External Memory k-Bisimulation Reduction of Big Graphs

Yongming Luo*, George H.L. Fletcher*, Jan Hidders†, Yuqing Wu‡, Paul De Bra*

*Eindhoven University of Technology, The Netherlands, {y.luo@, g.h.l.fletcher@, debra@win.}tue.nl
†Delft University of Technology, The Netherlands, a.j.h.hidders@tudelft.nl
‡Indiana University, Bloomington, USA, yuqwu@cs.indiana.edu

ABSTRACT

In this paper, we present, to our knowledge, the first known I/O efficient solutions for computing the k-bisimulation partition of a massive directed graph, and performing maintenance of such a partition upon updates to the underlying graph. Ubiquitous in the theory and application of graph data, bisimulation is a robust notion of node equivalence which intuitively groups together nodes in a graph which share fundamental structural features. k-bisimulation is the standard variant of bisimulation where the topological features of nodes are only considered within a local neighborhood of radius $k \geqslant 0$.

The I/O cost of our partition construction algorithm is bounded by $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$, while our maintenance algorithms are bounded by $O(k \cdot sort(|E_t|) + k \cdot sort(|N_t|))$. The space complexity bounds are $O(|N_t| + |E_t|)$ and $O(k \cdot |N_t| + k \cdot |E_t|)$, resp. Here, $|E_t|$ and $|N_t|$ are the number of disk pages occupied by the input graph's edge set and node set, resp., and $sort(n)$ and $scan(n)$ are the cost of sorting and scanning, resp., a file occupying n pages in external memory. Empirical analysis on a variety of massive real-world and synthetic graph datasets shows that our algorithms perform efficiently in practice, scaling gracefully as graphs grow in size.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; E.1 [**Data Structures**]: Graphs and networks

Keywords

graph bisimulation; structural index; external memory algorithm

1. INTRODUCTION

Massive graph-structured datasets are becoming increasingly common in a wide range of applications. Examples such as social networks, linked open data, and biological networks have drawn much attention in both industry and academic research. In reasoning over graphs, a fundamental and ubiquitous notion is that of bisimulation, which is a characterization of when two nodes in a graph share basic structural properties such as neighborhood connectivity. Bisimulation arises and is widely adopted in a surprisingly large range of research fields [28]. In data management, bisimulation partitioning (i.e., grouping together bisimilar nodes in order to reduce graph size) is often a basic step in indexing semi-structured datasets [23], and also finds fundamental applications in RDF [25] and general graph data (e.g., compression [5, 9], query processing [17], data analytics [8, 31]).

It is often the case that bisimulation reductions of real graphs result in partitions which are too refined for effective use. Hence, a notion of localized k-bisimulation has proven to be quite successful in data management applications (e.g., [11, 17, 26, 32]). k-bisimulation is the variant of bisimulation where topological features of nodes are only considered within a local neighborhood of radius $k \geqslant 0$. With a pay-as-you-go nature, k-bisimulation is cheaper to compute and maintain, cost adjustable, and faithfully representative of the bisimulation partition within the local neighborhood.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.
Copyright 2013 ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505752.

State of the art

Algorithms for bisimulation partitioning have been studied for decades, with well-known algorithms such as those of Paige and Tarjan [24] and more recent work (e.g., [7]), having effective theoretical behavior.

In practice, however, state-of-the-art solutions face a critical challenge: all known approaches for computing bisimulation are internal-memory based solutions[1]. As such, their inherently random memory access patterns do not translate to efficient I/O-bound solutions, where it is crucial to avoid such access patterns. Consequently, when processing graphs which do not fit entirely in main memory the performance of these algorithms decreases drastically.

The reality is that, in practice, many graphs of interest are too large to be processed in main memory. Indeed, massive graphs are now ubiquitous [8, 15]. Furthermore, the size of graphs will only continue to grow as technologies for generating and capturing data continue to improve and proliferate. We can safely conclude that it will become increasingly infeasible to apply existing internal-memory bisimulation partition algorithms in practice.

To process real graphs, therefore, we must necessarily turn to either external memory, distributed, or parallel solutions. There has been some work on parallel (e.g., [27, 30]) and distributed (e.g., [4]) approaches to bisimulation computation, and, recently, external memory solutions on restricted acyclic and tree-structured graphs [16]. However, to our knowledge there is no known effective solution for

[1]With the single exception of Hellings et al. [16] which we discuss below in Section 3.2.

computing bisimulation and k-bisimulation partitions on arbitrary graph structures in external memory. Such an algorithm would not only enable us to process big graphs on single machines, but also provide an essential step for parallel and distributed solutions (e.g., MapReduce [20]) to further scale their performance on real graphs. As noted in paper [20] and many other researches (e.g., [19]), in many cases, single machine external memory algorithms are more competitive than distributed algorithms due to their lack of communication overhead and their effective use of available infrastructure. Therefore, the study of external memory solutions is clearly warranted.

Our contributions

Given these motivations, we have studied external memory solutions for reasoning about k-bisimulation on arbitrary graphs. In this paper, we present the results of our study, which makes the following high-level contributions.

- We present the first known I/O efficient external memory based algorithm for constructing the k-bisimulation partition of a disk-resident graph. The I/O cost of this algorithm is bounded by $O(k \cdot \mathit{sort}(|E_t|) + k \cdot \mathit{scan}(|N_t|) + \mathit{sort}(|N_t|))$, with space complexity $O(|N_t| + |E_t|)$, where $|E_t|$ and $|N_t|$ are the number of disk pages occupied by the input graph's edge set and node set, resp., and $\mathit{sort}(n)$ and $\mathit{scan}(n)$ are the cost of sorting and scanning, resp., a file occupying n pages in external memory.
- We present the first known I/O efficient external memory based algorithms for performing maintenance on a disk-resident k-bisimulation graph partition, with I/O cost bounded by $O(k \cdot \mathit{sort}(|E_t|) + k \cdot \mathit{sort}(|N_t|))$, and space complexity $O(k \cdot |N_t| + k \cdot |E_t|)$.
- We present the results of an extensive empirical analysis of our solutions on a variety of massive real-world and synthetic graph datasets, showing that our algorithms not only perform efficiently, but also scale gracefully as graphs grow in size. For example, the 10-bisimulation partition of a graph having 1.4 billion edges can be computed with our solution within a day on commodity hardware, while this would take weeks, if not months, for a traditional in-memory algorithm to accomplish in the same environment.

The rest of the paper is organized as follows. In the next section we give our basic definitions and data structures used. We then describe in Section 3 our solution for constructing k-bisimulation partition. Next, Section 4 presents algorithms for keeping an existing partition up to date, in the face of updates to the underlying graph. Section 5 presents the results of our empirical study of all algorithms. We then conclude in Section 6 with a discussion of future directions for research.

2. PRELIMINARIES

2.1 Data model and definitions

Our data model is that of finite directed node- and edge-labeled graphs $\langle N, E, \lambda_N, \lambda_E\rangle$, where N is a finite set of nodes, $E \subseteq N \times N$ is a set of edges, λ_N is a function from N to a set of node labels $\mathcal{L}_N$, and λ_E is a function from E to a set of edge labels $\mathcal{L}_E$.

Definition 1. *Let k be a non-negative integer and $G = \langle N, E, \lambda_N, \lambda_E\rangle$ be a graph. Nodes $u, v \in N$ are called* k-bisimilar *(denoted as $u \approx^k v$), iff the following holds:*

1. $\lambda_N(u) = \lambda_N(v)$,
2. *if $k > 0$, then $\forall u' \in N[(u, u') \in E \Rightarrow \exists v' \in N[(v, v') \in E,\ u' \approx^{k-1} v'$ and $\lambda_E(u, u') = \lambda_E(v, v')$*[1]*]], and*
3. *if $k > 0$, then $\forall v' \in N[(v, v') \in E \Rightarrow \exists u' \in N[(u, u') \in E,\ v' \approx^{k-1} u'$ and $\lambda_E(v, v') = \lambda_E(u, u')]]$.*

It can be easily shown that the *k-bisimilar* relation is an equivalence relation.

We illustrate Definition 1 with an example. Consider the graph given in Figure 1. It is a small social network graph, in which nodes 1 and 2 are 0- and 1- bisimilar but not 2-bisimilar.

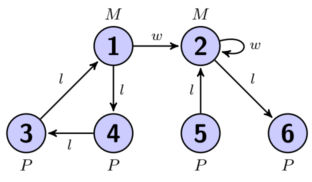


Figure 1: Example graph of a social network, where nodes 1 and 2 have label M (short for "manager"), and the other nodes have label P (short for "people"). The edge label l is short for "likes", while w is short for "works for".

Recall from Section 1 that our interest in this paper is in computing the k-bisimulation partition of a massive graph, and performing maintenance on the result under updates to the original graph. By *massive*, we mean that both the set of nodes and the set of edges of the graph are too big to fit into main memory. By a *partition* of the graph, we mean an assignment of each node u of the graph to a *partition block*, which is the unique subset of nodes in the graph of which the members are k-bisimilar to u.

In particular, we are interested in constructing partition "identifiers."

Definition 2. *A* k-partition identifier *for graph $G = \langle N, E, \lambda_N, \lambda_E\rangle$ and $k \geq 0$ is a set of $k+1$ functions $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ such that, for each $0 \leq i \leq k$, pId_i is a function from N to the integers, and, for all nodes $u, v \in N$, it holds that $pId_i(u) = pId_i(v)$ iff $u \approx^i v$.*

A fundamental tool in our reasoning about k-bisimulation is the notion of node signatures.

Definition 3. *Let $G = \langle N, E, \lambda_N, \lambda_E\rangle$ be a graph, $k \geq 0$, and $\mathcal{P} = \{pId_0, \ldots, pId_k\}$ be a k-partition identifier for G. The* k-bisimulation signature *of node $u \in N$ is the pair $sig_k(u) = (pId_0(u), L)$ where:*

$$L = \begin{cases} \emptyset & \text{if } k = 0, \\ \{(\lambda_E(u, u'), pId_{k-1}(u')) \mid (u, u') \in E\} & \text{if } k > 0. \end{cases}$$

We then have the following fact.

Proposition 1. *$pId_k(u) = pId_k(v)$ iff $sig_k(u) = sig_k(v)$ $(k \geq 0)$.*

Proof omitted.

Proposition 1 is the basis of all algorithms in this paper. The basic idea is that a node's k-bisimulation partition block can be determined by its k-bisimulation signature, which in turn is determined by the $(k-1)$-bisimulation partition of the graph. Intuitively, in order to compute the k-bisimulation partition, we compute the graph's *j-bisimulation* $(0 \leq j \leq k)$ partitions bottom-up, starting from $j = 0$. We call each such intermediate computation the *iteration j computation*.

It is straightforward to show that the k-bisimulation partition of a graph is unique. Hence, in the sequel, we can

[1]Note that we use $\lambda_E(u, u')$, instead of $\lambda_E((u, u'))$, for ease of readability.

safely talk about k-partition identifiers as unique objects. Also, note that we will use integer node identifier values to designate nodes in N. Therefore, in the following discussions the functions sig_k and pId_k both could take node identifiers (i.e., integers) as input.

Table 1: *k-bisimulation* for the example graph in Figure 1 ($k = 0, 1, 2$)

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, F)\}$	H
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{\}$	F	$B, \{\}$	K

Table 1 shows one way of assigning k-bisimulation ($k = 0, 1, 2$) partition identifiers and signatures for the example graph in Figure 1, where the *nId* denotes the unique identifier for each node, and $pId_i(nId)$ and $sig_j(nId)$ ($0 \leq i \leq 2$ and $0 < j \leq 2$) are presented accordingly. For $k = 0$, nodes are grouped into two partitions by node labels (given identifiers A and B). Then for $k = 1, 2$, signatures are constructed according to Definition 3, and then distinct partition identifiers are assigned to distinct signatures, following Proposition 1.

2.2 Data structures

We assume that graphs are saved on disk in the form of fixed column tables (node set as table N_t and edge set as table E_t). We also assume that these tables can have several copies sorted on different columns. In later discussions, we will use the notation $X.y$ to refer to column y of table X.

We have the following possible attributes for N_t:

nId	node identifier (note that this is the same as row identifier in the table; we leave this attribute here for clarity of the discussion).
nLabel	node label
pId_{old_nId}	*bisimulation* partition identifier for the given *nId* from last computation iteration
pId_{new_nId}	*bisimulation* partition identifier for the given *nId* from the current computation iteration
pId_{j_nId}	j *bisimulation* partition identifier for the given *nId* ($j = 0, 1, \ldots, k$)

and for E_t:

sId	source node identifier
tId	target node identifier
eLabel	edge label
pId_{old_tId}	*bisimulation* partition identifier for the given *tId* from last computation iteration

We further assume that we have a *signature storage facility* S, which stores the mapping between signatures and their corresponding partition identifiers. S is a data structure having only one idempotent function called *S.insert()*. For node $u \in N$, *S.insert()* takes $sig_j(u)$ ($0 \leq j \leq k$) as input, and provides $pId_j(u)$ as output. Essentially *S.insert()* implements the one to one mapping function from sig_j to pId_j. The implementation details of S will be discussed in Section 3.2.

For ease of discussion and investigation, we assume in what follows that the N_t and E_t are each just one file sequentially filled with fixed length records. Moreover, in this paper we make use of sort merge join to the extent possible, since it is a very basic way to achieve I/O efficient results. However, many possibilities could be explored for implementing these data structures (e.g., indexing techniques) and join algorithms to further optimize our presented results. We leave such investigations open for future research.

Finally, we also assume that we have a (possibly external memory based) priority queue available. In our empirical study below, we use the off-the-shelf I/O efficient priority queue implementation provided by the open source STXXL library [6].

2.3 Cost model

Since our focus is on disk-resident datasets, we use standard I/O complexity notions to analyze our algorithms [1]. The primary concern here is to minimize the number of I/Os needed to complete the task at hand.

Suppose we have table X, space to hold B disk pages in internal memory, and X occupies $|X|$ pages on disk. In what follows, we will use the following notation:

- $sort(|X|)$ denotes the number of I/Os when sorting table X on some given column(s). This will take $2|X|(1 + \lceil log_{B-1} \lceil \frac{|X|}{B} \rceil \rceil)$ I/Os for a standard external memory based merge sort.
- $scan(|X|)$ denotes the number of I/Os when scanning over table X. This will take $|X|$ I/Os.

3. CONSTRUCTING K-BISIMULATION PARTITIONS

We present our algorithm for k-bisimulation partition computation in Algorithm 1. The algorithm is inspired by Proposition 1, meaning for each node in the input graph, to construct its signature and find a one-to-one mapping number (partition identifier) for that signature.

In iteration $j = 0$, we assign distinct partition identifiers to nodes based on their *nLabel*s. For other iterations $j > 0$, our algorithm mainly performs two things for each node ID $uId \in \pi_{nId}(N_t)$ (line 14 to 17): (1) construct $sig_j(uId)$; and (2) insert $sig_j(uId)$ to S, record the returning $pId_j(uId)$ in the corresponding row in N_t. To prepare the necessary information for constructing $sig_j(uId)$, we need to fill in the missing columns of E_t (line 5 to 10). Several scans and sorts on tables are involved for each iteration. Note that some operations in the algorithm can be merged as one in practice. We present them separately just to make the presentation clearer. A detailed description is given in Section 3.1.

3.1 Details of Algorithm 1 (Build_Bisim())

Input and output. The input variables of Algorithm 1 are node table N_t, edge table E_t and k, which is the degree of local bisimilarity from Definition 1. The output variables are N_t and E_t. The schema of N_t is (*nId*, *nLabel*, pId_{0_nId}, pId_{old_nId}, pId_{new_nId}); the schema of E_t is (*sId*, *eLabel*, *tId*, pId_{old_tId}).

k* = 0, *line 2 to 4. According to Definition 1, $k = 0$ means nodes having the same labels should be assigned the same partition identifier. We achieve this by sorting N_t on *nLabel* column. When scanning N_t, for each new *nLabel* we encounter, we assign a new integer (e.g., a predefined counter) to the corresponding *nId*, filling it in the pId_{0_nId} and pId_{new_nId} columns. This will take $O(sort(|N_t|)) + O(scan(|N_t|))$ I/Os. Using a hash map could achieve the same goal as well, with the same I/O upper bound.

k* > 0, *line 5 to 18. For $k > 0$, we first perform a recursive call to the algorithm, ensuring we work in a bottom-up manner. For iteration 1 ($k = 1$), we sort N_t

Algorithm 1 Compute the *k-bisimulation* equivalence classes of a graph

```
 1: procedure BUILD_BISIM($N_t$, $E_t$, $k$)
 2:     if $k = 0$ then
 3:         fill in the $pId_{0\_nId}$ and $pId_{new\_nId}$ columns of $N_t$        ▷ $O(sort(|N_t|)) + O(scan(|N_t|))$
 4:         return ($N_t$, $E_t$)
 5:     ($N_t$,$E_t$)←BUILD_BISIM($N_t$, $E_t$, $k-1$)        ▷ $k > 0$, recursive call
 6:     if $k = 1$ then
 7:         $N_t \leftarrow$ sort($N_t$) by $nId$        ▷ $O(sort(|N_t|))$
 8:         $E_t \leftarrow$ sort($E_t$) by $tId$        ▷ $O(sort(|E_t|))$
 9:     scan $N_t$, move content of column $pId_{new\_nId}$ to $pId_{old\_nId}$        ▷ $O(scan(|N_t|))$
10:     fill in the $pId_{old\_tId}$ column of $E_t$        ▷ $O(scan(|E_t|)) + O(scan(|N_t|))$
11:     initialize $S$
12:     $F \leftarrow \pi_\alpha(E_t)$, where $\alpha = (sId, eLabel, pId_{old\_tId})$
13:     $F \leftarrow$ sort($F$) by $sId, eLabel, pId_{old\_tId}$, removing duplicates        ▷ $O(sort(|E_t|))$
14:     for each $uId \in \pi_{nId}(N_t)$ do        ▷ overall $O(scan(|E_t|)) + O(scan(|N_t|)) + cost\ of\ S$
15:         construct $sig_k(uId)$ from $F$        ▷ merge join with $F$
16:         $pId_k(uId) \leftarrow S.insert(sig_k(uId))$
17:         record $pId_k(uId)$ in $N_t.pId_{new\_nId}$ where $nId = uId$
18:     return ($N_t$, $E_t$)
```

and E_t on *nId* and *tId*, preparing them for later merge join operations. The algorithm's idea is to construct the signature of each node in order to distinguish it from other nodes according to the k-bisimilar relation. If we can properly fill in the pId_{old_tId} column of E_t, and join it with N_t on *nId*=*sId*, the information combined from columns $\{pId_{0\_nId}, eLabel, pId_{old\_tId}\}$ is enough for constructing the signature. The column *eLabel* is already filled in before algorithm starts. The column pId_{0_nId} is filled in during iteration 0 (line 2 to 4). The column pId_{old_tId} is filled in during each iteration $j > 0$ (line 10). Then for each node ID $uId \in N_t$, we get its $sig_k(uId)$, insert it to S in an I/O efficient way, getting $pId_k(uId)$ in return, and then placing this value in the pId_{new_nId} column of N_t.

At line 10 of Algorithm 1, to fill in the pId_{old_tId} column of E_t, we conduct a sort merge join of E_t and N_t (since both tables are sorted properly in iteration 1), replacing the content of pId_{old_tId} in E_t with pId_{old_nId} in N_t.

At line 15 of Algorithm 1, we sequentially construct the signature $sig_k(uId)$ for each $uId \in \pi_{nId}(N_t)$ according to Definition 3, and get the corresponding $pId_k(uId)$ (using *S.insert()*). All $pId_k(uId)$ will be written back to the pId_{new_nId} column of N_t (where *nId*=*uId*) right after, so that there is no random access to N_t. Note that although by definition sig_k is a set, we construct $sig_k(uId)$ as a string, maintaining elements of the set in sorted order. It is both an easy way for storing a set and handy for implementing S later on (e.g., using a trie).

3.2 Further discussion of Algorithm 1

Example run. If we assume the numbering scheme for S is a self-increased counter across iterations, Table 1 would be the intermediate results for running Algorithm 1 on the example graph in Figure 1 ($k = 2$), and Table 2 gives the final output of the algorithm.

Table 2: Output of Algorithm 1 on example graph in Figure 1 ($k = 2$)

(a) N_t

nId	*nLabel*	pId_{0_nId}	pId_{old_nId}	pId_{new_nId}
1	*M*	*A*	*C*	*G*
2	*M*	*A*	*C*	*H*
3	*P*	*B*	*D*	*I*
4	*P*	*B*	*E*	*J*
5	*P*	*B*	*D*	*I*
6	*P*	*B*	*F*	*K*

(b) E_t

sId	*eLabel*	*tId*	pId_{old_tId}
3	*l*	1	*C*
1	*w*	2	*C*
2	*w*	2	*C*
5	*l*	2	*C*
4	*l*	3	*D*
1	*l*	4	*E*
2	*l*	6	*F*

Early stopping condition. It is not always necessary to let the algorithm run k iterations. Indeed, it can be shown (proof omitted) that after a bounded number of computation iterations, Algorithm 1 would achieve the full (i.e., classical non-localized) bisimulation partition. We could detect this by simply checking the partition size each iteration produces. If two consecutive iterations produce the same number of partition blocks, this means that the algorithm already achieves the full bisimulation partition, and therefore it is safe to terminate the algorithm.

Data structures for S. The signature storage facility S clearly plays an important role in Algorithm 1. In principle, any data structure that permits an efficient set-equality check will be sufficient. Trie and dictionary are such data structures, for instance. During our experiments, we see that in many of the cases, partition sizes are small and the signatures are short, for which a main memory based data structure is enough. In other cases, signature length could reach several million and partition size into tens of millions, then we need some external memory based solution for S. We could, for example, sort all signatures from F in an I/O efficient way [2], then when scanning these signatures, partition identifiers are assigned. In this case, the overall cost of the *S.insert()* operation could still be bounded by $O(sort(|E_t|))$. Other disk based solutions, such as disk-based tries (e.g., String B-Tree [10] or [13]) or inverted files (e.g., [22]) could also be considered.

In our experiments we use BerkeleyDB (B-Tree or Hash index) to mimic a trie, which, as we show in the experimental results, has acceptable empirical behavior.

Complexity and correctness. We have the following characterization of Algorithm 1.

THEOREM 1. *Let $k \geq 0$ and $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph. Algorithm 1 computes the k-bisimulation partition of G with I/O complexity of $O(k \cdot sort(|E_t|) + k \cdot scan(|N_t|) + sort(|N_t|))$, and space complexity of $O(|N_t| + |E_t|)$.*

Proof omitted.

Differences with Hellings et al. As indicated in Section 1, the only known solutions for computing bisimulation on graphs in external memory are those of Hellings et al. [16], with I/O complexity of $O(sort(|N_t| + |E_t|))$. There are two critical differences between their work and ours. (1)

Targeting different problems. The solutions of Hellings et al. are designed specifically for the special case of acyclic graphs. Our approach does not rely on such structure, computing bisimulation regardless of the presence or absence of cycles in the graph. (2) *Using different techniques.* Hellings et al. compute partition blocks level by level, starting from the leaf nodes of the graph. Our approach constructs all partition blocks at each iteration, using data structures and processing strategies which are not tied to any (a)cyclic structure in the graph. In particular, the techniques of Hellings et al. do not generalize to graphs having cyclic structure.

4. MAINTENANCE OF K-BISIMULATION PARTITIONS

It is easy to show that any edge and node updates on a graph can potentially change the complete k-bisimulation partition of the graph. Therefore, in the worst case, the lower bound of such maintenance cost is the cost of recomputing the k-bisimulation partition from scratch. However, when dealing with real graphs, as we shall see in Section 5, in many cases there is still hope to use data structures such as S and priority queue to maintain the correct partition result instead of recomputing everything. In this section we propose several algorithms for this purpose.

For maintenance algorithms we assume that we have constructed the k-bisimulation partition of graph $G = \langle N, E, \lambda_N, \lambda_E \rangle$, where, as before, G's N_t and E_t are stored on disk, containing the historical information kept in N_t (Table 3); E_t is the same as in Algorithm 1, but has two copies with sort orders (*sId,tId*) and (*tId,sId*) to boost performance. We use E_{tst} and E_{tts} to refer to each of these copies.

Table 3: N_t for maintenance algorithms

nId	*nLabel*	pId_{0_nId}	pId_{1_nId}	...	pId_{k_nId}

We further assume that we save the signature storage facility S on disk, which we use and update throughout the maintenance process.

The maintenance problem includes the following subproblems.

Change k. If k increases, we carry out another iteration of computation. If k decreases, the result can be returned directly since we keep the history information in N_t.

*Add a set of new nodes (*ADD_NODES()*).* When adding a set of new nodes, we assume the new nodes are isolated, stored in the *newNodes* table, which has the same schema as N_t, and that $|newNodes| = O(|N_t|)$. We first sort N_t and *newNodes* by *nLabel*, then perform a merge join on the *nLabel* column to fill in the pId_{0_nId} column of *newNodes* for all the existing *nLabel*. For the missing ones, we request a new *pId* for each of the new *nLabel*. Then we get the $pId_1, \ldots, pId_k$ of the *newNodes* by inserting its pId_0 to S. At the end we append the whole *newNodes* to N_t. The I/O complexity of ADD_NODES() is bounded by $O(sort(|N_t|))$.

*Add a set of new edges (*ADD_EDGES()*).* For adding a set of edges, we assume that the edges are added between existing nodes. If this is not the case, we first call procedure ADD_NODES(). The new edges are stored in the *newEdges* table, having the same schema as E_t. For inserting one edge (s, l, t) to G, the potential changes are to $sig_j(s)$ $(1 \le j \le k)$, as well as those signatures of all ancestors of s within k steps. So the main work is to detect whether there is some change in $sig_j(s)$ and propagate those change(s) to its parent nodes' signatures in later iterations. We use a priority queue ***pQueue*** to record and process such changes in a systematic, level-wise manner. For some node ID *uId* and iteration j, *pQueue* stores the pair *(j,uId)* as priority reference. Then whenever we dequeue one element from *pQueue*, we get the smallest node ID from the lowest iteration (lowest priority reference). Therefore *pQueue* indicates those nodes whose signatures could change in each iteration level (from 1 up to k).

At the beginning of the algorithm, we enqueue (j, s) to *pQueue* $(\forall (s, l, t) \in E_t, 0 < j \le k)$. Then, while *pQueue* is not empty, we dequeue the list of (j, uId) pairs with the same j out of the queue, construct the new signature of each such *uId*, insert it to S, and compare the returning $pId_j(uId)$ with the old pId_{j_nId} value of *uId*. If the *pId* remains the same as the old one, we continue; if it changes, we record $pId_j(uId)$ in N_t, and enqueue all $(j + 1, vId)$ pairs to *pQueue* where $vId \in \pi_{sId}(\sigma_{tId=uId}(E_t))$. Pseudo code is given in Algorithm 2, and a detailed discussion is in Section 4.1.

Deletions. Deletions follow a similar idea to insertions. For example, when removing an edge (s, l, t), it is the same idea as adding one. We also (potentially) modify the signature of s, propagating changes to its ancestors via *pQueue*, then the reasoning is the same. When removing a node, we first remove each incoming edge and each outgoing edge for that node. Then we remove the node from N_t.

4.1 Details of Algorithm 2 (ADD_EDGES())

Input and output. The input variables of Algorithm 2 are node table N_t, edge tables E_{tst} and E_{tts}, the signature storage facility S, the new edge set *newEdges* and k. The output variables of Algorithm 2 are N_t, E_{tst}, E_{tts} and S. N_t's schema is given in Table 3, while E_{tst}, E_{tts} and *newEdges*'s schema is the same as E_t in Algorithm 1.

k* = 0*, line 2 to 3 of Algorithm 2. For $k = 0$, since all nodes' information is properly filled (including the pId_{0_nId} column in N_t), we only need to add new rows to E_{tst} and E_{tts} according to *newEdges*.

k* > 0*, line 4 to 20 of Algorithm 2. For $k > 0$, for each iteration, which is indicated by j in the algorithm, we need to (1) find out the potential nodes whose signatures could have changed; (2) check whether these signatures have been changed or not; and, (3) propagate any such changes to the parents of these nodes. To record the potential nodes and to perform the propagation, we use a priority queue *pQueue*. To check signature changes, we reuse the signature storage facility S.

When adding a new edge $(s, l, t) \in newEdges$ to the graph, all $sig_j(s)$ $(j > 0)$ have the potential to change, and hence we add all pairs (j, s), for $j \in \{1, \ldots, k\}$, to *pQueue*, indicating that we need to check the signature of s in every iteration (line 7 to 8). For each iteration $j > 0$, we dequeue from *pQueue* all node IDs in the smallest iteration j, remove duplicates, and save them to a temporary table M, so that M contains in sorted order all node IDs whose signatures would change in iteration j. Then we create an extra table F, preparing for signature constructions. This is achieved by performing a merge join of E_{tst} and M (where $E_{tst}.sId \in M$). Then we fill in $F.pId_{old\_tId}$ column, as in Algorithm 1.

Algorithm 2 Add a set of new edges to existing *k-bisimulation* partition

```
 1: procedure ADD_EDGES($N_t$, $E_{tst}$, $E_{tts}$, $S$, $newEdges$, $k$)        ▷ $newEdges$ is a table of new edges
 2:     if $k = 0$ then
 3:         merge $newEdges$ into $E_{tst}$ and $E_{tts}$                            ▷ $O(sort(|E_t|))$
 4:     else                                                                         ▷ $k > 0$
 5:         $N_t$← sort($N_t$) by $nId$                                              ▷ $O(sort(|N_t|))$
 6:         create empty priority queue $pQueue$                                      ▷ overall $O(sort(|N_t|))$
 7:         for $j \in \{1, \ldots, k\}$ and $(s, l, t) \in newEdges$ do
 8:             enqueue $(j, s)$ to $pQueue$
 9:         merge $newEdges$ into $E_{tst}$ and $E_{tts}$, fill in the $pId_{old\_tId}$ column   ▷ $O(sort(|E_t|))$
10:         while $pQueue$ is not empty do
11:             dequeue all pairs $(j, uId)$ from $pQueue$ with the same (i.e., smallest) $j$ value, save all distinct $uId$ to $M$
12:             $F \leftarrow \sigma_{sId \in M}(E_{tst})$                           ▷ merge join, $O(scan(|N_t|) + scan(|E_t|))$
13:             fill in the $pId_{old\_tId}$ column of $F$      ▷ $O(scan(|N_t|) + O(sort(|E_t|)) + O(scan(|E_t|)))$
14:             $H \leftarrow \pi_\alpha(F)$, where $\alpha$=($sId$, $eLabel$, $pId_{old\_tId}$)
15:             $H \leftarrow$ sort $H$ on $sId$, $eLabel$, $pId_{old\_tId}$, and remove duplicates   ▷ $O(sort(|E_t|))$
16:             for all $uId \in M$ do     ▷ scan $M$, $N_t$ and $H$, overall $O(scan(|N_t|)) + O(scan(|E_t|))$ + cost of $S$
17:                 construct $sig_j(uId)$ from $H$
18:                 $pId_j(uId) \leftarrow S.insert(sig_j(uId))$
19:                 if $pId_j(uId)$ is not the same as the corresponding value in $N_t.pId_{j\_nId}$ then
20:                     propagate changes to $N_t$ and $pQueue$                          ▷ $O(scan(|N_t|)) + O(scan(|E_t|))$
21:     return ($N_t$, $E_{tst}$, $E_{tts}$, $S$)
```

After projection on the (sId, $eLabel$, pId_{old_tId}) of F and removing duplicates, we get H (line 15), and are ready to construct the signatures. For each $uId \in M$, we construct $sig_j(uId)$ according to the signature definition. The idea of constructing the nodes' signatures is the same as line 15 of Algorithm 1, only in this case we are not considering every node but only those appearing in *pQueue* (and later in M).

We then call *S.insert(*sig_j*(uId))* for all such *uId*. If S returns the same $pId_j(uId)$ as recorded in $N_t.pId_{j\_nId}$, nothing will happen; otherwise we change the $N_t.pId_{j\_nId}$ entry of *uId* accordingly, and propagate the changes to *pQueue*. If $j < k$, we add all parents of *uId* to *pQueue* to indicate that we will check these nodes' signatures in the $j + 1$ iteration.

4.2 Further discussion of Algorithm 2

Example run. We present different behaviors of Algorithm 2 using two examples. Here we will extend the graph from Figure 1 as in Figure 2. The dashed lines in this figure indicate the two edges which we will add in our examples.

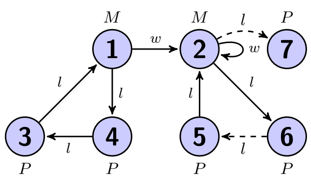


Figure 2: Updates on the example graph

First suppose we add edge $(2, l, 7)$ to the original graph of Figure 1, where node 7 is a new node with label P. Table 4 shows the resulting partition after this insertion. The new/changed part of the table is indicated in gray. When the algorithm starts, (1, 2) and (2, 2) are added to *pQueue*. Then after checking each of these, the algorithm finds no change in node 2's signature, therefore no change propagates, and the algorithm stops. We see that comparing with Table 1, the only thing that changes is to add one more row (node 7) to the table. Since node 7 does not have outgoing edges, adding one edge that points into node 7 will not change any existing nodes's signature. Node 7 belongs to the group of node 6, and no other node changes group membership.

In the second case, suppose we add edge $(6, l, 5)$ to the original graph of Figure 1. The algorithm first add (1, 6) and

Table 4: 2-*bisimulation* for the example graph after edge insertion $(2, l, 7)$

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, F)\}$	H
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{\}$	F	$B, \{\}$	K
7	B	$B, \{\}$	F	$B, \{\}$	K

(2, 6) to *pQueue*. Then in iteration 1, the algorithm detects that the signature of node 6 does change, and therefore adds one new pair (2, 2) to *pQueue*. In iteration 2, both node 2 and node 6's signatures are checked, and they are both changed. We see that in Table 5 $pId_2(1)$ and $pId_2(2)$ become the same, while $pId_2(6)$ changes from K to I.

Table 5: 2-*bisimulation* for the example graph after edge insertion $(6, l, 5)$

nId	$pId_0(nId)$	$sig_1(nId)$	$pId_1(nId)$	$sig_2(nId)$	$pId_2(nId)$
1	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
2	A	$A, \{(w, A), (l, B)\}$	C	$A, \{(w, C), (l, E)\}$	G
3	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
4	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J
5	B	$B, \{(l, A)\}$	D	$B, \{(l, C)\}$	I
6	B	$B, \{(l, B)\}$	E	$B, \{(l, D)\}$	J

Complexity and correctness. We have the following characterization of Algorithm 2.

THEOREM 2. *Let $G = \langle N, E, \lambda_N, \lambda_E \rangle$ be a graph and $k \geq 0$. After adding a set of new edges to G, Algorithm 2 correctly updates the k-bisimulation partition of G with I/O complexity of $O(k \cdot sort(|E_t|) + k \cdot sort(|N_t|))$, and space complexity of $O(k \cdot |N_t| + k \cdot |E_t|)$.*

Proof omitted.

When to switch back to Algorithm 1. As we will see in our empirical study (Section 5.3.4), it is not always beneficial to use Algorithm 2, since it performs extra work in each iteration. Heuristics could be adopted to decide when to switch back to Algorithm 1. For example, if at a certain iteration, most of the nodes are placed into *pQueue*, it is more beneficial to switch back to Algorithm 1. This could be done by simply checking the size of *pQueue* at the beginning of each iteration.

5. EMPIRICAL ANALYSIS

In this section we present the results of an in-depth experimental study of our algorithms. After introducing our set-up, we show the performance of the algorithms on both synthetic and real datasets. In these experiments, various aspects of the algorithms are investigated while other settings are fixed. A thorough analysis of the k-bisimulation result itself can be found in paper [21].

5.1 Experiment setting

Environment. The following experiments are run on a machine with 2.27 GHz Intel Xeon (L5520, 8192KB cache) processor, 12GB main memory, running Fedora 14 (64-bit) Linux. We use C++ to implement all the algorithms, using GCC 4.4.4 as the compiler. We use the open-source STXXL library [6] to construct the tables and perform the external memory sorting, and use Berkeley DB to implement S. One S is used for all computation iterations (as discussed in Section 3.2). In the experiments we do not exploit any parallelism and restrain ourselves with predefined buffer sizes. We record the running time as well as the I/O volume between the buffer and the disk system. Therefore, the performance (time) of the experiments are comparable to a commodity PC, and the I/O volume can be repeated on other systems. In the following experiments, we set both the STXXL buffer and Berkeley DB buffer to be 128MB, if not otherwise indicated. Please note that we run experiments for the Twitter dataset on a different machine (Intel Xeon E5520, 2.27 GHz, 8192KB cache, 70G main memory, same OS) for limited disk space reason, using a 512MB/512MB buffer setting.

Datasets. To prove the practicability of the algorithms, we experiment with various graph datasets. The datasets are collected from public repositories, ranging from synthetic data to real-world data, from several million of edges to more than 1.4 billion edges. In Table 6 we give a description of the datasets, as well as some simple statistics of them. All datasets are accessed on 15 May 2012. Note that due to space limitation, in the following we show the experiment results on a subset of the datasets when the result is representative enough.

Table 6: Description and statistics of the experiment datasets

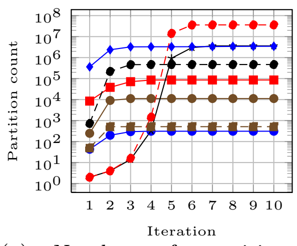
Data Name	Description	Node Count	Edge Count	Label on
Jamendo	A repository of music metadata in RDF format[1]	486,320	1,049,647	Edge
LinkedMDB	A repository of movie metadata in RDF format [14]	2,330,695	6,147,996	Edge
DBLP	An RDF format DBLP dump[2]	23,000,670	50,203,406	Edge
WikiLinks	A page-to-page linking graph of Wikipedia[3]	5,710,993	130,160,392	None
DBPedia	An early RDF dump of DBPedia[4]	38,615,135	115,305,444	Edge
Twitter	A following relationship graph of Twitter [18]	41,652,230	1,468,365,182	None
SP2B	A RDF data generator for arbitrarily large DBLP-like data [29]	280,908,393	500,000,912	Edge
BSBM	A RDF data generator for e-commerce use case [3]	8,886,078	34,872,182	Edge

[1]http://dbtune.org/jamendo/

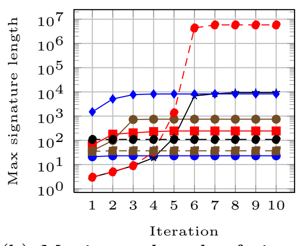
[2]http://thedatahub.org/dataset/l3s-dblp

[3]http://haselgrove.id.au/wikipedia.htm

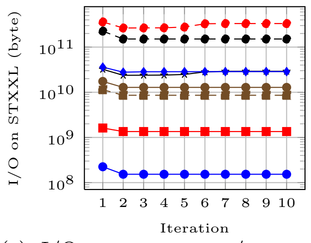
[4]http://www.cs.vu.nl/~pmika/swc/btc.html



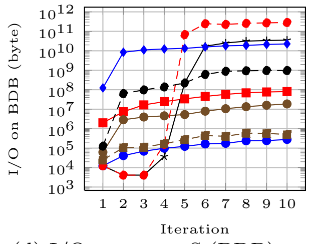
(a) Number of partition blocks



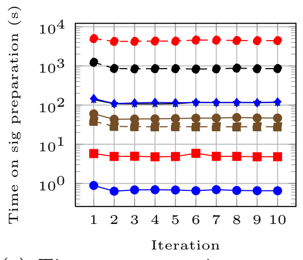
(b) Maximum length of signatures (integer count)



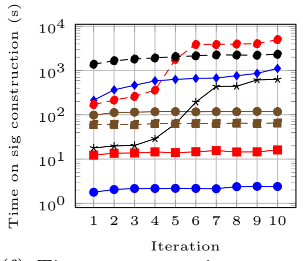
(c) I/O spent on sort/scan (STXXL)



(d) I/O spent on S (BDB)



(e) Time spent on signature preparation



(f) Time spent on signature construction and insertion

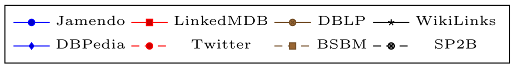


Figure 3: Experiment results for Algorithm 1 for real and synthetic datasets for each iteration ($k = 10$)

5.2 Experiments on the k-bisimulation construction algorithm (Build_Bisim())

In Figure 3 we show the experiment results for Algorithm 1 on all datasets. We compute the 10-bisimulation (i.e., $k = 10$) of these datasets, and measure many aspects of the running behavior for each iteration. Concerning time measurement, we run every experiment 5 times and take the average number. S uses BerkleyDB's B-Tree index in this experiment.

In Figure 3a, we show the number of partition blocks every iteration produces for all datasets. We see that the numbers vary from one dataset to another, where the difference is sometimes more than an order of magnitude, and interestingly, does not directly relate to the size of the dataset. In certain cases (e.g., Twitter) partition size is quite large. Moreover, many of the datasets (e.g., Jamendo, LinkedMDB, DBLP, etc.) reach full bisimulation after 5 iterations. In fact, all datasets (including Twitter) get sufficient partition result after 5 iterations of computation. Here we can reasonably argue that even for Twitter dataset, the partition results after 5 iterations are too refined (e.g., (partition count)/(node count) > 0.8).

Figure 3b shows the maximum length of signatures for each iteration. We observe that the signature length is usually quite short, especially comparing with the size of the graph. But there are still cases (e.g., Twitter) that the signature becomes very long (more than 1 million integers), which stresses the need for an I/O efficient solution for S.

Note that the synthetic datasets, such as BSBM and SP2B, reach their full bisimulation partition after 3 iterations of computations, and have rather short signatures, indicating that they are highly structured.

Figures 3c and 3d show the I/O volume spent on sorting/scanning (STXXL) and on interacting with S (Berkeley DB). We see for most of the datasets, there is no dramatic change cross different iterations. But for Wikilinks and Twitter, the two datasets which have very few partition blocks at the beginning and many at the end, there is a noticeable difference on S for different iterations. In this case I/O on S becomes a comparable factor with sort and scan (I/O on STXXL).

Figure 3e shows the time spent on preparing the signature (line 5 to 13 in Algorithm 1) for each iteration, which is quite stable for all datasets. Figure 3f shows the time on constructing the signature and insert into S (line 14 to 17 in Algorithm 1). In this case datasets with higher degrees tend to cost more time in later iterations, which correlate with their longer signatures and larger number of partition blocks. For all datasets, however, the operations on constructing and looking for signature are the dominant factor for each iteration. This brings us to think about further optimization tasks on construction of signature and implementation of S.

We can conclude that the algorithm is practical to use. It can process a graph with 100 million edges (e.g., WikiLinks and DBPedia) in under 700 seconds for one iteration, and performance scales (almost) linearly with the number of nodes and edges.

5.2.1 *Different implementations of* S

As we mentioned in Section 3.2, S could be implemented in several ways. we compare the overall I/O performance of BUILD_BISIM() using B-Tree and Hash indexes for S on several datasets. We notice that the B-Tree implementation slightly outperforms Hash Index for all datasets. This is most likely due to small caching effects and locality of references during construction of the signatures.

5.2.2 *The effect of different buffer sizes*

We allocate two buffers, one for scan and sort (STXXL buffer in our case), one for S (BerkeleyDB buffer in our case), in order to analyze the impact of buffer size on our algorithms. To illustrate, we take the DBPedia dataset since it is large enough to show buffer effects. For the sort/scan setting, we set the buffer size ranging from 16MB to 512MB, while keeping the S buffer to 128MB, recording the I/O between the buffer and the disk system. From Figure 4a we see that bigger buffer does improve the performance. But since we only gain in the external memory sorting part, a certain amount of I/Os is inevitable for each iteration. Note that the reason why iteration 1 has higher I/O cost is that in iteration 1 extra sorts on N_t and E_t are performed.

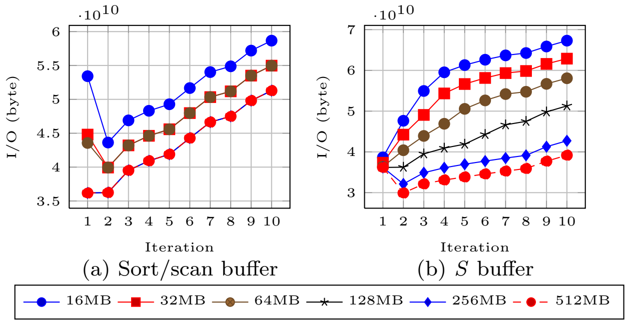


Figure 4: I/O for different buffer size setting for sort/scan and S ($k = 10$)

For the setting on S, we set the buffer size ranging from 16MB to 512MB, while keeping the sort/scan buffer to be 128MB, recording the I/O of the buffer to the disk system. From Figure 4b we also see that more buffer brings less I/O, as expected. However, in this case the buffer size change has a bigger impact on the I/O performance. This indicates that if we have a certain amount of memory space, it is more beneficial to allocate more memory to the S buffer than to the sort/scan buffer. Note that S buffer also shows quite high hit ratio during execution (more than 0.98 for DBPedia in all settings).

5.2.3 *Scalability*

In order to measure how well the algorithm scales, we generate different size of SP2B datasets (edge count 1M, 5M, 10M, 50M, 100M, 500M), and measure the I/O and elapsed time for each dataset. In Figure 5 we see that the time spent on each edge is on the order of 10^{-5} seconds, and the I/O spent on each edge is under 4000 bytes (which is one typical disk page size). The algorithm's performance scales (almost) linearly with the data size.

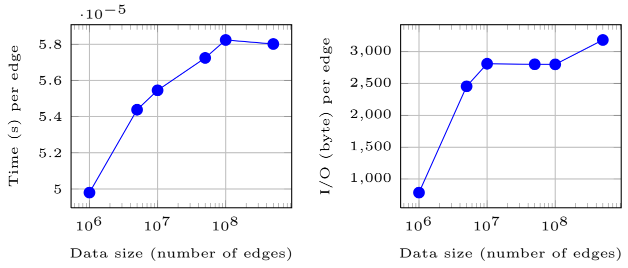


Figure 5: Time and I/O spent on each edge on average ($k = 10$)

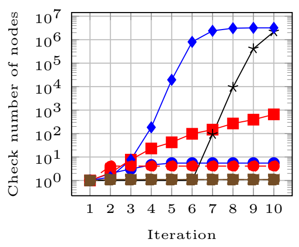
5.3 Experiments on the edge update algorithm (ADD_EDGES())

Edge updates are common operations for graph data. For our datasets, adding one edge means to add a link between two wiki pages (WikiLinks), to add more information to one publication or author (DBLP), to follow one more person (Twitter) and so on. Sometimes we would like to also add several edges together at once. So in this subsection we test the performance of Algorithm 2 (ADD_EDGES()), first adding a single edge and then adding a set of edges.

5.3.1 *Observations on single edge update*

To create the dataset for testing, we randomly take one edge from the edge set, perform BUILD_BISIM() on the rest of the dataset, and apply ADD_EDGES() on this edge. We believe the edge selection is more natural this way, since it take into account the distribution of edges among nodes. We repeat the experiment 10 times and take the average of the measured numbers. In Figure 6a we show how many nodes are checked for adding one edge to the graph in each iteration. In Figure 6b we show how many nodes actually change their partition IDs in each iteration. From the figures we see that the behavior varies for different datasets; graphs that have larger degrees tend to propagate more changes to later iterations, which complies with our intuition.

Since there is a chance that many nodes are changed but they may all belong to a certain set of partitions, we also examine how many partitions change their members in each iteration. We see that the behavior is closely related to that of Figure 6b.


(a) Number of nodes being checked

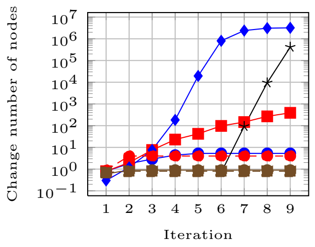

(b) Number of nodes that change partitions

Figure 6: Experiment on ADD_EDGES() when $k = 10$

5.3.2 *Comparison of* BUILD_BISIM() *and* ADD_EDGES() *(single edge update)*

After edge insertion, if there is no update algorithm available, the only choice to get the k-bisimulation partition is to execute the BUILD_BISIM() from scratch on the new dataset. So this would be the baseline for the ADD_EDGES() algorithm to compare. In the following we compare the overall I/O and time (Figure 7) of the two algorithms. We see that indeed the ADD_EDGES() algorithm always achieves a better performance than using BUILD_BISIM() to recompute the k-bisimulation partition result from scratch, with up to an order of magnitude improvement.

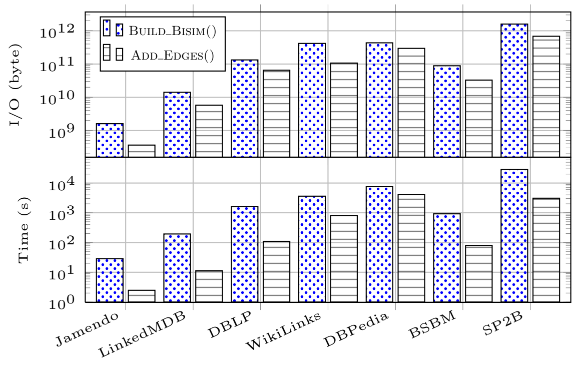
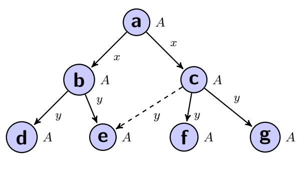


Figure 7: I/O and time comparison for BUILD_BISIM() and ADD_EDGES() after inserting one edge to the dataset ($k = 10$)

5.3.3 *Comparison of* BUILD_BISIM() *and* ADD_EDGES() *in extreme cases (single edge update)*

From the above experiments, we see that the performance of the algorithms are highly related to the datasets they process. For some datasets, the update algorithm is very much favorable while in other cases not so much. In the following, we would like to gain a better understanding of this phenomena.

We achieve this with two synthetic datasets, triggering both the extreme cases where the construction algorithm benefits the most and the update algorithm benefits the most. The first dataset, Dbest, shows a best-case scenario that the update algorithm can achieve relative to the construction algorithm. In this case we create a full k-ary tree, with edges pointing from parents to their children. When adding one edge to the tree, we add one edge to the leaf node, so that no node's signature would change after the insertion. In this case the update algorithm does the least amount of work, without propagating any change to further iterations during execution. Figure 8a shows an example of Dbest, which is a binary tree with height 3. The dashed edge is the newly added edge.


(a)

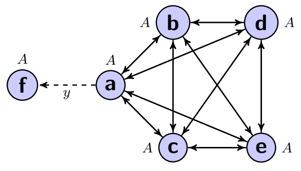

(b)

Figure 8: Examples for Dbest (8a) and Dworst (8b) datasets

The second dataset, Dworst, exhibits a worst-case scenario for the update algorithm, relative to construction. In this case we create a complete graph, with edges all labeled with x. Then when adding one more edge (labeled y) to one of the nodes, every other node in each iteration is affected and therefore all the nodes' signatures are changed. The update algorithm has to check all nodes in every iteration. Figure 8b shows an example of Dworst, a complete graph with 5 nodes. The dashed edge is the newly added edge.

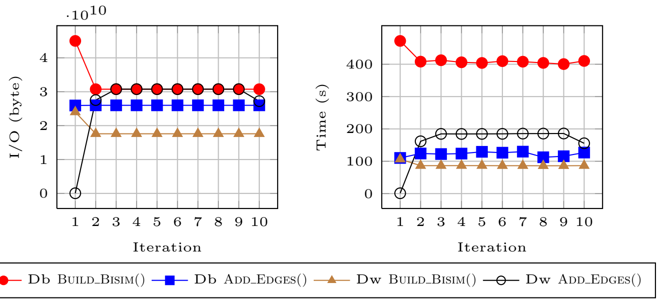


Figure 9: Time and I/O comparison for Db(est) and Dw(orst) by applying BUILD_BISIM() and ADD_EDGES() algorithms on both ($k = 10$)

We generate Dbest and Dworst on the scale of 100 million edges, and measure the elapsed time and I/O costs (Figure 9) for both the construction (BUILD_BISIM()) and edge update (ADD_EDGES()) algorithms in each iteration. We see that indeed for Dbest, the update algorithm shows a 4 times speed-up in time compared with the construction algorithm. For Dworst, the update algorithm is 2 times slower in time than the construction algorithm.

5.3.4 *Experiment on multiple edges update*

To test the performance of multiple edges update, we randomly select a set of edges from the dataset (edge count 1, 10, 100, ..., 1M), and apply the algorithm ADD_EDGES() upon them, recording the I/O and elapsed time performances. In Figure 10, we show the I/O improvement ratio and time speed up ratio (both construct/update) for all cases (taking the average). A gray line is drawn at $y = 1$ for both figures to split the space to indicate whether ADD_EDGES() performs better than BUILD_BISIM() or not. From the figure we see that for many of the datasets, it is beneficial to do batch update (ADD_EDGES()) up until 10^4 edges. An order of magnitude time speed up is observed for Jamendo, LinkedMDB and DBLP. In fact, if we consider the time cost for Jamendo and DBLP, it is always favorable to use ADD_EDGES() in all cases. For dataset DBPedia, however, changes propagate rapidly in the first few iterations, therefore the construction algorithm (BUILD_BISIM()) becomes a better choice when there are more than ten edges to be updated.

6. CONCLUSION AND FUTURE WORK

In this paper we have presented, to our knowledge, the first I/O efficient general-purpose algorithms for constructing and maintaining k-bisimulation partitions on massive

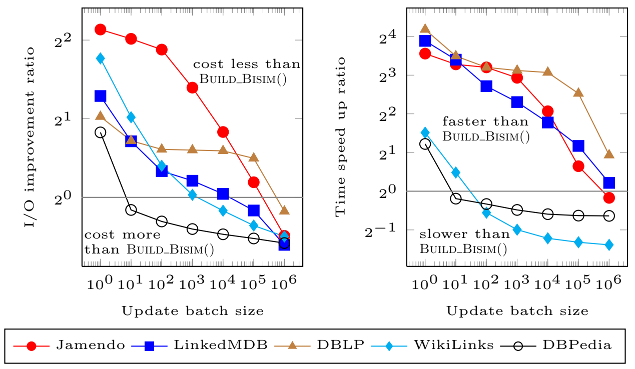


Figure 10: I/O (left) and time (right) improvement ratio $\frac{cost(\text{Build\_Bisim}())}{cost(\text{Add\_Edges}())}$ for batch edge updates ($k = 10$)

disk-resident graphs. A theoretical analysis showed, and an extensive empirical study confirmed, that our algorithms are not only efficient and practical to use, but also scale well with the size of the data.

We close by listing a few promising research directions for further study. First, it would be interesting to explore adaptations and extensions of our algorithms for alternative hardware platforms (e.g., multicore, SSD). Second, as we indicated at various points, many alternative data structures and join algorithms can be investigated for optimizing various aspects of the proposed algorithms. Third, because of their bulk streaming-based nature, many aspects of our algorithms naturally lend themselves to state-of-the-art parallel and distributed computing frameworks such as MapReduce. Studying the possibilities for leveraging our solutions to further scale the performance of these frameworks on real world graphs is certainly an interesting research direction. Last but not least, the ideas developed in this paper provide a basis for investigating related problems such as computing and maintaining *simulation* partitions in external memory (e.g., [12]).

Acknowledgments. The research of YW is supported by Research Foundation Flanders (FWO) during her sabbatical visit to Hasselt University, Belgium. The research of YL, GF, JH and PD is supported by the Netherlands Organisation for Scientific Research (NWO).

7. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988.
[2] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *STOC*, pages 540–548, El Paso, TX, USA, 1997.
[3] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 5(2):1–24, 2009.
[4] S. Blom and S. Orzan. Distributed state space minimization. *Int. J. STTT*, 7(3):280–291, 2005.
[5] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, Berlin, Germany, 2003.
[6] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, May 2008.
[7] A. Dovier, C. Piazza, and A. Policriti. An efficient algorithm for computing bisimulation equivalence. *Theor. Comp. Sci.*, 311(1-3):221–256, 2004.
[8] W. Fan. Graph pattern matching revised for social network analysis. In *ICDT*, pages 8–21, Berlin, Germany, 2012.
[9] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, Scottsdale, AZ, USA, 2012.
[10] P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
[11] G. H. L. Fletcher, D. Van Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. *Inf. Syst.*, 34(7):657–670, 2009.
[12] R. Gentilini, C. Piazza, and A. Policriti. From bisimulation to simulation: Coarsest partition problems. *J. Automated Reasoning*, 31:73–103, 2003.
[13] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. In *ALENEX*, pages 65–74, Kyoto, Japan, 2012.
[14] O. Hassanzadeh and M. P. Consens. Linked movie data base. In *LDOW*, 2009.
[15] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.
[16] J. Hellings, G. H. L. Fletcher, and H. Haverkort. Efficient external-memory bisimulation on DAGs. In *SIGMOD*, pages 553–564, Scottsdale, AZ, USA, 2012.
[17] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, San Jose, CA, USA, 2002.
[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, New York, NY, USA, 2010. ACM.
[19] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: large-scale graph computation on just a pc. In *OSDI*, pages 31–46, Berkeley, CA, USA, 2012.
[20] Y. Luo, Y. de Lange, G. H. L. Fletcher, P. De Bra, J. Hidders, and Y. Wu. Bisimulation Reduction of Big Graphs on MapReduce. In *BNCOD*, pages 189–203, Oxford, UK, 2013.
[21] Y. Luo, G. H. L. Fletcher, J. Hidders, P. De Bra, and Y. Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *GRADES*, pages 13:1–13:6, New York, NY, USA, 2013.
[22] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *SIGMOD*, pages 157–168, San Diego, CA, USA, 2003.
[23] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, Jerusalem, Israel, 1999.
[24] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16:973, 1987.
[25] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, Heraklion, Greece, 2012.
[26] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144, San Diego, CA, USA, 2003.
[27] S. Rajasekaran and I. Lee. Parallel algorithms for relational coarsest partition problems. *IEEE Trans. Parallel and Distributed Syst.*, 9(7):687–699, 1998.
[28] D. Sangiorgi and J. Rutten. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.
[29] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp^{2}bench: A sparql performance benchmark. In *ICDE*, pages 222–233, Washington, DC, USA, 2009. IEEE Computer Society.
[30] S. A. Smolka, O. Sokolsky, and S. Zhang. On the parallel complexity of bisimulation and model checking. *Modal Logic and Process Algebra: A Bisimulation Perspective in CSLI Lecture Notes*, 53:257–288, 1995.
[31] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD*, pages 567–580, New York, NY, USA, 2008. ACM.
[32] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of XML structural indexes. In *SIGMOD*, pages 491–502, Paris, France, 2004.