Bisimulation partitioning and partition maintenance

Jelle Hellings

July 2011

Bisimulation partitioning and partition maintenance

on very large directed acyclic graphs

MASTER'S THESIS

Jelle Hellings • exbisim@jhellings.nl

July 24, 2011

Department of Mathematics and Computer Science
Eindhoven University of Technology

Supervisor
dr. G. H. L. Fletcher

Committee members
prof. dr. P.M.E. de Bra
dr. G. H. L. Fletcher
dr. H. J. Haverkort

ABSTRACT

The combination of graphs and node bisimulation is widely used within and outside of computer science. One example of this combination is constructing indices for speeding up queries on XML documents. Thereby XML documents can be represented by trees and many index types for indexing XML documents utilize the notion of bisimulation. Thereby the notion of bisimulation is used to relate nodes that have equivalent behavior with respect to queries performed on the XML documents. By replacing these bisimilar nodes one can reduce the size of the XML document and as such speed up queries. The objective of this thesis is to develop techniques for constructing and maintaining bisimulation partitions. Thereby a bisimulation partition groups nodes based on bisimilarity. In this thesis we primarily focus on very large directed acyclic graphs. The results in this thesis can for example be used to index very large XML documents.

Our first goal is the development of external memory bisimulation partition algorithms. Bisimulation partitioning is already well studied for small graphs; this work is however hard to extend to an external memory environment. Therefore we develop a new algorithm; this algorithm has an expected IO complexity of $O(\text{Sort}(|N|) + \text{Sort}(|E|) + \text{PQ}(|E|))$. Thereby $|N|$ is the number of nodes in the input graph and $|E|$ is the number of nodes in the input graph. The notation $\text{Sort}(n)$ indicates the cost of external memory sorting a list with n fixed size elements; the notation $\text{PQ}(n)$ is used to indicate the cost of an external memory priority queue whereon at most n elements are stored. The behavior of this external memory bisimulation partitioning algorithm is also studied experimentally. This experiment shows that the algorithm is fast in practice; it can easily handle XML documents with a size of 55.8GB and directed acyclic graphs with at least a billion nodes and more than three billion edges.

The second goal is the investigation of partition maintenance in an external memory setting. Thereby we look at how a bisimulation partition can be kept up to date when the underlying graph is updated. Within a theoretical framework we proof that edge updates to a graph can change the entire bisimulation partition; subgraph updates however only affect the subgraph nodes in the bisimulation partition. We also provide an upper bound on the cost of partition maintenance by providing a naive algorithm for partition maintenance. For performing subgraph updates and edge updates we provide several sketches that can be of practical use; none of these sketches can however generally outperform the naive approach.

Lastly we focus on a practical application of bisimulation partitioning and partition maintenance; namely indexing XML documents for an XML database. Working with XML documents gives us the opportunity to utilize the simple structure of XML documents to optimize algorithms. In this setting we present an IO efficient 1-index construction algorithm with worst case IO complexity of $O(\text{Sort}(|N|) + \text{PQ}(|N|))$ and an IO efficient A(k)-index construction algorithm with worst case IO complexity of $O(\text{Scan}(k + |N|) + \text{Sort}(k|N|))$. We also provide a sketch for an F&B-index construction algorithm.

ACKNOWLEDGEMENTS

This thesis is the result of six months of research performed for my Master Thesis project. This project completes my master Computer Science & Engineering at the Eindhoven University of Technology. The project was performed internally at the Databases and Hypermedia group of the department of Mathematics and Computer Science; this under the daily supervision of dr. George Fletcher.

I would like to thank dr. George Fletcher for his daily guidance and support throughout the project. I also would like to thank him for introducing me to the topic during the Database Technology course and for fostering the development of my own research topic.

I also thank dr. Herman Haverkort for his expert knowledge on external memory algorithms and for his support throughout the project.

Jelle Hellings

CONTENTS

1. Introduction . . . 1
1.1 A small example: the 1-index . . . 2
1.2 Problem statement . . . 4
1.3 Overview . . . 4

2. Preliminaries . . . 5
2.1 Directed graphs . . . 5
2.2 Node and graph bisimulation . . . 7
2.3 Partitions and partition refinement . . . 9
2.4 Graph index . . . 12
2.5 External memory algorithms . . . 13
2.5.1 Memory model . . . 13
2.5.2 Complexity . . . 13

3. Bisimulation partitioning . . . 15
3.1 Online bisimulation partitioning . . . 15
3.1.1 Decision structures . . . 16
3.1.2 Online bisimulation . . . 19
3.2 Introducing time-forward processing . . . 21
3.2.1 The time-forward processing technique . . . 21
3.2.2 Time-forward processing online bisimulation partitioning algorithm . . . 22
3.3 On partition decision structures . . . 23
3.3.1 External memory search structures . . . 23
3.3.2 Query patterns . . . 24
3.3.3 Structural summary partition . . . 26
3.3.4 Using structural summaries for bisimulation partitioning . . . 28
3.4 External memory bisimulation partitioning . . . 30
3.5 Constructing maximum bisimulation graphs and graph indices . . . 34
3.6 Final notes . . . 35
3.6.1 Limitations on the external memory bisimulation partitioning algorithm . . . 35
3.6.2 Implementing external memory bisimulation . . . 36

4. Bisimulation partition maintenance . . . 38
4.1 Naive updating . . . 38
4.2 Maintenance complexity . . . 39
4.2.1 Update complexity for subgraph additions . . . 40
4.2.2 Update complexity for edge additions . . . 40
4.3 External memory algorithms for maintenance . . . 42
4.3.1 Adding subgraphs . . . 42
4.3.2 Removing subgraphs . . . 47
4.3.3 Edge updates . . . 48
4.4 Final notes . . . 50

5. Indexing XML documents 52
5.1 Preliminaries 52
5.1.1 The Extensible Markup Language 53
5.1.2 Variants on node bisimulation 54
5.2 External memory index construction for XML documents 57
5.2.1 Constructing the 1-index 58
5.2.2 Constructing the F&B-index 60
5.2.3 Constructing the A(k)-index 61
5.3 Partition maintenance for XML documents 63
5.3.1 Updating the 1-index 63
5.3.2 Updating the F&B-index 63
5.3.3 Updating the A(k)-index 63
5.4 Final notes 64

6. Experimental verification 65
6.1 Implementation overview 65
6.1.1 Low-level details 66
6.1.2 System specifications 67
6.2 Experiment description 68
6.3 Results 70
6.4 Conclusions 73

7. Conclusion 75
7.1 Overview 75
7.2 Future work 76
7.2.1 Practical implementations and verification 76
7.2.2 Practical partition maintenance 77
7.2.3 Internal memory bisimulation 78
7.2.4 Generalizing bisimulation partitioning 78
7.2.5 Generalizing index construction 79

Chapter 1

INTRODUCTION

Graphs provide an abstract model for describing elements and the relations between these elements. In practice graph-based models are used to describe many forms of data. An example of the usage of graphs in everyday life is the family tree. Also all kinds of networks are often portrayed as graphs; including public transportation networks, the Internet and social networks. Within computer science the usage of graphs is widespread. Some usages include representing data, systems and the behavior of systems. Obvious examples of the graph model being used to represent data are storing data in XML documents and storing data in RDF documents.

Figure 1.1: A common example of a graph used in everyday life: a map of the railroad network in the Netherlands. In this 'graph' the nodes represent stations; the edges between nodes represent railways connecting the stations. This image is taken from Wikimedia Commons.

The cost of computations performed on graphs typically depends on the size of the graph. Querying an XML document is a good example. To answer a query for some information on an XML document it is often necessary to traverse the entire document. This cost isn't a big issue when one query is performed. But when XML documents are used or stored in an XML database then one often queries these documents many times in a row.

We thus want to reduce the size of graphs as much as possible; such that later operations on the graph can be performed more efficiently. Here is where node bisimilarity comes into play. Node bisimilarity is an equivalence relation between nodes that is often used to reduce the size of graphs. Thereby bisimulation reduces the size of graphs by grouping all bisimilar equivalent nodes; after grouping each group of nodes is replaced by a single node. This approach works whenever the operations performed on the graph cannot distinguish between bisimilar nodes. This is the case for many query operations on graph databases; other operations include modal logic operators used in model checking.

The problem of grouping all bisimilar equivalent nodes is known as bisimulation partitioning. Due to its widespread application for optimizing performance of graph querying and model checking this problem is already studied in great detail in the past decades. Efficient internal memory solutions exist for general graphs, directed acyclic graphs, and also for the many variants of the node bisimilarity notion used for graph querying.

Internal memory solutions are however heavily restricted with respect to the size of the input graph. A possible solution for this size restriction is using external memory (for example hard disk drives). To make this solution feasible we however need algorithms optimized for external memory. The main goal of our investigation is to construct efficient external memory algorithms for bisimulation partitioning.

In our investigation we can focus on several classes of graphs. We have the very restricted class of trees and forests. The class of trees and forests is already heavily used to represent data; XML documents are one example of data described as trees. A less restricted class is the class of directed acyclic graphs; wherein general hierarchical relations can be described. An even less restricted class is the class of directed graphs; wherein any relation between data elements can be described. Due to the many usage scenarios wherein trees and directed acyclic graphs are used we have chosen to primarily focus on algorithms for performing bisimulation partitioning on these two classes.

We shall also look briefly at a second problem; namely partition maintenance. One can expect that graphs are subject of change over time. At these times one needs to recalculate the bisimulation partition for the changed graphs. This can be done by removing the old bisimulation partition of the graph and computing a new bisimulation partition from scratch. Such an approach might be costly when dealing with very large graphs. We thus also take a look at the problem on how a bisimulation partition can be kept up to date when the underlying graph changes.

1.1 A small example: the 1-index

The 1-index is a structural index for indexing XML documents and other graph-like data. This index can be used to speed up certain forms of path queries on these graphs; therefore the 1-index uses node bisimilarity[1] to reduce the size of the input document. Consider the XML document from figure 1.2a.

[1] More precise, the 1-index uses backward node bisimilarity. This is a variant of node bisimilarity, the basic change is that 1-index considers all edges in reverse. We shall have a closer look at the 1-index in Chapter 5.

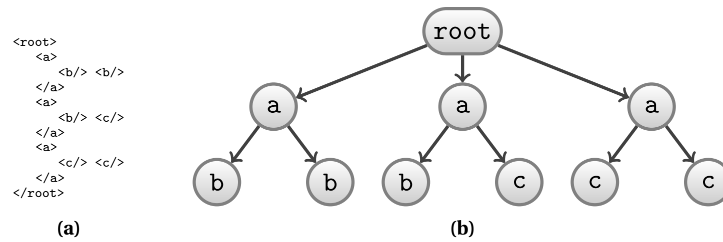


Figure 1.2: A very small example XML document. Figure 1.2a represents the XML document in plain text. This plain text document can also be represented by a tree as shown in Figure 1.2b.

We shall try to answer two types of queries on this graph; namely (1) *Can we reach some path* X *in the document; starting from the root?* and (2) *Return all elements found when following some path* X*; starting from the root.* The following two queries provide examples for these types of queries:

QUERY 1: is the path root/a/b reachable?
(answer: yes).

QUERY 2: give all nodes reachable by path root/a/b.
(answer: the three <b/> elements in the document).

The queries we have mentioned are easily answered by traversing all possible paths in the entire tree; starting from the root. Due to the size of the document one can directly see the answer to both queries; traversing the tree thus is perfectly fine. But for larger graphs we can imagine that traversing the entire graph is not such a good idea. Now consider Figure 1.3; this figure shows the tree that we obtain when grouping all nodes that are backward bisimilar equivalent[1].

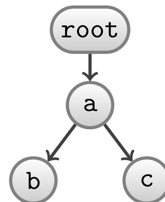


Figure 1.3: The indexed tree representation of the XML document from Figure 1.2. In this tree every group of backward bisimilar equivalent nodes has been replaced by a single node.

We can easily see how we can answer the first type of queries using the tree shown in Figure 1.3: simply traverse the graph. But answering the second type of queries is impossible on this graph as there is no relation to the nodes in the original graph. For answering the second type of queries one thus should maintain a mapping between groups of nodes from the source graph and the single node representing this group in the graph wherein every group of backward bisimilar nodes has been replaced by a single node. Such a mapping, together with the two graphs where between nodes are mapped, is called a structural index or 1-index. The structural index of the XML document from Figure 1.2 is shown in Figure 1.4.

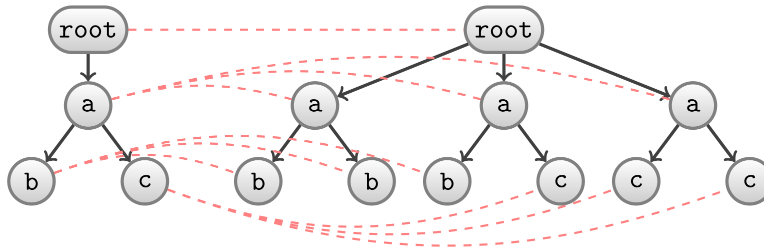


Figure 1.4: The structural index on the tree representation of the XML document from Figure 1.2.

1.2 Problem statement

The main goal of our work is to construct external memory algorithms and supporting data structures for performing bisimulation partitioning on directed acyclic graphs. The secondary goal of our work is to investigate partition maintenance in an external memory setting. To the best of our knowledge this work is the first contribution for the development of fully external memory bisimulation partitioning algorithms and partition maintenance algorithms.

1.3 Overview

This chapter serves as an introduction to the studied problems; namely bisimulation partitioning and partition maintenance for very large directed acyclic graphs. The next chapter, Chapter 2, shall introduce common theory and notation whereon the remainder of this work depends. With this common theory in mind we work on a solution for calculating the bisimulation partition of very large directed acyclic graphs; this solution is presented in Chapter 3. In Chapter 4 we take a look at partition maintenance.

After all this theoretical work we shift our attention to a practical application of the theory. We shall look at how bisimulation partitioning and partition maintenance can be used on XML documents in Chapter 5. Thereby we introduce algorithms for the construction of the 1-index, the F&B-index, and the A(k)-index of XML documents. We also look at what role partition maintenance has when updates are applied to XML documents. In Chapter 6 we present the results of a small scale implementation of the main algorithms introduced in this work, thereby we try to verify the efficiency claims made in previous chapters. In the last chapter, Chapter 7, we shall reflect on our findings and we present some topics for future investigations.

Chapter 2

PRELIMINARIES

In this chapter we introduce basic concepts and definitions whereon the contributions in this work depend. Section 2.1 introduces directed node-labeled graphs, Section 2.2 introduces node bisimulation and graph bisimulation, Section 2.3 introduces partitions and partition refinement, and Section 2.4 introduces graph indices. The last section, Section 2.5, presents an overview on external memory algorithms. Thereby some relevant concepts, data structures, and operations are presented for constructing efficient external memory (graph) algorithms.

2.1 Directed graphs

Central in this work is the notion of a directed acyclic node-labeled graph. Directed acyclic node-labeled graphs are a subset of directed node-labeled graphs. We shall use a very simple and general formalization for these directed node-labeled graphs. In a directed node-labeled graph every node has a label. This label represents the information represented by the elements in the graph. The edges in a directed node-labeled graph are directed; meaning that an edge from a node n to a node m does not imply that there is an edge from node m to node n. Edges represent relations between elements of information; the nodes. Thereby edges don't have any label or other information associated with them.

Definition 2.1. A graph is defined as a triple $G_{\mathcal{D}} = \langle N, E, l \rangle$; thereby N is a set of nodes, $E \subseteq N \times N$ is a directed edge relation and $l : N \to \mathcal{D}$ is a label function relating every node $n \in N$ with a label from some set $\mathcal{D}$. We shall refer to $\mathcal{D}$ as the label domain.

If we have nodes $n \in N, m \in N$ with $(n, m) \in E$ then n has an outgoing edge to node m and node m has an incoming edge from node n. We define $E(n) = \{m \in N : (n, m) \in E\}$ as the set of nodes that have an incoming edge from node n. We define $E'(n) = \{m \in N : (m, n) \in E\}$ as the set of nodes that have an outgoing edge to node n.

Node $m \in N$ is a child of node $n \in N$ if $m \in E(n)$, node $n\ inN$ is a parent of node $m \in N$ if $n \in E'(m)$. A node n is a root if it does not have parents; thus when $E'(n) = \emptyset$. A node n is a leaf is it does not have children; thus when $E(n) = \emptyset$.

In the remainder of this chapter we shall simply use graphs to refer to node-labeled graphs. We also directly simplify the notation and terminology used in Definition 2.1. We abstract from any details on the label domain $\mathcal{D}$. In the remainder we shall in general omit the label domain $\mathcal{D}$ altogether. We do however make some general assumptions on the label domain.

Assumption 2.2. We assume that there is a equivalence relation $=$ relating all equivalent labels from the label domain $\mathcal{D}$. We also assume that there is a total ordering on the labels from the label domain $\mathcal{D}$. When analyzing algorithms wherein labels are used we assume that every label can be stored in a fixed amount of storage.

Directed acyclic node-labeled graphs are a subset of directed node-labeled graphs. The subset of directed acyclic graphs can be easily defined in terms of paths between nodes; so we first introduce the notion of a path between nodes.

Definition 2.3. Let $G = \langle N, E, l \rangle$ be a graph, let $n_1 \in N, \ldots, n_i \in N; 1 \leq i$ be nodes. The sequence $n_1, \ldots, n_i$ is a path from node n_1 to node n_i if and only if for every pair of nodes $(n_j, n_{j+1}); 1 \leq j < i$ in the sequence

we have $(n_j, n_{j+1}) \in E$. If there is a path $n_1, \ldots, n_i$ then node n_1 has an (outgoing) path to node n_i and node n_i has a (incoming) path from node n_1.

With the notion of a path we can define the ancestors and descendants of a node. The ancestors of a node n are those nodes that have an outgoing path to node n. The descendants are those nodes that have an incoming path from node n.

Definition 2.4. Let $G = \langle N, E, l \rangle$ be a graph, let $n \in N$ be a node. The ancestors of node n are all nodes $m \in N$ such that there is a path from m to n. The descendants of node n are all nodes $m \in N$ such that there is a path from n to m. We define the ancestors function $A : N \rightarrow \wp(N)$ as $A(n) = \{m \in N : \text{there is a path from } m \text{ to } n\}$ and the descendants function $D : N \rightarrow \wp(N)$ as $D(n) = \{m \in N : \text{there is a path from } n \text{ to } m\}$.

Example 2.5. An example graph is shown in Figure 2.1a. In this graph the nodes `a`, `b` and `c` are roots. The nodes `g` and `h` are leafs. Node `e` has node `b` and `i` as parents and node `f` and `h` as children. In Figure 2.1b the ancestors of node `e` are highlighted; in Figure 2.1c the descendants of node `e` are highlighted. The path [`b`, `e`, `f`, `i`, `e`, `h`] is one of the many paths from node `b` to node `h`.

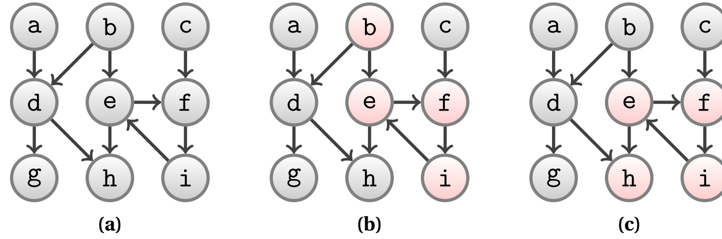


Figure 2.1: A directed node labeled graph; the text on each node represents the label of the node.

By using the definition of a path we can define the directed acyclic graphs as a subset of directed graphs.

Definition 2.6. A graph $G = \langle N, E, l \rangle$ is a directed acyclic graph if and only if there are no cycles in the graph. A node $n \in N$ is part of a cycle if and only if there is a path $n, \ldots, n$ from node n to node n. The graph G has cycles if there are nodes $n \in N$ that are part of a cycle.

In the following chapters in this work we shall generally use graph to refer to directed acyclic graphs; unless stated otherwise.

Example 2.7. The graph in Figure 2.1a has a path from node `e` to node `e`; and thus the graph has a cycle. As such this graph is not a directed acyclic graph. By removing an edge we can turn the graph into a directed acyclic graph; one of the candidates for removal is the edge (`e`, `f`).

By placing further restrictions on directed acyclic graphs we get trees. Trees form a frequently used subset of directed acyclic graphs. Among the applications of trees is information representation; XML documents are among the many data sources that can be fully represented by trees.

Definition 2.8. A graph $G = \langle N, E, l \rangle$ is a tree if and only if it is a directed acyclic graph wherein every node has at most one parent and exactly one node has no parents. This single node without any parents is called the root of the tree.

Collections of graphs can also be represented by a single graph. Thereby each individual graph in the collection is represented by a subgraph in the single graph.

Definition 2.9. Let $G = \langle N, E, l \rangle$ be a graph, let $G_s = \langle N_s, E_s, l_s \rangle$ be a graph. Graph G_s is a subgraph of graph G if and only if $N_s \subseteq N$ and:

(1) For every node $n \in N_s$ and every incoming edge $(m, n) \in E$ we have $m \in N_s$ and $(m, n) \in E_s$, and

(2) For every node $n \in N_s$ and every outgoing edge $(n, m) \in E$ we have $m \in N_s$ and $(n, m) \in E_s$.

In terms of subgraphs we can define a last class of graphs; namely forests.

Definition 2.10. A graph $G = \langle N, E, l \rangle$ is a forest if and only if every subgraph of graph G is a tree.

Note that every tree is a forest and every forest is a directed acyclic graph. Trees can be used to represent XML documents; forests are nothing more than a collection of trees. As such a forest can represent a collection of XML documents whereby each subgraph represents a single XML document.

Trees form a very restricted set of graphs; as such many problems can easier be solved on trees then on directed acyclic graphs. The same holds for directed acyclic graphs and directed graphs; whereby most problems become harder to solve efficiently (especially in external memory) when cycles are introduced. The restrictions on trees, forests and directed acyclic graphs also have their effects on the maximum number of edges as a function of the number of nodes.

Proposition 2.11. A directed graph $G = \langle N, E, l \rangle$ has at most $|N|^2$ edges. A directed acyclic graph $G = \langle N, E, l \rangle$ has at most $\frac{|N|(|N|-1)}{2}$ edges. A forest $G = \langle N, E, l \rangle$ has at most $|N| - 1$ edges. A tree $G = \langle N, E, l \rangle$ has exactly $|N| - 1$ edges.

For all directed acyclic graphs, forests, and trees we can topological order the nodes such that every parent node appears before all of its children. We can also reverse-topological order the nodes such that every parent node appears after all of its children.

Definition 2.12. Let $G = \langle N, E, l \rangle$ be a graph, let L be a list representation of all nodes N, let node n_i be the node at position i in list L. The list L is reverse-topological ordered if and only if every child node $n_j \in E(n_i)$ of node n_i has position $j < i$ in list L.

The reverse-topological ordering is particular useful whenever some computation of a property p for a node depends on the property p of the children of the node. We have a guarantee that the property p is already computed for all children of a node n before we start computing the property p for node n itself when we compute the property on all nodes in reverse-topological order. Thereby a reverse-topological order can be of help for achieving good performance.

Example 2.13. An example graph is shown in Figure 2.2. In this graph the nodes $N_s = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ are part of the same subgraph and the list $L = [\texttt{c}, \texttt{b}, \texttt{a}, \texttt{f}, \texttt{e}, \texttt{d}]$ provides a reverse-topological order on the nodes of the graph.

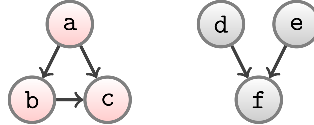


Figure 2.2: A directed acyclic node labeled graph; the text on each node represents the label of the node. The graph consists of two subgraphs; nodes belonging to the same subgraph have the same color.

2.2 Node and graph bisimulation

The second central notion in this work is node bisimilarity. Node bisimilarity is frequently used as an equivalence relation relating nodes that 'behave the same' from the perspective of some operation[1]. We first recall the definition of an equivalence relation.

Definition 2.14. Let R be a relation relating elements from some set $\mathcal{U}$. Relation R is an equivalence relation if and only if:

(1) R is reflexive; thus for all $e \in \mathcal{U}$ we have eRe,

(2) R is symmetric; thus for all $e_1 \in \mathcal{U}, e_2 \in \mathcal{U}$ with $e_1 R e_2$ we have $e_2 R e_1$, and

[1] In this work we primarily investigate bisimulation partitioning related to indexing graph databases. The notion of bisimulation is however used in many other fields; including fields outside of computer science. For a general in-depth overview on bisimulation, its history, and its usages we refer to [San09].

(3) R is transitive; thus for all $e_1 \in \mathcal{U}, e_2 \in \mathcal{U}, e_3 \in \mathcal{U}$ with $e_1 R e_2$ and $e_2 R e_3$ we have $e_1 R e_3$.

As said node bisimilarity is used to relate nodes that behave the same from the perspective of some operation. Thereby behaving the same means that the result of applying the operation on a node n will always give the same result as applying the operation on any other node that is bisimilar equivalent to node n. Examples of operations that cannot distinguish between bisimilar equivalent nodes are operations used in modal logic. Also several types of path queries cannot distinguish between nodes that are (backward) bisimilar equivalent

Definition 2.15. Let $G_1 = \langle N_1, E_1, l_1 \rangle, G_2 = \langle N_2, E_2, l_2 \rangle$ be graphs. Node $n \in N_1$ bisimulates node $m \in N_2$; denoted as $n \approx m$; if and only if:

(1) The nodes have the same label; $l_1(n) = l_2(m)$,

(2) For every node $n' \in E_1(n)$ there is a node $m' \in E_2(m)$ with $n' \approx m'$, and

(3) For every node $m' \in E_2(m)$ there is a node $n' \in E_1(n)$ with $n' \approx m'$.

Proposition 2.16. Node bisimulation is an equivalence relation; thus node bisimulation is reflexive, symmetric and transitive.

Note that nothing in Definition 2.15 prohibits nodes from the same graph to be bisimilar equivalent. To the contrary, in the largest part of this document we shall only focus on node bisimilarity between nodes in a single graph; most theory is thus also presented for single graphs.

Based on node bisimilarity we can also introduce an equivalence relation between graphs; relating graphs that behave the same from the perspective of some operation.

Definition 2.17. Let $G_1 = \langle N_1, E_1, l_1 \rangle, G_2 = \langle N_2, E_2, l_2 \rangle$ be graphs. Graph G_1 bisimulates graph G_2; denoted as $G_1 \approx_G G_2$; if and only if:

(1) For every node $n \in N_1$ there is a node $m \in N_2$ such that $n \approx m$, and

(2) For every node $m \in N_2$ there is a node $n \in N_1$ such that $n \approx m$.

Proposition 2.18. Graph bisimulation is an equivalence relation; thus graph bisimulation is reflexive, symmetric and transitive.

Example 2.19. Two graphs are shown in Figure 2.3a and Figure 2.3b; these graphs are bisimulation equivalent. The proof has been provided in Figure 2.3c wherein we show how nodes from the first graph are bisimulated by nodes in the second graph.

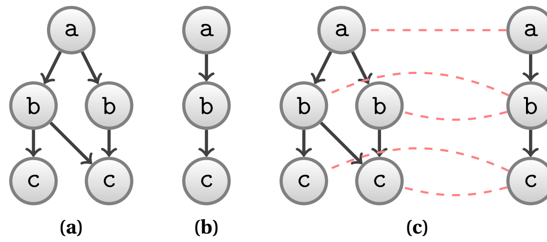


Figure 2.3: Figure 2.3a and Figure 2.3b both show a directed acyclic graph; in these graphs the text on each node represents the label of the node. In Figure 2.3c we show a relation between bisimilar equivalent nodes from the graph shown in Figure 2.3a with nodes from the graph shown in Figure 2.3b.

For the graph shown in Figure 2.3a we have that all nodes with label a bisimulate each other. We also have that all nodes with label b bisimulate each other.

With node and graph bisimilarity we can relate nodes and graphs that behave the same from the perspective of some operation. If this operation is expensive then it would be wise to execute it on a graph that is as small as possible; while staying bisimilar equivalent with the input. The smallest possible graph that bisimulates a graph G is called the maximum bisimulation graph of graph G.

Definition 2.20. Let $G = \langle N, E, l\rangle$ be a graph. Graph $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle$ is a maximum bisimulation graph of graph G if and only if:

(1) Graph G and $G_\downarrow$ are bisimilar equivalent; $G \approx_G G_\downarrow$, and

(2) For every other graph $G' = \langle N', E', l'\rangle$ with $G' \approx_G G$ it holds that $|N_\downarrow| \le |N'|$.

Let $n \in N$ be a node that bisimulates a maximum bisimulation graph node $n_\downarrow$. We define $E_\downarrow(n) = E_\downarrow(n_\downarrow), n_\downarrow \in N_\downarrow, n_\downarrow \approx n$ as the set of nodes that have an incoming edge from node $n_\downarrow$ bisimulated by node n. We define $E'_\downarrow(n) = E'_\downarrow(n_\downarrow), n_\downarrow \in N_\downarrow, n_\downarrow \approx n$ as the set of nodes that have an outgoing edge to node $n_\downarrow$ bisimulated by node n.

Example 2.21. The graph in Figure 2.3b is a maximum bisimulation graph of itself. The graph in Figure 2.3b is also a maximum bisimulation graph of the graph in Figure 2.3a.

Proposition 2.22. Let $G = \langle N, E, l\rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle$ be the maximum bisimulation graph for graph G. The maximum bisimulation graph $G_\downarrow$ is unique; any other maximum bisimulation graph for graph G' is isomorphic to $G_\downarrow$.

2.3 Partitions and partition refinement

The construction of a maximum bisimulation graph is not the only way to reduce the amount of expensive operations. We can also use the node bisimilarity equivalence relation to divide all nodes into equivalence classes.

Definition 2.23. Let R be an equivalence relation relating elements from some set $\mathcal{U}$, let $e \in \mathcal{U}$ be an element. The set $[e] = \{u \in \mathcal{U} : u R e\}$ is the equivalence class of element e. The relation R is an equivalence relation; thus for every element $e' \in [e]$ we have that the equivalence class $[e']$ is equivalent to $[e]$.

When we have placed all nodes in corresponding equivalence classes based on node bisimilarity; then we only have to execute the expensive operation on a single node per equivalence class. The result of this operation can then be shared by all nodes in the equivalence class. The set of all node bisimulation equivalence classes is usually called a bisimulation partition; which is a special form of a partition.

Definition 2.24. Let $G = \langle N, E, l\rangle$ be a graph. A partition block is a non-empty set of nodes; thereby a partition block of N is a subset of N. A partition of N is a set of partition blocks whereby for every node $n \in N$ there is exactly one partition block $p \in P$ such that $n \in p$.

Example 2.25. Let $G = \langle N, E, l\rangle$ be a graph. The set $\{N\}$ is a partition of N with a single partition block equal to the set N.

Based on Definition 2.24 we can define the bisimulation partition of a set of nodes as the set of all node bisimilarity equivalence classes with respect to the set of nodes.

Definition 2.26. Let $G = \langle N, E, l\rangle$ be a graph, let P be a partition of N. The partition P is a bisimulation partition if and only if every partition block $p \in P$ is equivalent to the node bisimulation equivalence class for all nodes $n \in p$. Stated otherwise; the following two conditions should hold for every node $n \in N$ placed in partition block $p \in P$:

(1) Every node $m \in N$ bisimulated by n is also placed in partition block p, and

(2) No node $m \in N$ not bisimulated by n is placed in partition block p.

Blocks in a bisimulation partition are called bisimulation partition blocks.

Any algorithm that computes the bisimulation partition of a set of nodes is called a bisimulation partition algorithm. For performance reasons we see that many bisimulation partition algorithms first calculate some easy computable partition and then refine this partition into the bisimulation partition. These algorithms do so by splitting the partition blocks from the easy computable partition until the resulting partition blocks are bisimulation partition blocks. This however only works if the bisimulation partition is a refinement of the easy computable partition.

Definition 2.27. Let $G = \langle N, E, l \rangle$ be a graph, let P_1, P_2 be partitions of N. Partition P_1 is a refinement of partition P_2 if and only if for every $p \in P_1$ there is exactly one $p' \in P_2$ with $p \subseteq p'$.

We can use functions to map nodes to a value; an example is the label function l relating every node with a label. These resulting node values can be used to create partitions wherein nodes are grouped on equivalent values; whereby each such group of nodes is placed in a separate partition block.

Definition 2.28. Let $G = \langle N, E, l \rangle$ be a graph, let $\mathcal{F} : N \rightarrow \mathcal{U}$ be a function mapping nodes to some value. The partition $P_{\mathcal{F}}$ is a node-value partition for function $\mathcal{F}$ and nodes N whenever it meets the following two conditions for every node $n \in N$ placed in partition block $p \in P$:

(1) Every node $m \in N$ with $\mathcal{F}(n) = \mathcal{F}(m)$ is also placed in partition block p, and

(2) No node $m \in N$ with $\mathcal{F}(n) \neq \mathcal{F}(m)$ is placed in partition block p.

Based on Definition 2.28 we can easily define the label partition as the node value partition using the label function.

Definition 2.29. Let $G = \langle N, E, l \rangle$ be a graph. The label partition P_l of N is defined as the node-value partition for function l.

Example 2.30. An example graph is shown in Figure 2.4. The label partition of this graph contains three partition blocks; namely partition block $\{\mathtt{a}, \mathtt{a}, \mathtt{a}\}$, partition block $\{\mathtt{b}, \mathtt{b}\}$ and partition block $\{\mathtt{c}\}$.

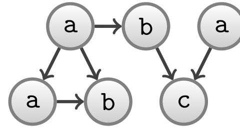


Figure 2.4: A directed acyclic node labeled graph; the text on each node represents the label of the node.

We shall provide a generic theorem that is useful to proof if the bisimulation partition is a refinement of a value-based partition for some function $\mathcal{F}$.

Theorem 2.31. Let $G = \langle N, E, l \rangle$ be a graph, let $\mathcal{F} : N \rightarrow \mathcal{U}$ be a function mapping nodes to some value, let $P_{\mathcal{F}}$ be a node-value partition for function $\mathcal{F}$ and nodes N. If we have $n \approx m$ implies $\mathcal{F}(n) = \mathcal{F}(m)$ then the bisimulation partition P is a refinement of $P_{\mathcal{F}}$.

Proof. Assume P is not a refinement of $P_{\mathcal{F}}$; there thus must be a partition block $p \in P$ such that there is no $p' \in P_{\mathcal{F}}$ with $p \subseteq p'$. Let n be a node in partition block p, let $p' \in P_{\mathcal{F}}$ be the partition block wherein all nodes with the value $\mathcal{F}(n)$ are placed. We have $n \in p'$. From $n \approx m$ implies $\mathcal{F}(n) = \mathcal{F}(m)$ we can conclude that every node in p has the same value. We thus have $p \subseteq p'$, thereby leading to a contradiction. □

Corollary 2.32. Let $G = \langle N, E, l \rangle$ be a graph. The bisimulation partition P of N is a refinement of the label partition P_l of N.

The rank of a node is another node-value that is used in fast bisimulation partitioning algorithms for directed acyclic graphs. The rank of a node is the length of the longest path starting at the node and ending in a leaf node.

Definition 2.33. Let $G = \langle N, E, l \rangle$ be a graph. The rank of a node $n \in N$ is defined as the length of the longest path starting at node n to any leaf node $m \in N$. The function *rank* maps nodes to their rank; this function is defined as:

$$rank(n) = \begin{cases} 0 & n \text{ is a leaf} \\ 1 + \max_{m \in E(n)} rank(m) & \text{otherwise} \end{cases}$$

Definition 2.34. Let $G = \langle N, E, l \rangle$ be a graph. The rank partition P_{rank} of N is defined as the node-value partition for function *rank*.

Example 2.35. An example graph is shown in Figure 2.5. The rank partition of this graph contains three partition blocks; namely partition block $\{b^0, c^0\}$, partition block $\{a^1, a^1, b^1\}$, and partition block $\{a^2\}$.

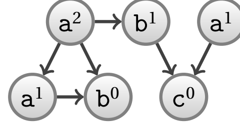


Figure 2.5: A directed acyclic node labeled graph; the text on each node represents the label of the node. The superscript on each node represents the rank of the node.

We shall first show that bisimilar nodes have the same rank; such that we can use Theorem 2.31 to proof that the bisimulation partition is a refinement of the rank partition.

Theorem 2.36. Let $G = \langle N, E, l \rangle$ be a graph, let $n \in N, m \in N$ be nodes. We have $n \approx m$ implies $rank(n) = rank(m)$.

Proof. The proof is by induction on the rank of nodes.

BASE CASE: Let n be a node with rank 0, let node m be a node with $n \approx m$. According to Definition 2.33 node n does not have children. If node m would have children; then the third requirement in Definition 2.15 is violated. As such node m cannot have children and thus also has rank 0.

INDUCTION HYPOTHESIS: Let n be a node with rank up to r. For every node m with $n \approx m$ it holds that $rank(m) = rank(n)$.

INDUCTION STEP: Let node n be a node with rank $r+1$, let node m be a node with $n \approx m$. We shall show (1) that $rank(m) \geq r+1$ and (2) that $rank(m) \leq r+1$.

(1) Assume $rank(m) < r+1$. According to Definition 2.33 there must be a child node n' of node n with $rank(n') = r$. According to Definition 2.15 there must be a child node m' of node m with $n' \approx m'$. According to the induction hypotheses this node m' must have rank r; using Definition 2.33 we can conclude that $rank(m) < r+1$ cannot hold. Thus by contradiction we have proven $rank(m) \geq r+1$.

(2) Assume $rank(m) > r+1$. According to Definition 2.33 there must be a child node m' of node m with $rank(m') \geq r+1$. According to Definition 2.15 there must thus be a child node n' of node n with $n' \approx m'$. From Definition 2.33 we can derive that $rank(n') \leq r$. Using the induction hypothesis we can conclude $r+1 \leq rank(m') = rank(n') \leq r$. This leads to a contradiction; thereby proving $rank(m) \leq r+1$.

Combining (1) and (2) we can only conclude $rank(m) = r+1$.

□

Corollary 2.37. Let $G = \langle N, E, l \rangle$ be a graph. The bisimulation partition P of N is a refinement of the rank partition P_{rank} of N.

Example 2.38. Let $G = \langle N, E, l \rangle$ be the graph shown in Figure 2.6.

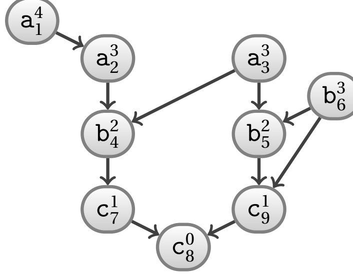


Figure 2.6: A directed acyclic node labeled graph; the text on each node represents the label of the node. The subscript on each node represents a unique identifier and the superscript on each node represents the rank of the node.

The set $P_N = \{N\}$ is a partition of N containing a single partition block N. The set $P_l = \{\{\mathtt{a}_1^4, \mathtt{a}_2^3, \mathtt{a}_3^3\}, \{\mathtt{b}_4^2, \mathtt{b}_5^2, \mathtt{b}_6^3\}, \{\mathtt{c}_7^1, \mathtt{c}_8^0, \mathtt{c}_9^1\}\}$ is a partition of N where nodes are grouped on equivalent label. The set $P_{rank} = \{\{\mathtt{a}_1^4\}, \{\mathtt{a}_2^3, \mathtt{a}_3^3, \mathtt{b}_6^3\}, \{\mathtt{b}_4^2, \mathtt{b}_5^2\}, \{\mathtt{c}_7^1, \mathtt{c}_9^1\}, \{\mathtt{c}_8^0\}\}$ is a partition of N where nodes are grouped on equivalent rank. The set $P = \{\{\mathtt{a}_1^4\}, \{\mathtt{a}_2^3, \mathtt{a}_3^3\}, \{\mathtt{b}_4^2, \mathtt{b}_5^2\}, \{\mathtt{b}_6^3\}, \{\mathtt{c}_7^1, \mathtt{c}_9^1\}, \{\mathtt{c}_8^0\}\}$ is the bisimulation partition of N.

The partition P_N is refined by all other partitions. The bisimulation partition P is a refinement of partition P_{rank} and also of partition P_l.

The bisimulation partition and the maximum bisimulation graph are defined in terms of the same notion; namely node bisimilarity. As such it is not a surprise that there is a clear relation between the bisimulation partition and the maximum bisimulation graph of the same graph.

Proposition 2.39. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G, let P be the bisimulation partition of graph G. There is a bijection relating every partition block $p \in P$ with a maximum bisimulation graph node $n_\downarrow \in N_\downarrow$.

2.4 Graph index

We have described that maximum bisimulation graphs and bisimulation partitions can be used to optimize performance. For practical purposes; these two separate notions are not always sufficient. The maximum bisimulation graph misses any relation with the nodes in the original graph and the bisimulation partition misses information on how partition blocks are related by the edges between nodes. To overcome these restrictions we introduce the graph index as a combination of the maximum bisimulation graph and the bisimulation partition.

Definition 2.40. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G. A directed node-labeled graph index for graph G is defined as a quadruple $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$. Thereby $p : N_\downarrow \to \wp(N)$ is a bisimulation partition function relating maximum bisimulation graph nodes $n_\downarrow$ with the bisimulation partition block containing the nodes in the original graph bisimulated by $n_\downarrow$; we thus define $p(n_\downarrow)$ by $p(n_\downarrow) = \{n \in N : n_\downarrow \approx n\}$.

The maximum bisimulation graph is directly represented in the graph index. Without too much work we can also derive the bisimulation partition from a graph index.

Proposition 2.41. Let $G = \langle N, E, l \rangle$ be a graph, let $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$ be the graph index of graph G. The set $P = \{p(n_\downarrow) : n_\downarrow \in N_\downarrow\}$ is the bisimulation partition of nodes N.

The other way is also possible; given a bisimulation partition we can easily construct a maximum bisimulation graph; and thus also the index. In Remark 2.42 we shall informally describe a procedure to achieve this.

Remark 2.42. Let $G = \langle N, E, l \rangle$ be a graph, let P be the bisimulation partition. The maximum bisimulation graph $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ can be constructed in terms of graph G and bisimulation partition P.

Let $p \in P$ be any partition block with node $n \in p$ being a node in this partition block. Introduce a single maximum bisimulation graph node $n_{\downarrow}$ for ever partition block p; this node $n_{\downarrow}$ has label $l(n)$.

After creating all maximum bisimulation graph nodes one can create the maximum bisimulation graph edges. Let $p \in P$ be any partition block represented by maximum bisimulation graph node $n_{\downarrow}$, let $p' \in P$ be any partition block represented by maximum bisimulation graph node $n'_{\downarrow}$. Introduce a single maximum bisimulation graph edge $(n_{\downarrow}, n'_{\downarrow})$ if and only if there are nodes $n \in p, n' \in p'$ with $(n, n') \in E$.

2.5 External memory algorithms

We investigate algorithm operating on data that does not fit in main memory. Therefore we need to use secondary memory. In general secondary memory is slow, the performance of algorithms using secondary memory are thus often dominated by the access patterns for secondary memory. Traditional computational and complexity models do not take these access patterns into account; as these models are mainly used to analyze the number of operations executed (by some processing unit).

Traditional computational and complexity models are as such not sufficient for analyzing the performance of algorithm utilizing secondary memory. In this section we introduce a computational and complexity model that does take the memory hierarchy into account; this model is much better equipped to analyze performance of algorithms that utilize secondary memory.

We only give a small overview; our overview is in no way complete. For a more in-depth look into algorithms in an hierarchical memory model we refer to [MSS03]. For graph algorithms in an hierarchical memory model we refer to [Zeh02].

2.5.1 Memory model

We utilize a two level memory model. Therein the first memory level consists of fast memory with a limited size. The second memory level is slow but has a practically unlimited size. In general the first memory level represents internal memory and the second memory level represents storage space available on hard disk drives or other forms of slow external storage. As such algorithms utilizing this secondary memory level during their operations are called external memory algorithms.

Assumption 2.43. The internal memory can store a total of M units of data.

During operation of an external memory algorithm data needs to be transferred between internal and external memory. For hard disk drives the duration of each transfer is determined by the latency and the transfer time. Thereby the latency is the time it takes to move the read and write head of the hard disk drive into the position where data needs to be transferred to or transferred from. The transfer time is the time it takes to transfer all data once the read and write head is into position.

For hard disk drives the duration of transfers of small amounts of data is dominated by the latencies. Hard disk drives utilize several mechanics to keep transfer times for adjacent blocks of data low. As such efficient external memory algorithms transfer data in larger blocks instead of transferring small amounts of data. Thereby all transferred data should be useful, just wrapping every tiny transfer into a transfer of a larger block of useless data will not make an algorithm efficient.

Definition 2.44. Let B be (a close to) optimal unit of data to transfer during a single transfer between internal and external memory. A disk block is a chunk of data of size B; B is called the block size.

The duration of a data transfer of B units of data should be dominated by the transfer times. In general the optimal value for B depends on specific details of the hard disk drives, hardware and software caches, and details of the file system and operating system.

Definition 2.45. Any transfer of data between internal and external memory is called an IO operation.

2.5.2 Complexity

Performance of algorithms utilizing external memory is restricted by the speed of external memory. During analysis of external memory algorithms we thus not only need to take the number of operations executed by a processing unit into account, but also the number of IO operations performed.

Definition 2.46. The IO complexity of an algorithm is the (asymptotic) number of IOs performed by an algorithm.

Many IO efficient algorithms utilize scanning and sorting of data as basic operations.

Proposition 2.47. Scanning represents reading data from external memory in the order it is stored or writing data to external memory in the order it will be stored. The IO complexity of scanning N units of data is $\Theta(\text{SCAN}(N)) = \Theta(\frac{N}{B})$ IOs.

Proposition 2.48. Data stored in external memory can be sorted with respect to some total ordering. The IO complexity of sorting N units of data is $\Theta(\text{SORT}(N)) = \Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ IOs.

Besides the algorithmic complexity for scanning and sorting the IO cost for IO efficient algorithms is determined by the data structures used. A crucial data structure used by many IO efficient external memory directed acyclic graph algorithms is the priority queue.

Proposition 2.49. A priority queue is a data structure serving as a container for data elements. Data elements can be added to the container in any order using the ADD operation. The data element with the highest priority in the queue can be retrieved efficiently without performing any IOs; this by using the TOP operation. The queue also allows one to remove the data element with the highest priority by using the POP operation. The total IO cost associated with adding and removing N elements to the priority queue is $\Theta(\text{PQ}(N)) = \Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{B}))$ IOs.

Chapter 3

BISIMULATION PARTITIONING

Bisimulation partitioning a graph is a well studied problem wherefore several good internal memory solutions exist. A well known solution is the partitioning algorithm by Robert Paige and Robert E. Tarjan [PT87]. This algorithm has a worst case runtime complexity of $O(|E|\log(|N|))$ and a memory usage of $O(|N|+|E|)$. For directed acyclic graphs several refinements of the algorithm by Paige and Tarjan exists [DPP01, GBH10]. These refinements improve the worst case runtime complexity for directed acyclic graphs to $O(|N|+|E|)$.

The existence of fast internal memory algorithms does not directly imply the existence of fast and IO efficient external memory algorithms. Adapting the algorithms based on the work by Paige and Tarjan seems problematic as these algorithms require direct access to nodes and their children. There have however been attempts to implement the algorithm by Paige and Tarjan in an external memory environment. The work by [HDFJ10] is an example; they implement the algorithm on top of a relational database and show that this solution works for not-too-large directed graphs.

Due to the problematic nature of adapting internal memory algorithms we have chosen to investigate an alternative approach. Our approach tries to minimize access to parts of the graph that are not expected to be in main memory. We do so by reading the graph sequentially and place each sequentially read node in the right partition block with only the information we have available locally. Thereby we have restricted ourselves to directed acyclic graphs with a reverse-topological ordering on their nodes. These restrictions give us access to several useful external memory graph algorithm techniques.

A sketch of this approach is presented in Section 3.1. The missing details of the sketch are introduced in Section 3.2 and Section 3.3. The resulting IO efficient bisimulation partitioning algorithm is presented in Section 3.4. Details on how the IO efficient bisimulation partition algorithm can be utilized to construct maximum bisimulation graphs and graph indices are presented in Section 3.5. In Section 3.6 we conclude our findings by discussing some practical considerations when using the bisimulation partitioning algorithm.

3.1 Online bisimulation partitioning

The main principle of an online algorithm is that it makes decisions based only on the information it has already seen. We shall try to adhere to an even stronger principle; namely that decisions made by the algorithm are based only on the information the algorithm is currently investigating. This without looking to information it has already investigated. Such an algorithm adhering to the strong online principle is likely to need to maintain some data structures for supporting making the right decisions. Online algorithms are excellent candidates for external memory algorithms; as a one-way sequential read over the input can be performed IO efficient. Thereby the algorithm must however only rely on supporting data structures that have a bounded size or can be implemented IO efficient.

Applying the strong principle of online algorithms on bisimulation partitioning results in an algorithm that can decide for each node n to which partition block this node n belongs; this by only inspecting the node n and some supporting data structures. The following algorithm provides a sketch of an online bisimulation partitioning algorithm operating on reverse-topological ordered graphs.

Algorithm 3.1 Online bisimulation partitioning algorithm (outline)

Require: Directed acyclic graph $G = \langle N, E, l \rangle$.

Ensure: The output is a pair (n, p) for every $n \in N$; with p an identifier for the bisimulation partition block whereto n belongs.

1: P is a decision structure
2: **for all** $n \in N$, in reverse-topological order **do**
3: **print** $(\text{QUERY}(P, \text{KEY}(n)), n)$

In the sketched algorithm we use a supporting data structure that decides, for every node n, to which bisimulation partition block it belongs. This data structure is queried with a key derived from node n; this query returns a bisimulation partition block. When this data structure works correctly; then the given algorithm will trivially calculate a valid bisimulation partitioning.

3.1.1 Decision structures

The outline suggests a decision structure based on a mapping between a search key (derived from the node) and bisimulation partition blocks. Our first objective is thus to find a suitable key. Therefore we shall introduce the node-bisimilarity value.

Definition 3.1. Let $G = \langle N, E, l \rangle$ be a directed acyclic graph, let $n \in N$ be a node. We define the node-bisimilarity value $v_\approx(n)$ of node n as $v_\approx(n) = (l(n), \{v_\approx(n') : n' \in E(n)\})$.

The node-bisimilarity value of a node is defined inductively in terms of the node-bisimilarity values of children. Thereby it closely resembles bisimilarity as defined in Definition 2.15. The node-bisimilarity value is only useful as a search key if we can derive the same node-bisimilarity value from two nodes if and only if these nodes are bisimilar equivalent.

Theorem 3.2. Let $G = \langle N, E, l \rangle$ be a directed acyclic graph, let $n \in N, m \in N$ be nodes. We have $v_\approx(n) = v_\approx(m)$ if and only if $n \approx m$.

Proof. The proof is by induction on the rank of the nodes.

BASE CASE: Let node n be a node with $rank(n) = 0$. We have $v_\approx(n) = (l(n), \emptyset)$. We shall proof (1) that $v_\approx(n) = v_\approx(m)$ holds for nodes m with $n \approx m$ and (2) that $v_\approx(n) \neq v_\approx(m)$ holds for nodes m with $n \not\approx m$.

(1) Let m be a node with $m \approx n$. We have $v_\approx(m) = (l(m), \emptyset)$; according to Definition 2.15 we have $l(n) = l(m)$ and thus $v_\approx(n) = v_\approx(m)$.

(2) Let m be a node with $m \not\approx n$ and $v_\approx(n) = v_\approx(m)$. We thus have $l(m) = l(n)$ and node m does not have children. According to Definition 2.15 this implies $n \approx m$, contradicting our assumptions. By contradiction we have $v_\approx(n) \neq v_\approx(m)$ for $m \not\approx n$.

INDUCTION HYPOTHESIS: Let n be a node with rank up to r. We have $v_\approx(n) = v_\approx(m)$ if and only if $n \approx m$ for every node m.

INDUCTION STEP: Let node n be a node with $rank(n) = r + 1$. We have $v_\approx(n) = (l(n), S_n)$. We shall proof (1) that $v_\approx(n) = v_\approx(m)$ holds for nodes m with $n \approx m$ and (2) that $v_\approx(n) \neq v_\approx(m)$ holds for nodes m with $n \not\approx m$.

(1) Let m be a node with $m \approx n$. We have $v_\approx(m) = (l(m), S_m)$ and according to Definition 2.15 we have $l(n) = l(m)$. Definition 2.15 guarantees that for every node $n' \in E(n)$ there is a node $m' \in E(m)$ with $n' \approx m'$. These nodes n' have $rank(n') \leq r$; thus according to the induction hypothesis we have $v_\approx(n') = v_\approx(m')$. Therefrom we can conclude $S_n \subseteq S_m$. In the same way we can prove $S_n \supseteq S_m$; thereby proving $S_n = S_m$ and thus $v_\approx(n) = v_\approx(m)$.

(2) Let m be a node with $m \not\approx n$ and $v_\approx(n) = v_\approx(m)$. We thus have $l(m) = l(n)$, proving that Definition 2.15, condition (1) holds. We also have $S_n = \{v_\approx(m') : m' \in E(m)\}$. For each node $n' \in E(n)$ there thus is a node $m' \in E(m)$ with $v_\approx(n') = v_\approx(m')$. We have $rank(n') \leq r$; thus according to the induction hypothesis $n' \approx m'$ holds for each n', proving that Definition 2.15, condition (2) holds. In the same way we can prove that Definition 2.15, condition (3) holds; thereby proving $n \approx m$. By contradiction we have $v_\approx(n) \neq v_\approx(m)$ for $m \not\approx n$.

□

In Theorem 3.2 we see that the node-bisimilarity value as defined in Definition 3.1 can fill in the role of a search key. On top of this search key we shall define a first decision structure.

Definition 3.3. Let $pds_{v_\approx}$ be a partition decision structure using node-bisimilarity values as search keys. We represent $pds_{v_\approx}$ as a list of node-bisimilarity values. This list is empty when newly constructed. The position of a node-bisimilarity value in $pds_{v_\approx}$ serves as a unique bisimulation partition block identifier.

The KEY operation is defined as $\text{KEY}(n) = v_\approx(n)$. The QUERY operation searches the list and if the node-bisimilarity value is found; then the position in the list is returned. If the node-bisimilarity value is not found; then a new entry is added to the end of the list and the position of this entry is returned.

The partition decision structure $pds_{v_\approx}$ provided the functionality required by the outline in Algorithm 3.1. This partition decision structure thus can be used in combination with the outline in Algorithm 3.1 to produce a working bisimulation partitioning algorithm. This will however result in a non-optimal algorithm due to the usage of an unordered list as a search structure.

There are more efficient data structures for storing and searching (key, value)-pairs. We shall however stick to the unordered list for the moment; as it is a very simple structure to analyze the underlying properties of any partition decision structure. The first property we take a look at is the cumulative storage needed to store all the keys (the node-bisimilarity values). For this storage analysis we first introduce a size measurement for node-bisimilarity values.

Definition 3.4. Let $G = \langle N, E, l \rangle$ be a graph, let $n \in N$ be a node with $v_\approx(n) = (l(n), S)$. The size of $v_\approx(n)$ is given by $|v_\approx(n)| = 1 + \sum_{v \in S} |s|$.

Proposition 3.5. Let $G = \langle N, E, l \rangle$ be a graph, let $n \in N$ be a node. If node labels can be stored in a fixed amount of storage; then the storage needed for the node-bisimilarity value $v_\approx(n)$ of node n is $\Theta(|v_\approx(n)|)$.

Using the size defined for node-bisimilarity values we can prove worst case lower bounds on the size of node-bisimilarity values.

Theorem 3.6. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph for graph G, let $n \in N$ be a node. The worst case lower bound on $|v_\approx(n)|$ is given by $2^{|N_\downarrow|-1}$.

Proof. Let $G = \langle N, E, l \rangle$ be the transitive closure graph defined as $N = \{n_1, \ldots, n_{|N|}\}$, $E = \{(n_j, n_i) : 1 \leq i < j \leq |N|\}$. In Figure 3.1 we show a transitive closure graph containing 4 nodes.

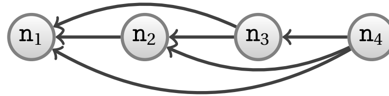


Figure 3.1: A transitive closure graph with 4 nodes.

Note that the transitive closure graph is already a maximum bisimulation graph; every node thus has a unique node-bisimilarity value. By induction we shall proof that $|v_\approx(n_i)| = 2^{i-1}$.

BASE CASE: Node n_1 has node value $v_\approx(n_1) = (l(n_1), \emptyset)$ and thus $|v_\approx(n_1)| = 1 = 2^0$.

INDUCTION HYPOTHESIS: For nodes $n_1, \ldots, n_j$ we have $|v_\approx(n_j)| = 2^{j-1}$.

INDUCTION STEP: Node n_{j+1} has edges to every node $n_1, \ldots, n_j$. The node-bisimilarity value for node n_{j+1} is thus given by $v_{\approx}(n_{j+1}) = (l(n_j), \{v_{\approx}(n_1), \ldots, v_{\approx}(n_j)\})$. The size of this node-bisimilarity value is given by $|v_{\approx}(n_{j+1})| = 1 + \sum_{1 \le i \le j} |v_{\approx}(n_i)|$; using the induction hypothesis we get $|v_{\approx}(n_{j+1})| = 1 + \sum_{1 \le i \le j} 2^{i-1} = 2^j$

□

The worst case lower bound on the size of node-bisimilarity values shows that using node-bisimilarity values for search keys in any decision structure will not result in a fast structure. We can however use the partition decision structure $pds_{v_{\approx}}$ to reduce the size of node-bisimilarity values.

In the partition decision structure $pds_{v_{\approx}}$ each node-bisimilarity value is mapped one-to-one to a unique integer; namely the position of the node-bisimilarity value in the decision structure. If we have a node-bisimilarity value $(l(n), S)$ for node n then we can replace each node-bisimilarity value $v \in S$ by the position of v in the partition decision structure. Based on this idea we introduce the node-decision value.

Definition 3.7. Let $G_{\mathcal{D}} = \langle N, E, l \rangle$ be a graph, let $n \in N$ be a node. The node-decision value $v_{pds}(n)$ is a tuple $v_{pds}(n) \in (\mathcal{D} \times \wp(\mathbb{N}))$.

Let *pds* be a partition decision structure using node-decision values as search keys. We represent *pds* as a list of node-decision values. This list is empty when newly constructed. The position of a node-decision value in list *pds* serves as a unique bisimulation partition block identifier.

The QUERY operation searches the list and if the node-decision value is found; then the position in the list is returned. If the node-decision value is not found; then a new entry is added to the end of the list and the position of this entry is returned.

We define the node-decision value $v_{pds}(n)$ of node n with respect to a partition decision structure *pds* as $v_{pds}(n) = (l(n), \{\text{QUERY}(pds, v_{pds}(m)) : m \in E(n)\})$.

We shall refer to partition decision structures using node-decision values as search keys as partition decision structures. The definition of node-decision values directly imposes a limitation on the usage of partition decision structures: the node-decision value of a node n can only be determined when the node-decision values of the children of node n are already evaluated. This limitation is not problematic for our applications as we assume that nodes are processed in a reverse-topological order. Before we analyze the properties of the partition decision structure we shall proof that node-decision values can be used as valid search keys for nodes.

Theorem 3.8. Let $G = \langle N, E, l \rangle$ be a directed acyclic graph, let $n \in N, m \in N$ be nodes, let *pds* be a partition decision structure. When we query nodes in reverse-topological order and only determine the node-decision value of a node when needed for querying; then we have $v_{pds}(n) = v_{pds}(m)$ if and only if $n \approx m$ and $n \approx m$ if and only if $\text{QUERY}(pds, v_{pds}(n)) = \text{QUERY}(pds, v_{pds}(m))$.

Proof (sketch). The proof is by induction on the rank of the nodes; and is similar to the proof for Theorem 3.2. Thereby utilize that the first time the node-decision value for node n is evaluated, is when node n is queried. Also utilize that the children of node n are queried before node n; and thus the node-decision values of the children of n are present in the partition decision structure. Both properties follow from the reverse-topological ordering wherein nodes are queried. □

We have introduced the node-decision value as an alternative to the node-bisimilarity value whereby we have tried to reduce the storage needed for partition decision structures. We shall now analyze the exact storage needs for partition decision structures; therefore we first introduce a size measurement for node-decision values.

Definition 3.9. Let $G = \langle N, E, l \rangle$ be a graph, let *pds* be a partition decision structure, let $n \in N$ be a node with $v_{pds}(n) = (l(n), S)$. The size of $v_{pds}(n)$ is given by $|v_{pds}(n)| = 1 + |S|$.

Proposition 3.10. Let $G = \langle N, E, l \rangle$ be a graph, let $n \in N$ be a node. If node labels and positions in the partition decision structure can be stored in a fixed amount of storage; then the storage needed for the node-decision value $v_{pds}(n)$ of node n is $\Theta(|v_{pds}(n)|)$.

Using the size defined for node-decision values we can prove strong bounds on the size of the node-decision value and on the partition decision structure.

Theorem 3.11. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G, let *pds* be a partition decision structure, let $n \in N$ be a node. The size of $v_{pds}(n)$ is given by $|v_{pds}(n)| = 1 + |E_\downarrow(n)|$.

Proof. Let $v_{pds}(n) = (l(n), S)$. According to Theorem 3.8 all bisimilar equivalent child nodes of node n will have the same node-decision value and querying the *pds* with these values will result in the same partition block identifier. The set S thus contains one node-decision value per group of bisimilar equivalent child nodes of n. The node $n_\downarrow \approx n, n_\downarrow \in N_\downarrow$ will have one child per group of bisimilar equivalent child nodes of n; thus $|E_\downarrow(n)| = |S|$. □

Theorem 3.11 proves a strict bound on the size of individual node-decision values. Based on this result we can easily proof a strict lower bound on the size of any (implementation of a) partition decision structure.

Theorem 3.12. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G. The size of the partition decision structure for graph G is $|N_\downarrow| + |E_\downarrow|$.

Proof. According to Theorem 3.8 all bisimilar equivalent nodes will have the same node-decision value. The partition decision structure thus contains one entry for every group of bisimilar equivalent nodes. Each group of bisimilar equivalent nodes is represented by a single node $n_\downarrow \in N_\downarrow$. According to Theorem 3.11 the size for the entry for node $n_\downarrow$ is $1 + |E_\downarrow(n_\downarrow)|$. Summarizing over every group of bisimilar equivalent nodes in graph G gives a total size of $|N_\downarrow| + |E_\downarrow|$. □

The size of the partition decision structure as proven by Theorem 3.12 does not stand in the way of an efficient bisimulation partition algorithm. Using the node-decision value as search key in a partition decision structure is thus a viable approach.

Theorem 3.12 also hints at a correlation between the partition decision structure of a graph and the maximum bisimulation graph of the same graph. We end our investigation of decision structures by taking a closer look at this correlation.

Theorem 3.13. Let $G = \langle N, E, l \rangle$ be a graph, let *pds* be the partition decision structure obtained after running the online bisimulation algorithm. The partition decision structure *pds* is a list representation of the maximum bisimulation graph $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ of graph G.

Proof (sketch). Let $(l(n), S)$ be the i-th element in the partition decision structure. This i-th entry corresponds to the maximum bisimulation graph node with node identifier i and label $l(n)$. In this representation set S contains the node identifiers of children of the maximum bisimulation graph node with node identifier i. □

From Theorem 3.13 it follows that the combination of the outline in Algorithm 3.1 and a partition decision structure calculates not only a bisimulation partition but also a maximum bisimulation graph. Thereby the algorithm calculates all the components for constructing a graph index.

3.1.2 Online bisimulation

With the partition decision structure we have introduced a decision structure that can be combined with the outline in Algorithm 3.1; resulting in a functional online bisimulation partitioning algorithm. This online bisimulation partitioning algorithm is presented in Algorithm 3.2.

Algorithm 3.2 Online bisimulation partitioning algorithm

Require: Directed acyclic graph $G = \langle N, E, l \rangle$.
Ensure: The output is the pair (n, p) for every $n \in N$; with p an identifier for the bisimulation partition block whereto n belongs.

1: *pds* is an empty (key, value) mapping
2: **for all** $n \in N$, in reverse-topological order **do**
3: $(* \; v_{pds}(n) \leftarrow \text{KEY}(n) \; *)$
4: $v_{pds}(n) \leftarrow (l(n), \{pds[v_{pds}(m)] : m \in E(n)\})$
5: $(* \; p \leftarrow \text{QUERY}(pds, \text{KEY}(n)) \; *)$
6: **if** *pds* does not contain the key $v_{pds}(n)$ **then**
7: $pds[v_{pds}(n)] \leftarrow |pds|$
8: $p \leftarrow pds[v_{pds}(n)]$
9: **print** (p, n)

We have already mentioned that the unordered list is not the most efficient data structure for storing and searching (key, value)-pairs. Therefore we shall assume that we use a more suitable data structure.

Assumption 3.14. Let the partition decision structure be implemented by some efficient data structure. Each query (lookups, insertions) on this data structure has an amortized cost of $O(L)$.

With Assumption 3.14 we can study runtime complexity and memory usage of Algorithm 3.2.

Theorem 3.15. The worst case runtime complexity of Algorithm 3.2 is $O((|N| + |E|)(1 + L))$.

Proof. The main loop of the algorithm is executed $|N|$ times. For calculating the node-decision value of a node n we visit all outgoing edges of node n; thereby visiting every edge once during the execution of the algorithm. The partition decision structure is queried once for every node; and once for every edge. □

According to Theorem 3.12 the memory usage of Algorithm 3.2 is lower bounded by $O(|N_\downarrow| + |E_\downarrow|)$; but exact values depend on the used data structure.

Remark 3.16. In Algorithm 3.2 the *pds* is queried for every outgoing edge of every node n when determining the node-decision value of node n. These edge queries can be eliminated by maintaining a mapping between nodes n and the partition block identifier of the partition block wherein node n is placed.

One way of implementing this mapping is by annotating each node n with the value $q[n] \leftarrow p$ (after line 8 in Algorithm 3.2). The partition decision structure of a node n' can then be retrieved by replacing line 4 with $v_{pds}(n') \leftarrow (l(n'), \{q[m'] : m' \in E(n')\})$. The worst case runtime complexity for this alternative is $O(|N|(1 + L) + |E|)$. This alternative does however increase memory usage to $O(|N| + |E_\downarrow|)$.

Algorithm 3.2 and the alternative described in Remark 3.16 can both be considered runtime efficient when appropriate data structures are used to implement the partition decision structure. We are however primarily interested in the IO efficiency of the algorithms. There are two important reasons why Algorithm 3.2 and the alternative from Remark 3.16 are not IO efficient.

The first reason is the partition decision structure; we haven't yet introduced an IO efficient implementation for this partition decision structure. The second reason is the way wherein node-decision values are calculated; during calculation of the node-decision value of node n we visit the outgoing edges of node n; this counteracts the whole benefit of online processing of the input.

In Section 3.2 we shall introduce additional techniques for removing the need to visit any edge in the graph; thereby making the entire algorithm online according to the strong principle. In Section 3.3 we take a look at how we can circumvent the IO efficiency problems introduced by the partition decision structure.

3.2 Introducing time-forward processing

In Algorithm 3.2 and the alternative described in Remark 3.16 we traverse the outgoing edges of node n during the calculation of the node-decision value of node n. When we remove these explicit edge traversals we can improve the locality of the algorithm; as such making it easier to implement the algorithm in an IO efficient way.

We can remove these explicit edge traversals by introducing a supporting structure that can be used to send information from child to parent. We thereby send the value $q[m]$ as described in Remark 3.16 from node m to parent node n. The idea of sending node information from children to parents in a directed acyclic graph is a variant of the technique called time-forward processing. A detailed general description of time-forward processing can be found in [Zeh02] and [MSS03]. We shall only describe how we use time-forward processing for eliminating edge traversals in our online bisimulation algorithm.

3.2.1 The time-forward processing technique

Central for time-forward processing is using an IO efficient priority queue to send information from one node to another node in the graph. Time-forward processing can be used when nodes have unique ordered node identifiers and whereby nodes have access to all the node identifiers of their parents. We shall first introduce a formal graph representation for directed acyclic graph based on these constraints.

Definition 3.17. Let $G = \langle N, E, l \rangle$ be a directed acyclic graph. Graph G can be represented by list representation $\mathcal{L}$. In list representation $\mathcal{L}$ every node $n \in N$ is represented by a list element $e = (i, l(n), E'(n))$ where

(1) i is the node identifier of node n; corresponding to the position of element e in the list,

(2) $l(n)$ is the label of the node, and

(3) $E'(n)$ is the list of parent nodes of n; every parent node represented by its node identifier.

Additional we require that the nodes in a graph represented in list representation $\mathcal{L}$ are reverse-topological ordered. For a list element $e = (i, l(n), E'(n))$ and parent node identifier $m \in E'(n)$ we thus have $i < m$.

Example 3.18. Let $G = \langle N, E, l \rangle$ be the graph shown in Figure 3.2.

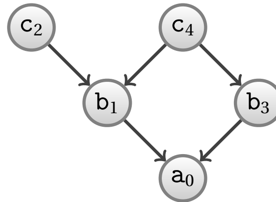


Figure 3.2: A directed acyclic node labeled graph; the subscript on each node represents a unique node identifier.

The list $[(0, a, [1,3]), (1, b, [2,4]), (2, c, []), (3, b, [4]), (4, c, [])]$ represents this graph in $\mathcal{L}$ representation.

Assume we have a graph represented by list L in list representation $\mathcal{L}$ and a value v calculated on node n_i with node identifier i. This value should be send to a parent node n_j with identifier j, $i < j$. We can put the value v on a priority queue Q with priority j.

Also assume we process all nodes in order of their position in list L; and each node only placed values on the priority queue for its parents. We also assume that each node n_i removes all values with priority i from the top of the queue (if any of these values are on top of the queue). Under these conditions the top of the queue can only contain values with a priority of at least i during the processing of node n_i; thus the top will contain all messages intended for node n_i (if any are present).

In this setting we thus can use an IO efficient priority queue to send information from nodes n to all parent nodes of node n in an IO efficient way.

3.2.2 Time-forward processing online bisimulation partitioning algorithm

We can combine time-forward processing with Algorithm 3.2. Thereby we shall also incorporate the ideas described in Remark 3.16. This results in the algorithm presented in Algorithm 3.3.

Algorithm 3.3 Online bisimulation partitioning algorithm (using time-forward processing)

Require: Directed acyclic graph $G = \langle N, E, l \rangle$ in list representation $\mathcal{L}$.
Ensure: The output is the pair (n, p) for every $n \in N$; with p an identifier for the bisimulation partition block whereto n belongs.

```
 1: pds is an empty (key, value) mapping
 2: Q is an empty priority queue
 3: for all (n, l(n), E'(n)) ∈ G, in order of node identifier do
 4:    S is an empty set
 5:    for all Top(Q) = (n, p) for some p ∈ ℕ do
 6:       S ← S ∪ {p}
 7:       Pop(Q)
 8:    v_pds(n) ← (l(n), S)
 9:    if pds does not contain the key v_pds(n) then
10:       pds[v_pds(n)] ← |pds|
11:    p ← pds[v_pds(n)]
12:    (∗ Send p to parents of n ∗)
13:    for all m ∈ E'(n) do
14:       Add(Q, (m, p))
15:    print (p, n)
```

We shall first take a look at the correctness of Algorithm 3.3. The correctness of the algorithm follows from the correctness of the alternative for Algorithm 3.2 described in Remark 3.16 and from the correctness of the used time-forward processing technique. Before we can analyze the runtime and IO complexity of Algorithm 3.3 we need to take a look at how the set S can be constructed efficiently (starting at line 4).

In the way we have introduced time-forward processing we have assumed that the ordering on node identifier determines the priority for any (node identifier, partition block identifier)-pair placed on the priority queue. In such setting all the partition block identifiers for a node are placed unordered on the priority queue. This complicates the insertion of partition block identifier p into set S (line 6) as we need to take care of duplicates; thereby introducing the need for a complex set data structures or expensive duplicate checks on S.

We can however change the ordering used by the priority queue for determining priorities to guarantee that for each node the partition block identifiers are retrieved in order. Let the lexicographical ordering of the (node identifier, partition block identifier)-pairs determine the priority of elements placed on the priority queue. Now any pair (i, p) only has the same priority as other pairs (i', p') whenever these pairs are equivalent; this only happens when a partition block identifier is send to the same node several times. The priority queue utilizing a lexicographical ordering for determining node priority thus guarantees that partition block identifiers are added to set S in an ordered way.

Assumption 3.19. The priority queue Q in Algorithm 3.3 uses a lexicographical ordering of the (node identifier, partition block identifier)-pairs. The set S is implemented as a list. Due to the ordering guaranteed by the priority queue we only have to check if the last partition block identifier added to set S is equivalent to the one stored on top of the queue; only then do we have a duplicate. We thus only add new elements to list S when the top of the priority queue is different from the last element in list S.

With this implementation of set S we can perform the runtime and IO complexity analysis of Algorithm 3.3. Thereby we shall use Assumption 3.14 for the runtime cost on querying the partition decision structure.

Theorem 3.20. The worst case runtime complexity of Algorithm 3.3 is $O(|N|(1+L)+|E|+\mathrm{PQ}(|E|))$.

Proof (sketch). In total $|E|$ elements are added and removed from the priority queue; this introduces an additional runtime cost of $O(\mathrm{PQ}(|E|))$ to the cost of the alternative for Algorithm 3.2. □

The runtime complexity does not have to be equivalent to the IO complexity. As a last step we shall analyze the IO complexity. We have not yet discussed any details on the partition decision structure. As such we can only analyze the IO cost of the algorithm without any operation involving the *pds*.

Theorem 3.21. The worst case IO complexity of Algorithm 3.3 is $O(\textsc{Scan}(|N|+|E|)+\mathrm{PQ}(|E|))$ when excluding any operation wherein the partition decision structure is involved.

Proof. Algorithm 3.3 reads the entire graph in list representation $\mathcal{L}$ sequentially; reading the input thus costs $O(\textsc{Scan}(|N|+|E|))$ IOs. The accumulative size of data structure S; when implemented as described in Assumption 3.19; is $|E|$. The total IO cost for the data structure S thus is $O(\textsc{Scan}(|E|))$. The algorithm will add an element to the priority queue for every edge; only these elements are removed. Thereby the total IO cost for the priority queues is $O(\mathrm{PQ}(|E|))$ IOs. □

With Algorithm 3.3 we have presented an algorithm that adheres to the strong online principle. This algorithm can however not yet be considered to be an IO efficient external memory algorithm; the algorithm is missing details on how the partition decision structure should be implemented. In the next section we shall try to fill in these missing details.

3.3 On partition decision structures

The efficiency of Algorithm 3.3 still relies on the efficiency of the partition decision structure. In this section we shall investigate if and how we can implement an IO efficient partition decision structure. Therefore we shall first make a practical assumption on the node-decision values.

Assumption 3.22. Let n be a node with node-decision value $(l(n), S)$. We assume that set S is represented by an ordered list. Assumption 3.19 describes how this can be achieved by utilizing Algorithm 3.3.

Assumption 3.22 simplifies any implementation of the partition decision structures. The assumed ordering on node-decision values simplifies comparison and hashing of node-decision values as every node-decision value has only a single ordered representation. Thereby Assumption 3.22 makes it easier to analyze the usage of data structures for implementing the partition decision structure.

We shall start our investigation for a partition decision structure with a small survey of useful data structures. These data structures will show practical limits on the efficiency of the partition decision structure. We then take a look at possible query patterns on the partition decision structure and how we can optimizing the query patterns to reduce the total IO cost for any implementation of the partition decision structure.

3.3.1 External memory search structures

The partition decision structure is nothing more than a one-to-one mapping between node-decision values and unique numeric identifiers. There are many data structures developed for storing and querying (key, value)-pairs. These data structures include lists, B^+ trees and hash tables.

Theorem 3.11 already proves that node-decision values have a variable size. The size of a node-decision value is upper bounded by the number of outgoing edges the node has. Thus the (key, value)-pairs stored in the partition decision structure have a non-fixed size. Moreover the search key can be arbitrary large; even larger than the block size B. This property of the search key rules out the usage of B^+ trees. We can however use a variant of the balanced search trees; namely string B-trees [FG99, MSS03]. We do so by representing the ordered node-decision value $(l, \{s_1, \ldots, s_n\})$ as a string $l s_1 \ldots s_n$.

For all mentioned data structures the IO cost to read entries from external memory is at least a single IO. In the case that every query needs to be served from external memory we thus have a total worst case IO cost that is lower bounded by $\Omega(|N|)$. The total IO cost might be even higher due to the cost associated with querying large node-decision values.

3.3.2 Query patterns

For a partition decision structure implemented by any of the mentioned data structures we have seen that the worst case lower bound on the number of IOs is $\Omega(|N|)$ when every query to the partition decision structure is served from external memory. We shall analyze the possible query patterns to see of this worst case lower bound can occur in practice. Therefore we make some assumptions on the query cost.

Assumption 3.23. Let the partition decision structure be implemented by some efficient data structure. Each query (lookups, insertions) on this data structure has an amortized IO cost of $O(L)$.

We also make an assumption on the memory usage for the partition decision structure.

Assumption 3.24. The partition decision structure can keep one (key, value)-pair in main memory. We thereby assume that the result of the last query is kept in main memory.

Note that Assumption 3.24 can only hold when the node-decision keys are bounded in size; as such the assumption does not reflect the worst possible situation whereby node-decision keys are so large that they don't fit in main memory. Under Assumption 3.24 we only have to pay the IO cost $O(L)$ once for consecutive queries with the same node-decision value.

Example 3.25. Let $G = \langle N, E, l \rangle$ be the graph containing j identical chains of length i. The nodes in this graph are defined as $N = \{n_1^1, \ldots, n_i^1, \ldots, n_1^j \ldots n_i^j\}$, the edges are defined as $E = \{(n_{k+1}^m, n_k^m) : (1 \le k < i) \wedge (1 \le m \le j)\}$, and all nodes have the same label. This graph is shown in Figure 3.3.

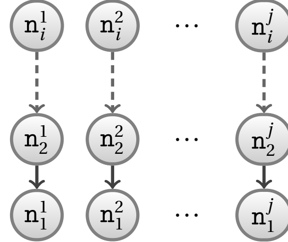


Figure 3.3: A graph containing j chains of length i. The subscript on each node represents the unique identifier of a node within its chain. The superscript on each node indicates the chain whereto the node belongs.

For fixed values $k, 1 \le k \le i$ we have that all nodes $n_k^m, 1 \le m \le j$ are bisimilar equivalent to each other. The maximum bisimulation graph of graph G thus is a single chain. The bisimulation partition P of nodes N is given by $P = \{s_k : s_k = \{n_k^m : 1 \le m \le j\} \wedge (1 \le k \le i)\}$.

Now assume that we have a list L representing graph G in list representation $\mathcal{L}$. This list L must represent all nodes in a valid reverse-topological order. One possible reverse-topological order can be achieved when all nodes n_k^m are lexicographically ordered on (m, k).

This reverse-topological ordering guarantees that every query will be a query with a different search key as the previous query. Thus for each query, and for each node, we need to pay an IO cost of $O(L)$. The total IO cost will thus be $O(L|N|)$.

The list, hash table and string B-trees all have $L \le 1$ in the worst case. Combining this with Example 3.25 proves that in the worst case we would have $\Omega(|N|)$ IOs for querying the partition decision structure. We shall now look at a query pattern for the same graph that causes much less IOs than this worst case.

Example 3.26. Once again consider the graph presented in Example 3.25, now with another reverse-topological order on the nodes of the graph. The new reverse-topological order is achieved by ordering

all nodes n_k^m lexicographically on (k, m). Now the queries are grouped in i groups containing j queries with the same key per group. As such a total of $|N_\downarrow|$ queries need to be served from disk with an IO cost of $O(L|N_\downarrow|)$.

When we take a closer look at Example 3.26 then we can conclude that any IO cost for the example is unnecessary. The used reverse-topological ordering already groups all bisimilar equivalent nodes together; so all queries with the same node-decision value are performed consecutive. The partition decision structure thus doesn't need to keep any entry in the partition decision structure after the algorithm has queried the entire group of bisimilar equivalent nodes. As such there is no need to write any entry to external memory.

Example 3.26 is however an artificial example wherein the nodes provided as input are ordered on their bisimulation partition block. We thus cannot expect an equivalent situation for all graphs; as this would remove the need for any complex bisimulation partitioning algorithm. We can however generalize the idea applicable to Example 3.26 for reducing the need to keep partition decision structure entries.

If the nodes of the input graph are grouped such that for any two bisimilar equivalent nodes in the graph we have that they are placed in the same group. Then we only need to maintain the partition decision entries related to these nodes as long as we are processing the group. When all nodes in the group are processed; then we can discard all partition decision structure entries created for the group of nodes. We can formalize the conditions for this generalization by stating that the nodes in the input graph should be partitioned into some partition P_i; such that the maximum bisimulation partition P is a refinement of P_i.

Definition 3.27. Let $G = \langle N, E, l \rangle$ be a graph, let P_i be a partition of N such that the bisimulation partition P of N is a refinement of P_i. If the nodes N in the graph are grouped with respect to partition P_i then P_i is the initial partition of graph G. We refer to partition blocks $p_i \in P_i$ as initial partition blocks.

We shall first proof that any initial partition of a graph meets the condition that any two bisimilar equivalent nodes in the graph are placed in the same partition.

Theorem 3.28. Let $G = \langle N, E, l \rangle$ be a graph, let P_i be an initial partition of graph G, let $n \in N$ be a node and $p \in P_i$ be the partition block with $n \in p$, let $m \in N$ be a node. We have $n \approx m$ implies $m \in p$.

Proof. Assume $n \approx m$ holds; due to Definition 2.26 we can conclude that there is a partition block $p' \in P$ with $n \in p'$ and $m \in p'$. According to Definition 2.27 there must be a partition block $p'' \in P_i$ with $p' \subseteq p''$ and thus $n \in p''$ and $m \in p''$. Definition 2.24 guarantees that only a single partition block in P_i contains node n; we thus have $p'' = p$ and $m \in p$. □

The online bisimulation algorithms we have been working on only work when the nodes in the input graph are reverse-topological ordered. Thus when we want to process nodes based on the order imposed by an initial partition; then it must be possible that the partition blocks in the initial partition can be ordered such that all nodes are reverse-topological ordered. A candidate initial partition that meets this condition is the rank partition defined in Definition 2.34; whereby we can simply order the partition blocks on increasing rank of the nodes in each partition block.

Theorem 3.29. Let $G = \langle N, E, l \rangle$ be a graph, let P_i be an initial partition of graph G wherein the partition blocks can be ordered such that the nodes N are reverse-topological ordered. When the initial partition blocks are processed based on this reverse-topological order; then the partition decision structure only has to keep (key, value)-pairs for node-decision values of nodes that are part of the initial partition block that is currently being processed.

Proof. Let $n \in N$ be a node placed in initial partition block $p_i \in P_i$. According to Theorem 3.28 all nodes $m \in N, n \approx m$ are also placed in partition block p_i. Only these nodes m will have the same node-decision value as node n. As such the partition decision entry $(v_{pds}(n)$, value) for node n is only needed when processing the nodes in partition block p_i. □

Theorem 3.29 shows that using initial partitions other then $\{N\}$ can improve the performance of Algorithm 3.3 quite a bit; this by optimizing the pattern wherein the partition decision structure is queried.

3.3.3 Structural summary partition

To maximize the improvement gained from optimizing the query patterns one wants to find an easy-computable initial partition wherein each partition block is as small as possible. Many internal memory algorithms use some initial partition. The algorithms based on the work by Robert Paige and Robert E. Tarjan often use label partitions. For directed acyclic graphs rank partitions are used. From Corollary 2.32 and Corollary 2.37 we can conclude that the label partition and the rank partition both are initial partitions.

Label equivalence or rank equivalence does in no way guarantee a close to optimal initial partition wherein partition blocks are as small as possible. A way to get an optimal initial partition is by grouping nodes on node-bisimilarity values. This approach is however impractical due to the storage needed for the node-bisimilarity value. We have solved this problem before by using the one-to-one mapping between node-bisimilarity values and partition block identifiers. This resulted in the node-decision value of Definition 3.7; we however need a partition decision structure for this approach to work.

In the definition of node-decision values we use the partition decision structure as a function mapping other node-decision values to numeric identifiers. We can replace this partition decision structure function by other functions; thereby defining other types of node values.

Definition 3.30. Let $G_{\mathcal{D}} = \langle N, E, l\rangle$ be a graph, let $\mathbb{F}$ be a set, let $\mathcal{F} : \mathcal{D} \times \wp(\mathbb{F}) \to \mathbb{F}$ be a function. We define the node-value $v_{\mathcal{F}}(n)$ of node n with respect to this function $\mathcal{F}$ as $v_{\mathcal{F}}(n) = \mathcal{F}(l(n), \{v_{\mathcal{F}}(m) : m \in E(n)\})$.

Note the resemblance between Definition 3.30 and the node-decision value. Just as with the node-decision value this definition of the node-value $v_{\mathcal{F}}$ implies that the node-value of a node n can only be determined when the node-value of the children of node n are already evaluated. We have proven a strong relation between node bisimulation and node-decision values; we shall now look at the relation between node bisimulation and node-decision values.

Theorem 3.31. Let $G = \langle N, E, l\rangle$ be a directed acyclic graph, let $n \in N, m \in N$ be nodes in this graph. We have $n \approx m$ implies $v_{\mathcal{F}}(n) = v_{\mathcal{F}}(m)$.

Proof. The proof is by induction on the rank of the nodes.

BASE CASE: Let node n be a node with $rank(n) = 0$. We have $v_{\mathcal{F}}(n) = \mathcal{F}(l(n), \emptyset)$. Let m be a node with $m \approx n$. We have $v_{\mathcal{F}}(m) = \mathcal{F}(l(m), \emptyset)$. According to Definition 2.15 we have $l(n) = l(m)$ and thus $v_{\mathcal{F}}(n) = v_{\mathcal{F}}(m)$.

INDUCTION HYPOTHESIS: Let n be a node with rank up to r. We have $n \approx m$ implies $v_{\mathcal{F}}(n) = v_{\mathcal{F}}(m)$ for every node m.

INDUCTION STEP: Let node n be a node with $rank(n) = r+1$. We have $v_{\mathcal{F}}(n) = (l(n), S_n)$. Let m be a node with $m \approx n$. We have $v_{\mathcal{F}}(m) = (l(m), S_m)$ and according to Definition 2.15 we have $l(n) = l(m)$. Definition 2.15 guarantees that for every node $n' \in E(n)$ there is a node $m' \in E(m)$ with $n' \approx m'$. These nodes n' have $rank(n') \leq r$; thus according to the induction hypothesis we have $v_{\mathcal{F}}(n') = v_{\mathcal{F}}(m')$. Therefrom we can conclude $S_n \subseteq S_m$. In the same way we can prove $S_n \supseteq S_m$; thereby proving $S_n = S_m$ and thus $v_{\mathcal{F}}(n) = v_{\mathcal{F}}(m)$.

□

For any node-value $v_{\mathcal{F}}$ with respect to any function $\mathcal{F}$ we can define an initial partition by grouping nodes on equivalent node-value. We can pick good functions $\mathcal{F}$ based on particular properties of the (expected) input graphs. This approach is however limited to graphs wherefore such properties are known. For a more general approach we can use a hash function as function $\mathcal{F}$.

Definition 3.32. Let $\mathbb{H}$ be a finite set. The function $\mathcal{H} : \mathcal{U} \to \mathbb{H}$ is called a hash function; it maps keys drawn from some domain $\mathcal{U}$ to a hash value. The hash function is typically a many-to-one function. Thereby it is unavoidable that there are collisions. We speak of a collision if we have a pair of unequal values $u_1 \in \mathcal{U}, u_2 \in \mathcal{U}, u_1 \neq u_2$ that have the same hash value; thus $\mathcal{H}(u_1) = \mathcal{H}(u_2)$.

Using the hash function we can calculate a node-value for every node. On these node-values we can define an initial partition by grouping nodes with equivalent node-values. To guarantee that the partition blocks in such an initial partition can be ordered such that the nodes are reverse-topological ordered we not only group on equivalent node-value but also on equivalent rank. We can also include grouping of nodes on equivalent label; this removes the need to check on labels later on during bisimulation partitioning. We refer to the resulting partition as the structural summary partition.

Definition 3.33. Let $G = \langle N, E, l \rangle$ be a graph, let $\mathcal{H} : \mathcal{D} \times \wp(\mathbb{H}) \to \mathbb{H}$ be a hash function. The structural summary $\mathcal{S}(n)$ of a node n is defined as $\mathcal{S}(n) = (rank(n), l(n), v_{\mathcal{H}}(n))$. The structural summary partition $P_{\mathcal{S}}$ is defined as the node-value partition for function $\mathcal{S}$. The node-value $v_{\mathcal{H}}(n)$ is called the node-hash value of node n.

Theorem 3.34. Let $G = \langle N, E, l \rangle$ be a graph, let $P_{\mathcal{S}}$ be the structural summary partition of nodes N. The bisimulation partition P is a refinement of the structural summary partition $P_{\mathcal{S}}$.

Proof. Node bisimulation equivalence implies rank equivalence, label equivalence and equivalence on the node-hash value. As such node bisimulation equivalence implies structural summary equivalence. Using Theorem 2.31 we can conclude that P is a refinement of $P_{\mathcal{S}}$. □

With a good hash function $\mathcal{H}$ the structural summary partition is expected to be close to the bisimulation partition. Thereby a good hash function distributes the hash values of a set of inputs uniformly over the domain $\mathbb{H}$. Furthermore a good hash function distributes the hash values of a set of inputs randomly; whereby any correlations between the values in the set of inputs are not reflected in the distribution of the hash values. Any hash function can however have collisions whereby non-equivalent input values results in equivalent node-hashes. Structural summaries thus do not remove the need for a bisimulation partitioning algorithm to further refine the input to a bisimulation partition.

In most applications whereby a hash function (with collisions) is used these collisions don't have a great effect. In these applications the collisions are localized; we shall however show that this is not the case for node-hash values.

Example 3.35. Let $G = \langle N, E, l \rangle$ be the graph shown in Figure 3.4.

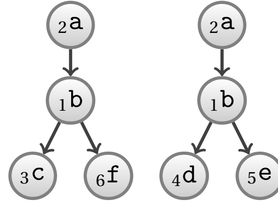


Figure 3.4: A graph wherein each node is annotated with a node-hash value. This node-hash value is placed as a subscript on the left of the node label. For calculating the node-hash value of each node we have mapped the node labels to integers i by mapping each letter from the alphabet to its position in the alphabet (thus $a \to 1, \ldots, z \to 26$). The node-hash value of a node n is then calculated as $v_{\mathcal{H}}(n) = (i + \sum_{v \in \{v_{\mathcal{H}}(m) : m \in E(n)\}} v) \bmod 10$.

We see that the nodes with label b are not bisimilar equivalent; these nodes have children with completely different labels. Due to a collision the nodes with label b do have the same node-hash value. The nodes with label a only have edges to these nodes b; the nodes with label a thus get the same node-hash value. The collision for the node-hash value of the nodes with label b thus has propagated to the nodes with label a.

Collision propagation on node-hash values as described in Example 3.35 increases the probability for collisions on node-hash values. In the worst case collision propagation will have a cumulative effect. On the other hand the effect of collisions is reduced by including rank and label equivalence in the structural summary partition. As such we expect that the structural summary partition will turn out to be a good initial partition; with almost as much partition blocks as the bisimulation partition. The

definition of the structural summary partition at least guarantees that the structural summary partition is at least as good as partitions based on label and/or rank equivalence.

3.3.4 Using structural summaries for bisimulation partitioning

We have introduced structural summary partitions and we have shown that the usage of such an initial partition as input for Algorithm 3.3 is expected to greatly reduce the cost for every query on the partition decision structure. This improvement is achieved by reducing the number of entries maintained by the partition decision structure at any given time. For this approach to work we do however need a way to convert arbitrary graphs in list representation $\mathcal{L}$ into some initial structural summary partition. We shall thus introduce a formal graph representation for directed acyclic graphs wherein we can express structural summary partitions.

Definition 3.36. Let $G = \langle N, E, l \rangle$ be a graph, let $P_{\mathbb{S}}$ be the structural summary partition of N. Graph G can be represented by list representation $\mathcal{L}_{\mathbb{S}}$. In list representation $\mathcal{L}_{\mathbb{S}}$ every partition block $p \in P$ is represented by a list element (l', L) where

(1) l' is the common label of the nodes in L, and

(2) L is a list whereby each node $n \in L$ is represented by a list element $e = (i, E'(n))$ where

  (a) i is the node identifier of node n, and

  (b) $E'(n)$ is the list of parent nodes of n; every node represented by its node identifier.

Additional the partition blocks (as represented by list elements) are lexicographically ordered on increasing structural summary.

The node identifier i is defined with respect to its overall position in the list representation $\mathcal{L}_{\mathbb{S}}$ of a graph. When we read a graph in list representation $\mathcal{L}_{\mathbb{S}}$ sequentially, thereby reading each list of nodes in each element (l', L) sequentially; then the i-th node list element we read will have node identifier i.

Example 3.37. Let $G = \langle N, E, l \rangle$ be the graph shown in Figure 3.5.

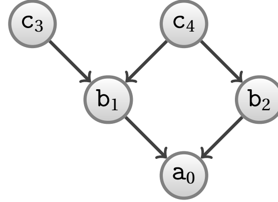


Figure 3.5: A directed acyclic node labeled graph; the subscript on each node represents a unique node identifier.

The list $[(a, [(0, [1, 2])]), (b, [(1, [3, 4]), (2, [4])]), (c, [(3, []), (4, [])])]$ represents this graph in $\mathcal{L}_{\mathbb{S}}$ representation.

With this representation $\mathcal{L}_{\mathbb{S}}$ we can introduce an algorithm that takes a graph in list representation $\mathcal{L}$ and produces the same graph in list representation $\mathcal{L}_{\mathbb{S}}$. Such an algorithm needs to calculate ranks and node-hash values for each node; the definition of rank and node-hash value hint at the application of time-forward processing. This approach leads to Algorithm 3.4.

Algorithm 3.4 introduces a shorthand notation for reading and removing elements from the priority queue; we first introduce this shorthand notation before looking at the details of the algorithm.

Remark 3.38. Let Q be a priority queue. The expression $(n, c) = \textsc{Top}(Q)$ checks if the top element on queue Q is intended for node n; if this is the case then the value c is returned and the top element is popped. If the top element is not intended for node n, then no additional values are read from the queue.

Algorithm 3.4 Converting graphs in $\mathcal{L}$ representation into graph in $\mathcal{L}_S$ representation

Require: Directed acyclic graph $G = \langle N, E, l \rangle$ in list representation $\mathcal{L}$.
Ensure: List L is a directed acyclic graph $G = \langle N, E, l \rangle$ in list representation $\mathcal{L}_S$.

1: N', E', L are empty lists
2: $Q_{rank}, Q_{\mathcal{H}}$ are empty priority queues
3: **for all** $(n, l(n), E'(n)) \in G$, in order of node identifier **do**
4: $\quad rank(n) \leftarrow \max(\{r+1 : (n,r) = \textsc{Top}(Q_{rank})\} \cup \{0\})$
5: $\quad v_{\mathcal{H}}(n) \leftarrow \mathcal{H}(l(n), \{s : (n,s) = \textsc{Top}(Q_{\mathcal{H}})\})$
6: $\quad \textsc{Add}(N', (rank(n), l(n), v_{\mathcal{H}}(n), n))$
7: $\quad$ **for all** $m \in E'(n)$ **do**
8: $\quad\quad \textsc{Add}(E', (m, n))$
9: $\quad\quad \textsc{Add}(Q_{rank}, (m, rank(n)))$
10: $\quad\quad \textsc{Add}(Q_{\mathcal{H}}, (m, v_{\mathcal{H}}(n)))$
11: $\textsc{Sort}(N')$ on lexicographical order
12: Give nodes new identifiers based on the new position in N'
13: Change identifiers for edges E' based on new node identifiers
14: $\textsc{Sort}(E')$ on (child node identifier, parent node identifier)
15: **for all** structural summaries $(r, l', v_{\mathcal{H}})$, lexicographical ordered **do**
16: $\quad L_p$ is an empty list
17: $\quad$ **for all** node $n, (r, l', v_{\mathcal{H}}, n) \in N'$, in order of node identifier **do**
18: $\quad\quad \textsc{Add}(L_p, (n, \{m : (m,n) \in E'\}))$
19: $\quad \textsc{Add}(L, L_p)$

Algorithm 3.4 gives little details on how the node identifiers in list E' and list N' can be updated IO efficient (line 11–14). We shall first provide some details on how to achieve this before we take a look at the correctness and the IO complexity of the algorithm

Remark 3.39. We can renumber the nodes in N' by sequentially updating the list, replacing each old node identifier by the new position in the list. During this update process we can construct a renumber list R containing (old node identifier, new node identifier)-pairs.

For updating the node identifiers in edge list E' we use the renumber list R. Sort this list R on old node identifier. To update the child node identifiers in the edge list we sort the edge list on child node identifiers; we then sequentially read list R and at the same time sequentially update the child node identifiers in the edge list. To update the parent node identifiers in the edge list we sort the edge list on parent node identifier; we then sequentially read list R and at the same time sequentially update the parent node identifiers in the edge list.

The correctness of Algorithm 3.4 follows from Remark 3.39, the correctness of the time-forward processing technique and from Definitions 2.33, 3.33 and 3.36. We shall now show that Algorithm 3.4 performs the conversion in an IO efficient way.

Theorem 3.40. The worst case IO complexity of Algorithm 3.4 is $O(\textsc{Sort}(|N|) + \textsc{Sort}(|E|) + \text{PQ}(|E|))$.

Proof. We shall proof the IO complexity for the three parts of the algorithm separately.

LINES 4–10: The graph is read sequentially; lists E' and N' are written sequentially. These reads and writes cost $O(\textsc{Scan}(|N| + |E|))$ in total. The algorithm will also add an element to both priority queues for every edge; these elements are later removed. Thereby the total IO cost for this part of the algorithm is $O(\textsc{Scan}(|N| + |E|) + \text{PQ}(|E|))$ IOs.

LINES 11–14: Using Remark 3.39 these operations can be achieved by sequentially reading and updating lists N', E' and reading and constructing list R. The lists N', E' and R are also sorted; thereby the total IO cost for this part of the algorithm is $O(\textsc{Sort}(|N|) + \textsc{Sort}(|E|))$.

LINES 15–19: The ordering of lists N' and E' achieved in the previous part of the algorithm guarantees that during this part of the algorithm both lists can be read sequentially. The list L is sequentially written. The list L_p is sequentially written and then copied to list L; the size of list L_p will accumulatively be $O(|N|+|E|)$. Thereby the total IO cost for this part of the algorithm is $O(\text{SCAN}(|N|+|E|))$.

□

We can easily adapt Algorithm 3.3 such that it can operate on graphs in list representation $\mathcal{L}_{\mathbb{S}}$. This adaption of Algorithm 3.3 leads to Algorithm 3.5; which we shall only briefly analyze.

Algorithm 3.5 Online bisimulation partitioning algorithm (on graphs in list representation $\mathcal{L}_{\mathbb{S}}$)

Require: Directed acyclic graph $G = \langle N, E, l\rangle$ in list representation $\mathcal{L}_{\mathbb{S}}$.
Ensure: The output is the pair (n, p) for every $n \in N$; with p an identifier for the bisimulation partition block whereto n belongs.

1: $i \leftarrow 0$
2: Q is an empty priority queue
3: **for all** $(l', L) \in G$, in order of the list representation G **do**
4: $\quad$ pds^L is an empty (key, value) mapping
5: $\quad$ **for all** $(i, E'(n)) \in L$, in order of node identifier **do**
6: $\quad\quad$ $v_{pds}(n) \leftarrow (l', \{p : (n, p) = \text{TOP}(Q)\})$
7: $\quad\quad$ **if** pds^L does not contain the key $v_{pds}(n)$ **then**
8: $\quad\quad\quad$ $pds^L[v_{pds}(n)], i \leftarrow i, i+1$
9: $\quad\quad$ $p \leftarrow pds^L[v_{pds}(n)]$
10: $\quad\quad$ **for all** $m \in E'(n)$ **do**
11: $\quad\quad\quad$ $\text{ADD}(Q, (m, p))$
12: $\quad\quad$ **print** (p, n)

Algorithm 3.5 also uses the notation introduced in Remark 3.38 for reading values from the priority queue. Correctness of Algorithm 3.5 follows directly from correctness of Algorithm 3.3. The runtime and IO complexity analysis for Algorithm 3.5 is also along the same lines as the analysis for Algorithm 3.3; resulting in the same runtime and IO complexity.

The localized partition decision structures pds^L used in Algorithm 3.5 are only of the same size as the partition decision structure *pds* used in Algorithm 3.3 when an input graph only has nodes with the same label and rank. In all other cases the localized partition decision structures pds^L is always smaller then the partition decision structure *pds*. When we assume that the cost of querying the partition decision structure depends on the size of the partition decision structure; then the queries performed by Algorithm 3.5 are in total cheaper as those performed by Algorithm 3.3. As such we expect Algorithm 3.5 to be faster as Algorithm 3.3. Thereby we however note that Algorithm 3.5 introduces a considerable cost for running Algorithm 3.4.

3.4 External memory bisimulation partitioning

Algorithm 3.5 is expected to give good performance in many cases. There is however always a probability on collision and collision propagation. We need additional processing on the partition blocks in the structural summary partition to reduce the probability on collisions and remove any probability on collision propagation.

We are in the best position for this additional processing on the partition block p in the structural summary partition when we start partitioning the nodes in partition block p (line 5 in Algorithm 3.5). At this moment all child nodes of the nodes in partition block p are placed in bisimulation partition blocks. The priority queue will thus contain all information to construct the node-decision values for all nodes in partition block p.

When we calculate these node-decision values; then we can sort the nodes in partition block p on their node-decision values. Sorting might be a bit tricky as the size of node-decision values is not

fixed. We can however use the same approach as used for the node-hash value: use a hash function to compress the node-decision values to a fixed size.

Definition 3.41. Let $G = \langle N, E, l \rangle$ be a graph, let $\mathbb{H}$ be a finite set and $\mathcal{H}^L : \wp(\mathbb{N}) \to \mathbb{H}$ be a hash function, let $n \in N$ be a node with node-decision value $v_{pds}(n) = (l(n), S)$, let P be the initial partition, let $p \in P$ be the structural summary partition block wherein node n is placed. The local structural summary $\mathcal{S}^L(n)$ of node n is defined as $\mathcal{S}^L(n) = (|S|, \mathcal{H}^L(S))$. The local structural summary partition of nodes p is defined as the node-value partition of p for function $\mathcal{S}^L$. The value $\mathcal{H}^L(S)$ for node n is called the local node-hash value of node n.

Proposition 3.42. Let $G = \langle N, E, l \rangle$ be a directed acyclic graph, let $n \in N, m \in N$ be nodes, let *pds* be a partition decision structure. When we query nodes in reverse-topological order and only determine the node-decision value of a node when needed for querying; then we have $v_{pds}(n) = v_{pds}(m)$ implies $\mathcal{S}^L(n) = \mathcal{S}^L(m)$ and thus $n \approx m$ implies $\mathcal{S}^L(n) = \mathcal{S}^L(m)$.

We can use the local structural summary to refine the single initial partition block into several local structural summary partition blocks. Property 3.42 together with Theorem 2.31 guarantees that these local structural summary partition block can be refined to bisimulation partition blocks.

The local structural summary depends directly on the node-decision values. Thereby several nodes with different node-decision values can locally end up with the same local structural summary due to local collisions. Propagation of these collisions is however not possible. We expect that the total number of bisimulation partition blocks that share the same rank, label, node-hash and local structural summary is very small when using a good local hash function $\mathcal{H}^L$. With a good local hash function $\mathcal{H}^L$ we thus expect that in each local structural summary partition block the number of entries for a localized partition decision structure is upper bounded by some constant.

We can enhance Algorithm 3.5 by including local refinement of initial partition blocks into local structural summary partitions. This results in Algorithm 3.6.

Algorithm 3.6 External memory online bisimulation partitioning algorithm

Require: Directed acyclic graph $G = \langle N, E, l \rangle$ in list representation $\mathcal{L}_{\mathcal{S}}$.
Ensure: The output is the pair (n, p) for every $n \in N$; with p an identifier for the bisimulation partition block whereto n belongs.

1: $i \leftarrow 0$
2: Q is an empty priority queue
3: **for all** $(l', L) \in G$, in order of the list representation G **do**
4:   N' is an empty list
5:   **for all** $(i, E'(n)) \in L$, in order of node identifier **do**
6:     $S \leftarrow \{p : (n, p) = \textsc{Top}(Q)\}$
7:     $v_{pds}(n) \leftarrow (l', S)$
8:     $\textsc{Add}(N', (|S|, \mathcal{H}^L(v_{pds}(n)), i, E'(n), S))$
9:   $\textsc{Sort}(N')$ on lexicographical order
10:   **for all** local summary $(c, v^L_{\mathcal{H}})$, lexicographical ordered **do**
11:     pds^L is an empty (key, value) mapping
12:     **for all** node n with $((c, v^L_{\mathcal{H}}), i, E'(n), S) \in N'$, in order of the list N' **do**
13:       **if** pds^L does not contain the key (l, S) **then**
14:         $pds^L[(l', S)], i \leftarrow i, i + 1$
15:       $p \leftarrow pds^L[(l', S)]$
16:       **for all** $m \in E'(n)$ **do**
17:         $\textsc{Add}(Q, (m, p))$
18:       **print** (p, n)

Before we can analyze Algorithm 3.6 we need to take a look at some of the details of the algorithm. First is the list N'; storing and sorting this list seems hard as it is storing variable sized entries. This was the exact problem we tried to circumvent by introducing local structural summaries. Therefore we give a description on how this list N' can be implemented in an IO efficient way.

Remark 3.43. The list N' contains entries $(c, v^L_{\mathcal{H}}, i, E'(n), S)$; these entries have a variable size. We can however move the variable sized values $E'(n)$ and S to separate data structures; thereby fixing the size of each entry in list N'. We do so by placing a representation of $E'(n)$ in its own list $L_{E'}$ and a representation of S in its own list L_S. Thereby for every $m \in E'(n)$ list $L_{E'}$ will contain an entry $(c, v^L_{\mathcal{H}}, i, m)$ and for every $v_{pds} \in S$ list L_S will contain an entry $(c, v^L_{\mathcal{H}}, i, v_{pds})$.

The result is N' represented by three separate lists, the entries in each list have a fixed size. These lists can be sorted on lexicographical order. The sequential read of list N' starting at line 10 can then be implemented by sequentially reading the three lists at the same time. The accumulative size of this implementation of list N' is $O(|N|)$. The accumulative size of list $L_{E'}$ and list L_S is $O(|E|)$.

We also have to give details on how the partition decision structure will be implemented.

Remark 3.44. We assume that the number of collisions within a single local structural summary partition block is upper bounded by some constant h_c. Let p be such a local structural summary partition block wherein every node shares the same local structural summary $(c, v^L_{\mathcal{H}})$. From Theorem 3.11 it follows that for every node $n \in p$ the size of the node-decision value $v_{pds}(n)$ is given by $|v_{pds}(n)| = c + 1$.

Under this conditions we can implement the local partition decision structure as a list of (node-decision value, partition block identifier)-pairs. This structure will contain at most h_c entries; and each entry will have size $\Theta(c)$. The total cost of a single lookup is then upper bounded by $O(\text{SCAN}(c h_c)) = O(\text{SCAN}(c))$.

Note that we can freely drop the node label from the partition decision structure and from the remainder of Algorithm 3.6. Label equivalence for nodes in the same initial partition is guaranteed as the initial partition is based on structural summaries.

We shall first look at the correctness of Algorithm 3.6; we then analyze the IO complexity for Algorithm 3.6.

Theorem 3.45. Algorithm 3.6 calculates the bisimulation partition.

Proof (sketch). Correctness of Algorithm 3.6 follows from its construction as described in this chapter. Thereby we use that the input of the algorithm is a structural summary partition. Theorem 3.34 proofs that this structural summary partition can be refined to the bisimulation partition. Each partition block in the structural summary partition is refined to the local structural summary partition.

According to Proposition 3.42 all nodes with the same node-decision value must be placed in the same local structural summary partition block. Using partition decision structures nodes are then assigned an identifier; thereby nodes are assigned the same identifier if and only if they have the same node-decision value. These identifiers provide a grouping on node-decision value. Theorem 3.31 proofs that this grouping on node-decision value is a bisimulation partition. □

We shall first take a look at the expected IO complexity of the algorithm. After this analysis we shall look at the worst case IO complexity of the algorithm.

Theorem 3.46. The expected IO complexity of Algorithm 3.6 is $O(\text{SORT}(|N|) + \text{SORT}(|E|) + \text{PQ}(|E|))$.

Proof. We assume the implementations described in Remark 3.43 and Remark 3.44. Using the implementation of list N' will result in an IO cost of $O(\text{SORT}(|N|) + \text{SORT}(|E|)$ for reading the input graph; refining each initial partition block and then reading the refined partition blocks. Further the algorithm will add an element to the priority queue for every edge; these elements are later removed; thereby introducing an IO cost of $O(\text{PQ}(|E|))$.

Each local partition decision structure can be seen as a single bucket in a virtual hash table with the node-decision value as the hash key. We thus can expect at most a constant h_c different node-decision values being stored in each local partition decision structure. Thereby we have forced that all entry in a single local decision structure have the same size. A query for an entry $v_{pds}(n)$ thus is expected to cost at most $\text{SCAN}(h_c|v_{pds}(n)|)$. The accumulative cost of queries on local partition decision structures is thus given by $O(\sum_{n \in N} \text{SCAN}(h_c|v_{pds}(n)|)) \leq O(\text{SCAN}(h_c|E|)) = O(\text{SCAN}(|E|))$. □

Nothing guarantees the expected complexity of Algorithm 3.6. Therefore we shall also analyze the worst case complexity of Algorithm 3.6.

Theorem 3.47. The worst case IO complexity of Algorithm 3.6 is $O(\textsc{Sort}(|N|) + \textsc{Sort}(|E|) + \text{PQ}(|E|) + \textsc{Scan}(|N||E_\downarrow|))$.

Proof. We assume the implementations described in Remark 3.43 and Remark 3.44. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G, let p be a local structural summary partition block wherein every node has local structural summary $(c, v^L_{\mathcal{H}})$. We shall consider the case where $c = 0$ and the case where $c \neq 0$ separately.

Case $c = 0$: The partition block p only contains leaf nodes; each leaf node having the same label. According to Definition 2.15 all leaf nodes with the same label are bisimilar equivalent. As such the local partition decision structure for partition block p will contain at most a single entry. According to Remark 3.44 this single entry has a fixed size of $\Theta(c+1) = \Theta(1)$. We thus can conclude that the local partition decision structure for partition block p can be maintained in internal memory; thereby not leading to any IO cost.

Case $c \neq 0$: According to Theorem 3.29 all bisimilar nodes are placed in the same local structural summary partition block p'. As such only the local partition decision structure for partition block p' will contain an entry for these nodes. For every node $n_\downarrow \in N_\downarrow$ there will thus be a single local partition decision structure wherein an entry for node $n_\downarrow$ is created. We assume the worst case wherein all these entries are placed in the same partition decision structure *pds*.

From $c \neq 0$ we can conclude that only for nodes $n_\downarrow \in N_\downarrow$ with $|E_\downarrow(n_\downarrow)| \neq 0$ there will be an entry in the partition decision structure *pds*. According to Remark 3.44 the entry for node $n_\downarrow$ has size $\Theta(|E_\downarrow(n_\downarrow)|)$. Therefore the total size for the partition decision structure is upper bounded by $O(\sum_{n_\downarrow \in N_\downarrow} |E_\downarrow(n_\downarrow)|) = O(|E_\downarrow|)$.

In the worst case the partition decision structure *pds* is queried $|N|$ times, once for every node in the graph. This results in a worst case upper bound on the IO cost for performing all queries on local partition decision structures of $O(\textsc{Scan}(|N||E_\downarrow|))$.

□

If one is truly concerned about worst case behavior; then picking the list as a data structure for implementing the local partition decision structures is not recommended. The usage of a hash table doesn't make much sense either; each local partition decision structure can be seen as a hash table bucket in a virtual hash table. The only remaining data structure mentioned in Subsection 3.3.1 is the string B-tree.

Proposition 3.48. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G, let *pds* be a partition decision structure implemented as a string B-tree. The partition decision structure *pds* can answer queries (lookups of existing entries and insertions of new entries) for node-decision values v_{pds} in $O(\textsc{Scan}(|v_{pds}|) + \log_B(|N_\downarrow|))$ IOs.

The worst case analysis for Algorithm 3.6 utilizing a string B-tree for implementing the partition decision structures differs from the worst case analysis provided in Theorem 3.47. As such we shall provide a separate worst case analysis for Algorithm 3.6 utilizing a string B-tree for implementing the partition decision structures.

Theorem 3.49. The worst case IO complexity of Algorithm 3.6 is $O(\textsc{Sort}(|N|) + \textsc{Sort}(|E|) + \text{PQ}(|E|) + |N| \log_B(|N_\downarrow|))$.

Proof. We assume the implementations described in Remark 3.43 while using a string B-tree implementation for the partition decision structures. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G, let p be a local structural summary partition block wherein every node has local structural summary $(c, v^L_{\mathcal{H}})$. We shall reconsider the case where $c \neq 0$; see Theorem 3.47 for the case $c = 0$.

CASE $c \neq 0$: Based on Theorem 3.47 we assume that the upper bound on the size of the partition decision structure is $|E_\downarrow|$. We also assume there is at most a single insert for every $n_\downarrow \in N_\downarrow$ and a single query for every node $n \in N$.

The upper bound on the total cost for all inserts is given by $\sum_{n_\downarrow \in N_\downarrow}(\textsc{Scan}(|v_{pds}|) + \log_B(|N_\downarrow|)) = O(\textsc{Scan}(|E_\downarrow|) + |N_\downarrow| \log_B(|N_\downarrow|))$. The upper bound on the total cost for all queries is given by $\sum_{n \in N}(\textsc{Scan}(|v_{pds}|) + \log_B(|N_\downarrow|)) = O(\textsc{Scan}(|E|) + |N| \log_B(|N_\downarrow|))$. This results in a worst case upper bound on the IO cost for performing all queries on local partition decision structures implemented as string B-trees of $O(\textsc{Scan}(|E|) + |N| \log_B(|N_\downarrow|))$.

□

For worst cases one can expect that the list implementation of the local partition decision structures is outperformed by the string B-tree implementation of the partition decision structures. According to Theorem 3.47 and Theorem 3.49 the provided implementations of Algorithm 3.6 are, in the worst case, not IO efficient. With decent hash functions one can however expect an IO efficient algorithm; as proven by Theorem 3.46.

3.5 Constructing maximum bisimulation graphs and graph indices

The result of running Algorithm 3.6 on a graph is a list of (partition block identifiers, node identifiers). Thereby the node identifiers do not correspond to the node identifiers used in the original graph; this due to the assignment of new node identifiers to nodes in Algorithm 3.4. This reassignment was necessary for constructing a structural summary partition. In this section we shall discuss how we can post-process the results of Algorithm 3.6 for constructing the maximum bisimulation graph and/or the graph index of the original input graph.

According to Theorem 3.13 the maximum bisimulation graph can be represented by a partition decision structure. Algorithm 3.6 can easily be adapted such that it maintains the entire partition decision structure. We can simply add a statement to the algorithm such that every new local partition decision structure entry is also appended to a global list *pds*. This global list *pds* represents the entire partition decision structure; and thereby list *pds* is a list representation of the maximum bisimulation graph. By construction Algorithm 3.6 places a specific order on the nodes in list *pds*.

Proposition 3.50. Let $G = \langle N, E, l \rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow \rangle$ be the maximum bisimulation graph of graph G in partition decision structure representation *pds* calculated by Algorithm 3.6, let $(l(n), S)$ be an entry in list *pds* representing a maximum bisimulation graph node. The maximum bisimulation graph nodes in list *pds* are lexicographically ordered on (rank, label, node-hash, $|S|$, $\mathcal{H}^L(l(n), S)$).

In several algorithms we have seen that every node needs access to the set of parents of the node; the time-forward processing depends on these sets. The partition decision structure representation of maximum bisimulation graphs does not provide these sets of parents of a node; instead every node has access to the set of children of the node. We can however trivially convert the partition decision structure representation of a maximum bisimulation graph into a list representation $\mathcal{L}$ of a maximum bisimulation graph. Algorithm 3.7 provides an outline for this conversion.

Algorithm 3.7 Converting maximum bisimulation graphs from *pds* representation to $\mathcal{L}$ representation

Require: Maximum bisimulation graph $G_{\downarrow} = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow} \rangle$ in *pds* representation.
Ensure: List L is the maximum bisimulation graph $G_{\downarrow} = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow} \rangle$ in list representation $\mathcal{L}$.

```
1:  i ← 0
2:  N′, E′, L are empty lists
3:  for all (l, S) ∈ G, in order of the list representation G do
4:      ADD(N′, (i, l))
5:      for all m ∈ S do
6:          ADD(E′, (i, m))
7:      i ← i + 1
8:  SORT(E′) on (child node identifier, parent node identifier)
9:  for all (i, l) ∈ N′, in order of the list N′ do
10:     ADD(L, (i, l, {m : (i, m) ∈ E′}))
```

Algorithm 3.7 has an IO complexity of $O(\text{SCAN}(|N_{\downarrow}|) + \text{SORT}(|E_{\downarrow}|))$. We shall not provide any correctness or IO complexity proofs for Algorithm 3.7 as these follow directly from the algorithm.

For constructing the graph index we probably need a bisimulation partition wherein the node identifiers correspond to the node identifiers used in the original graph. The bisimulation partition P produced by Algorithm 3.6 does not contain the original node identifiers; as these are replaced during the construction of the structural summary partition. We can however utilize the temporary list R proposed in Remark 3.39. List R provides a mapping between the node identifiers used in P and the node identifiers used in the original graph. After sorting lists R and P on the new node identifier one can sequentially read list R and at the same time sequentially update list P.

After updating the node identifiers in list P one can sort the list on partition block identifier. When the list P is sorted on partition block identifier; then it is easy to merge list P and the maximum bisimulation graph (either represented by a partition decision structure *pds* or represented in list representation $\mathcal{L}$) into a single representation of a graph index. The exact details of the index constriction are left open as these details depend completely on the future purpose of the graph index.

3.6 Final notes

In this chapter we have developed an IO efficient external memory bisimulation partitioning algorithm. We have also presented theory to support the development of this algorithm. In Subsection 3.6.1 we discuss the limitations on the developed algorithm. Subsection 3.6.2 concludes this chapter with useful details and considerations for implementing the algorithm.

3.6.1 Limitations on the external memory bisimulation partitioning algorithm

Algorithm 3.6; and all other related algorithms are restricted in two ways in the input they can accept. We have (1) the input should be a directed acyclic graph and (2) the directed acyclic graph should be reverse-topological sorted. Both restrictions are not easy to overcome.

First we take a look at the second restriction. To the best of our knowledge there are no truly IO efficient algorithms for topological sorting a directed acyclic graph. The (asymptotically) fastest algorithms use depth-first search; the worst case IO complexity for the best known depth-first search algorithm is $O((|N| + \text{SCAN}(|E|))\log(|N|))$ [BGVW00]. There are however a number of heuristic approaches that seem to perform acceptable [ACLZ11]. One can however expect that including topological sorting before running any of the bisimulation partitioning algorithms will greatly increase runtime and IO complexity.

This leaves us with the question if we can develop IO efficient bisimulation partitioning algorithms for directed graphs and/or for non-ordered directed acyclic graphs. Bisimulation partitioning approaches are likely to include some form of edge traversal; this stems directly from Definition 2.15. With dropping reverse-topological ordering we cannot make assumptions on the order of these edge traversals.

Breadth-first search and depth-first search are algorithms that primarily perform edge traversals; for these algorithms fast internal memory algorithms are known. There are however no IO efficient external memory algorithms for these problems; even though efficient 'edge traversal' internal memory solutions exist. This makes it very unlikely that an IO efficient bisimulation partitioning algorithm for general directed graphs and/or for non-ordered directed acyclic graphs is easily constructible.

Therefore one is more likely to develop acceptable performing general bisimulation partitioning algorithms by using heuristic approaches. Developing heuristic approaches is basically the same approach as currently investigated for other 'edge traversal' problems; such as topological sorting.

3.6.2 Implementing external memory bisimulation

Algorithm 3.6 is not optimized for any particular (expected) type of input and any particular type of output. When implementing an external memory bisimulation partitioning such as Algorithm 3.6 it is in generally a good idea to utilize the properties of the (expected) type of input and the type of output to improve (expected) performance. Therefore we shall give some brief ideas that can be used. Some of these ideas are applicable when calculating indices for XML documents; this specific case is handled in more depth in Chapter 5.

(1) The bisimulation partitioning algorithms are presented as if they are incrementally providing more details. One can however utilize another view; namely that the presented algorithms are increasingly less dependent on internal memory for storing the partition decision structure. This reduction on memory dependence is paid with an increase in IO complexity.

    When we implement the partition decision structure by using efficient internal memory string B-trees (see Subsection 3.3.1 for some details) then we get a hierarchy of algorithms. In this hierarchy Algorithm 3.3 is the fastest algorithm, but it can only handle graphs whose partition decision structure fits entirely in internal memory. This is the case for well-structured data. Algorithm 3.6 is the slowest algorithm; but it can handle all reverse-topological ordered directed acyclic graphs.

    When one expects input to be well-structured; and thus resulting in a small partition decision structure; then it is a good idea to pick Algorithm 3.3 over the more expensive combination of Algorithm 3.4 and Algorithm 3.6.

(2) Even when the partition decision structure does not fit entirely in internal memory we can work with the assumption that a local partition decision structure for an initial partition block fits in internal memory. With this assumption we can utilize Algorithm 3.5 over Algorithm 3.6; saving the work for locally refining initial partition blocks.

    If we cannot assume that every local partition decision structures for every initial partition block fits in internal memory, then we can still work with the assumption that most local partition decision structures will fit in internal memory. In these cases we don't need to locally refine the initial partition block. This assumption can be implemented by reserving a fixed amount of memory for a local partition decision structure. If this fixed amount of memory is used; then the nodes having node-decision values that are not present in the partition decision structure are stored in list N' for further local refinement. Thereby the total size of list N' and the number of nodes in the locally refined partition blocks is expected to be strongly reduced.

(3) For processing local structural summary partition blocks in Algorithm 3.6 (lines 10–18) we can primarily depend on an internal memory partition decision structure; whose size is bounded by some constant. We should only store elements in external memory when this internal memory partition decision structure is full.

(4) The choice of hash functions in Algorithm 3.4 and Algorithm 3.6 is important. When the used hash functions result in many collisions then the node-hash and local node-hash values are bad indicators for node bisimilarity. This increases the probability on worst-case behavior.

    Furthermore a hash function resulting in bad node-hash values will result in an initial partition containing very large partition blocks; reducing the probability that initial partition blocks can be

completely processed in internal memory (see 2). Likewise a hash function resulting in bad local node-hash values reduces the probability that local partition blocks can be processed completely in internal memory (see 3).

(5) When some (easily) computable node-value is a strong indicator for node bisimilarity; then this indicator can be included in the structural summary or in the local structural summary. This leads to the same improvements as seen in 4.

(6) Algorithm 3.4 is a very expensive algorithm; especially assigning new identifiers to every node and edge (lines 11–14); as described in Remark 3.39; will cost multiple sort and scan operations. We shall now look at an alternative approach.

Assume that Algorithm 3.4 results in initial partition P. Each partition block $p \in P$ will have a unique key as its identifier; namely the structural summary shared by all nodes in the partition block. Now assume we can maintain a mapping entry (unique key, partition block identifier, count) for every partition block. The total size of this mapping is upper bounded by $O(|N_{\downarrow}|)$.

We can utilize this mapping to assign composite identifiers (rank, partition block identifier, count) to every node n. For every node n being placed in some initial partition block p we look up the current count in the mapping. This count is assigned to node n as the unique node identifier of node n with respect to the other nodes in the same initial partition block. After assigning the count we increment it by one. This composite identifier can directly be assigned to the nodes in list N and to the child node identifiers stored in list E. When we use these composite identifiers; then only the node identifiers of parent nodes stored in list E needs to be updated.

For efficiency one would require that the mapping (unique key, partition block identifier, count) fits in main memory. For reducing the size of this mapping one can choose to sacrifice the quality of the resulting initial partition by restricting the initial partition to the rank partition or the rank, label partition. The exact savings for such an approach depends on the specifics of each case. Measurements can be of help to decide if the reduction of the cost for executing Algorithm 3.4 outweighs the increase in cost for Algorithm 3.6.

(7) Algorithm 3.4 can be skipped altogether if it is guaranteed that the nodes in the input graph are already ordered on rank. With this ordering on rank we already have an initial partition that allows us to use localized partition decision structures (see Theorem 3.29). With such a rank-ordered input graph we can simply utilize Algorithm 3.5 or Algorithm 3.6 directly on the provided input.

Locally refining an initial partition block as performed by Algorithm 3.6 costs less IOs then computing a structural summary as performed by Algorithm 3.4. The decrease in IO cost by removing Algorithm 3.4 thus more than compensates for the possible increase in IO cost for using a less optimal initial partition as input for Algorithm 3.6.

(8) When the fan out of nodes is bounded; then the node-decision values have a fixed size. In this case we can skip using local refinement and using partition decision structures altogether. In Algorithm 3.6 we simply sort list N' on the exact node-decision value. After sorting list N' one can process the nodes in list N' per node-decision value. Thereby each distinct node-decision value indicates a distinct bisimulation partition block. For this specific case the worst case upper bound on Algorithm 3.6 is $O(\textsc{Sort}(|N|) + \textsc{Sort}(|E|) + \text{PQ}(|E|))$.

Chapter 4

BISIMULATION PARTITION MAINTENANCE

In Chapter 3 we have introduced an algorithm for bisimulation partitioning directed acyclic graphs; this in an IO efficient way. We have also indicated how the introduced algorithm can be utilized to create the graph index of the input graph during its operations. We shall now look at how we can keep this graph index up to date when small changes are made to the input graph. This as an alternative to recalculating the entire index from scratch every time a small change is made to the input graph.

There has been some research on partition maintenance in internal memory. Work on exact updates on bisimulation partitions has been presented in [Sah07, DCXB11]; these approaches seem to have a worst case runtime complexity that is not lower then recalculating the entire index from scratch. In [KBNS02] algorithms are presented for maintaining the partition in an inexact way; thereby the updated partition is a refinement of the bisimulation partition. Such an inexact update will result in suboptimal results, but the resulting partition can still be used. In practical cases the results of these inexact updates are however expected to be close to optimal.

For the presented algorithms in this chapter we shall only consider exact updates to directed acyclic graphs stored in external memory. Thereby we keep the assumption that all nodes are reverse-topological sorted. The theory in this chapter is however applicable to cyclic and acyclic directed graphs. In Section 4.1 we shall present a native approach to partition maintenance during graph updates. This will give a useful upper bound; which will be used as a point of comparison for alternative algorithms. In Section 4.2 we introduce a complexity model for partition maintenance. In this complexity model we shall study lower bounds on the performance of all possible partition maintenance algorithms.

We shall take a look at practical approaches for performing partition maintenance in Section 4.3. The updates we look at are the addition of subgraphs and the addition of edges. We also briefly look at removal of subgraphs and removal of edges. These updates provide the basic operations wherein any graph-changing operation can be expressed. In Section 4.4 we make some final notes on partition maintenance.‘

4.1 Naive updating

Many operations can be considered to update graphs; we shall only focus on operations that add or remove subgraphs and edges. Any other update on graphs can easily be constructed as a sequence of these basic operations.

Definition 4.1. Let $G = \langle N, E, l\rangle$ be a directed graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle$ be the maximum bisimulation graph of graph G, let $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$ be the graph index of graph G, let P be the bisimulation partition of N, let $\mathcal{M}$ be an operation on graph G. The operation $\mathcal{M}$ is a graph update operation if and only if:

(1) The operation adds or removes a subgraph, or

(2) The operation adds or removes an edge between two nodes already part of graph G.

The update operation $\mathcal{M}$ applied on graph G results in the directed graph $G^{\mathcal{M}} = \langle N^{\mathcal{M}}, E^{\mathcal{M}}, l^{\mathcal{M}}\rangle$. The maximum bisimulation graph of graph $G^{\mathcal{M}}$ is given by $G^{\mathcal{M}} = \langle N^{\mathcal{M}}, E^{\mathcal{M}}, l^{\mathcal{M}}\rangle$, the graph index of graph $G^{\mathcal{M}}$ is given by $I^{\mathcal{M}} = \langle N^{\mathcal{M}}_\downarrow, E^{\mathcal{M}}_\downarrow, l^{\mathcal{M}}_\downarrow, p^{\mathcal{M}}\rangle$, and the bisimulation partition of $N^{\mathcal{M}}$ is given by $P^{\mathcal{M}}$.

A naive way to update an index is by computing it from scratch, for example by running Algorithm 3.6 on the updated graph. We shall refer to this approach as the naive updating.

Theorem 4.2. Let $G = \langle N, E, l \rangle$ be a directed graph, let $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$ be the graph index of graph G, let $\mathcal{M}$ be an update operation on graph G and $G^\mathcal{M} = \langle N^\mathcal{M}, E^\mathcal{M}, l^\mathcal{M} \rangle$ be the updated graph. Updating the graph index I using the naive approach has expected IO complexity of $O(\text{Sort}(|N^\mathcal{M}|) + \text{Sort}(|E^\mathcal{M}|) + \text{PQ}(|E^\mathcal{M}|))$.

Proof. Apply Algorithm 3.6 on graph $G^\mathcal{M}$. □

4.2 Maintenance complexity

Naive updating gives us an upper bound on the complexity of updating bisimulation partitions. We shall now establish lower bounds for updating. The actual lower bound for updating depends on the representations of the graph and the cost for maintaining any auxiliary data structures. For analyzing the lower bound we abstract from all these (implementation specific) details; therefore we introduce update complexity.

Definition 4.3. Let $G = \langle N, E, l \rangle$ be a directed graph with graph index $I = \langle N_\downarrow, I_\downarrow, l_\downarrow, p \rangle$, let $\mathcal{M}$ be an update operation, let $G^\mathcal{M} = \langle N^\mathcal{M}, E^\mathcal{M}, l^\mathcal{M} \rangle$ be the updated graph with graph index $I^\mathcal{M} = \langle N_\downarrow^\mathcal{M}, E_\downarrow^\mathcal{M}, l_\downarrow^\mathcal{M}, p^\mathcal{M} \rangle$. The update complexity for update $\mathcal{M}$ is given by the minimum number of changes needed to change the partition p into $p^\mathcal{M}$. Thereby we count the number of nodes that are added to or removed from partition blocks.

Analyzing updates on graphs in terms of update complexity gives theoretical lower bounds on all possible update algorithms. The update complexity does not take changes to the graph index into account. For purely updating the bisimulation partition this is acceptable. When index updates are studied; then we also need to include the cost of changes to the remainder of the index.

Definition 4.4. Let $G = \langle N, E, l \rangle$ be a directed graph with graph index $I = \langle N_\downarrow, I_\downarrow, l_\downarrow, p \rangle$, let $\mathcal{M}$ be an update operation, let $G^\mathcal{M} = \langle N^\mathcal{M}, E^\mathcal{M}, l^\mathcal{M} \rangle$ be the updated graph with graph index $I^\mathcal{M} = \langle N_\downarrow^\mathcal{M}, E_\downarrow^\mathcal{M}, l_\downarrow^\mathcal{M}, p^\mathcal{M} \rangle$. The index update complexity for update $\mathcal{M}$ is given by the minimum number of changes needed to change I into $I^\mathcal{M}$. Thereby we count the number of index nodes, index edges and partition blocks that are added to the graph index; the number of index nodes, index edges and partition blocks that are removed from the graph index; and the number of nodes that are added to or removed from partition blocks.

The main benefit of index update complexity over update complexity is that it provides better lower bounds for updating graph indices and maximum bisimulation graphs. It is reasonable to claim that the index update complexity provides a practical lower bound for any update algorithm. One can expect to need some auxiliary structure describing relations between bisimulation partitions when only these partitions need to be kept up to date. The graph index is an example of a structure describing relations between bisimulation partitions.

Therefore our main focus shall be on proving lower bounds for index update complexity. We can however easily derive the update complexity from the index update complexity as update complexity of an operation is included in the index update complexity of the operation.

Remark 4.5. Update complexity and index update complexity do not take the cost for performing the update operations on the underlying graph into account. This cost does not play any role in creating an up to date bisimulation partition or graph index. The cost of updating the graph itself can however place practical limitations on graph updates.

We shall only investigate the index update complexity for adding subgraphs and adding edges. The index update complexity of removing a subgraph is equivalent to the index update complexity for adding the same subgraph. This follows from the observation that removing a subgraph only undoes any changes made while adding the same subgraph. The same observation can be made for edge additions and edge removals.

4.2.1 Update complexity for subgraph additions

We shall first study the index update complexity for adding subgraphs. Thereby we first take a look at the best case index update complexity.

Theorem 4.6. Let $G = \langle N, E, l\rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$, let $G_s = \langle N_s, E_s, l_s\rangle$ be a graph, let $\mathcal{M}$ be the update operation adding graph G_s to graph G. In the best case the index update complexity for this update is $\Omega(|N_s|)$.

Proof. In the best case we can assume that for every subgraph node $n_s \in N_s$ there is a pre-existing index node $n_\downarrow \in N_\downarrow$ with $n_s \approx n_\downarrow$. In this case we only have to add every node n_s to the partition block $p(n_\downarrow)$. Thereby a total of $|N_s|$ updates have been made to the partition blocks in the graph index. □

The best case index update complexity only holds when the structure of the added subgraph is already present in the original graph. In the worst case we need to add this entire structure to the graph index.

Theorem 4.7. Let $G = \langle N, E, l\rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$, let $G_s = \langle N_s, E_s, l_s\rangle$ be a graph with maximum bisimulation graph $G_{s\downarrow} = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow}\rangle$, let $\mathcal{M}$ be the update operation adding subgraph G_s to graph G. In the worst case the index update complexity for this update is $O(|N_s| + |E_{s\downarrow}|)$.

Proof. Consider the maximum bisimulation graph node $n_{s\downarrow} \in N_{s\downarrow}$ such that no index node $n_\downarrow \in N_\downarrow$ exists with $n_{s\downarrow} \approx n_\downarrow$. There thus is no partition block wherein nodes $n_s \in N_s, n_s \approx n_{s\downarrow}$ can be placed. For this node $n_{s\downarrow}$ we need to create an index node with an accompanying partition block in the graph index. Thereby we also need to introduce a representation for every outgoing edge of $n_{s\downarrow}$.

After assuring that for every maximum bisimulation graph node $n_{s\downarrow} \in N_{s\downarrow}$ there is an index node $n_\downarrow \in N_\downarrow$ with $n_{s\downarrow} \approx n_\downarrow$ one can place all nodes N_s of subgraph G_s into the appropriate partition blocks in the graph index. Thereby a total of $|N_s| + 2|N_{s\downarrow}| + |E_{s\downarrow}|$ updates have been made to the graph index. □

Corollary 4.8. Let $G = \langle N, E, l\rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$, let $G_s = \langle N_s, E_s, l_s\rangle$ be a graph, let $\mathcal{M}$ be the update operation adding subgraph G_s to graph G. The update complexity for this update is $\Theta(|N_s|)$.

4.2.2 Update complexity for edge additions

We shall now study the index update complexity for adding edges. Thereby we shall first take a look at the best case index update complexity.

Theorem 4.9. Let $G = \langle N, E, l\rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$, let $n \in N, m \in N$ be nodes, let $\mathcal{M}$ be a graph update operation adding edge (n, m) to E. In the best case the index update complexity for this update is $\Omega(0)$.

Proof. Let $m' \in E(n)$ with $m' \approx m$. We have $v_\approx(n) = (l(n), S)$ with $v_\approx(m') \in S$. Adding edge (n, m) to the graph will not change this node-bisimilarity value as $v_\approx(m) = v_\approx(m')$. As such node n keeps the same node-bisimilarity value and according to Theorem 3.2 it stays bisimilar equivalent to the same nodes. Therefore no changes are needed to the graph index to keep the graph index up to date. □

Example 4.10. In this example we shall shown an example of an edge update on a graph that does not have any effect on the graph index. An example graph is shown in Figure 4.1a; together with the maximum bisimulation graph and the bisimulation partition. In Figure 4.1b the resulting graph after an edge addition is shown; also together with the maximum bisimulation graph and the bisimulation partition.

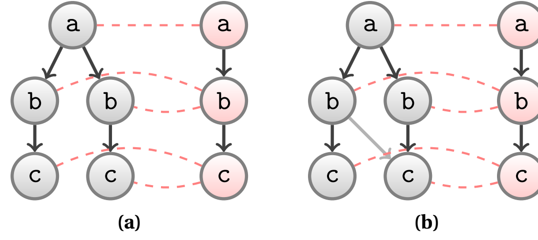


Figure 4.1: Two graphs are shown together with their maximum bisimulation graph (highlighted) and a relation relating nodes from the graph with bisimilar equivalent nodes in the maximum bisimulation graph. The newly added edge in graph Figure 4.1b is given a lighter gray color.

As one can see the edge addition does not have an effect on the bisimulation partition or the maximum bisimulation graph. The edge change shown in Figure 4.1 thus has a total index update complexity of 0.

An edge change can change the bisimulation partition block wherein nodes should be placed. The index update complexity for an edge change depends on the maximum number of nodes that need to be moved to other bisimulation partition blocks after the edge change. We shall first take a look at this maximum number.

Theorem 4.11. Let $G = \langle N, E, l \rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$, let $n \in N, m \in N$ be nodes, let $\mathcal{M}$ be a graph update operation adding edge (n, m) to E. Only the node-bisimilarity value for the ancestors $A(n)$ of node n can change by the edge addition.

Proof (sketch). When m is not bisimilar equivalent to any child node $m' \in E(n)$ then the node-bisimilarity value of node n is changed. When the node-bisimilarity value of node n is changed; then for every parent node $p \in E'(n)$ of node n the node-bisimilarity value can change. This can propagate up to all ancestors of node n. □

The results form Theorem 4.11 suggests that in the worst case almost all nodes in the graphs can be moved to other bisimulation partition blocks. Every node that is moved to another partition block can result in the creation or removal of partition blocks, graph index nodes and graph index edges. Thus in the worst case an edge change can have an effect on the entire graph index.

Theorem 4.12. Let $G = \langle N, E, l \rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$, let $n \in N, m \in N$ be nodes, let $\mathcal{M}$ be a graph update operation adding edge (n, m) to E. In the worst case the index update complexity for this update is $O(|N| + |E|)$.

Proof (sketch). According to Theorem 4.11 the node-bisimilarity value for the nodes $A(n)$ can change. This can cause all these nodes to be placed in their own partition block. For the graph index we need to introduce new index nodes related to the new partition blocks and index edges for connecting these new index nodes to the other index edges. The node n can have $O(|N|)$ ancestors, these ancestors nodes have at most $O(|E|)$ outgoing edges. □

Example 4.13. In this example we shall shown an example of an edge update on a graph that has a worst case effect on the graph index. Let $G = \langle N, E, l \rangle$ be the graph shown in Figure 4.2a. This graph is shown together with the maximum bisimulation graph and the bisimulation partition. In Figure 4.2b the resulting graph after an edge addition is shown; also together with the maximum bisimulation graph and the bisimulation partition.

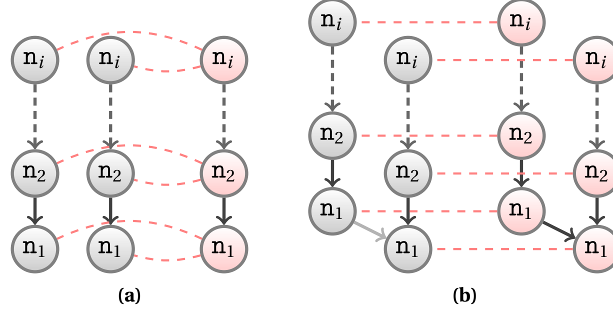


Figure 4.2: Two graphs are shown together with their maximum bisimulation graph (highlighted) and a relation relating nodes from the graph with bisimilar equivalent nodes in the maximum bisimulation graph. The newly added edge in graph Figure 4.2b is given a lighter gray color.

As one can see the edge addition doubles the size of the graph index and doubles the amount of partition blocks. Thereby half of the nodes in these partitions are moved to these new partition blocks. The edge change shown in Figure 4.2 thus has a total index update complexity of $O(|N|+|E|)$ and update complexity of $O(|N|)$.

We can easily provide worst cases examples for graphs with $|N|$ nodes and $O(|N|^2)$ edges. In the above example we can simply replace the chains by transitive closure chains, and the edge addition would still exhibit the worst case index update complexity.

Corollary 4.14. Let $G = \langle N, E, l\rangle$ be a directed graph with graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$, let $n \in N, m \in N$ be nodes, let $\mathcal{M}$ be a graph update operation adding edge (n, m) to E. The worst case update complexity for this update is $O(|N|)$.

4.3 External memory algorithms for maintenance

The naive approach already provides a general purpose algorithm for updating bisimulation partitions and graph indices. We shall now investigate alternative approaches; for these approaches we shall see if and when they outperform the naive approach. We thereby focus on updating graph indices; from the solutions for updating graph indices one can easily derive solutions for updating bisimulation partitions. In Subsection 4.3.1 we shall look at several algorithms for adding subgraphs to existing graph indices. Some considerations on subgraph removal are presented in Subsection 4.3.2. The last subsection, Subsection 4.3.3, shall primarily look at the challenges one faces when performing edge updates on graph indices.

4.3.1 Adding subgraphs

Assume we have a graph $G = \langle N, E, l\rangle$ and a graph $G_s = \langle N_s, E_s, l_s\rangle$. The graph G has graph index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$ represented by partition decision structure *pds* and bisimulation partition P. The graph G_s has graph index $I_{s\downarrow} = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow}, p\rangle$ represented by partition decision structure pds_s and bisimulation partition P_s. We are going to add graph G_s to graph G; resulting in graph $G^{\mathcal{M}} = \langle N^{\mathcal{M}}, E^{\mathcal{M}}, l^{\mathcal{M}}\rangle$. After this update we want to update the graph index I into the graph index $I^{\mathcal{M}} = \langle N_\downarrow^{\mathcal{M}}, E_\downarrow^{\mathcal{M}}, l_\downarrow^{\mathcal{M}}, p^{\mathcal{M}}\rangle$.

Without any loss of generality we can assume that the graph G is much larger than the graph G_s, we thus should avoid working on graph G directly. The graph index I of graph G can however be used to relate new nodes from graph G_s with existing nodes in graph G. We shall formalize this idea with the notion of maximum-merge graphs.

Definition 4.15. Let $G = \langle N, E, l\rangle, G_s = \langle N_s, E_s, l_s\rangle$ be directed graphs, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle, G_{s\downarrow} = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow}\rangle$ be the maximum bisimulation graphs for graph G and graph G_s. The graph $G' = \langle N_\downarrow \cup$

$N_{s\downarrow}, E_{\downarrow} \cup E_{s\downarrow}, l_{\downarrow} \cup l_{s\downarrow}\rangle$ is called the maximum-merge graph of graphs G and G_s. The bisimulation partition P' of graph G' is called the maximum-merge partition of graphs G and G_s. The partition blocks $p' \in P'$ are called maximum-merge partition blocks.

The maximum-merge graph G' of the graphs G and G_s is a graph containing two subgraphs; namely the maximum bisimulation graphs of graph G and graph G_s. Thereby the maximum-merge partition of graph G' relates bisimilar equivalent nodes from the indices of graph G and graph G_s. Before putting the maximum-merge partition to use we shall further analyze it.

Theorem 4.16. Let $G = \langle N, E, l\rangle, G_s = \langle N_s, E_s, l_s\rangle$ be directed graphs, let $G' = \langle N_{\downarrow} \cup N_{s\downarrow}, E_{\downarrow} \cup E_{s\downarrow}, l_{\downarrow} \cup l_{s\downarrow}\rangle$ be the maximum-merge graph of graphs G and G_s, let P' be the index-merge partition of graphs G and G_s. Every partition block $p' \in P'$ will contain at most two nodes; namely at most a single node $n_{\downarrow} \in N_{\downarrow}$ and at most a single node $n_{s\downarrow} \in N_{s\downarrow}$.

Proof. Assume we have a maximum-merge partition block $p' \in P'$ containing two distinct maximum bisimulation graph nodes n, m originating from the same maximum bisimulation graph. We have $n \approx m$, thus the size of the maximum bisimulation graph can be reduced by merging nodes n and m to a single maximum bisimulation graph node; leading to a contradiction. □

Example 4.17. Figure 4.3 shows the process of creating a maximum-merge graph. Figure 4.3a and Figure 4.3b show two different graphs together with their graph indices. These graph indices are taken and put in a separate graph; the maximum-merge graph. The resulting maximum-merge graph, together with its graph index; is shown in Figure 4.3c.

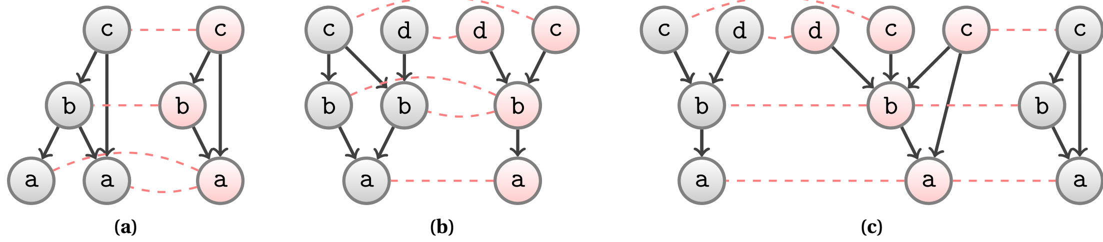


Figure 4.3: Three graphs are shown together with their maximum bisimulation graphs (highlighted) and a relation relating nodes from each graph with bisimilar equivalent nodes in their maximum bisimulation graph.

In Figure 4.3c the nodes in the same maximum-merge partition block are related to the same graph index node. One can easily see that each maximum-merge graph index node is related to at most two nodes. Furthermore each maximum-merge graph index node is only related to at most a single node from the original graph indices of the graphs from Figure 4.3a and Figure 4.3b.

A partition block in a valid partitioning cannot be empty; Theorem 4.16 thus allows three possible compositions for each maximum-merge partition block. We shall now look at an interpretation of these compositions.

Corollary 4.18. Let $G = \langle N, E, l\rangle, G_s = \langle N_s, E_s, l_s\rangle$ be directed graphs, let $I = \langle N_{\downarrow}, E_{\downarrow}, l_{\downarrow}, p\rangle, I_s = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow}, p_s\rangle$ be the graph indices for graph G and graph G_s, let $G' = \langle N_{\downarrow} \cup N_{s\downarrow}, E_{\downarrow} \cup E_{s\downarrow}, l_{\downarrow} \cup l_{s\downarrow}\rangle$ be the maximum-merge graph of graphs G and G_s, let P' be the maximum-merge partition of graphs G and G_s, let $p' \in P'$ be a maximum-merge partition block. The partition block p' can have the following three compositions:

CASE (1) $p' = \{n_{\downarrow}, n_{s\downarrow}\}$ FOR NODES $n_{\downarrow} \in N_{\downarrow}, n_{s\downarrow} \in N_{s\downarrow}$: This composition indicates that every node in partition block $p(n_{\downarrow})$ is bisimilar equivalent to every node in partition block $p_s(n_{s\downarrow})$. For updating we thus only need to merge the nodes in partition block $p_s(n_{s\downarrow})$ into partition block $p(n_{\downarrow})$.

CASE (2) $p' = \{n_\downarrow\}$ FOR A NODE $n_\downarrow \in N_\downarrow$: This composition indicates that there is no node in graph G_s that is bisimilar equivalent to the nodes in partition block $p(n_\downarrow)$. As such no updates are needed to keep the partition block $p(n_\downarrow)$ and the index node $n_\downarrow$ up to date.

CASE (3) $p' = \{n_{s\downarrow}\}$ FOR A NODE $n_{s\downarrow} \in N_{s\downarrow}$: This composition indicates that there is no node in graph G that is bisimilar equivalent to the nodes in partition block $p_s(n_{s\downarrow})$. For keeping the graph index I of graph G up to date we thus need to add a new index node representing partition block $p_s(n_{s\downarrow})$. Together with this new index node we need to add a representation for every edge $E_{s\downarrow}(n_{s\downarrow})$ to the graph index I.

Example 4.19. We have taken the graphs shown in Figure 4.3a and Figure 4.3b; and combined them to a single graph. We have combined these graphs with the graph index of the maximum-merge graph shown in Figure 4.3c.

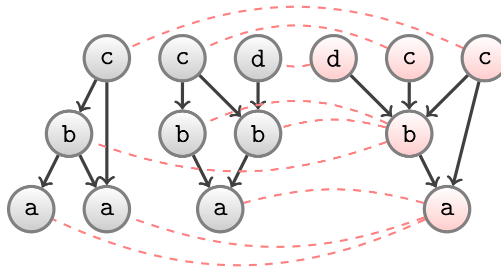


Figure 4.4: A graph consisting of two subgraphs is shown together with the maximum bisimulation graph (highlighted); also the relation relating index nodes with bisimilar equivalent graph nodes is shown.

As one can see the maximum-merge graph index is also a valid graph index on the combined graph of the two graphs shown in Figure 4.3a and Figure 4.3b. A closer look shows that we can use transitivity of the relation relating bisimilar nodes to connect the maximum-merge index with the nodes in the combined graph.

We can easily generalize the presented theory; including Corollary 4.18; to the case where we don't have an index on the graph G_s. One can replace the maximum bisimulation graph of graph G_s in the maximum-merge graph G' by graph G_s itself.

Case 1 from Corollary 4.18 can then be translated to a maximum-merge partition block $p' \in P'$ composed of a single node $n_\downarrow \in N_\downarrow$ and many nodes from N_s. Case 2 can be translated to a maximum-merge partition block $p' \in P'$ composed of only a single node $n_\downarrow \in N_\downarrow$. Case 3 can be translated to a maximum-merge partition block $p' \in P'$ composed of nodes only from N_s.

Example 4.20. We have taken the graph index from Figure 4.3a and the graph from Figure 4.3b; the graph index and the graph are combined in a single graph. The result is shown in Figure 4.5.

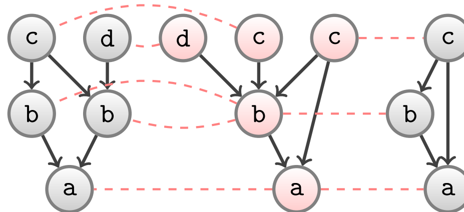


Figure 4.5: A graph consisting of two subgraphs is shown together with the maximum bisimulation graph (highlighted); also the relation relating index nodes with bisimilar equivalent graph nodes is shown.

As one can see the maximum-merge graph index stays the valid graph index on the combined graph. A closer look shows that at most one node from the graph index of Figure 4.3a is related to a single index node.

Corollary 4.18 already hints at a high level informal description on how to keep graph index I up to date when graphs are added to graph G. We shall further work out this high level description.

General outline

For the general outline we first describe the process of updating graph index I to graph index $I^{\mathcal{M}}$ by translating the features in pds_s and in P_s. Therefore we assume there is a way to calculate a partition block identifier update list L_U. The partition block identifier update lists consists of entries (old partition block identifier, new partition block identifier). Thereby the old partition block identifiers are partition block identifiers used in graph index I_s. The new partition block identifiers are the partition block identifiers used in graph index I.

We assume that the bisimulation partitions P and P_s are represented by lists of (node identifier, partition block identifier)-pairs as produced by Algorithm 3.6. For the partition decision structure *pds* we assume that it is either implemented as a list or as string B-trees. For the partition decision structure pds_s we assume that it is implemented as a list. After describing the general approach for subgraph addition we shall provide two approaches to calculating this partition block identifier update list.

STEP 1: Calculate a partition block identifier update list L_U.

STEP 2: Translate the bisimulation partition P_s and add P_s to bisimulation partition P.

Recall that the bisimulation partitions P and P_s are represented by lists consisting of (node identifier, partition block identifiers) entries. Before we append list P_s to list P we need to update the node identifiers and the partition block identifiers. For updating the partition block identifiers we can utilized the partition block identifier update list L_U. Sort L_U on old partition block identifier and P_s on partition block identifier; then sequentially read the sorted list L_U and sequentially update list P_s.

For updating the node identifiers in P_s we need to establish how node identifiers are represented in graph G. A simple approach would be to increment every node identifier in P_s by a constant c, whereby c is larger than any node identifier used for nodes already in graph G. This approach guarantees unique node identifiers, but it introduces additional work on updating the edges E_s. Another possibility would be to create a graph identifier for graph G_s and giving every node from graph G_s a composite node identifier (graph identifier, node identifier). Thereby the graph G_s stays easily recognizable as a subgraph in graph G and no changes to the edges E_s are needed. When the second approach is used for updating the node identifiers, then the IO complexity of this step is $O(\text{SORT}(|N_s|)+\text{SORT}(|N_{s\downarrow})|)$.

STEP 3: Translate the partition decision structure pds_s and add new entries to partition decision structure *pds*.

We can turn partition decision structure pds_s into a list of maximum bisimulation graph nodes N' and a list of maximum bisimulation graph edges E' by utilizing the first part of Algorithm 3.7. Note that all node identifier used in N' and E' are partition block identifiers valid only in pds_s. These partition block identifiers need to be translated to the partition block identifiers used in *pds*. After this translation we reconstruct the entries from N'; entries not yet placed in *pds* are appended to list *pds*.

For updating the partition block identifiers we can utilized the partition block identifier update list L_U. Lists N' and E' can be updated by first sorting them on the field that needs to be updated, and then sequentially reading list L_U and at the same time sequentially updating the field that needs to be updated.

The IO complexity for translating pds_s is at most $O(\text{SORT}(|N_{s\downarrow}|)+\text{SORT}(|E_{s\downarrow}|))$. The IO complexity for adding the translated pds_s to a list implementation of *pds* is at most $O(\text{SCAN}(|N_{s\downarrow}|)+\text{SCAN}(|E_{s\downarrow}|))$. The IO complexity for adding the translated pds_s to a string B-tree implementation of *pds* is at most $O(\text{SCAN}(|E_{s\downarrow}|)+|N_{s\downarrow}|\log_B(N_{\downarrow}^{\mathcal{M}}))$.

Maximum-merge partition block identifier update list

The first approach to calculate the partition block identifier update list is by calculating the maximum-merge partition and using this partition as a template to construct an update list.

STEP A: Create the maximum-merge graph for graphs G and G_s.

Take the partition decision structure pds_s and increment all partition block identifiers used in pds_s by c, whereby c is a constant larger than the maximum partition block identifier used in *pds*. We can utilize Algorithm 3.7 to construct the maximum-merge graph in $\mathcal{L}$ representation from the constructed partition decision structure. The total IO complexity for this step is $O(\text{SORT}(|N_\downarrow| + |N_{s\downarrow}|) + \text{SORT}(|E_\downarrow| + |E_{s\downarrow}|))$ when the partition decision structures are implemented as lists.

STEP B: Calculate the maximum-merge partition.

Perform Algorithm 3.6. The total expected IO complexity for this step is $O(\text{SORT}(|N_\downarrow| + |N_{s\downarrow}|) + \text{SORT}(|E_\downarrow| + |E_{s\downarrow}|) + \text{PQ}(|E_\downarrow| + |E_{s\downarrow}|))$.

STEP C: Translate maximum-merge partition into partition block identifier update list.

According to Corollary 4.18 the maximum-merge partition has three types of partition blocks. For case (1) partition blocks an entry (partition block identifier of $n_\downarrow$, partition block identifier of $n_{s\downarrow}$) is added to the partition block identifier update list. Case (2) partition blocks do not introduce changes to the graph index, and thus can be ignored. For Case (3) partition blocks an entry (fresh partition block identifier, partition block identifier of $n_{s\downarrow}$) is added to the partition block identifier update list. Thereby the fresh partition block identifier is a fresh partition block identifier not yet used in *pds*. When one wants to maintain the relation between partition block identifier and position in the partition decision structure; then one should pick incremental values starting with $|N_\downarrow|$. The IO complexity for this step is $O(\text{SCAN}(|N_\downarrow| + |N_{s\downarrow}|))$.

Partition decision structure partition block identifier update list

The second approach to calculate the partition block identifier update list is by external memory bisimulation partitioning of the graph index I_s while using the partition decision structure *pds*. For supporting efficient queries on *pds* we assume that the partition decision structure *pds* is implemented by a string B-tree.

STEP A: Perform online bisimulation partitioning on the graph index I_s as performed in Algorithm 3.3. Thereby use a copy of the partition decision structure *pds* as the partition decision structure. This results in a bisimulation partition P'_s of the nodes $N_{s\downarrow}$.

The resulting bisimulation partition P'_s is a partition block identifier update list. The total cost of this step is $O(\text{SCAN}(|N_{s\downarrow}|) + \text{SCAN}(|E_{s\downarrow}|) + \text{PQ}(|E_{s\downarrow}|) + |N_{s\downarrow}| \log_B(|N_\downarrow^{\mathcal{M}}|))$. Note that by using the partition decision structure *pds* directly we have incorporated step 3 from the outline into this step.

Complexity for subgraph addition

The total IO complexity for graph addition using maximum-merge partition block identifier update lists is $O(\text{SORT}(|N_s|) + \text{SORT}(|N_\downarrow| + |N_{s\downarrow}|) + \text{SORT}(|E_\downarrow| + |E_{s\downarrow}|) + \text{PQ}(|E_\downarrow| + |E_{s\downarrow}|))$. This makes this approach especially useful when graphs have small indices; this is the case when graphs are well structured.

The total IO complexity for graph addition using partition decision structure partition block identifier update list is $O(\text{SORT}(|N_s|) + \text{SORT}(|E_{s\downarrow}|) + \text{PQ}(|E_{s\downarrow}|) + |N_{s\downarrow}| \log_B(|N_\downarrow^{\mathcal{M}}|))$. This makes this approach faster when the graph G_s or its index I_s are very small; even when the graph index I is very large. This makes this approach especially useful when individual graphs are small or have small indices; even when the resulting graph G has a complicated and large index.

4.3.2 Removing subgraphs

Assume we have a graph $G = \langle N, E, l \rangle$ and a subgraph $G_s = \langle N_s, E_s, l_s \rangle$ whereby $N_s \subseteq N$, $E_s \subseteq E$, and $l_s \subseteq l$. The graph G has index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p \rangle$, the subgraph G_s has maximum bisimulation graph $G_{s\downarrow} = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow} \rangle$. We are going to remove the subgraph G_s from graph G resulting in graph $G^{\mathcal{M}}$. After removal of subgraph G_s we want to update the graph index I to the graph index $I^{\mathcal{M}}$ of graph $G^{\mathcal{M}}$. We shall present a general approach for performing this update.

STEP 1: Remove every subgraph node $n_s \in N_s$ from its partition block; remove empty partition blocks.

For analyzing this step we need to make an assumption on the implementation of the partition blocks. Let us start with the assumption that partition blocks are stored in a bisimulation partition P represented by a list of (node identifier, partition block identifier)-pairs. This representation is the output produced by Algorithm 3.6. Thereby we assume that this bisimulation partition is ordered on node identifier; which is the case for the output of Algorithm 3.6.

When the set of nodes N_s is ordered on node identifier; then an easy approach would be to read the set N_s sequentially and at the same time sequentially remove matching nodes from list P. For supporting removals in the list we need to use an external memory version of a linked list data structure. Thereby this step has total IO complexity of $O(\text{SORT}(|N_s|) + \text{SCAN}(|N|))$. An alternative approach would be to use binary search on the list to find each element from set N_s in list P. This would result in an IO complexity of $O(|N_s| \log_B(|N|))$. Both approaches will be very costly when graph G is large.

We can restrict the removal of subgraphs; only allowing the removal of previously added graphs. This restriction allows for more efficient approaches. Let us assume that in this restricted setting each node identifier is a composite identifier (graph identifier, node identifier); see also Step 2 in the general outline described for subgraph addition in Subsection 4.3.1. Now we only have to remove the nodes with the same graph identifier. When we maintain the order (node identifier, partition block) on the bisimulation partition, then we can remove all nodes by finding the first node with the specified graph identifier and then sequentially remove all nodes with the specified graph identifier. Thereby the IO complexity is $O(\text{SCAN}(|N_s|) + \log_B(|N|))$.

Note however that a bisimulation partition ordered on (node identifier, partition block identifier) does not allow fast lookups for all nodes placed in a single partition block. These lookups can be sped up by ordering the bisimulation partition on (partition block identifier, composite node identifier). In this setting we can use a separate mapping (graph identifier, partition block identifiers) for facilitating a fast removal of the nodes N_s from their partition block. This mapping (graph identifier, partition block identifiers) relates a subgraph with all partition blocks wherein nodes from this subgraph are placed.

In this setting we can remove all nodes $n_s \in N_s$ by utilizing the mapping (graph identifier, partition block identifiers). For every partition block in the mapping we utilize binary search to look up the first entry with the specified graph identifier. We can then sequentially remove all nodes in the partition block with the specified graph identifier The total IO complexity for this approach is $O(\text{SCAN}(|N_s|) + |N_{s\downarrow}| \log_B(|N|))$.

For supporting the last two approaches we also need to change the way wherein graphs are added to a graph. Thereby both the ordering in the bisimulation partition must be maintained and (optionally) a mapping (graph identifier, partition block identifiers) must be maintained. For implementation of an ordered bisimulation partition we can use specialized data structures such as the B^+ tree. Thereby the cost for inserting a set of nodes during graph addition is basically the same as the cost for removing these nodes.

Partition blocks can become empty during the process of node removal from partition blocks. These partition blocks are no longer present in the implementations described above; for other implementations one should make sure to remove any representation of empty partition blocks.

STEP 2: Remove index nodes and index edges related to removed partition blocks.

If a partition block $p(n_\downarrow)$ becomes empty during the execution of the previous step then the index node $n_\downarrow$ related to this partition block needs to be removed from the graph index. Also the incoming and outgoing index edges for the index node $n_\downarrow$ need to be removed from the index edge. If the graph

index is represented by a partition decision structure; then this can be achieved by removing entry i for every empty partition block with partition block identifier i.

Theorem 4.21. Let $G = \langle N, E, l\rangle$ be a graph, let $G_\downarrow = \langle N_\downarrow, E_\downarrow, l_\downarrow\rangle$ be the maximum bisimulation graph of graph G, let $G_s = \langle N_s, E_s, l_s\rangle$ be a subgraph of graph G, let $G_{s\downarrow} = \langle N_{s\downarrow}, E_{s\downarrow}, l_{s\downarrow}\rangle$ be the maximum bisimulation graph of graph $G_{s\downarrow}$. We remove subgraph G_s from graph G resulting in graph $G^{\mathcal{M}}$ with maximum bisimulation graph $G_\downarrow^{\mathcal{M}}$. If we have node $n_\downarrow \in N_\downarrow, n \notin N_\downarrow^{\mathcal{M}}$, then for any parent node $m_\downarrow \in E'_\downarrow(n_\downarrow)$ we have $m_\downarrow \notin N_\downarrow^{\mathcal{M}}$.

Proof. Assume we have $m_\downarrow \in E'_\downarrow(n_\downarrow), m_\downarrow \in N_\downarrow^{\mathcal{M}}$. This can only happen when node $m_\downarrow$ is bisimulated by a node $m \in N, m \notin N_s$. There must be a child node $n \in N^{\mathcal{M}}$ of node m; else node $m_\downarrow$ cannot have node $n_\downarrow$ as a child. There thus is a child node $n \in N, n \notin N_s$ of node m with $n \approx n_\downarrow$. This contradicts that node $n_\downarrow$ is only bisimulated by nodes in N_s. □

In a partition decision structure each entry stores a node together with all its outgoing edges. Theorem 4.21 thus proofs that removing entries from the partition decision structure will also remove all index edges that need to be removed.

Example 4.22. In Figure 4.6 we show the graph and graph index from Figure 4.4; now after the removal of an entire subgraph from the graph. As one sees the graph index now contains a node not longer connected to any graph node. This index node, together with its outgoing edges, should be removed during index maintenance.

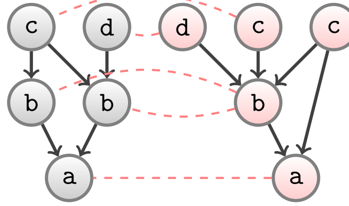


Figure 4.6: A graph is shown together with the maximum bisimulation graphs (highlighted) and a relation relating nodes from the graph with bisimilar equivalent nodes in the maximum bisimulation graph.

Removal of entries from the partition decision structure invalidates the relation between the partition block identifier of partition block $p(n_\downarrow)$ and the position of the entry for node $n_\downarrow$ in the partition decision structure. This can be resolved by utilizing a (partition block identifier, partition decision structure entry) mapping to represent the partition decision structure. The B^+ tree is a data structure that can be used for implementing such a mapping. Note that in the worst case at most $|N_{s\downarrow}|$ entries are removed from the partition decision structure. These entries have a total size of $O(|N_{s\downarrow}| + |E_{s\downarrow}|)$. The IO complexity in a setting whereby a B^+ tree is used to find and remove entries is at most $O(\text{SCAN}(|E_{s\downarrow}|) + |N_{s\downarrow}| \log_B(|N_\downarrow|))$.

Total IO complexity for removing a subgraph is largely dependent on the exact representation of the graph index. From the complexities described in the two steps one can easily derive the cost for any described presentation.

4.3.3 Edge updates

Assume we have a graph $G = \langle N, E, l\rangle$ and nodes $n \in N, m \in N$. The graph G has index $I = \langle N_\downarrow, E_\downarrow, l_\downarrow, p\rangle$. We are going to add or remove the edge (n, m) from graph G resulting in graph $G^{\mathcal{M}}$. After this update we want to update the graph index I to the graph index $I^{\mathcal{M}}$ of graph $G^{\mathcal{M}}$. We shall first analyze the practical complexity of this problem, and then we shall analyze a propagation based approach for edge addition.

Practical complexity of edge changes

From Theorem 4.12 we can conclude that in the worst case the entire graph index will be restructured after an edge change. As such it is safe to assume that no approach will be asymptotically faster than recalculating the entire index with a fast bisimulation partitioning algorithm. The naive approach is thus already the best solution for general cases. But even this naive approach is hard to implement.

For external memory graph algorithms the existence of an ordering on the graph nodes is crucial to guarantee good performance and/or low memory overhead. Edge additions can break the existing reverse-topological ordering on the nodes, thereby making it impossible to run the naive approach. As such the first problem to solve is an efficient reestablishment of the reverse-topological order.

Assumption 4.23. Let $G = \langle N, E, l\rangle$ be a graph. Function $\text{ID} : N \to \mathbb{N}$ gives the unique node identifier of a graph node. Let $n_i \in N$ be the node with $\text{ID}(n_i) = i$, let $n_j \in N$ be the node with $\text{ID}(n_j) = j$. We assume $i < j$ if and only if node n_i is ordered before n_j in the reverse-topological order used in graph G.

Assume edge (n, m) is added to graph G. Two possible cases are possible, namely $\text{ID}(n) < \text{ID}(m)$ and $\text{ID}(n) > \text{ID}(m)$. In the case $\text{ID}(n) < \text{ID}(m)$ the reverse-topological order used in graph G is invalidated. We thus need to assign a new node identifier to node n; such that $\text{ID}(n) > \text{ID}(m)$ holds. Along the same line as Theorem 4.11 one can prove that in the worst case an update to the node identifier of node n can propagate to all ancestors $A(n)$ of node n. Consider for example the case wherein for every ancestor node $n' \in A(n)$ of node n we have $\text{ID}(n') < \text{ID}(m)$.

Based on this analysis we can also conclude that the naive approach performs to much work, we only need to update the nodes $A(n)$. A more optimized approach would only update the nodes $A(n)$. Such an optimized approach can be much faster than the naive approach; especially when the set $A(n)$ is relatively small. We shall now present a simple approach for edge additions; this approach can easily be adapted to cover edge removals.

Edge update propagation

The addition of edge (n, m) can break the overall reverse-topological order. We however see that the reverse-topological ordering between nodes remains valid for the nodes in set $A(n)$. Also the reverse-topological order on the nodes in $N - A(n)$ remains valid. We can thus still perform time-forward processing on the set of nodes $A(n)$.

Let Q_{ID} and Q be priority queues. The priority queue Q_{ID} is used to send node identifiers of children to parents. If a node n receives a node identifier i with $i > \text{ID}(n)$ then the node identifier of node n needs to be updated. The initial value for priority queue Q_{ID} is a message $\text{ID}(m)$ send to node n. The priority queue Q is used to send (old partition block identifier, new partition block identifier) messages from children to parents. Thereby a message indicates that a child node is moved from the partition block with the old partition block identifier to the partition block with the new partition block identifier. The initial value for priority queue Q is a message ($-$, partition block identifier of node m) send to node n; whereby $-$ indicates that a new partition block identifier should be included in the calculation of the node-decision value of node n.

We shall also maintain lists L_{ID} and L_E, list L_{ID} contains (old node identifier, new node identifier)-pairs and list L_E contains the outgoing edges of every node $A(n)$. These lists are used to update the outgoing edges of nodes $A(n)$ after updating the node identifiers of nodes $A(n)$. In our analysis we use the notation $|E(S)| = \sum_{n \in S} |E(s)|$ and $|E'(S)| = \sum_{n \in S} |E'(s)|$ to indicate the total number of outgoing edges and the total number of incoming edges for a set of nodes S.

We shall now look at the steps taken to process a single node $m \in A(n)$; thereby we assume that the nodes $A(n)$ are processed in reverse-topological order. The cost of accessing individual nodes $m \in A(n)$ is analyzed after we have presented and analyzed the steps for processing the node $m \in A(n)$.

STEP 1: Give node m a valid node identifier; send the new node identifier to the parents of m.

Read all the changed node identifiers for children of node m from the priority queue Q_{ID}. If a node identifier i is send to node m with $i > \text{ID}(m)$ then an unused node identifier j larger than any node identifier send to m is picked and assigned to node m. When a new node identifier is assigned to node

m, then this new node identifier is send to the parents of node m. The total IO complexity for this step is $O(\text{PQ}(|E'(A(n))|))$.

STEP 2: Add incoming edges of node m to list L_E, add (old node identifier m, new node identifier m) to list L_{ID} when the node identifier of node m changed.

The total cost for this step is $O(\text{SCAN}(|E'(A(n))|)+\text{SCAN}(|A(n)|))$.

STEP 3: Recalculate the node-decision value for node m.

This step can be implemented efficiently if every node has access to the partition block identifiers of children. This can be implemented by storing a list L_m that contains, for every node m, all the partition block identifiers it received during bisimulation partitioning (all the values send to node m during Algorithm 3.6). Thereby we assume that this list L_m is strictly ordered, as described in Assumption 3.19 Algorithm 3.6 can guarantee a strict ordering on list L_m.

We can update list L_m by sequentially reading the list and updating the entry found at the top of queue Q. The ordering on queue Q guarantees that we can update all entries in a single sequential read. At the same time we can recalculate the node-decision value of node m. Thereby the total IO complexity for this step is $O(\text{SCAN}(|E(A(n))|)+\text{PQ}(|E'(A(n))|))$.

STEP 4: Assign node m to the partition block belonging to the recalculated node-decision value of m.

For this step we utilize a partition decision structure that can be queried efficiently; we shall assume we have a string B-tree implementation of the partition decision structure. When the partition block whereto node m is assigned changes; then we need to remove node m from the old partition block and we need to send a message (old partition block, new partition block) to every parent of node m using queue Q.

For removing nodes from a partition block we refer to the first step performed for subgraph removal. The total IO complexity for this step will be $O(\text{SCAN}(|E(A(n))|)+|A(n)|\log_B(|N|)+\text{PQ}(|E'(A(n))|))$.

STEP 5 (AFTER PROCESSING ALL NODES m): Update the incoming edges of every node $m \in A(n)$.

Sort list L_{ID} on old node identifier and use this list to update the edges in L_E. Do so by sequential reading list L_{ID} and sequential updating list L_E. After updating the edge list L_E the edges of each node $m \in A(n)$ can be reassigned. The IO complexity for this step is $O(\text{SORT}(|A(n)|)+\text{SORT}(|E'(A(n))|)$ when we exclude the cost for finding the incoming edge list of every node m.

Complexity of edge addition propagation

The total IO complexity for edge addition propagation is given by the cost for performing each step for each node. There is however also a cost related to finding each node $m \in A(n)$, the list of incoming edges $E'(m)$ and the list L_m containing the partition block identifiers of children of the node m. Therefore we need a B^+ tree or a similar search structure. Including the cost for finding nodes in this search structure results in the total IO complexity of $O(|A(n)|\log(|N|)+\text{SORT}(|A(n)|)+\text{SORT}(|E'(A(n))|)+\text{SCAN}(|E(A(n))|)+\text{PQ}(|E'(A(n))|))$ for edge addition propagation.

The update propagation approach for edge addition can easily be adapted for edge removal. Thereby all work performed to maintain a reverse-topological ordering is unnecessary as the topological order cannot break by removing an edge.

4.4 Final notes

In this chapter we have investigated partition maintenance. Thereby we have provided upper bounds on the cost of partition maintenance by presenting a naive method for partition maintenance after graph updates. We have also provided lower bounds on the cost of any partition maintenance approach. On top of these results we have presented approaches for performing partition maintenance

after subgraph addition, subgraph removal, edge addition and edge removal. We shall conclude with a brief investigation on the limitations of partition maintenance.

In our analysis on the lower bound cost for partition maintenance we have only looked at the amount of changes made to the index. Our analysis does not consider the consequences of these changes on the runtime or IO complexity of any practical algorithms. This does not pose big problems for edge updates. We have seen that for edge updates a worst case update will affect the entire graph. As such one cannot expect to find a method to perform general edge updates that is always faster than the naive approach of calculating a new bisimulation partition for the updated graph.

For subgraph updates we have seen that the lower bound cost only depends on the size of the subgraph. We have only presented approaches that depend on the size of the entire graph (or are not IO efficient). We thus might question if there are any subgraph update approaches that are IO efficient and whose complexity only depends on the size of the subgraph. We can however present an informal argument why such an efficient approach is not expected to exist.

Thereby we shall first take a look at subgraph removal. Let $G_s = \langle N_s, E_s, l_s \rangle$ be a subgraph that is removed from a graph G. Thereby we assume we have $|N_\downarrow| \geq B|N_{s\downarrow}|$; whereby B is the block size. In this case it is easy to image that each index node $n_{s\downarrow} \in N_{s\downarrow}$ and/or each partition block related to index node $n_{s\downarrow}$ is placed in its own disk block. Therefore we need at least $|N_{s\downarrow}|$ IOs to update these index nodes and/or partition blocks during subgraph removal. In the worst case we have $|N_{s\downarrow}| = |N_s|$. As such it is not expected that a general subgraph removal algorithm would be IO efficient and only dependent on the size of the subgraph for any form of input.

We shall now look at subgraph addition. When the result of partition maintenance after subgraph addition is some structured bisimulation partition (for example to allow subgraph removal); then we can rerun the argument for subgraph removal. To maintain a structured bisimulation partition we need to access and update individual (possibly pre-existing) partition blocks, according to the argument for subgraph removal we can't force these accesses to be IO efficient. The same reasoning also rules out the existence of a fully IO efficient version of a subgraph addition algorithm that uses a lookup structure (such as a partition decision structure).

We can also rule out any variant of the maximum-merge approach that uses the graph index of the graph; as these approaches all depend on the size of the remainder of the graph (or graph index). What is left is the possible existence of an IO efficient subgraph addition algorithm that does not perform lookups, does not use the graph index, and does not provide any structured way of storing the bisimulation partition.

Let assume we have such an IO efficient subgraph addition algorithm. This algorithm must be able to relate nodes from the subgraph with bisimilar equivalent nodes in another graph in an IO efficient way. We have however prohibited the algorithm to have any knowledge of this other graph. Such an IO efficient subgraph addition algorithm thus can be utilized for easy bisimulation partitioning. The subgraph addition algorithm can easily relate every node from a subgraph with bisimilar equivalent nodes in the maximum bisimulation graph of the subgraph. We don't even need to construct this maximum bisimulation graph; as the subgraph addition algorithm cannot have knowledge of this maximum bisimulation graph. As such the subgraph addition algorithm would provide an easy way to perform bisimulation partitioning. Therefore the existence of such a general IO efficient subgraph addition algorithm is not very likely.

A last note on the approaches on partition maintenance after graph updates is in place. None of the approaches we presented could generally outperform the naive approach of recalculating the entire bisimulation partition of a graph. In this section we have even argued that more efficient approaches are not very likely to exist. Furthermore the approaches we presented did make assumptions on the data structures used to represent the maximum bisimulation graph and bisimulation partition. For practical applications other data structures can be needed to represent the maximum bisimulation graph and bisimulation partition. In these cases further attention is needed for updating these data structures; this can further complicate partition maintenance.

Chapter 5

INDEXING XML DOCUMENTS

With the rise of the World Wide Web hierarchical tree-like data formats have taken a major role for representing and exchanging information. Well known examples of these hierarchical tree-like data formats are HTML[1] and XML[2]. Together with the development of these data formats there is a considerable development of query languages for extracting information from data stored in XML or HTML documents.

Query languages such as XQuery[3] rely heavily on path queries. A path query on an XML document only retrieves those nodes in a tree representation of the XML document that are part of a specified path. XQuery depends on the path query language XPath[4]. Also other XML-related technologies for processing XML documents utilize XPath, one example is the family of document transformation techniques XSL[5]. Path queries thus have a central role in XML related technologies. Therefore several index types have been developed to speed up these path queries.

We have already seen one of these index types; namely the 1-index [MS99]. In Section 1.1 we have shown how the 1-index can be utilized to answer simple path queries on XML documents. The 1-index utilizes backward node bisimulation to group nodes with equivalent incoming paths. As such the 1-index does not look at outgoing paths. Thereby the 1-index cannot be utilized to answer all path queries. A more general index is the F&B-index [ABS00]; this index looks at incoming and outgoing paths. Thereby the F&B-index can cover all twig queries. A twig query matches only those nodes whose relations to other nodes can be described by a twig (pattern represented by a tree). The F&B-index not only covers twig queries, it is also the smallest index to cover these twig queries [KBNK02].

The 1-index and F&B-index both group nodes based on their relations with the remainder of the document. Thereby these indices can be utilized to answer many different types of queries; the drawback is that these indices can be very large. Therefore a large group of indices have been developed that only look at paths of limited length. A general representative of this group of limited path length indices is the A(k)-index [KSBG02]; the A(k)-index groups nodes with equivalent incoming paths of length at most k. For an overview of many other indices used on XML documents we refer to [GC07].

In this chapter we shall focus on the 1-index, the F&B-index and the A(k)-index; thereby we have picked three general representatives of the many bisimulation-based indices used for indexing XML documents. In Section 5.1 we introduce the XML data format, the three index types and related theory. Section 5.2 presents efficient index construction algorithms for XML documents. We then take a look at how the format of XML documents can be utilized for updating indexed XML documents; the resulting approaches are presented in Section 5.3. We conclude our findings in Section 5.4 by discussing some limitations on the investigated approaches.

5.1 Preliminaries

In this section we introduce terminology and theory used in the remainder of the chapter. We shall first take a look at the format of XML documents; and how this format relates to trees. We then introduce

[1] See `http://www.w3c.org/HTML/`.
[2] See `http://www.w3c.org/XML/`.
[3] See `http://www.w3c.org/XQuery/`.
[4] See `http://www.w3c.org/XPath/`.
[5] See `http://www.w3c.org/XSL/`.

the bisimulation notions used by the 1-index, F&B-index and the A(k)-index.

5.1.1 The Extensible Markup Language

We shall utilize the terminology introduced by the W3C[6] whenever possible. We provide a brief introduction to the format of XML documents; for more in-depth information on the format of XML documents and on XML-related technologies we refer to the resources mentioned in the footnotes.

XML documents can be used to store and exchange data. XML documents are often represented by trees. In such an XML tree the tree nodes are representations of XML elements and attributes. In XML documents an XML element starts with a start-tag and ends with an end-tag. The start- and end-tag contain the name of the XML element; this name serves as the label of the tree nodes represented by XML elements. Start-tags can have attributes; each attribute consists of a name and a value. Generally attributes are represented by a distinct type of nodes in the tree representation of the XML document.

The XML data format allows variants on this idea; nodes without any content nested in them can be represented by an empty element. For empty elements the start- and end-tag is combined in a single empty-element tag. Within the start- and end-tag for an XML element child elements and textual content is nested. In the examples and algorithms in this chapter we shall assume that only XML elements and attributes are included in the indexed data tree. One can however easily generalize the examples, theory and algorithms presented in this chapter to include all features of the XML document in the indexed data tree.

Example 5.1. In Figure 5.1a we show the plain-text representation of an XML document. The document starts with the processing instruction <?xml encoding='UTF-8'?>. A processing instruction does not represent information; it is an instruction to applications describing how these applications should interpret this XML document. This particular instruction describes which character set is used for representing the plain text characters in the document.

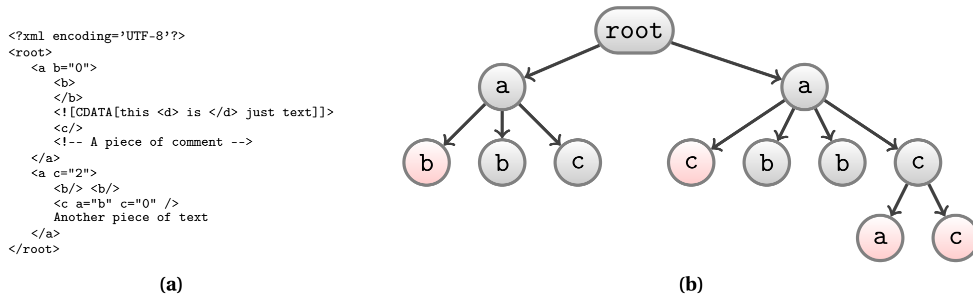


Figure 5.1: An example XML document. Figure 5.1a represents the XML document in plain text. This plain text document can also be represented by a tree as shown in Figure 5.1b; thereby we have highlighted the tree nodes representing attributes.

The first XML element starts with the start-tag <root>; this start-tag begins the XML element with name root. The content of this element ends with the end-tag </root>. Another XML element starts with the start-tag <a b="0">; this tag is nested directly within the <root> tag. As such the XML element started by <a b="0"> is a direct child of the XML element with name root. The start-tag <a b="0"> also has a single attribute b="0"; this attribute has name b and value 0. The element <c/> is the first empty-element tag representing a node without any children.

The document also contains some other features; including two different pieces of textual content. The first one is <![CDATA[this <d> is </d> just text]]>, this represents the piece of text this <d> is </d> just text. By wrapping the text within <![CDATA[and]]> we express that <d> and </d> should not be interpreted as start- and end-tags but as normal text. A simpler piece of textual content is Another piece of text, this piece does not contain any special characters. The last feature

[6] See http://www.w3c.org/standards/xml/.

in the document is a comment. The element <!-- A piece of comment --> is enclosed within <!-- and -->; indicating that the content A piece of comment can safely be ignored during further processing.

In Figure 5.1b we show the tree-representation of the structure of the XML document. In this representation we have only included XML elements and attributes; other features are not included.

Any part of an XML document that can be inserted as a child of an XML element is called a document fragment. A document fragment can be a single attribute, a piece of text, an entire XML element or even multiple XML elements. A document fragment can be represented by a forest containing zero or more trees. When the document fragment represented by some forest is added to some XML element represented by tree node e; then the roots of the trees in the forest representation of the document fragment will become direct children of tree node e.

Example 5.2. In Figure 5.1a we show the plain-text representation of an XML document. The part <b/> <b/> is an example of a document fragment. In this case the document fragment can be represented by two trees; both having a single node (namely b).

The standardized interface for reading and constructing XML documents is the Document Object Model[7]. Due to the many constructs used in XML documents this interface is not the ideal way to presenting abstract algorithms on XML documents. Therefore we introduce an abstract interface for reading XML documents.

Assumption 5.3. We have a function Next for reading XML documents. This function reads XML documents sequentially and returns the next start-tag or end-tag. For simplicity we assume that this function presents all relevant features (elements, empty elements, attributes) of the XML document by explicit start and end-tags. Lastly we assume that start-tags have a name. A representation of this name has been used as a label; which can be looked up by the function l.

The function Next can been seen as a very-abstracted representation of common XML reader interfaces[8]. These interfaces are intended to provide a simple interface for fast sequential reading of XML documents; this with a low memory footprint. Thereby these XML reader interfaces are well suited for allowing IO efficient processing of very large XML documents.

Due to the highly-structured format of XML documents the combination of the Next function and XML documents provide both a topological order and a reverse-topological order on the nodes represented by XML elements and attributes.

Theorem 5.4. Let D be an XML document. If we order the nodes represented by the XML document in the order wherein the function Next reads the start-tags of these nodes; then this order is a topological order on these nodes. If we order the nodes in the XML document in the order wherein the function Next reads the end-tags of these nodes; then this order is a reverse-topological order on these nodes.

Proof. Start- and end-tags are nested; whereby child nodes are placed between the start- and end-tag of its parent node. As such a start-tag of a child node is always read after the start-tag of its parent node; we thus have a topological order. The end-tag of a child node is always read before the end-tag of its parent node; we thus have a reverse-topological order. □

XML documents are often automatically generated and/or used by a formal protocol for (automatic) information exchange. Thereby many documents conform to a predefined structural description, for example the structural description defined by a XML Schema[9] or by a Document Type Definition[10]. As a result many XML documents are highly structured and thus are expected to have relative small indices.

5.1.2 Variants on node bisimulation

The graph index defined in Definition 2.40 uses node bisimulation; thereby nodes are grouped based on the structure of their descendants. The path queries shown in Section 1.1 return nodes based on

[7] See http:/www.w3c.org/DOM/.
[8] See for example the interface XmlReader used in the .NET Framework and the libxml2 XMLReader interface.
[9] See http://www.w3c.org/standards/xml/schema.
[10] See http://www.w3c.org/TR/REC-xml/#dt-doctype.

the structure of their ancestors. The graph index thus is not a good index to help answering these path queries. A better suited index type that can answer these queries is the 1-index; the 1-index utilizes backward node bisimilarity.

Definition 5.5. Let $G_1 = \langle N_1, E_1, l_1\rangle, G_2 = \langle N_2, E_2, l_2\rangle$ be graphs. Node $n \in N_1$ backward bisimulates node $m \in N_2$; denoted as $n \approx_{\mathrm{B}} m$; if and only if:

(1) The nodes have the same label; $l_1(n) = l_2(m)$,

(2) For every node $n' \in E'_1(n)$ there is a node $m' \in E'_2(m)$ with $n' \approx_{\mathrm{B}} m'$, and

(3) For every node $m' \in E'_2(m)$ there is a node $n' \in E'_1(n)$ with $n' \approx_{\mathrm{B}} m'$.

In this chapter we shall refer to the normal node bisimilarity as defined in Definition 2.15 as forward node bisimilarity. The only difference between forward node bisimilarity and backward node bisimilarity is that forward node bisimilarity looks at the structure of descendants and backward node bisimilarity looks at the structure of ancestors.

In previous chapters we extensively used theory derived from forward node bisimilarity in combination with properties of directed acyclic graphs to construct IO efficient algorithms. Forward and backward node bisimilarity are defined and used in similar ways. As such a large part of the theory developed for forward node bisimilarity can be adapted such that this theory is applicable to backward node bisimilarity.

Proposition 5.6. Let $G = \langle N, E, l\rangle$ be a graph, let $G' = \langle N', E', l'\rangle$ be the graph obtained from graph G by reversing every edge; thus with $N' = N$, $E' = \{(n, m) : (m, n) \in E\}$, and $l' = l$, let P be the forward bisimulation partition of N and let P' be the backward bisimulation partition of N'. We have $P = P'$

Proposition 5.6 hints at a general approach for adapting theory developed for forward node bisimilarity to theory applicable to backward node bisimilarity. One only has to reverse the edges used in the theory developed for forward node bisimilarity. Using this approach we shall introduce the backward rank of a node.

Definition 5.7. Let $G = \langle N, E, l\rangle$ be a graph. The backward rank of a node $n \in N$ is defined as the length of the longest path from any root node m to node $n \in N$. The function $rank_B$ maps nodes to their rank; this function is defined as:

$$rank_B(n) = \begin{cases} 0 & n \text{ is a root} \\ 1 + \max_{m \in E'(n)} rank_B(m) & \text{otherwise} \end{cases}$$

In this chapter we shall refer to the rank defined in Definition 2.33 as forward rank.

Example 5.8. In Figure 5.2 we have taken the tree from Figure 5.1b and annotated each node in the tree with its backward and forward rank.

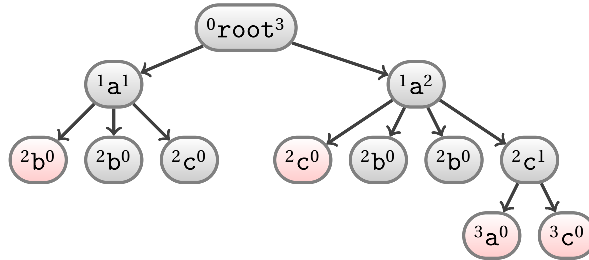


Figure 5.2: Tree representation of the XML document from Example 5.1. Thereby we have highlighted the tree nodes representing attributes and we have annotated each node with its forward rank (superscript on the right) and its backward rank (superscript on the left).

The relation that exists between forward node bisimulation and the forward rank also exists between backward node bisimulation and the backward rank. For backward bisimulation we can also introduce the backward node-bisimilarity value and the backward node-decision value; this in the same way same way as we have introduced backward rank.

There are a lot of queries that cannot be answered directly by using the 1-index. Examples are more complex twig queries; these not only look to the structure of ancestors of node, but also to the structure of descendants. A query can for example result in nodes having particular attributes; while also meeting some path query. For this type of queries we need a more complex index; an index that can answer these twig queries is the F&B-index. The F&B-index utilizes F&B bisimilarity to group nodes.

Definition 5.9. Let $G_1 = \langle N_1, E_1, l_1 \rangle, G_2 = \langle N_2, E_2, l_2 \rangle$ be graphs. Node $n \in N_1$ F&B bisimulates node $m \in N_2$; denoted as $n \approx_{\text{F\&B}} m$; if and only if:

(1) The nodes have the same label; $l_1(n) = l_2(m)$,

(2) For every node $n' \in E_1(n)$ there is a node $m' \in E_2(m)$ with $n' \approx_{\text{F\&B}} m'$,

(3) For every node $m' \in E_2(m)$ there is a node $n' \in E_1(n)$ with $n' \approx_{\text{F\&B}} m'$,

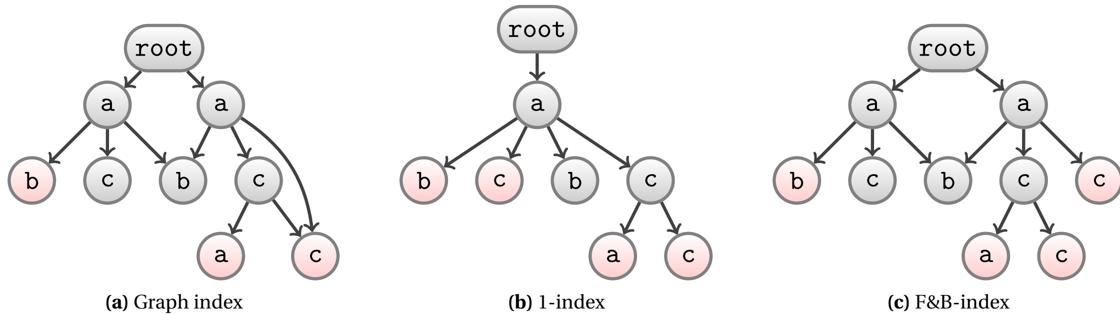
(4) For every node $n' \in E'_1(n)$ there is a node $m' \in E'_2(m)$ with $n' \approx_{\text{F\&B}} m'$, and

(5) For every node $m' \in E'_2(m)$ there is a node $n' \in E'_1(n)$ with $n' \approx_{\text{F\&B}} m'$.

The naive way for constructing the F&B-index starts with calculating the F&B bisimulation partition using repeated refinement of an initial partition P. First refine this partition P using forward node bisimilarity; then refine the resulting partition using backward node bisimilarity. Repeat these refinement steps until no changes are made during refinement; the resulting partition is the F&B bisimulation partition.

For the F&B-index on trees we only need one such refinement step. We only need to refine the initial partition once on forward node bisimilarity and refine the result once on backward node bisimilarity. The resulting partition is called the F+B-partition and can be used to construct the F+B-index. For trees this F+B-index is equivalent to the F&B-index [GBH10].

Example 5.10. In Figure 5.3 we show the graph index, the 1-index, and the F&B index of the tree representation of the XML document presented in Example 5.1.



(a) Graph index **(b)** 1-index **(c)** F&B-index

Figure 5.3: Three indices on the XML document tree presented in Figure 5.1b.

The three indices are all different; as such it is easy to see that the three indices all capture different aspects of the original XML document.

We have shown examples of the 1-index and the F&B-index in Example 5.10. One immediately sees that these indices can be big. The size of indices can be reduced; one way to reduce the size of an index is by limiting the structure one looks at when grouping nodes. For a node n we can restrict ourselves to the structure made by those ancestor nodes $m \in A(n)$ that have a path to node n with a length of at most k. There is a large class of graph indices that use this idea in some form or another; thereby

providing indices that are often much smaller than the 1-index or F&B-index. From this large class we shall look at a general representative; namely the A(k)-index.

The A(k)-index utilizes backward node k-bisimulation. Backward node k-bisimulation is a variant of backward node bisimulation. Thereby backward node k-bisimulation groups nodes n based on only the ancestors nodes that have an outgoing path of length at most k to node n.

Definition 5.11. Let $G_1 = \langle N_1, E_1, l_1 \rangle, G_2 = \langle N_2, E_2, l_2 \rangle$ be graphs, let $n \in N_1, m \in N_2$ be nodes. Backward k-bisimilarity is defined inductively.

BASE CASE: Node n and node m are backward 0-bisimilar equivalent, denoted as $n \approx^0 m$, if and only if node n and node m have the same label; thus $l_1(n) = l_2(m)$.

INDUCTIVE DEFINITION: Node n and node m are backward k-bisimilar equivalent, denoted as $n \approx^k m$, if and only if:

(1) The nodes are backward k-1-bisimilar equivalent; thus $n \approx^{k\text{-}1} m$,

(2) For every node $n' \in E_1'(n)$ there is a node $m' \in E_2'(m)$ with $n' \approx^{k\text{-}1} m'$, and

(3) For every node $m' \in E_2'(m)$ there is a node $n' \in E_1'(n)$ with $n' \approx^{k\text{-}1} m'$.

The A(k)-index, with small values for k, will in general give smaller indices as the 1-index. Note however that when the length of every path is less then k; then the 1-index and the A(k)-index are equivalent. The reduction in the size of the index provided by the A(k)-index does not come for free. The cost of the smaller index is a reduction in the number of queries the A(k)-index can answer (directly).

Example 5.12. Consider the tree in Figure 5.4. From the tree in Figure 5.4 we can create several different A(k)-indices.

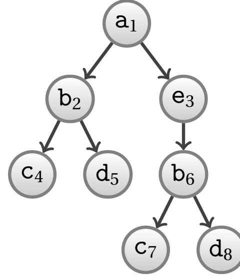


Figure 5.4: A tree graph; the text on each node represents the label of the node. The subscript on each node represents a unique identifier.

First we give the backward 0-bisimulation partition of the nodes in this graph; given by $\{\{a_1\}, \{b_2, b_6\}, \{c_4, c_7\}, \{d_5, d_8\}, \{e_3\}\}$. The backward 1-bisimulation partition is given by $\{\{a_1\}, \{b_2\}, \{b_6\}, \{c_4, c_7\}, \{d_5, d_8\}, \{e_3\}\}$. The last partition we look at is the backward 2-bisimilarity partition. This partition is given by $\{\{a_1\}, \{b_2\}, \{e_3\}, \{b_6\}, \{c_4\}, \{d_5\}, \{c_7\}, \{d_8\}\}$.

Note that backward k-bisimulation partitions on this graph; with $k > 2$; are all equivalent to the backward 2-bisimulation partition. Also the backward bisimulation partition of this graph is equivalent to the backward 2-bisimulation partition.

5.2 External memory index construction for XML documents

With the theory presented in the previous section we are ready to work out specialized external memory algorithms for constructing indices on XML documents. Thereby we not only provide index construction algorithms; we shall also show how the properties of XML documents can be utilized in the construction of IO efficient algorithms.

5.2.1 Constructing the 1-index

In Subsection 3.6.2 we have outlined several ideas that can be used to implement a fast version of external memory bisimulation partitioning. The structure of XML documents provides several ways to utilize these ideas. We shall start with idea 6; we shall show that we can efficiently create a composite node identifier based on backward rank.

Proposition 5.13. Let D be an XML document, let t be the start-tag read with function NEXT for an XML element or attribute represented by node n. The backward rank of node n is given by the difference between the number of start-tags that are read and the number of end-tags that are read before reading tag t.

We can utilize Proposition 5.13 to efficiently evaluate the backward rank of each node; this by maintaining a counter that is incremented when reading a start-tag and decremented when reading an end-tag. By extending this idea we can easily assign a unique composite node identifier (backward rank, unique identifier with respect to the nodes having this backward rank) to each node.

Assumption 5.14. We assume that each node in an XML document is assigned a composite node identifier (backward rank, unique identifier with respect to the nodes having this backward rank).

Assumption 5.14 can only be used if there is a way to efficiently calculate these composite node identifiers. Remark 5.15 sketches how the composite node identifiers can be constructed.

Remark 5.15. Assume we have a counter r and a list R. The counter r maintains the backward rank for the node represented by the next start-tag. The list R is a mapping between backward rank and the number of nodes that are already assigned this backward rank; thus $R[i] = j$ indicates that we have already read j nodes with backward rank i. We maintain a pointer to the r-th element in this list R. The list R is an empty list on initialization, counter r is initialized to 0.

Now when we read a start-tag representing a node n during the processing of an XML document; then this node is assigned backward rank r. Now we read the value $R[r]$ using the pointer; when no value is present at $R[r]$ then this value is initialized by 0. The value $R[r]$ gives the number of nodes read before node n that also have backward rank r. We use this value as the unique identifier with respect to the nodes having backward rank r. We then increment the value $R[r]$, we increment r by one and move the pointer to the next element in the list R. When an end-tag is read; then we only have to maintain r and the pointer to $R[r]$. We do so by decrementing r by one and moving the pointer to the previous element in list R.

The approach in Remark 5.15 makes it easy to determine the composite node identifier for any node n. We can also utilize this approach to determine the composite node identifier of any parent of node n; Remark 5.16 sketches how the composite node identifier of a parent can be constructed.

Remark 5.16. The backward node rank of node m is given by $rank_B(n)-1$. The unique identifier with respect to the nodes having backward rank $rank_B(m)$ can be found by reading the value $R[rank_B(m)]$. When node n is read, then node m must be the last node that is read with rank $rank_B(m)$; the value $(rank_B(m), R[rank_B(m)]-1)$ thus is the composite node identifier of node m.

The value of $R[rank_B(m)]$ can easily be determined by using the pointer to $R[r]$ when we are processing node n. Before constructing the composite node identifier of node n we have $r = rank_B(m)+1$; after construction of the composite node identifier of node n we have $r = rank_B(m)+2$. We thus can retrieve the value $R[rank_B(m)]$ by performing a constant amount of move operations on the pointer to list R.

Remark 5.15 can be utilized to easily partition all nodes on rank; Remark 5.16 can be utilized to easily represent all edges in the XML document tree as composite node identifier pairs. Thereby there is no need for the expensive renumbering operations performed by Algorithm 3.4. One can however expect that this results in a very bad initial partition. We shall now see that this is not a problem when processing XML documents. Therefore we shall show how idea 5 and idea 8 from Subsection 3.6.2 can be utilized to eliminate the need for a partition decision structure.

In a tree the fan-in is upper bounded by 1 incoming edge; according to idea 8 we thus have node-decision values whose size is upper bounded. During local refinement we thus can replace the local

node-hash value by the actual node-decision value of each node. Thereby local refinement by sorting produces a bisimulation partition of the initial partition blocks; this without using any partition decision structure. Using the sketched ideas results in Algorithm 5.1.

Algorithm 5.1 Online backward bisimulation partitioning algorithm for XML documents

---

Require: XML document D.
Ensure: The output is the pair $((rank_B(n), i), p)$ with $(rank_B(n), i)$ the composite node identifier for node n and p an identifier for the backward bisimulation partition block whereto n belongs.

1: $r \leftarrow 0$
2: N, E, R are empty lists
3: **for all** $e \leftarrow \textsc{Next}(D)$ **do**
4:   **if** e is a start-tag **then**
5:     **if** $|R| = r$ **then**
6:       $\textsc{Add}(R, 0)$
7:     **if** $r \neq 0$ **then**
8:       $\textsc{Add}(E, ((r-1, R[r-1]-1), (r, R[r])))$
9:     $\textsc{Add}(N, ((r, R[r]), l(e))$
10:     $R[r], r \leftarrow R[r]+1, r+1$
11:   **else if** e is a end-tag **then**
12:     $r \leftarrow r-1$

13: $\textsc{Sort}(N)$ on lexicographically order
14: $\textsc{Sort}(E)$ on lexicographically order

15: $i \leftarrow 0$
16: Q is a priority queue
17: **for all** rank r in list N **do**
18:   N', E' are empty lists
19:   **for all** elements $((r, c), l) \in N$ **do**
20:     $p \leftarrow \{p : ((r, c), p) = \textsc{Top}(Q)\}$
21:     $\textsc{Add}(N', (l, p, c))$
22:     **for all** elements $((r, c), (r+1, c')) \in E$ **do**
23:       $\textsc{Add}(E', (l, p, c, c'))$
24:   $\textsc{Sort}(N')$ on lexicographically order
25:   $\textsc{Sort}(E')$ on lexicographically order
26:   **for all** label l, partition block identifier p in list N' **do**
27:     **for all** elements $(l, p, c) \in N'$ **do**
28:       **print** $((r, c), i)$
29:       **for all** elements $(l, p, c, c') \in E'$ **do**
30:         $\textsc{Add}(Q, ((r+1, c'), i))$
31:     $i \leftarrow i+1$

Algorithm 5.1 is a combination of Algorithm 3.4 and Algorithm 3.6 optimized by the optimizations described in this subsection. The correctness of Algorithm 5.1 thus follows by construction. Due to the optimizations and the use of XML documents we can provide a strict worst case upper bound on the IO complexity of Algorithm 5.1.

Theorem 5.17. The worst case IO complexity of Algorithm 5.1 is $O(\textsc{Sort}(|N|) + \text{PQ}(|N|))$.

Proof. The algorithm constructs lists N, E, N', E'; these lists have accumulative size of $O(|N|)$. These lists are sequentially written; sorted and then sequentially read. The total IO cost for these lists thus is $O(\textsc{Sort}(|N|))$. The list R has maximum size $O(|N|)$; the read and write patterns to this list are not sequentially; but they are localized. One only has to read the next element when a start-tag is encountered; and return to the previous element in the list when an end-tag is encountered. The total IO cost for list R

thus is $O(\text{SCAN}(|N|))$. In total $|E| = \Theta(|N|)$ elements are added and removed from the priority queue; this introduces an additional IO cost of $O(PQ(|N|))$. □

Remark 5.18. For clarity Algorithm 5.1 is not presented in an optimized fashion. We shall give some brief ideas that can be used for optimizing implementations of Algorithm 5.1.

(1) We have stored backward ranks quit liberal in the algorithm. Many of these stored backward rank values are unnecessary and can be removed from actual implementations to reduce the total size of the used data structures.

   (a) Edges $((r_1, i_1), (r_2, i_2))$ stored in E can be simplified to $((r_1, i_1), i_2)$. In trees we always have $r_2 = r_1 + 1$; the backward rank of child nodes is always one higher then the backward rank of parent nodes.

   (b) Each initial partition block p contains all nodes with a specific backward rank r. All children of nodes in this partition block have backward rank $r + 1$. The priority queue will (at any given time) only contain nodes that are placed in the same partition block; all these nodes have the same backward rank. The backward rank can thus be omitted when sending messages to children utilizing the priority queue.

(2) As with Algorithm 3.6 it is a good idea to perform operations in internal memory whenever possible; see the first three ideas described in Subsection 3.6.2.

5.2.2 Constructing the F&B-index

The F+B-index provides a simple approach to calculating the F&B-index on a XML document. One simply runs Algorithm 3.6 to calculate the forward node bisimulation partition P and then runs an adapted version of Algorithm 3.6 to refine this partition P by refining each partition block on backward node bisimilarity. This results in the F+B partition of the XML document; which is equivalent to the F&B partition.

In Subsection 5.2.1 we have however seen that the backward node bisimulation partition of an XML document can be constructed very efficiently. A better approach would thus be reverse the two partitioning steps; thus to calculate a 'B+F partition'. We shall first show that this approach will yield a partition equivalent to the F+B partition.

Theorem 5.19. Let $G =< N, E, l >$ be a graph, let P_{F+B} be the F+B partition of nodes N, let P_{B+F} be the B+F partition of nodes N. These partitions are equivalent; thus $P_{\text{F+B}} = P_{\text{B+F}}$.

Proof (sketch). Both partitions are node-value partitions on the value (forward node-bisimilarity value, backward node-bisimilarity value). □

We thus can utilize Algorithm 5.1 to calculate an initial backward node bisimulation partition P_{B} that is used as input for Algorithm 3.6. We can however easily enhance Algorithm 5.1 such that during backward node bisimulation partitioning each node is also annotated with a forward rank.

Proposition 5.20. Let D be an XML document, let t be the end-tag read with function NEXT for an XML element or attribute represented by node n, let r be the maximum forward rank of any child node of n. The forward rank of node n is given by $rank(n) = r + 1$.

We can utilize Proposition 5.20 to efficiently evaluate the forward rank of each node. Remark 5.21 sketches how the construction of node identifiers can be performed by utilizing a stack.

Remark 5.21. Assume we have a counter r and stack S. The counter r maintains the backward rank for the node represented by the next start-tag; see Remark 5.15 on how this counter r is maintained. The top of the stack S maintains the maximum forward rank of any descendants of the last node we have encountered with rank $r - 1$; thus the top of the stack S contains the maximum forward rank of any descendants of the current node. The stack S is an empty stack on initialization.

Now when we read a start-tag represents a node n during the processing of an XML document; then this node is assigned backward rank r. We have not yet encountered any descendants of this node n;

thus we push 0 onto the stack. When an end-tag for node n is read; then the forward rank of node n is given by $rank(n) = \textsc{Top}(s) + 1$; we then pop the top of the stack. If the stack is not empty (thus when node n has a parent); then we replace the new top of the stack by $\max(rank(n), \textsc{Top}(s))$.

Note that Remark 5.21 describes a simplified version of time-forward processing to calculate forward ranks. Forward ranks provide enough information for an initial partition whereon Algorithm 3.6 can operate. We can however further enhance Algorithm 5.1 such that during backward node bisimulation partitioning each node is also annotated with a forward node-hash value. Remark 5.22 sketches how we can integrate the computation of node-hash values into Algorithm 5.1 by using time-forward processing.

Remark 5.22. Assume we have a counter r, list R and priority queue $Q_{\mathcal{H}}$. The counter r maintains the backward rank for the node represented by the next start-tag. The list R is a mapping between backward rank and the number of nodes that are already assigned this backward rank. See Remark 5.15 on how this counter r and list R are maintained. The priority queue $Q_{\mathcal{H}}$ is used to send node-hash values from child nodes to their parent nodes.

When an end-tag for node n is read; then we can use counter r and list R to determine the composite node identifier i of node n; see Remark 5.15 for details. The node-hash value for node n is now given by $v_{\mathcal{H}}(n) = \mathcal{H}(l(n), \{s : (i, s) = \textsc{Top}(Q_{\mathcal{H}})\})$. If node n has a parent node (when $r \neq 0$); then the node-hash value $v_{\mathcal{H}}(n)$ must be send to this parent node. We can utilize counter r and list R to determine the composite node identifier j of the parent of node n; to send $v_{\mathcal{H}}(n)$ to the parent of node n we add the value $(j, v_{\mathcal{H}}(n))$ to the priority queue $Q_{\mathcal{H}}$.

By integrating the ideas outlined in Remark 5.21 and Remark 5.22 into Algorithm 5.1 we construct an algorithm that performs backward node bisimulation partitioning and at the same time annotates each node with the forward rank and forward node-hash value needed for constricting the structural summary partition used as input for Algorithm 3.6.

Note that Algorithm 3.6 gets a refinement of the structural summary as input (refined on backward node bisimulation); this further reduces the size of initial partition blocks and thus allows more work to be performed entirely in internal memory. The output of Algorithm 3.6 will be a refinement of the forward node bisimulation partition (namely the F+B partition). A global partition decision structure thus can contain several entries that have the same forward node-decision value as a key. For constructing a useful global partition decision structure one thus should use a combination of forward node-decision values and backward node-decision values as a key; as this key is unique for partition blocks in the F+B partition.

5.2.3 Constructing the A(k)-index

The A(k)-index seems similar to the 1-index; but there is a major difference between the two. According to Theorem 2.36 all backward bisimilar nodes have the same backward rank. This does however not have to hold for backward k-bisimilar nodes; see Example 5.12 for some examples. We thus cannot use backward rank to localize the partitioning computations. We can however express backward node k-bisimilarity in another way; namely in terms of k-traces.

Definition 5.23. Let $G = \langle N, E, l \rangle$ be a graph, let $n_1 \in N, \dots, n_i \in N; 1 \leq i$ be nodes, let $n_1, \dots, n_i$ be a path[11] from node n_1 to node n_i. The sequence $l(n_1), \dots, l(n_i)$ is a trace.

Let $T = l_1, \dots, l_i; 1 \leq i$ be a trace. The k-trace of T is a sequence containing the last k elements in the sequence T; we thus define the k-trace of T as $l_{i-k} \dots l_i$. Thereby we assume that traces whose length is less than k are prefixed by at least $k - i$ occurrences of a special label λ; we thus define $l_j = \lambda, j < 1$. The special label λ should thereby only be used for prefixing too-short traces.

The k-trace can easily be represented by a fixed size value; we shall now show that k-trace values can be used as a backward k-bisimilarity replacement for the backward node-decision value used for constructing the 1-index.

[11] Note that we use a slightly different definition for a path as Definition 2.3. For defining k-traces we allow paths consisting of a single node.

Theorem 5.24. Let $G = \langle N, E, l \rangle$ be a tree, let $r \in N$ be the root node of tree G, let $n \in N, m \in N$ be nodes, let T_n^{k+1} be the k+1-trace from node r to node n, let T_m^{k+1} be the k+1-trace from node r to node m. We have $n \approx_B^k m$ if and only if $T_n^{k+1} = T_m^{k+1}$.

Proof. The proof is by induction on k.

BASE CASE: For $k = 0$ we have that the 1-traces T_n and T_m only contain the labels of node n and node m. Definition 5.11 defines backward 0-bisimilarity as label equivalence; as such $n \approx_B^0 m$ if and only if $T_n^1 = T_m^1$ holds.

INDUCTION HYPOTHESIS: For any value of $k \leq i$ we have $n \approx_B^k m$ if and only if $T_n^{k+1} = T_m^{k+1}$.

INDUCTION STEP: Let n' be the parent node of n, let m' be the parent node of m, let $T_{m'}^j$ be the j-trace from node r to node m', let $T_{n'}^j$ be the j-trace from node r to node n'. According to Definition 5.11 we have $n \approx_B^{i+1} m$ if and only if $(l(n) = l(m))$ and $(n \approx_B^i m)$ and $(n' \approx_B^i m')$.

According to the induction hypothesis we have $n' \approx_B^i m'$ if and only if $T_{n'}^{i+1} = T_{m'}^{i+1}$ and we have $n \approx_B^i m$ if and only if $T_n^{i+1} = T_m^{i+1}$. We can rewrite $T_n^{i+1} = T_m^{i+1}$ to $T_{n'}^i \mathbin{+\!\!+} l(n) = T_{m'}^i \mathbin{+\!\!+} l(m)$. We have $T_{n'}^i = T_{m'}^i$ whenever $T_{n'}^{i+1} = T_{m'}^{i+1}$. As such we have $n \approx_B^{i+1} m$ if and only if $T_{n'}^{i+1} = T_{m'}^{i+1}$ and $l(n) = l(m)$. We have $T_n^{i+2} = T_{n'}^{i+1} \mathbin{+\!\!+} l(n)$ and $T_m^{i+2} = T_{m'}^{i+1} \mathbin{+\!\!+} l(m)$ thus we have $n \approx_B^{i+1} m$ if and only if $T_n^{i+2} = T_m^{i+2}$.

□

We can easily use a stack to store the labels of all parents of the current node; this 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 elements we get the k-trace. This leads to a very straight forward A(k)-construction algorithm for XML documents, this algorithm is presented in Algorithm 5.2.

Algorithm 5.2 Online backward k-bisimulation partitioning algorithm for XML documents

Require: XML document D.
Ensure: The output is the pair (i, p) with i the node identifier for node n and p an identifier for the backward k-bisimulation partition block whereto n belongs.

```
1:  N, S are empty lists
2:  for all 0 ≤ i < k do
3:     ADD(S, λ)
4:  i ← 0
5:  for all e ← NEXT(D) do
6:     if e is a start-tag then
7:        PUSH(S, l(e))
8:        ADD(N, (S[|S| − (k + 1), |S|), i))
9:        i ← i + 1
10:    else if e is a end-tag then
11:       POP(S)
12: SORT(N) on lexicographically order
13: p ← 0
14: for all k-trace t in list N do
15:    for all (t, i) ∈ N do
16:       print (i, p)
17:    p ← p + 1
```

The correctness of Algorithm 5.2 follows directly from Theorem 5.24. We shall now analyze the IO complexity of the algorithm.

Theorem 5.25. The worst case IO complexity of Algorithm 5.1 is $O(\textsc{Scan}(k+|N|)+\textsc{Sort}(k|N|))$.

Proof. The stack S will contain at most $k+|N|$ labels; this stack can be read and written to with a total IO cost of $O(\textsc{Scan}(k+|N|))$. The list N will contain at most $|N|$ entries; each entry having a size given by $\Theta(k)$; the total IO cost for reading, writing and sorting list N is thus given by $O(\textsc{Scan}(k+|N|)+\textsc{Sort}(k|N|))$. □

5.3 Partition maintenance for XML documents

The structure of XML documents restricts the allowed update operations on an XML document. None of the operations studied in Chapter 4 are allowed; as all these operations can turn the XML document into a non-tree graph. The structure of XML documents does however allow some update operations; namely the addition of a document fragment to a node and removal of a document fragment from a node.

When we store XML documents in a single large container; then we can also allow subgraph addition and removal. For an XML document container subgraph addition corresponds with adding an XML document to the container, subgraph removal corresponds with removing an XML document from the container. We shall briefly look at all three index types to see what the possibilities and difficulties are for supporting these update operations.

5.3.1 Updating the 1-index

The 1-index is comparable to the graph index studied in Chapter 4. We can however utilize that each node in an XML document has at most a single parent. The node-decision values for nodes in XML documents thus have a fixed size. Therefore we can replace string B-trees for storing the index by the simpler B^+ tree data structure.

For document fragment addition to a node n with index node $n_\downarrow$ we can easily adapt subgraph addition. We can simply add the 1-index of the document fragments as children of $n_\downarrow$; we can then perform the same approach as described for subgraph addition in Subsection 4.3.1. Removal of document fragments and removal of subgraphs can both be handled in the same way as subgraph removal described in Subsection 4.3.2.

5.3.2 Updating the F&B-index

For subgraph addition and removal we can stick with the approach described in Subsection 4.3.2. Thereby we repeat a remark stated in Subsection 5.2.2; one should use a combination of forward node-decision values and backward node-decision values as a key for partition blocks in a F+B partition.

Updating the F&B-index after document fragment updates is a bit more complicated as updating the 1-index. Performing document fragment updates to node n implies edge updates. For the F&B-index one sees that updating the index after document fragment updates can affect the document fragment and its new ancestors. These new ancestors are node n and the ancestors $A(n)$ of node n. As such these ancestors are those nodes on the path from the root node r to node n; whereby root node r is the root node of the XML document wherein node n is placed.

We can utilize the same approach as we used for the construction of the F+B index. Updating after a document fragment can be implemented by first adjusting the backward bisimulation partition of the affected nodes and then adjusting the forward bisimulation partition of the affected nodes. For adjusting the backward bisimulation partition of the affected nodes one can look at the approach for the 1-index described in the previous subsection. For adjusting the forward bisimulation partition of the affected nodes one can look at the edge propagation approach described in Subsection 4.3.3.

5.3.3 Updating the A(k)-index

For subgraph addition and removal from an A(k)-index one can easily adopt the approach described in Chapter 4. Thereby we note that we have utilized k+1-traces as a unique partition block key to

determine in which backward k-bisimilarity partition block each node n should be placed. These $k+1$-traces are completely independent from the graph wherein node n is placed. As such no renaming step is necessary when merging several backward k-bisimilarity partitions.

For document fragment addition in A(k)-indices we can utilize the same idea as used for document fragment addition in 1-indices. If a document fragment is added to node n represented by index node $n_{\downarrow}$; then we can simply let the document fragment be a child of the index node $n_{\downarrow}$. An easy way to achieve this is by prefixing the trace of each node in the document fragment by the k-trace of node n.

For practical purposes one might use string B-trees with prefix searching for lookups on $k+1$-traces. When we store $k+1$ in reverse order then we can also range query the string B-tree with j-traces where $j \leq k$. As such this structure for storing the backward k-bisimulation partition also provides easy access to backward j-bisimulation partitions.

5.4 Final notes

This chapter provides construction algorithms for indexing XML documents using the 1-index, the F&B-index, and the A(k)-index. We have also provided sketches for updating these indices after changes to the indexed XML documents are made. Thereby we have utilized the structure of XML documents to our advantage. The approaches presented in this chapter thus are limited to their applicability.

The 1-index can easily be constructed for topological ordered directed acyclic graphs; this by utilizing Algorithm 3.6. For the F&B-index we cannot easily extend the approach presented in this chapter; as for non-tree graphs it does not hold that the F+B-index of the graph is equivalent to the F&B-index of the graph. Also for the A(k)-index we see that the theory presented in this chapter only holds for trees. For directed (acyclic) graphs the k-trace between a root node r and a node n is not a key that can determine the backward k-bisimulation partition block wherein node n should be placed.

Extending the F&B-index and the A(k)-index to directed acyclic graph is difficult. For the F&B-index we see that processing in one direction does not directly yield a good index; one-direction processing needs to be repeated until no changes are made. An algorithm that processes nodes in multiple directions at the same time is hard to construct in an IO efficient way. For the A(k)-index we see that we cannot use backward ranks to localize complicated decision making (by localizing partition decision structures). As such a more powerful approach then the approach used for bisimulation partitioning is needed to make backward k-bisimulation partitioning of directed acyclic graphs IO efficient.

Chapter 6

EXPERIMENTAL VERIFICATION

In the previous chapters we have presented external memory algorithms for bisimulation partitioning of directed acyclic graphs and XML documents. Analysis showed that these algorithms are expected to have low runtimes and low IO complexities. Theoretically expected low runtimes and IO complexities for an algorithm do however not always imply a practically fast algorithm. Therefore we have set up a small scale experiment to see if the algorithms are also practically fast.

We have primarily restricted this small scale experimental to the external memory bisimulation algorithm presented in Algorithm 3.6. This algorithm; and the theory it builds upon; provides the foundation whereupon the remainder of our work is based. For the experiment we have implemented Algorithm 3.6 together with supporting algorithms; on these implementations we have performed several measurements to see how the algorithms behave in practice. Thereby we have also made a comparison between Algorithm 3.6 and the more specialized approach for XML documents presented in Algorithm 5.1.

In our experiments we have studied the behavior of Algorithm 3.6 with respect to the size of its input and the available amount of internal memory. The comparison between Algorithm 3.6 and Algorithm 5.1 only looks at the behavior as a function of the size of the input. The questions we want to answer with the experiments and details of the experiments can be found in Section 6.2. Before introducing the details of the experiments we introduce relevant details of the implementations; this can be found in Section 6.1. The results of these experiments are presented in Section 6.3. In Section 6.4 we try to answer the questions stated in Section 6.2; thereby concluding on the experimental results.

6.1 Implementation overview

For the experiments we have implemented each tested algorithm and each supporting tool as a separate program. With this setup it is easy to perform measurements on each algorithm. This results in three classes of programs; namely programs implementing the algorithms we are interested in, internal developed tools to aid performing measurements, and externally developed tools to aid performing measurements. We shall first introduce the three programs that implement the algorithms we are interested in.

`dagfpdagfps`: Implements Algorithm 3.4. The `dagfpdagfps` program reads graphs in $\mathcal{L}$ representation and transforms them into graphs in $\mathcal{L}_S$ representation. The program supports two types of output. The output graph can be an initial partition based on structural summaries and the output graph can be an initial partition based on rank and label. The initial partition based on rank and label represents the worst-case structural summary.

`exbisim`: Implements Algorithm 3.6. The `exbisim` program reads graphs in $\mathcal{L}_S$ representation and outputs a bisimulation partition of this graph.

`xmlbbisim`: Implements Algorithm 5.1. The `xmlbbisim` program reads trees represented by XML documents and outputs a backward-bisimulation partition of the tree. This program only uses XML elements and attributes as nodes in the tree representation of XML documents.

The three algorithms are implemented without any of the possible implementation-specific improvements described in Subsection 3.6.2 and improvements described in Remark 5.18. For testing the three implemented algorithms we need several programs to generate input graphs and convert them to the expected graph representations. Therefore we have developed the following tools:

`gen`: The `gen` program is a simple graph generator. The program can create random directed acyclic graphs and trees. The program can also create chains and transitive closure chains. There is limited support for controlling label assignment. This program generates the input for most experiments; thereby the program does not try to represent any particular class of directed acyclic graphs.

For generating these random graphs we use a simple approach. First the generator generates a label for the node. This label is picked from a limited set; thereby the generator uses a subset of the available labels. The specific subset is selected by using information derived from the node identifier. The generator then tries to repeatedly assign edges with probability p to the generated node. When no edge is assigned then the generator moves on to generating the next node. An n-th edge for node i is thus generated with probability p^n. For generating an edge for the i-th node the random generator picks a node identifier in the range $0 \leq j < i$; if the picked node identifier is already a child of the i-th node then this generated edge is ignored.

`dagdagfp`: The `dagdagfp` program reads graphs in the format produced by `gen` and outputs the graphs in $\mathcal{L}$ representation. The $\mathcal{L}$ representation of the graph can be used as input for the `dagfpdagfps` program.

`graphstat`: The `graphstat` program gives statistics about graphs. It accepts input in any format used by the various programs. The output includes the number of nodes, number of edges and the number of labels. For certain types of input the program can provide additional details such as the maximum rank assigned to a node in the graph.

`xmldagfpr`: The `xmldagfpr` program reads trees represented by XML documents and outputs the trees in $\mathcal{L}$ representation. The $\mathcal{L}$ representation of the graph can be used as input for the `dagfpdagfps` program. This program is only used to compare Algorithm 3.6 and Algorithm 5.1. This program only uses XML elements and attributes as nodes in the tree representation of XML documents. Thereby the program automatically reverses all edges in this tree representation such that the backward node bisimulation partition is calculated when this tree is given as input to Algorithm 3.4 and Algorithm 3.6.

During testing we also used an externally developed program; namely the `xmlgen` program provided by the XML Benchmark Project[1]. This program is used to generate large XML documents for benchmarking XML-related algorithms. We have used version 0.92 of `xmlgen`. Further we have used various measurements scripts that generated test data and executed the various programs to prepare the generated test data and measure the performance of the implemented algorithms.

6.1.1 Low-level details

All programs are written in C++; the source code can be found at `http://jhellings.nl/projects/exbisim/`. All source code is compiled with the Microsoft (R) C/C++ Optimizing Compiler (version 16.00.40219.01 for x64). We have used the following third party libraries for crucial functionality:

STXXL: Standard Template Library for Extra Large Data Sets: This library provides IO efficient external memory implementations for common algorithms and data structures. From this library we have used the vector data structure, the priority queue data structure, and the general sort algorithm. We have used version 1.3.1 of the library; retrieved from SourceForge[2].

[1] See `http://www.xml-benchmark.org/`.

[2] See `http://sourceforge.net/projects/stxxl/files/stxxl/1.3.1/stxxl-1.3.1.tar.gz/download`.

LIBXML2: This library provides several interfaces for reading and writing XML documents. We have used the XMLReader API for reading XML documents. We have used version 2.7.8 of the libxml2 library; retrieved from xmlsoft.org[3].

BOOST C++ LIBRARIES: This library provides functionality supporting many useful tasks. The STXXL library depends on some of the Boost libraries. Further we have used the program options library for reading command line options. We have used version 1.46.1; retrieved from SourceForge[4].

The implementation used 32bit unsigned integers for storing node identifiers, labels, hash values, ranks and other pieces of information. The storage space needed to store a single node stored as a (node identifier, label identifier)-pair thus is 8bytes. The STXXL library utilizes a disk block size of 2MB; as such at most 262144 nodes can be stored in a single disk block. This results in a lower bound on the IOs per (number of nodes, number of edges) of $\frac{1}{262144} \approx 3.8 \cdot 10^{-6}$. In practice we shall see higher values as we not only store nodes but also edges and other auxiliary data. Furthermore we are likely to perform several IOs per node and edge as information is read, written and sorted several times.

We have assured ourselves of the correctness of the implemented algorithms in several ways. First we have proven that the implemented approaches should theoretically work. We have also run the programs on small inputs and manually verified the outcomes. We did also test the programs on large predictable inputs and verified the outcome for these inputs. Thereby we have tested if the programs worked on groups of (topological) chains. Lastly we have verified if several implementations gave the same result for the same input. Thereby we have compared an internal memory bisimulation partitioning implementation with the external memory bisimulation partitioning implementation. We have also checked if the XML backward bisimulation implementation provided by `xmlbbisim` did give the same results as the external memory bisimulation partitioning algorithm provided by `exbisim`.

6.1.2 System specifications

All programs are tested on a Dell XPS 15 (L501X) laptop with an Intel Core i5-560M Processor and 4GB of main memory. We have used the internal hard disk drive of this system; a Seagate Momentus ST9500420AS; for temporary storage. Thereby the internal hard disk drive is used for sorting and for storing temporarily data structures such as the used lists and priority queues. All storage used on this drive is managed by the STXXL library.

During testing the laptop was connected to an external USB hard disk drive; the Western Digital Elements WDBAAU0010HBK-01. This external hard disk drive is only used for storing input and output of programs. Thereby it is guaranteed that each program reads its input sequentially from the external hard disk drive and writes its output sequentially to the same external hard disk drive. We have performed some raw performance benchmarks on these hard disk drives using the HD Tach 3.0.4.0[5] benchmark program; see Table 6.1 for the results.

Disk	Random access [ms]	Average speed [MB/s]	Burst speed [MB/s]
ST9500420AS	17.4	87.8	179.2
WDBAAU0010HBK-01	14.9	36.7	37.4

Table 6.1: Raw performance benchmarks on the internal and external hard disk drive.

We have performed the experiments under Microsoft Windows 7 x64 (Home Premium) using the high performance power plan. We have tested and measured the behavior of each program at least once before performing the experiments. Thereby we have checked and verified that at no prolonged amount of time the CPU load is high and that at any prolonged amount of time the IO throughput is high. We have also checked that memory consumption is as expected (both for the program and for the system).

[3] See `ftp://xmlsoft.org/libxml2/libxml2-2.7.8.tar.gz`.
[4] See `http://sourceforge.net/projects/boost/files/boost/1.46.1/boost_1_46_1.7z/download`.
[5] See `http://www.simplisoftware.com/Public/index.php?request=HdTach`.

These checks have been performed to assure that there are no anomalies in the implementation and in the environment that could result in useless experimental results. After these verifications we have performed the real measurements on the experiments.

6.2 Experiment description

We have performed four small scale experiments. During each experiment we have measured the runtime of each program and the total number of IOs performed by the algorithms. For determining the total number of IOs we have only looked at the IOs performed by the STXXL library (to the internal hard drive); as only these IOs are a necessary part of the algorithm. During measurements we also collect statistics; including statistics on the input graphs and resulting partitions. With our experiments we have tried to answer the following questions:

DOES ALGORITHM 3.6 BEHAVE AS ONE CAN EXPECT FROM THE THEORY? Thereby we have looked at IO efficiency and runtime scalability with respect to the size of input graphs. We have also looked at IO efficiency and runtime with respect to the memory usage.

DOES ALGORITHM 3.6 BENEFIT FROM A GOOD INITIAL PARTITION? Thereby we have compared the performance of the algorithm when it has a bad initial partition (with few initial partition blocks) as input with the performance of the algorithm when it has a good initial partition (with many initial partition blocks) as input.

HOW DOES ALGORITHM 3.6 COMPARE WITH MORE OPTIMIZED APPROACHES? Thereby we have compared the performance of Algorithm 3.6 and Algorithm 5.1 for performing backward node bisimulation partitioning on large XML documents.

We have set up four small scale experiments that should aid in answering these questions.

EXPERIMENT 1: In this experiment we measure results that should help answering the first two questions. In this experiment we have performed external memory bisimulation of graphs; thereby we have used initial partitions based on structural summaries and initial partitions based on rank and label. For this experiment we have created graphs whose size is a function of the number of nodes; we have created graphs that have between $100 \cdot 10^6$ and $1000 \cdot 10^6$ nodes. Figure 6.1 provides the program flow used in this experiment. Every graph has an average of three to four edges per node. The file size of the input graphs ranges from 2.1GB to 21.2GB.

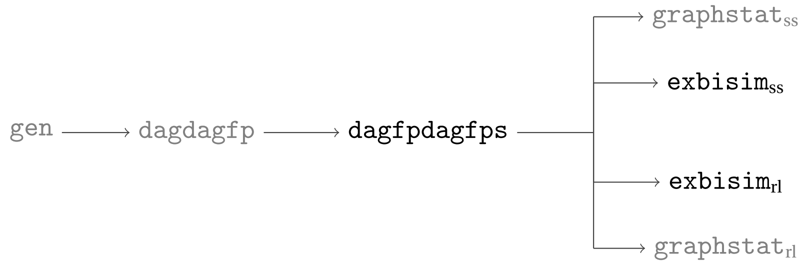


Figure 6.1: Visualization of the program flow for EXPERIMENT 1; in this figure an arrow indicates that the output of one program is used as the input for another program. We have only performed measurements on the highlighted programs (`dagfpdagfps` and `exbisim`). The subscript $_{rl}$ stands for an initial partition based on rank and label. The subscript $_{ss}$ stands for an initial partition based on structural summaries.

EXPERIMENT 2: In this experiment we measure results that should help answering the first question. In this experiment we have performed external memory bisimulation of graphs; thereby we have used initial partitions based on structural summaries. For this experiment we have created graphs

with 50000 nodes and a variable amount of edges; namely between 0 and $1249 \cdot 10^6$ edges. Figure 6.2 provides the program flow used in this experiment.

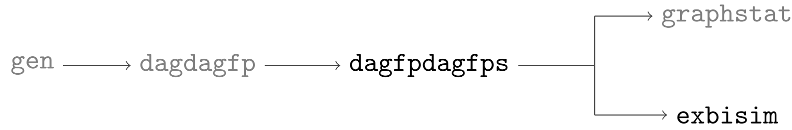


Figure 6.2: Visualization of the program flow for EXPERIMENT 2; in this figure an arrow indicates that the output of one program is used as the input for another program. We have only performed measurements on the highlighted programs (`dagfpdagfps` and `exbisim`).

EXPERIMENT 3: In this experiment we measure results that should help answering the first question. In this experiment we have performed external memory bisimulation of graphs. For this experiment we have created a single graph with 10^8 nodes and $3.3 \cdot 10^8$ edges. On this graph we have calculated the structural summary partition; on this structural summary partition we have performed external memory bisimulation partitioning. Thereby we have used versions of `exbisim` and `dagfpdagfps` that are constrained to a limited memory usage. We have used values between 4MB and 512MB for the amount of memory each data structure and sort operation is allowed to use. Figure 6.3 provides the program flow used in this experiment.

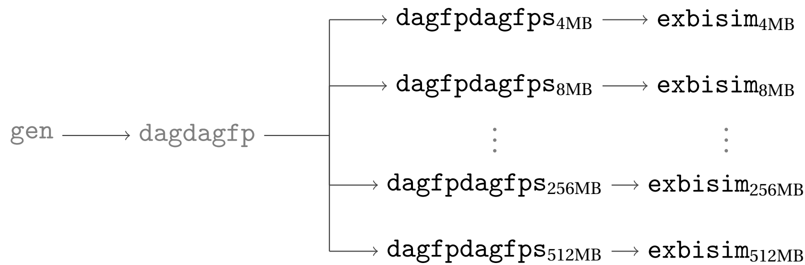


Figure 6.3: Visualization of the program flow for EXPERIMENT 3; in this figure an arrow indicates that the output of one program is used as the input for another program. We have only performed measurements on the highlighted programs (`dagfpdagfps` and `exbisim`). The subscript indicates the amount of memory each data structure and sort operation is allowed to use.

EXPERIMENT 4: In this experiment we measure results that should help answering the last question. In this experiment we have performed external memory bisimulation of XML documents. Thereby we have compared Algorithm 5.1 with the combination of Algorithm 3.4 and Algorithm 3.6. For this experiment we have created XML documents using the `xmlgen` program provided by the XML Benchmark Project. For the generation of XML documents we have used scaling factors between 50 and 500. Figure 6.4 provides the program flow used in this experiment.

Figure 6.4: Visualization of the program flow for EXPERIMENT 4; in this figure an arrow indicates that the output of one program is used as the input for another program. We have only performed measurements on the highlighted programs (`xmldagfpr`, `xmlbbisim`, `dagfpdagfps` and `exbisim`).

6.3 Results

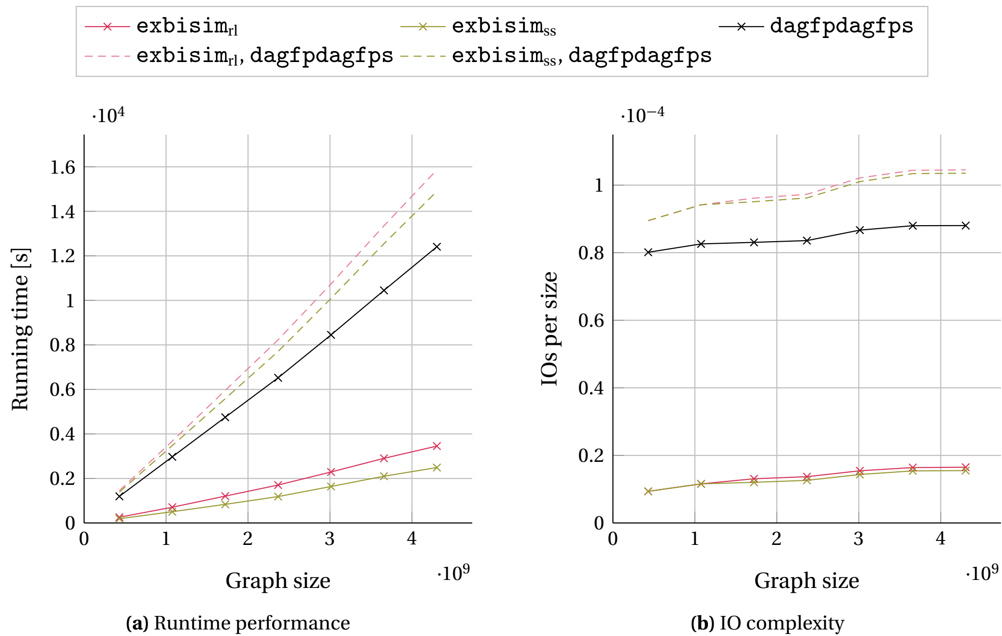
We shall present the relevant measurements performed for each experiment.

EXPERIMENT 1: In Figure 6.5 we have plotted the runtime and IO scalability as a function of the size of the input graph. Thereby we have plotted the results for performing external memory bisimulation partitioning on two types of initial partitions; namely the structural summary partition and the partition based on rank and label. Table 6.2 shows the accuracy of these initial partitions and the resulting local refined partitions with respect to the bisimulation partition. In Table 6.3 the number of local collisions occurring during bisimulation partitioning is shown.

EXPERIMENT 2: In Figure 6.6 we have plotted the runtime and IO scalability as a function of the size of the input graph.

EXPERIMENT 3: In Figure 6.7 we have plotted the runtime and IO scalability as a function of the amount of memory each data structure and sort operation is allowed to use.

EXPERIMENT 4: In Table 6.4 we have given an overview of the scaling factor and other measurements of the size of an XML document (document size and the number of extracted nodes). In Figure 6.8 we have plotted the runtime and IO scalability as a function of the scaling factor used by `xmlgen` to generate XML documents.



(a) Runtime performance

(b) IO complexity

Figure 6.5: Results of EXPERIMENT 1. In Figure 6.5a the runtime is plotted as function of the size of the input. In Figure 6.5b the number of IOs performed per node and edge is plotted as function of the size of the input. We have plotted results for individual programs and for entire tool chains.

Nodes	$\texttt{exbisim}_{ss}$ Initial [%]	Refined [%]	*(max)*	$\texttt{exbisim}_{rl}$ Initial [%]	Refined [%]	*(max)*
$1.0 \cdot 10^8$	99.86	100.00	15	0.02	99.96	33,391
$2.5 \cdot 10^8$	99.84	100.00	21	0.01	99.98	83,047
$4.0 \cdot 10^8$	99.83	100.00	24	0.00	99.92	132,730
$5.5 \cdot 10^8$	99.82	100.00	25	0.00	99.97	182,525
$7.0 \cdot 10^8$	99.81	100.00	25	0.00	99.95	232,238
$8.5 \cdot 10^8$	99.81	100.00	25	0.00	99.97	282,043
$1.0 \cdot 10^9$	99.80	100.00	26	0.00	99.96	331,530

Table 6.2: Results of EXPERIMENT 1. This table provides a comparison of some statistics on the initial partition and on the locally refined partitions of partition blocks in the initial partition. The $\texttt{exbisim}_{ss}$ columns show these statistics for external memory bisimulation partitioning on a structural summary partition. The $\texttt{exbisim}_{rl}$ columns show these statistics for external memory bisimulation partitioning on an initial partition based on rank and label. Each column 'Initial' shows the size of the initial partition as a percentage of the size of the bisimulation partition; each column 'Refined' shows the cumulative size of all refined partitions of initial partition block; and each column '*(max)*' shows the maximum number of partition blocks wherein an initial partition block is refined. The number of bisimulation partitions in the output ranged from $70 \cdot 10^6$ for the smallest graph to $708 \cdot 10^6$ for the largest graph.

Nodes	$\texttt{exbisim}_{ss}$ Collisions	*(max)*	$\texttt{exbisim}_{rl}$ Collisions	*(max)*
$1.0 \cdot 10^8$	0	0	28,709	3
$2.5 \cdot 10^8$	0	0	44,245	4
$4.0 \cdot 10^8$	0	0	216,302	6
$5.5 \cdot 10^8$	0	0	108,743	5
$7.0 \cdot 10^8$	0	0	236,386	5
$8.5 \cdot 10^8$	0	0	203,958	6
$1.0 \cdot 10^9$	0	0	278,918	6

Table 6.3: Results of EXPERIMENT 1. Some statistics on the usage of local partition decision structures during bisimulation partitioning. The $\texttt{exbisim}_{ss}$ columns show these statistics for external memory bisimulation partitioning on a structural summary partition. The $\texttt{exbisim}_{rl}$ columns show these statistics for external memory bisimulation partitioning on an initial partition based on rank and label. Each column 'Collision' shows the cumulative amount of collisions for all local partition decision structures. Each column '*(max)*' shows the maximum number of collisions per local partition decision structure. A collision corresponds to the addition of a new entry to a non-empty local partition decision structure.

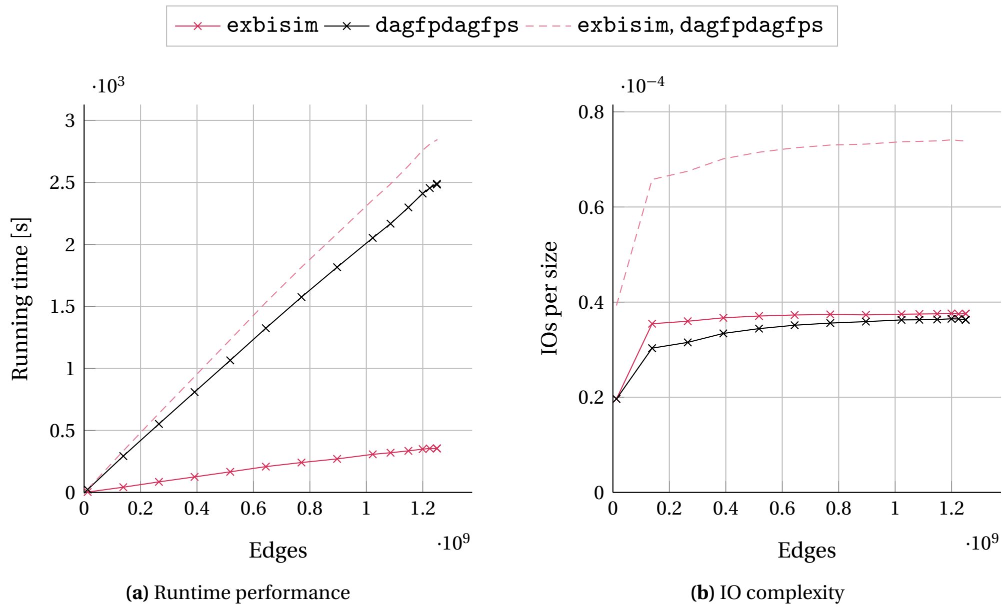


Figure 6.6: Results of EXPERIMENT 2. In Figure 6.6a the runtime is plotted as function of the number of edges in the input. In Figure 6.6b the number of IOs performed per node and edge is plotted as function of the number of edges in the input. We have plotted results for individual programs and for the entire tool chain.

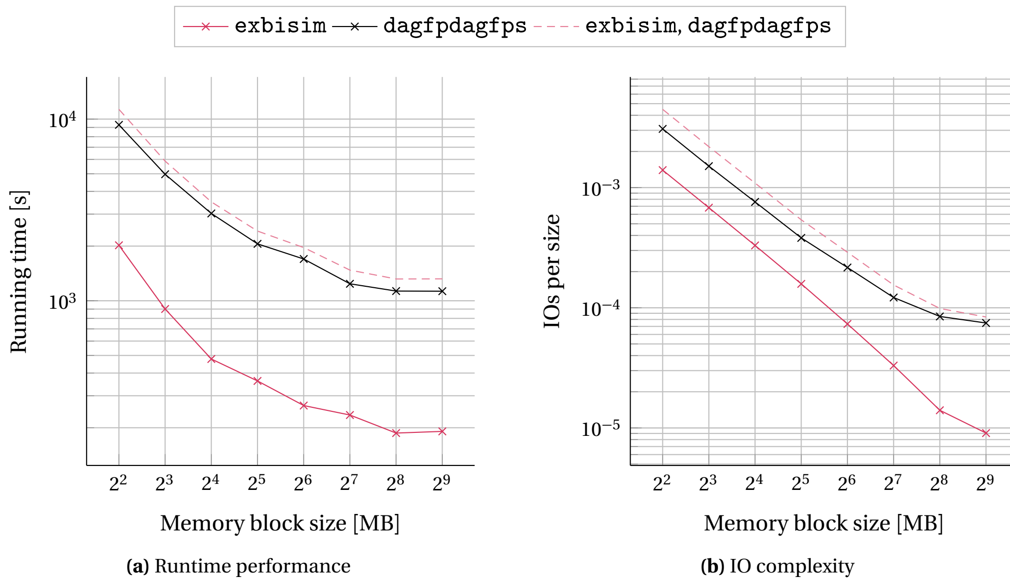
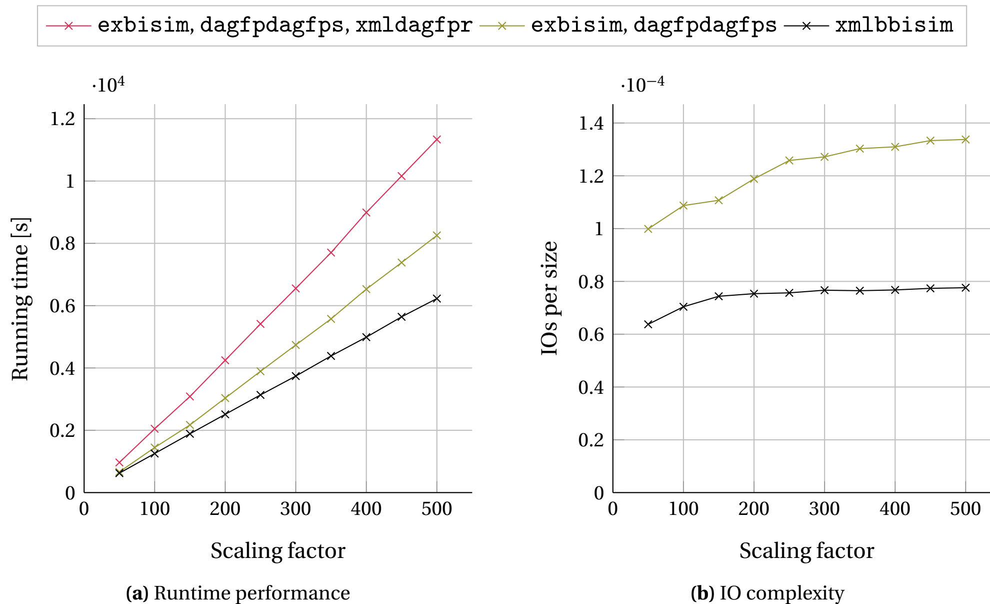


Figure 6.7: Results of EXPERIMENT 3. In Figure 6.7a the runtime is plotted as function of the amount of memory data structures and sort operations can use. In Figure 6.7b the number of IOs performed per node and edge is plotted as function of the amount of memory data structures and sort operations can use. We have plotted results for individual programs and for the entire tool chain.

Scaling factor	Document size [GB]	Extracted nodes
50	5.6	$1.03 \cdot 10^8$
100	11.1	$2.05 \cdot 10^8$
150	16.7	$3.08 \cdot 10^8$
200	22.3	$4.11 \cdot 10^8$
250	27.9	$5.13 \cdot 10^8$
300	33.5	$6.16 \cdot 10^8$
350	39.0	$7.19 \cdot 10^8$
400	44.6	$8.21 \cdot 10^8$
450	50.2	$9.24 \cdot 10^8$
500	55.8	$1.03 \cdot 10^9$

Table 6.4: Results of EXPERIMENT 4. Statistics on the size of generated XML documents. Thereby the column 'Scaling factor' shows the scaling factor used as input for the `xmlgen` program. The column 'Document size' shows the size of the produced XML documents. The column 'Extracted nodes' shows the number of XML elements and attributes that are represented by nodes in the tree representation of the XML document. On the nodes in this tree representation we calculate the backward bisimulation partition.



(a) Runtime performance

(b) IO complexity

Figure 6.8: Results of EXPERIMENT 4. In Figure 6.8a the runtime is plotted as function of the scaling factor. In Figure 6.8b the number of IOs performed per node and edge is plotted as function of the scaling factor. We have plotted results for the entire tool chain. Thereby we have not plotted the IO cost for running `xmldagfpr`; as we have only measured IO complexity of presented algorithms. Note that the functionality of `xmldagfprs` can easily be integrated into `dagfpdagfps`.

6.4 Conclusions

In Section 6.2 we have stated three questions related to the behavior of the external memory bisimulation partitioning algorithm. With the results presented in Section 6.3 we can answer these questions.

DOES ALGORITHM 3.6 BEHAVE AS ONE CAN EXPECT FROM THE THEORY? From the results of EXPERIMENT 1

and EXPERIMENT 2 we can conclude that the runtime of the algorithm scales near-linear with respect to the number of nodes and with respect to the number of edges. We also see that the algorithm stays IO efficient for any amount of nodes and edges. The IO cost comes close to the absolute lower bound and stays well below a single IO per node or a single IO per edge.

From the results of EXPERIMENT 3 we see that the algorithm operates IO efficient for all the tested amounts of available internal memory. We do however see that within the range 4MB to 128MB increasing the amount of available memory does strongly increases the performance and reduces IO cost of the algorithm. Additional increasing of the size of internal memory beyond 128MB does not result in an evenly strong increase of performance or reduction of the IO cost.

DOES ALGORITHM 3.6 BENEFIT FROM A GOOD INITIAL PARTITION? The results of EXPERIMENT 1 clearly show that on a unstructured input (as generated by `gen`) the rank and label based initial partition is a very bad predictor for the bisimulation partition. As such the structural summary partition; a highly structured initial partition; provides better performance and lower IO cost during partitioning. The cause of this improvement can be explained by the reduction of collisions.

HOW DOES ALGORITHM 3.6 COMPARE WITH MORE OPTIMIZED APPROACHES? The comparison made in EXPERIMENT 4 clearly show that a fine-tuned algorithm for bisimulation partitioning (in this case an optimized backward bisimulation partitioning algorithm for XML documents) does improve performance.

Chapter 7

CONCLUSION

The objective of this thesis was to develop techniques for constructing and maintaining bisimulation partitions of large directed acyclic graphs. These techniques can for example be used for constructing graph databases wherein graphs are indexed using bisimulation-based indices. In Section 1.2 we have stated two research goals for this objective. The first goal is the development of external memory bisimulation partition algorithms. The main theoretical contributions for this goal are presented in Chapter 3. The second goal is the investigation of partition maintenance in an external memory setting. Therefore the main contributions are presented in Chapter 4. We can already conclude that the goals set out in the problem statement are met. A more detailed conclusion and overview of our results is presented in Section 7.1. We end our work by providing an overview of possible future (research) work; this overview is presented in Section 7.2.

7.1 Overview

The first goal of our work is the development of external memory bisimulation partition algorithms. Therefore we have developed an external memory bisimulation partitioning algorithm for directed acyclic graphs. This algorithm is designed as an online algorithm; whereby the algorithm processes the nodes in the order wherein they are stored. Thereby the algorithm decides for each node in which bisimulation partition block it should be placed; this by utilizing only the information gathered on the node. For making this decision we have introduced the partition decision structure. We have analyzed access patterns to this partition decision structure; and we have introduced several hash-based techniques to optimize these access patterns. This leads to an expected IO efficient bisimulation partitioning algorithm.

The developed algorithm has an expected IO complexity of $O(\textsc{Sort}(|N|)+\textsc{Sort}(|E|)+\mathrm{PQ}(|E|))$. For the worst case IO complexity we have presented two possible implementations. The first implementation uses a list to store the partition decision structure; this implementation has worst case complexity of $O(\textsc{Sort}(|N|)+\textsc{Sort}(|E|)+\mathrm{PQ}(|E|)+\textsc{Scan}(|N||E_{\downarrow}|))$. The other implementation uses a string B-tree to store the partition decision structure thereby achieving a worst case IO complexity of $O(\textsc{Sort}(|N|)+\textsc{Sort}(|E|)+\mathrm{PQ}(|E|)+|N|\log_B(|N_{\downarrow}|))$. With a small experimental study we have verified that the algorithm is IO efficient in practice; this for graphs containing up to 10^9 nodes and $4\cdot10^9$ edges. Thereby the experiment did not show any reasons to question the scalability of the algorithm for graphs with much more than 10^9 nodes.

The external memory bisimulation algorithm is accompanied with a theoretical framework. This framework is also applicable to other (partitioning) problems. We have presented some of these applications for indexing XML documents. Thereby we have shown efficient external memory algorithms for constructing the 1-index, the F&B-index, and the A(k)-index of XML documents. The 1-index construction algorithm has a worst case IO complexity of $O(\textsc{Sort}(|N|)+\mathrm{PQ}(|N|))$. We have compared this 1-index construction algorithm with the external memory bisimulation partition algorithm. Both algorithms could easily handle XML documents with a size of 55.8GB (representing trees containing 10^9 nodes). Thereby the experiment shows that the specialized 1-index construction algorithm was faster than the general approach. For the construction of an F&B-index we have provided details on how the external memory bisimulation partitioning algorithm can be utilized. For the construction of an A(k)-

index we have provided an algorithm with worst case IO complexity of $O(\text{Scan}(k+|N|)+\text{Sort}(k|N|))$.

The second goal of our work is the investigation of partition maintenance in an external memory setting. For partition maintenance we have provided theoretical analysis of the worst case complexity. We have also provided practical approaches for updating partitions; thereby utilizing the data structures and algorithms developed for the external memory bisimulation partitioning algorithm. Lastly we have shown how partition maintenance can play a role for XML documents. Thereby we have given an overview of useful XML document update operations. We have also provided details on how to adapt the described general partition maintenance approaches for maintaining the 1-index, the F&B-index, and the A(k)-index of XML documents.

We have introduced update complexity and index update complexity for studying the lower bounds on the cost of partition maintenance. These complexities express the minimum number of changes one needs to make to a bisimulation partition and/or to a graph index to update it after the underlying graph has been modified. Within this framework we have studied the lower bound on the cost of four operations; namely subgraph addition, subgraph removal, edge addition, and edge removal.

We have proven that the lower bound on the cost for performing index updates for adding or removing a subgraph G_s from a graph index is $O(|N_s|+|E_{s\downarrow}|)$. For supporting subgraph addition we have developed an approach around the idea of maximum-merge graphs. Thereby we use node bisimilarity to relate index nodes of several graph indices. We can then use transitivity of node bisimilarity to relate all bisimilar equivalent graph nodes. We have provided two sketches for subgraph addition algorithms based on maximum-merge graphs. The first approach has expected IO complexity of $O(\text{Sort}(|N_s|)+\text{Sort}(|N_\downarrow|+|N_{s\downarrow}|)+\text{Sort}(|E_\downarrow|+|E_{s\downarrow}|)+\text{PQ}(|E_\downarrow|+|E_{s\downarrow}|))$. This approach is only fast whenever the graphs have small indices. As such this approach is best suited whenever the graphs are highly structured. The second approach has worst case IO complexity of $O(\text{Sort}(|N_s|)+\text{Sort}(|E_{s\downarrow}|)+\text{PQ}(|E_{s\downarrow}|)+|N_{s\downarrow}|\log_B(|N_\downarrow^{\mathcal{M}}|))$. This approach is only fast when the subgraph is small. As such this approach is best suited additions of graphs that have a small index. For subgraph removal we have provided a sketch for an algorithm with IO complexity of $O(\text{Scan}(|E_{s\downarrow}|)+|N_{s\downarrow}|\log_B(|N_\downarrow|))$. This approach is also best suited for removal of subgraphs that have a small index.

For edge changes we have proven that the lower bound on the worst case index update cost is $\Theta(|N|+|E|)$; this already rules out a general and practically fast algorithm for edge updates. Using the idea of edge change propagation we have provided a sketch for an edge update algorithm with worst case IO complexity of $O(|N_n|\log_B(|N|)+\text{Sort}(|N_n|)+\text{Sort}(|E'(N_n)|)+\text{Scan}(|E(N_n)|)+\text{PQ}(|E'(N_n)|))$. Thereby N_n is the set of affected nodes; for an updated edge (n,m) this includes node n and its ancestors. Edge updates using this propagation approach are only fast when the number of affected nodes is small.

From the results we can conclude that the first goal has been reached; we have developed an IO efficient bisimulation partitioning algorithm and we have validated its performance in a small scale experiment. For the secondary goal; investigating partition maintenance; we have shown lower bounds and upper bounds on the complexity of partition maintenance. Thereby we have concluded that it is unlikely that a general approach for partition maintenance exist that is (asymptotically) faster than recalculating the entire bisimulation partition. We have however provided approaches that are applicable in some practical settings.

7.2 Future work

This work focuses on bisimulation partitioning and partition maintenance. Thereby our work resulted in useful theory and algorithms for large directed acyclic graphs. Based on these results there are many topics for further research; we shall list several suggestions.

7.2.1 Practical implementations and verification

We have only performed a small scale experiment to see if the developed external memory bisimulation partitioning algorithm behaves as expected. Thereby we have primarily used random generated graphs that do not represent real-life data and we haven't paid any attention to the way the results of our algorithm are stored. We haven't looked at partition maintenance; as the performance of partition

maintenance is highly dependent on the used external memory data structures for storing the graph index. These data structures used for storing the graph index highly depend on the purpose of the index. This all leads to the following question: How does bisimulation partitioning and partition maintenance perform in a practical setting?

One of such practical settings is an XML database. Thereby several questions and issues pop up during implementation. Which data structures should one use for storing the constructed bisimulation-based index (the graph index, the 1-index, the A(k)-index, or the F&B-index)? What is the impact of this data structure on the cost of constructing the index? What is the impact of this data structure on the cost of maintaining the index? What is the impact of this data structure on the cost of using the index for querying the XML database?

For practical purposes; such as querying XML databases; one can expect to need more than just the list of (node identifier, partition block identifier)-pairs produced by the algorithms in this paper. There is already some work on storing and querying XML document indices in external memory [WJW+05]. One thus might want to investigate how the structure presented in [WJW+05] can be combined with the index construction algorithms presented in this paper in an efficient way.

There might however be better approaches. In our work we have already mentioned some data structures that can be used to represent graph indices. We have for example seen how the string B-tree can be used for storing partition decision structures. When the string B-tree is combined with the A(k)-index on XML documents we see that this structure can provide all A(i)-indices for $i \leq k$. In the same way we can utilize the string B-tree for storing the 1-index. An alternative approach is thus to investigate if the data structures used in this paper can be utilized for performing queries.

7.2.2 Practical partition maintenance

We have provided several approaches to perform partition maintenance after some update. None of these approaches where truly satisfactory. We have seen that edge changes are just expensive. For subgraph additions and removals we have provided some reasons why general efficient approaches are unlikely to exist.

Subgraph addition and removal are very useful operations; as they model the construction of a graph index over a collection of graphs. As such fast approaches are of use. One can investigate if (practically) faster approaches exist when the algorithms for subgraph addition and subgraph removal use specialized data structures. One might for example consider a buffered subgraph addition or removal approach wherein changes are only made to those parts of the index when these parts are accessed. Note that such a buffered approach does in no way invalidate the comments made in Section 4.4. These delayed updates need to happen at some point; this without the guarantee that other updates are included.

This buffered approach might however be fast in practice as it hides the cost of a single update by 'smearing' it out over many operations. Furthermore for certain data loads this buffered approach is able to group several updates to the same part of the index graph and perform them as one update; thereby reducing the overall IO cost. A possible implementation for the graph index might be to buffer the update in the buffer for a virtual leaf node that has all leaf nodes as parent nodes. Relevant parts of the update are then pushed forward to the right index node when needed. The complexity of this approach lies in performing these forward pushes. For an XML document indexed with the 1-index this complexity does not exist, as each index node has only one child node per label. We can thus easily push down the right parts of an update to the right child index node based on this label.

One might also want to look at the performance of the partition maintenance algorithms in a setting wherein the external memory is provided by solid-state drives. Good solid-state drives have much lower seek latencies then ordinary hard disk drives; as such random reads and writes of disk blocks are at the same level of performance as sequential read and writes of disk blocks. Using solid-state drives thus allows one to use a more random pattern on IO behavior. Thereby solid-state drives can provide higher performance then ordinary hard disk drives; especially for problems that are not expected to be very sequential in nature (such as partition maintenance). For optimal usage of solid-state drives one should further take into account that solid-state drives often have non-symmetric read and write speeds. For

solid state drives the write speed is often less than the read speed; moreover the write speed can degrade when the solid state drive gets older or when free space becomes sparse on the solid state drive.

7.2.3 Internal memory bisimulation

Our external memory bisimulation partitioning algorithm can easily be used as an internal memory algorithm for bisimulation partitioning of directed acyclic graphs. We thus provide an alternative for fast internal algorithms [PT87, DPP01, GBH10]. Thereby we directly note that our approach is asymptotically slower than other approaches for directed acyclic graphs.

We can however improve our approach for an internal memory setting; thereby the first thing to remove would be the priority queue. Also the performed sorting operations are not needed; as one can directly use a hash table for refinement. Thereby we however need another way for efficiently removing duplicates during node-decision and node-hash value evaluation. We also need a fast approach for getting node-decision and node-hash values in some standard format that allows fast comparisons and hashing. After tackling these issues the resulting algorithm can be compared with other bisimulation partitioning algorithms to see how it performs in practice.

We can also use the theory and algorithms developed in this work for the development of a hybrid algorithm. We can for example improve the performance of the Paige and Tarjan algorithm by providing a good initial partition. Part of our theory focused on constructing such a good initial partition. For constructing a good initial partition for directed graphs we however need to extend node-hash values. For node-hash values to work for directed graphs we need to look at cycles (or more general: nodes that are part of some strongly connected component[1]). A simple solution would be to assign every node in a strongly connected component the node-hash value $S_{\mathcal{H}}$. One can try to further improve performance by processing initial partition blocks in 'reverse-topological order'. This order can be calculated by replacing each strongly connected component by a single node S.

For better results one should construct better node-hash values for strongly connected components. Therefore one could extend the node-hash value by utilizing properties of the strongly connected component. One could also set up a general theory for node-bisimulation values and node-hash values in a directed graph setting. For all cases the node-hash based initial partition can be at least as good as a simple label partitioning. Thereby the node-hash based initial partition can increasing performance of the Paige and Tarjan algorithm by reducing the number of refinements each partition block needs.

7.2.4 Generalizing bisimulation partitioning

We have only provided an external memory bisimulation partitioning algorithm for reverse-topological ordered directed acyclic graphs. In Subsection 3.6.1 we have already argued that a general IO efficient solution for directed acyclic graphs and directed graphs might be hard to find. One research topic is finding (heuristic) approaches for general classes of graphs.

We have already developed some technology that can be applied on certain directed graphs. When a large directed (acyclic) graph has a small index; and this large graph and can be subdivided in small subgraphs; then we can use the subgraph addition method to construct the entire bisimulation partition. Thereby we can calculate maximum-merge graphs for each subgraph and the index (in memory, using for example Paige and Tarjan [PT87]) and thereby constructing a full index over the entire graph. This can be generalized by grouping nodes starting at leaf nodes and ending at root nodes. Thereby groups are processed one at a time. After processing the nodes in a group we can replace all nodes placed in the same partition block by a single node; thereby reducing the size of the entire graph during bisimulation partitioning.

Also other extensions into less-restricted cases are useful. One approach is to perform Paige and Tarjan, this approach is taken in [HDFJ10]. The approach taken in [HDFJ10] does however not seem to result in a fully IO efficient algorithm; we can however improve this method in several ways. The main way is by constructing a good initial partition beforehand. We have already discusses how this can be achieved for internal memory bisimulation partition algorithms. We can also try to construct a partially-external memory algorithm; for example by allowing each graph node and/or index node to keep a fixed

[1] A strongly connected component is a set of nodes S; whereby every node $n \in S$ has a path to all nodes in the set.

amount of data in internal memory. This can introduce a useful algorithm for many practical cases as modern computer systems can already keep information of billions of nodes in internal memory.

7.2.5 Generalizing index construction

For the construction of the 1-index, the F&B-index and the A(k)-index we have only presented solutions for XML documents. These solutions can easily be generalized to work on all trees and forests. But only the 1-index can easily be generalized to topological ordered directed acyclic graphs (by using Algorithm 3.6). For practical purposes one can be interested in the construction of the F&B-index and/or the A(k)-index on directed (acyclic) graphs. We have already noted the difficulties one can expect for directed graph bisimulation partitioning; we shall thus restrict ourselves to topological ordered directed acyclic graphs.

For the A(k)-index we have presented an approach for trees; this approach cannot be generalized to general graphs as k-trace equivalence is not equivalent to backward node k-bisimilarity for non-tree nodes. For directed acyclic graphs we thus need a more complicated algorithm. For the A(k)-index one cannot utilize the backward rank to localize backward k-bisimulation partitioning. One can however use an inductive approach to localize backward k-bisimulation partitioning. Thereby we perform k steps. At step j we calculate the backward j-bisimulation partition; we do so by first sending backward j-1-bisimulation partition block identifiers from parent nodes to child nodes. On these values we can construct a node-hash value. Ordering on these node-hash values would localize the backward node j-bisimulation partitioning calculation. The details for such an approach need to be further worked out; we can however already say that for large values of k the approach will result in many steps and thus will be much slower then calculating the 1-index.

The F&B-index is another index with practical value. For trees we have seen that the F&B-index is easily calculated by calculating the F+B-index. For directed acyclic graphs this approach will not work. We have however already given a native approach. We have described that repeated forward and backward bisimulation partition refinement will result in the F&B bisimulation partition. We have already provided IO efficient algorithms for performing each refinement step. This solution does however not seem very satisfactory. More efficient approaches should use an alternative form of processing. This might however be hard; as F&B bisimilarity of a node depends on all ascendants and descendants of the node. As such processing the node in one time-forward processing step seems impossible; a single time-forward processing step can collect information from either the ancestors of a node or the descendants of a node; not from both. Finding a general fast algorithm for the construction of the F&B-index on directed acyclic graphs is thus also non-trivial.

BIBLIOGRAPHY

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu, *Data on the web: from relations to semistructured data and xml*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[ACLZ11] D. Ajwani, A. Cosgaya-Lozano, and N. Zeh, *Engineering a topological sorting algorithm for massive graphs*, International Workshop on Algorithm Engineering and Experiments (ALENEX 2011), 2011.

[BGVW00] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook, *On external memory graph traversal*, Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), SODA '00, Society for Industrial and Applied Mathematics, 2000, pp. 859–860.

[DCXB11] Jintian Deng, Byron Choi, Jianliang Xu, and Sourav S. Bhowmick, *Optimizing incremental maintenance of minimal bisimulation of cyclic graphs*, DASFAA (1), 2011, pp. 543–557.

[DPP01] Agostino Dovier, Carla Piazza, and Alberto Policriti, *A fast bisimulation algorithm*, CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification (London, UK), Springer-Verlag, 2001, pp. 79–90.

[FG99] Paolo Ferragina and Roberto Grossi, *The string b-tree: a new data structure for string search in external memory and its applications*, J. ACM **46** (1999), 236–280.

[GBH10] Nils Grimsmo, Truls Amundsen Bjørklund, and Magnus Lie Hetland, *Linear computation of the maximum simultaneous forward and backward bisimulation for node-labeled trees*, XSym, 2010, pp. 18–32.

[GC07] Gang Gou and Rada Chirkova, *Efficiently querying large xml data repositories: A survey*, IEEE Trans. on Knowl. and Data Eng. **19** (2007), 1381–1403.

[HDFJ10] Ala' Hawash, Anton Deik, Bilal Farraj, and Mustafa Jarrar, *Towards query optimization for the data web: disk-based algorithms: trace equivalence and bisimilarity*, Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications (New York, NY, USA), ISWSA '10, ACM, 2010, pp. 17:1–17:7.

[KBNK02] Raghav Kaushik, Philip Bohannon, Jeffrey F Naughton, and Henry F Korth, *Covering indexes for branching path queries*, Proceedings of the 2002 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '02, ACM, 2002, pp. 133–144.

[KBNS02] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy, *Updates for structure indexes*, Proceedings of the 28th international conference on Very Large Data Bases, VLDB '02, VLDB Endowment, 2002, pp. 239–250.

[KSBG02] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, *Exploiting local similarity for indexing paths in graph-structured data*, Data Engineering, 2002. Proceedings. 18th International Conference on, 2002, pp. 129–140.

[MS99] Tova Milo and Dan Suciu, *Index structures for path expressions*, Database Theory – ICDT'99 (Catriel Beeri and Peter Buneman, eds.), Lecture Notes in Computer Science, vol. 1540, Springer Berlin / Heidelberg, 1999, pp. 277–295.

[MSS03] Ulrich Meyer, Peter Sanders, and Jop Sibeyn (eds.), *Algorithms for memory hierarchies: advanced lectures*, Springer-Verlag, Berlin, Heidelberg, 2003.

[PT87] Robert Paige and Robert E. Tarjan, *Three partition refinement algorithms*, SIAM J. Comput. **16** (1987), no. 6, 973–989.

[Sah07] Diptikalyan Saha, *An incremental bisimulation algorithm*, Proceedings of the 27th international conference on Foundations of software technology and theoretical computer science (Berlin, Heidelberg), FSTTCS'07, Springer-Verlag, 2007, pp. 204–215.

[San09] Davide Sangiorgi, *On the origins of bisimulation and coinduction*, ACM Trans. Program. Lang. Syst. **31** (2009), 15:1–15:41.

[WJW+05] Wei Wang, Haifeng Jiang, Hongzhi Wang, Xuemin Lin, Hongjun Lu, and Jianzhong Li, *Efficient processing of xml path queries using the disk-based f&b index*, Proceedings of the 31st international conference on Very large data bases, VLDB '05, VLDB Endowment, 2005, pp. 145–156.

[Zeh02] Norbert Zeh, *I/o-efficient graph algorithms*, 2002.