λ not in the range of l.

The k-traces, which are easily represented by fixed-size values, are used for identifying backward k-bisimilar equivalent nodes, as follows.

PROPOSITION 1. *Let $G = \langle N, E, l\rangle$ be a tree, $m, n \in N$, and $k \geq 0$. Then $n \approx^k m$ if and only if $T_n^{k+1} = T_m^{k+1}$.*

While processing an XML document, we can use a stack to store the labels of all parents of the current node n by pushing the label of a node onto the stack when we encounter a start-tag and popping the top of the stack when we encounter an end-tag. By taking the topmost $k + 1$ elements we get T_n^{k+1}, the $(k + 1)$-trace of n. This leads to a simple $A(k)$-index construction algorithm for XML documents, a sketch of which is presented in Algorithm 7.

THEOREM 3. *Given an XML-document of N nodes, we can compute its $A(k)$-index in $O(k\text{SORT}(|N|))$ IOs.*

5. EMPIRICAL ANALYSIS

To investigate the empirical behavior of our algorithms, we have performed four separate experiments. All experiments were performed on a standard laptop with an Intel Core i5-560M processor and 4GB of main memory. We have used the internal hard disk drive of this system for sorting and for storing temporary data structures such as priority queues. We have used the open source C++ library STXXL to perform all disk IO, (except for reading the initial input

Algorithm 6 XML 1-index, phase 2: sort by backward bisimilarity equivalence class

Input: file of XML nodes N' as records $(rank, idOnLevel, origId, label)$, sorted lexicographically by $(rank, idOnLevel)$;
file of edges E' as records $(parentRank, parentIdOnLevel, childIdOnLevel)$,
sorted lexicographically by $(parentRank, parentIdOnLevel)$;
Output: file of XML nodes B as records $(origId, bisimId)$

create empty file B of records $(origId, bisimId)$
create priority queue Q of records $(rank, idOnLevel, parentBisimId)$, ordered by $(rank, idOnLevel)$
insert $(-1, 0, 0)$ in Q (sentinel for root)
$lastBisimId \leftarrow 0$
create empty file $Group$ of records $(label, parentBisimId, origId, children)$

for all $(r, idOnLevel, origId, label) \in N'$, in order **do**
  extract $(r, idOnLevel, parentBisimId)$ from Q
  create an empty list $children$
  while next record of E' has $parentRank = r$ and $parentIdOnLevel = idOnLevel$ **do**
    read $(parentRank, parentIdOnLevel, childIdOnLevel)$ from E
    append $childIdOnLevel$ to $children$
  append $(label, parentBisimId, origId, children)$ to $Group$

  if N' has no more records with $rank = r$ **then**
    sort $Group$ lexicographically by $(label, parentBisimId)$
    for all $(label, parentBisimId, origId, children) \in Group$, in order **do**
      if $label$ and $parentBisimId$ are not the same as in previous record of $Group$ **then**
        $lastBisimId \leftarrow lastBisimId + 1$
      append $(origId, lastBisimId)$ to B
      for all $childIdOnLevel \in children$ **do**
        insert $(r + 1, childIdOnLevel, lastBisimId)$ in Q
    erase $Group$

return B

Algorithm 7 $A(k)$-index construction for XML documents

Input: XML document D.
Output: file of XML nodes B as records $(origId, trace)$

create empty file B of records $(origId, trace)$
create empty stack S
push k dummy labels λ onto S

for all tags tag of D, in order **do**
  if tag is a start tag **then**
    determine node identifier $origId$ and label $label$
    push $label$ onto S
    append ($origId$, top $k + 1$ elements of S) to B
  else if tag is an end tag **then**
    pop one label from S

Sort B by $trace$
return B

and writing the final output). The default configuration for STXXL, which we used, is to use direct IO, i.e., to completely bypass file system buffering. Further details can be found in [16].[2]

Experiment 1. In this experiment we measured the performance of the general bisimulation algorithm from Section 3.5 as a function of the number of nodes in the input graph. We also considered the difference between starting with a good initial partition (by Algorithm 4, based on rank, label, and hash value) and starting with a poor initial partition (by Algorithm 1, based on rank and label only), in order to measure the impact of a good initial partition on performance.

For this experiment, random graphs having between $100 \cdot 10^6$ and $1000 \cdot 10^6$ nodes were created using the generator described in Appendix A. Every graph had an average of three to four edges per node. The file size of the input graphs ranged between 2.1GB and 21.2GB. The number of bisimulation partition blocks in the output ranged from $70 \cdot 10^6$ for the smallest graph to $708 \cdot 10^6$ for the largest graph. For the largest input we have measured a total of 35919 reads from disk; 35091 writes to disk. Thereby a total 70.1GB was read and 68.5GB was written. These measurements only include temporary file usage (priority queues and sorting); not the reading of input and writing of output. In Figure 3 we have plotted the results for this experiment.

Experiment 2. In this experiment we measured the performance of the general bisimulation algorithm from Section 3.5 as a function of the number of edges in the input graph. To this end, we created graphs having $5 \cdot 10^4$ nodes and between 0 and $1249 \cdot 10^6$ edges, using the generator as described in Appendix A. In Figure 4 we have plotted the results for this experiment.

Experiment 3. In this experiment we measured the performance the general bisimulation algorithm of Section 3.5 on a single graph as a function of the amount of available memory (per data structure). For this experiment we fixed

[2]Open-source code of the full C++ implementation of the algorithms and supporting tooling used in our analysis can be found at `http://jhellings.nl/projects/exbisim/`.

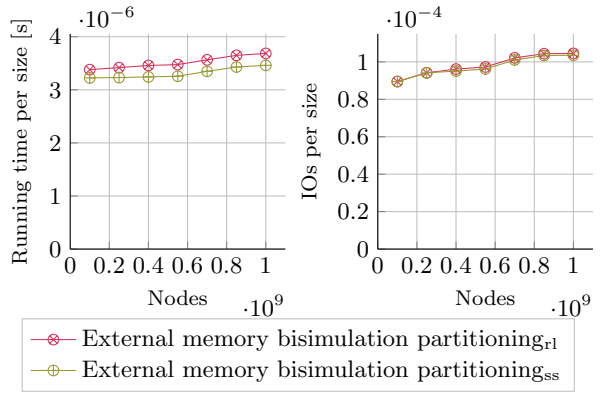


Figure 3: Performance of the bisimulation algorithm from Section 3 (Experiment 1). On the left, running time per node and edge is plotted against the number of nodes in the input. On the right, the number of IOs performed per node and edge is plotted against the number of nodes in the input. The subscript rl indicates an initial partition based on rank and label, the subscript ss indicates an initial partition based on rank, label, and hash value.

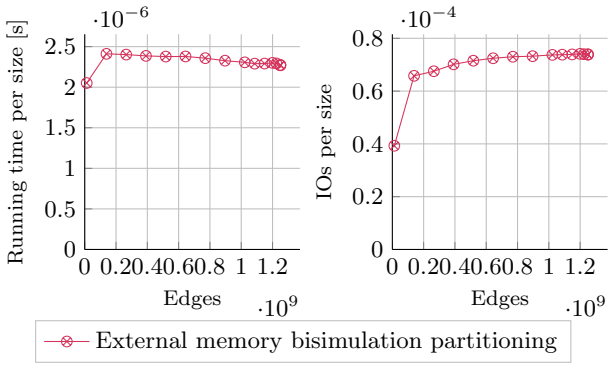


Figure 4: Performance of the bisimulation algorithm from Section 3 (Experiment 2). On the left, running time per node and edge is plotted against the number of edges in the input. On the right, the number of IOs performed per node and edge is plotted against the number of edges in the input.

a single graph with 10^8 nodes and $3.3 \cdot 10^8$ edges, generated as described in Appendix A. On this graph we performed external memory bisimulation partitioning, using versions of the algorithm from Section 3 constrained to a limited memory usage. We used values between 12 MB and 1.5 GB for the amount of memory the algorithm is allowed to use. In Figure 5 we have plotted the results for this experiment.

Experiment 4. In this experiment we compared the performance of, on one hand, the general DAG bisimulation algorithm from Section 3.5, and, on the other hand, the specialized algorithm from Section 4.1 for 1-index construction on XML documents. The performance of both algorithms is measured as a function of the size of the input graph. For this experiment we created XML documents using the `xmlgen` program provided by the XML Benchmark Project.[3]

[3]We have used version 0.92 of `xmlgen`, see `http://www.xml-benchmark.org/` for details.

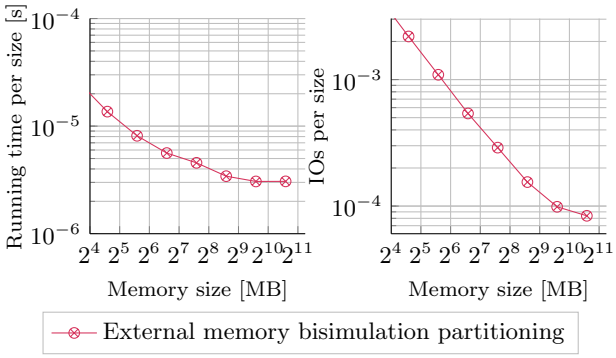


Figure 5: Impact of available internal memory on the performance of the bisimulation algorithm from Section 3 (Experiment 3). On the left, running time per node and edge is plotted against the amount of available memory. On the right, the number of IOs performed per node and edge is plotted against the amount of available memory. Note that the amount of available memory excludes stack space used by local variables and the memory used for buffers (256MB in total).

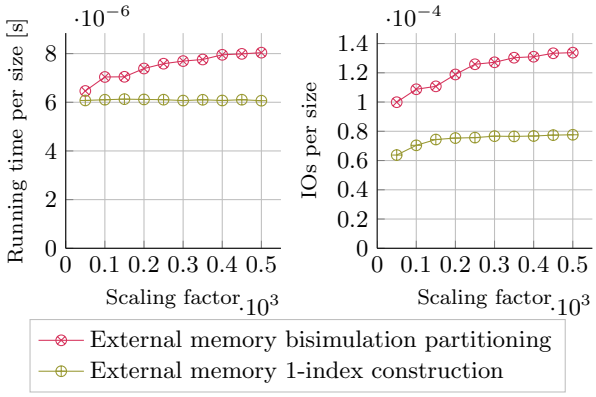


Figure 6: Comparing the performance of the DAG bisimulation algorithm of Section 3 and the 1-index algorithm of Section 4.1 (Experiment 4). On the left, running time per node is plotted as a function of the scaling factor. On the right, the number of IOs performed per node is plotted as a function of the scaling factor.

For the generation of XML documents we have used scaling factors between 50 and 500, resulting in documents with sizes between 5.6GB (10^8 nodes) and 55.8GB (10^9 nodes). In Figure 6 we have plotted the results for this experiment.

Analysis of results. The experiments all show that under all tested conditions, the general algorithm from Section 3 is and stays IO-efficient, even when available memory is artificially limited or when the number of nodes and edges is very high. We also see from Experiment 4 that specializations of our algorithm can outperform the general algorithm with a good margin. In particular, we were able to process an 55.8 GB XML document of 10^9 nodes, generated by software from the XMark XML benchmark project, in 104 minutes on a standard laptop with a standard hard disk.

Experiment 1 further shows that a good initial partition

(by rank, label and hash value) improves performance over the less-refined initial partitions (by rank and label only). A deeper look into the results of this experiment show that this performance improvement is due to a high increase in the number of partition blocks in the input for the second phase. This is as expected and does partly account for the improvement of performance. The results also show a reduction of the collisions on the secondary hash values that are computed from bisimilarity families (as explained in Section 3.5) — in fact, collisions were completely eliminated [16].

Experiment 3 shows that the algorithm does benefit from having more memory at its disposal. However, the impact of an increase in available memory becomes less significant for larger amounts of available memory.

From the results of the experiments and from the structure of the algorithm we do not expect that certain types of DAGs will have a much better performance than others. An in-depth look into the running time performance shows that it is mainly dominated by the first phase; and within this phase the majority of time is spend on sorting and renumbering the entire graph (last lines of Algorithm 1 and Algorithm 4). This sorting and renumbering is unaffected by any particular graph structures.

6. CONCLUDING REMARKS

In this paper we have developed the first IO-efficient bisimulation partitioning algorithm for DAGs. We also developed specializations of our general algorithm to compute well-known variants of bisimulation for disk-resident XML data. We have complemented our theoretical analysis of these algorithms with an empirical investigation which established their practicality on graphs having billions of nodes and edges.

The proposed algorithms are simple enough for practical implementation and use, for example in the design and implementation of scalable indexing data structures to facilitate efficient search and query answering in a wide variety of real-world applications of DAG-structured data, as discussed in Section 1.

Future work. The conceptual and practical results developed here pave the way for a variety of further investigations. We conclude the paper with a brief discussion of some of these.

Generalizing bisimulation partitioning. DAGs are adequate for representing XML data and other practical types of hierarchical data. However, for some applications, including querying RDF graphs and general graph databases, cycles in the data are common. Looking at the current state of general external-memory graph algorithms [20], it is not clear that solutions for IO-efficient bisimulation partitioning on cyclic graphs are likely to exist. One can however focus research on heuristic approaches to achieve acceptable performance in many cases, as is common for many external-memory algorithms (e.g., [3]). Extending our algorithms with such heuristics to handle cycles is an interesting direction for further study.

Partition maintenance. One can expect that a practical data set might be subject to modifications over time. Upon modification, it becomes necessary to update any bisimulation partition maintained on the data. Of course, this maintenance can be performed by throwing out the old partition and computing a new one from scratch. It is easy to show that, in the worst-case, partition maintenance can indeed be as expensive as calculating a new partition from scratch. In many practical cases, however, such a drastic approach is avoidable. For example, approximations of bisimulation which are cheaper to maintain may be acceptable. Internal-memory approaches to incremental partition maintenance in this spirit have been proposed, e.g., [11, 18, 26]. Studying such practical maintenance schemes for disk-resident data is another interesting direction for future research.

Practical output formatting. In the empirical validation of our algorithms, we have not considered any particular output format. An interesting research problem is to consider adapting our algorithms such that their output is usefully structured for some intended applications. For example, on XML documents one can explore the combination of our algorithms with the on-disk data structure studied by Wang et al. [29].

7. REFERENCES

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML.* Morgan Kaufmann, San Francisco, 2000.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] D. Ajwani, A. Cosgaya-Lozano, and N. Zeh. Engineering a topological sorting algorithm for massive graphs. Proc. Algorithm Engineering and Experimentation (ALENEX), 2011.

[4] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[5] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory (extended abstract). In *Proc. ACM Symp. on Theory of Computation (STOC)*, pages 540–548, 1997.

[6] M. Ben-Ari, T. Milo, and E. Verbin. Querying DAG-shaped execution traces through views. In *WebDB*, Providence, RI, USA, 2009.

[7] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, Berlin, 2003.

[8] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. Symp. on Discrete Algorithms (SODA)*, pages 139–149, 1995.

[9] V. Christophides, G. Karvounarakis, D. Plexousakis, M. Scholl, and S. Tourtounis. Optimizing taxonomic semantic web queries using labeling schemes. *J. Web Sem.*, 1(2):207–228, 2004.

[10] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.

[11] J. Deng, B. Choi, J. Xu, and S. S. Bhowmick. Optimizing incremental maintenance of minimal bisimulation of cyclic graphs. In *DASFAA*, pages 543–557, Hong Kong, 2011.

[12] D. Deutch and T. Milo. A quest for beauty and wealth (or, business processes for database researchers). In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 1–12, Athens, Greece, 2011.

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

[14] G. H. L. Fletcher, D. Van Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. *Inf. Syst.*, 34(7):657–670, 2009.

[15] N. Grimsmo, T. A. Bjørklund, and M. L. Hetland. Linear computation of the maximum simultaneous forward and backward bisimulation for node-labeled trees. In *XSym*, pages 18–32, Singapore, 2010.

[16] J. Hellings. Bisimulation partitioning and partition maintenance on very large directed acyclic graphs. Master's thesis, Eindhoven Univ. of Technology, 2011.

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD International Conference on Management of Data*, pages 133–144, Madison, Wisconsin, 2002.

[18] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *VLDB*, pages 239–250, Hong Kong, 2002.

[19] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, pages 129–140, 2002.

[20] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for memory hierarchies: advanced lectures*. Springer-Verlag, Berlin, Heidelberg, 2003.

[21] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 277–295, Jerusalem, 1999.

[22] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2-3):99–241, 2010.

[23] K.-K. Muniswamy-Reddy and M. Seltzer. Provenance as first class cloud data. *SIGOPS Oper. Syst. Rev.*, 43:11–16, 2010.

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

[25] G. Palla, I. J. Farkas, P. Pollner, I. Derényi, and T. Vicsek. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, 10(12):123026, 2008.

[26] D. Saha. An incremental bisimulation algorithm. In *Proc. Found. of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 204–215, New Delhi, India, 2007.

[27] D. Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31:15:1–15:41, 2009.

[28] B. Smith et al. The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. *Nature Biotechnology*, 25:1251–1255, 2007.

[29] W. Wang, H. Jiang, H. Wang, X. Lin, H. Lu, and J. Li. Efficient processing of XML path queries using the disk-based F&B index. In *VLDB*, pages 145–156, Trondheim, Norway, 2005.

[30] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3:171–200, 2005.

APPENDIX

A. GENERATING BENCHMARK DATA

Developed as part of our open-source experimental framework [16], the `gen` program is a benchmark graph generator. The program can be configured to create random DAGs, trees, chains and transitive-closure chains, with control of basic features such as node label assignment and graph size. We used `gen` to generate the input to Experiments 1–3, discussed in Section 5. The generator does not try to represent any particular class of graph structures, instead focusing on the worst-case scenario of random structure, to stress-test our algorithms.

The program uses a direct approach for generating the input for Experiment 1. First, `gen` creates n nodes. To each node v, `gen` assigns a label from a limited set of labels that depends on n. Then `gen` selects children to be connected to v by repeatedly flipping a coin that comes up heads with a certain probability p, that is given as a parameter to `gen`. Whenever the coin comes up heads, a new child for v is selected from the nodes that were generated before v; if the new child was already a child of v, it is ignored. As soon as the coin comes up tails, our program `gen` stops selecting children for v, and moves on to generating the next node.

The program uses a slightly different approach for generating the random graphs used in Experiment 2. Again, `gen` creates n nodes and assigns labels in the way described above. To create edges, `gen` considers every pair of nodes u, v such that u was generated before v, and creates an edge from u to v with probability p.

For Experiment 3 we use the same approach as for Experiment 1.