Structure preserving graph sampling and methods for partitioning graphs a bisimulation case study

van Heeswijk, W.J.A.

Award date:
2014

Structure preserving graph sampling and methods for partitioning graphs

Wouter van Heeswijk

Eindhoven, August 29, 2014

Technische Universiteit Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

Structure preserving graph sampling and methods for partitioning graphs

A bisimulation case study

Wouter van Heeswijk

Supervisors:
dr. George Fletcher
dr. Mykola Pechenizkiy

Assessment committee:

dr. Rui Castro	*Stochastics Section*
dr. George Fletcher	*Information Systems Section*
dr. Herman Haverkort	*Algorithms & Visualization Section*
dr. Mykola Pechenizkiy	*Information Systems Section*

Eindhoven, August 29, 2014

Abstract

Using the mathematical concept of a graph we can model many problems, and predict the behavior of a system or find solutions to a problem. With the advent of massive data a new problem arises: we have to be able to efficiently analyze or run calculations on very large graphs. One solution is to sample the graph, and perform our analysis on the smaller subgraph. However, this subgraph must maintain some of the properties of the original graph in which we are interested. Little work has been done on structure preserving sampling of graphs, and we contribute to get this subject going. We analyze the quality of eight sampling algorithms in how well they preserve the structure of a graph as the sampling ratio changes. For our case study we chose to preserve the bisimulation partition of a graph as its structural property. Additionally, we present a new algorithm which is intended to compute the bisimulation partition of a graph, but sacrifices correctness for efficiency. Our results show that this new algorithm is significantly faster than a traditional algorithm, but still gives a solution which is completely correct for our data sets.

Preface

Acknowledgments
During my education at the TU/e I enjoyed a great variety of aspects of computer science. I was interested in so many topics that it was difficult to choose a single problem, or even an area of computer science, to focus on for five months.
I want to thank George for pointing me to the problem of graph sampling, and his amazing guidance throughout the project. I also want to thank Mykola for his excellent consistent feedback, which made sure the project stayed on track. Big thanks to Rui and Herman too for their feedback on the thesis and assessing my project.
I want to thank Bram, Bram and Josh, for being such great colleagues. It was a real joy studying alongside you, making various projects and working on courses so much more fun.
Lastly I want to thank Yongming for his pointers at graph data sets, and Sander for allowing me to use his machine to run the experiments on.

Letter to my future self
Dear future me,

This year was the first year when it did not snow during the winter. Right now Netherlands is in the semi-finals against Argentina in the world cup. I've switched the Amazon cloud server to a Linux machine instead of a Windows machine; it's a faster yet costs less, so hopefully you can use some of that saved money when you're reading this. Finally being finished with school and university feels like such a relief, I wonder if you're thinking about it the same way. I'm also working on getting my driver's license, I wonder how good at driving I'll be in the future. After this summer I'm going to look for a job, but I have no idea what I'll end up doing.

Ciao!
24 year old me

Contents

Chapter 1

Introduction

Motivation In many fields of science and engineering, data is often modeled as a graph consisting of nodes and edges. One example could be road maps, where places are modeled as nodes and roads between those places are modeled as edges. We are often interested in finding patterns or structure in such graphs for various reasons. For example, it can help us predict the behavior of the systems modeled as graphs, or help us find nodes with particular properties.

Problem statement Our hardware's capabilities are ever growing, as predicted by Moore's law in 1965 [24]. This also means that our storage for data is growing rapidly, and huge graphs with billions of nodes are not uncommon (think of the World Wide Web, social media, state spaces of complex systems, and so on). This requires that we must be able to efficiently (with low complexity) do analysis on these graphs. One solution to this problem is to take a small sample of the graph and do our traditional analysis on that small subgraph. Indeed, a recent report by the American Academy of Sciences [25] highlights the importance of sampling as a data-gathering process and methodology for data reduction. However, for this to make sense, certain graph properties must be maintained, or be predictable, as we sample the original graph. Efforts made by [17][19] for example focus on estimating the structure of the original graph based on a sample.

Approach In this thesis we evaluate the quality of a number of sampling algorithms in maintaining structure. The structural property we have chosen is bisimulation, and we make a case study using this relation. Bisimulation captures the behavior of a graph, and can be used to derive equalities between the nodes of a graph [27][28]. Bisimulation finds applications in process theory, bioinformatics, sociology, structured data management, visualization and so on [5][4][14][30][11]. Therefore it is an interesting structural property which we would like to maintain as we sample a graph. Similar to results from [18] we want to investigate what happens to the quality of the samples when we vary our sampling fraction. To do so, we introduce two metrics (correctness and coverage) which measure how well the samplers maintain the bisimulation structure of a graph.

We observe that for some data sets, bisimulation can easily be maintained through sampling, while for others it cannot. Paradoxically, we cannot guarantee to perfectly sample a graph without at least knowing its bisimulation partition. Our aim is to provide an overview of what types of data sets are well suited for certain sampling methods, and which are not. We also find that it can be useful to design a sampling algorithm keeping in mind some knowledge of the structure of the original graph. This confirms previous research which further investigates how we can tailor sampling algorithms to suit particular needs [9].

Sometimes it is not necessary to achieve 100% accuracy when deriving the bisimulation partition of graph. That is: for the intended application it is acceptable that some small fraction of the partition blocks are missing, or if a few edges are missing or added. An example would be for the visualization of a graph modulo its bisimulation partition, it may be acceptable for the user to get a rough idea of the structure of the graph without complete accuracy. To serve this purpose, we also introduce a new algorithm which tries to compute the bisimulation partition of a graph as best it can, but is more efficient than traditional algorithms which are 100% accurate. Thus it sacrifices correctness for efficiency.

We evaluate the performance of our algorithm analytically and empirically. As we report in this thesis, we observed a significant speedup of our algorithm compared to a traditional algorithm, whilst still being 100% accurate itself as well. The accuracy of the more efficient algorithm which we present depends on the quality of some hash functions. We have also implemented a distributed variant of the new algorithm and one of the traditional algorithms, to compare the performance in a distributed setting.

Organization of thesis In Chapter 2 we introduce definitions and notation. In Chapter 3 we present the sampling algorithms, as well as the bisimulation partition algorithms. In Chapter 4 we discuss the data sets used in our experiments. In Chapter 5 we present our results and discuss them, drawing conclusions from our observations. In Chapter 6 we will conclude on the work we have done, and what our most important conclusions are. In Appendix A we show some of the properties of the data sets which we have used. In Appendix B we formally prove the correctness of some of the partition algorithms, and that there is a graph which is the smallest of all graphs bisimilar to it.

Chapter 2

Preliminaries

In this chapter we shall introduce the necessary definitions, concepts, and notation which need to be understood to fully comprehend everything in this document. Many things should be familiar to the reader with a background in Computer Science or Mathematics. Care has been taken to comply with standard notation used in the field of Computer Science as much as possible. There are also some completely new concepts which (to the best of our knowledge) have not been used anywhere else.

2.1 Labeled directed multigraph

We consider a labeled directed multigraph G. We denote the set of nodes by V and the set of edges by E. Note that we consider edges to be first-class entities in order to work with labeled parallel edges. As such, from the edge itself it is not clear what its source node, target node, and label are, and hence we introduce functions to indicate these as well. The functions $s : E \to V$ and $t : E \to V$ map edges to their source and target node respectively. The functions $\lambda_V : V \to \Sigma_V$ and $\lambda_E : E \to \Sigma_E$ map nodes and edges to their labels respectively. By Σ_V and Σ_E we denote the alphabets of node labels and edge labels respectively. Formally, our graph is an 8-tuple:

$$G = (V, E, s, t, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E).$$

For readability reasons we shall make use of the following shorthand notation:

$$u \xrightarrow{e} v \text{ denotes that } e \text{ is an edge going from node } u \text{ to node } v.$$

Figure 2.1 shows an example of a labeled directed multigraph. The example consists of 9 nodes and 10 edges. Each node is identified by a number (1 to 9) and its label is shown following a colon. For example: node 1 has label A and node 6 has label C. Edges have labels next to them as well. For example: the edge from node 3 to node 4 has label a, and the two parallel edges from node 5 to node 7 have labels a and b. Often we will abbreviate “labeled directed multigraph” by simply saying “graph”.

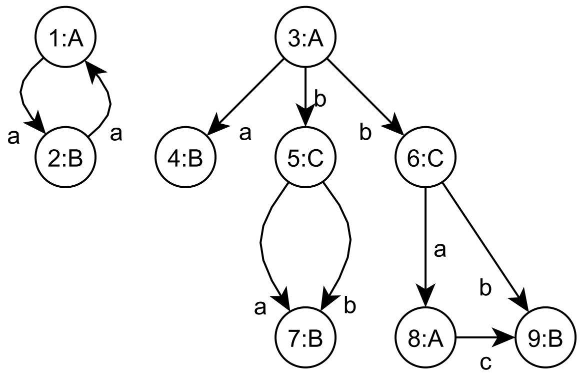


Figure 2.1: Example of a labeled directed multigraph with 9 nodes and 10 edges

2.2 (Un)bounded bisimulation and partitioning

Bisimulation captures the behavior of a node in terms of its outgoing edges and the behavior of the target nodes of those edges. We consider k-bisimulation where we look at a fixed depth of this behavior. Two nodes u and v are k-bisimilar if and only if (1) their labels are equal, (2) for every edge $u \xrightarrow{e} u'$ there exists an edge $v \xrightarrow{e'} v'$ such that the labels of e and e' are equal and u' and v' are $k-1$-bisimilar, and (3) symmetrically for outgoing edges of v.

More formally, given a labeled directed multigraph $G = (V, E, s, t, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ and a non-negative integer k, nodes $u, v \in V$ are k-bisimilar (denoted $u \approx^k v$) if and only if the following three conditions hold:

1. $\lambda_V(u) = \lambda_V(v)$,
2. if $k > 0$ then $\forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow$
   $\exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{k-1} v' \wedge \lambda_E(e) = \lambda_E(e')$, and
3. if $k > 0$ then $\forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow$
   $\exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{k-1} u' \wedge \lambda_E(e) = \lambda_E(e')$.

Note that the k-bisimilarity relation is an equivalence relation, and hence we can partition the set of nodes V into equivalence classes. The set of partition blocks (or equivalence classes) is defined by $\mathcal{P}_k = \left\{[v]_{\approx^k} \middle| v \in V\right\}$. So far we have looked at bisimulation for an arbitrary non-negative integer k. For all graphs there is a critical value $0 \leq k_{max} < |V|$ for which the following holds:

1. $\forall k \in \mathbb{N} : k > k_{max} \Rightarrow \mathcal{P}_{k_{max}} = \mathcal{P}_k$, and
2. if $k_{max} > 0$ then $\mathcal{P}_{k_{max}} \neq \mathcal{P}_{k_{max}-1}$.

For a proof of the existence of k_{max} see Proposition 2 in Appendix B. Informally the partition of the $\approx^{k_{max}}$ bisimulation relation is the most refined of all bisimulation partitions. For any values of $0 \leq k < k_{max}$ we call $\mathcal{P}_k$ a bounded bisimulation partition. For any values of $k \geq k_{max}$ we call $\mathcal{P}_k$ the unbounded bisimulation partition. The latter will often be abbreviated to $\mathcal{P}$ unless stated otherwise. Similarly the relation $\approx^{k_{max}}$ will often be abbreviated to $\approx$.

Figure 2.2 shows our example graph with the $k = 0$ and $k = 1 = k_{max}$ partitions illustrated by a coloring. Every partition block is assigned a unique color and its nodes are displayed with that color.

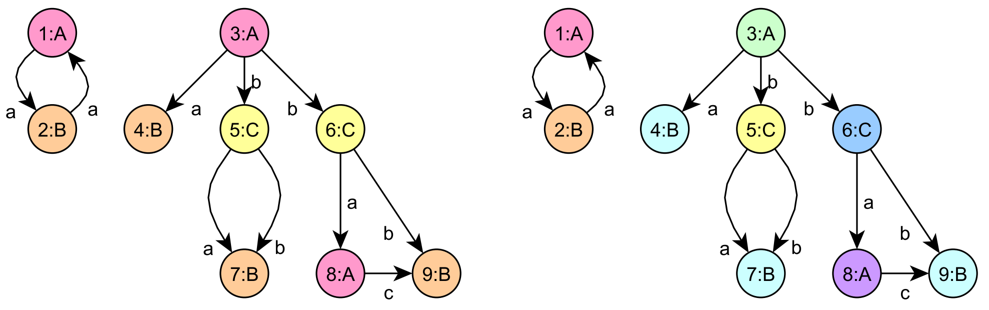


Figure 2.2: Example of a labeled directed multigraph, with the 0-bisimulation (left) and 1-bisimulation (right) partitions illustrated by a coloring of the partition blocks

2.3 Inter-graph bisimulation

To reason about bisimilarity of nodes from two (different) graphs, we introduce the notion of inter-bisimilarity here. For two graphs

$$G_1 = (V_1, E_1, s_1, t_1, \Sigma_{V_1}, \Sigma_{E_1}, \lambda_{V_1}, \lambda_{E_1})$$

$$G_2 = (V_2, E_2, s_2, t_2, \Sigma_{V_2}, \Sigma_{E_2}, \lambda_{V_2}, \lambda_{E_2})$$

we say that nodes $u \in V_1$ and $v \in V_2$ are k-inter-bisimilar (denoted $u \simeq^k v$) if and only if the following three conditions hold:

1. $\lambda_{V_1}(u) = \lambda_{V_2}(v)$,
2. if $k > 0$ then $\forall u' \in V_1, e \in E_1 : u \xrightarrow{e} u' \Rightarrow$
   $\exists v' \in V_2, e' \in E_2 : v \xrightarrow{e'} v' \wedge u' \simeq^{k-1} v' \wedge \lambda_{E_1}(e) = \lambda_{E_2}(e')$, and
3. if $k > 0$ then $\forall v' \in V_2, e \in E_2 : v \xrightarrow{e} v' \Rightarrow$
   $\exists u' \in V_1, e' \in E_1 : u \xrightarrow{e'} u' \wedge v' \simeq^{k-1} u' \wedge \lambda_{E_2}(e) = \lambda_{E_1}(e')$.

We say that graphs G_1 and G_2 are k-bisimilar if and only if the following two conditions holds:

1. for every $u \in V_1$ there exists a $v \in V_2$ such that $u \simeq^k v$, and
2. for every $v \in V_2$ there exists a $u \in V_1$ such that $v \simeq^k u$.

Put simply: we require that a node from one graph can be simulated by a node from the other graph, and vice versa. If this is the case for all nodes of either graph, then they are bisimilar.

2.4 Partition reduced graph

Let $G = (V, E, s, t, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$ be a labeled directed multigraph and $\mathcal{P}$ its bisimulation partition. We can derive a minimal graph from G modulo its bisimulation partition. This minimal graph contains nodes and edges such that it is bisimilar to G, according to the definition given in Section 2.3. Out of all graphs that exist that are bisimilar to G, this minimal graph has the least amount of nodes and the least amount of edges. We obtain this minimal graph by having each partition block in $\mathcal{P}$ be a node in the reduced graph. The edges are then added such that an edge exists from partition block P to Q with some unique label if there is a node in P with an edge to a node in Q with that label. Formally, we obtain the graph with the following construction:

1. let $\mathcal{P}$ be the bisimulation partition of G;

2. let $\lambda_{\mathcal{P}}(P) = \lambda_V(v)$ for all $P \in \mathcal{P}$ and arbitrary $v \in P$;
3. let $E' = \left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}$;
4. let $s'(P, Q, l) = P$ for all $(P, Q, l) \in E'$;
5. let $t'(P, Q, l) = Q$ for all $(P, Q, l) \in E'$;
6. let $\lambda_{E'}(P, Q, l) = l$ for all $(P, Q, l) \in E'$.

Then $R = (\mathcal{P}, E', s', t', \Sigma_V, \Sigma_E, \lambda_{\mathcal{P}}, \lambda_{E'})$ is the minimal bisimulation partition reduced graph of G. A proof that R is indeed a minimally reduced graph can be found in Proposition 2 in Appendix B. Figure 2.3 shows a minimally reduced version of our example graph for $k = 1$. Our original example graph has a partition block containing three nodes (4, 7 and 9) which collapsed into node 6 of the minimally reduced graph.

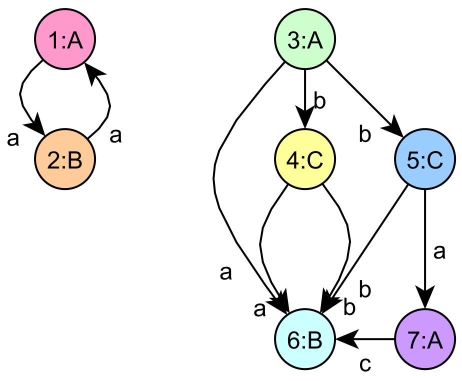


Figure 2.3: Our example of a labeled directed multigraph which has been minimally reduced with respect to its 1-bisimulation partition

2.5 Partition correctness and coverage

Standard metrics Our goal is to measure how well some sampling algorithms preserve the bisimulation partition of a graph. To accomplish this we will introduce two metrics. The correctness metric (denoted P for precision) will measure how many of the sampled graph's partition blocks are indeed also present in the original graph's partition. The coverage metric (denoted R for recall) will measure how many of the original graph's partition blocks are also part of the sampled graph's partition.

In order to compute these two metrics we shall give definitions here. Suppose we have two graphs $G_1 = (V^1, E^1, s^1, t^1, \Sigma_V^1, \Sigma_E^1, \lambda_V^1, \lambda_E^1)$ and $G_2 = (V^2, E^2, s^2, t^2, \Sigma_V^2, \Sigma_E^2, \lambda_V^2, \lambda_E^2)$ with disjoint node sets $V^1 \cap V^2 = \emptyset$. We will merge these two graphs into one $G = (V^1 \cup V^2, E^1 \cup E^2, s^1 \cup s^2, t^1 \cup t^2, \Sigma_V^1 \cup \Sigma_V^2, \Sigma_E^1 \cup \Sigma_E^2, \lambda_V^1 \cup \lambda_V^2, \lambda_E^1 \cup \lambda_E^2)$. On this merged graph we can compute the bisimulation partition $\mathcal{P}$. We can then consider the partition blocks covered by each of the initial two graphs. Let $\mathcal{P}_1 = \left\{[v]_{\approx} \in \mathcal{P} \middle| v \in V^1\right\}$ and $\mathcal{P}_2 = \left\{[v]_{\approx} \in \mathcal{P} \middle| v \in V^2\right\}$ denote those partition blocks which contain nodes from G_1 and G_2 respectively. Then we define the correctness and coverage as follows:

$$P(G_1, G_2) = \frac{|\mathcal{P}_1 \cap \mathcal{P}_2|}{|\mathcal{P}_2|}$$

$$R(G_1, G_2) = \frac{|\mathcal{P}_1 \cap \mathcal{P}_2|}{|\mathcal{P}_1|}$$

Note that these metrics are defined asymmetrically: they represent the correctness and coverage of graph G_2 onto G_1.

Using these two definitions we can get an idea about how well the sampled graph preserves the bisimulation partition of the original graph. The higher these values are, the better the sampled graph represents the original graph with respect to bisimulation. We must ensure however, by definition of the metrics, that the set of nodes from the original graph is disjoint with the set of nodes from the sampled graph. In order to achieve this we can transform both graphs such that nodes $v_1 \in V^1$, $v_2 \in V^2$ become tuples with the original node and an identifier of which graph it came from $v_1' = (v_1, 1)$, $v_2' = (v_2, 2)$. By transforming the graphs in this way we can, once we obtained a partition of the merged graph, easily find out which of the two graphs (or both) a partition block occurs in by inspecting its member nodes.

In Figure 2.4 we see our example graph (indicated by circular nodes) merged with a smaller graph (indicated by square nodes). If we were to compute the correctness and coverage of the "square" graph onto the original "circle" graph, we would get $P(circle, square) = \frac{4}{4} = 100\%$ and $R(circle, square) = \frac{4}{7} \approx 57\%$.

Weighted metrics Often we observe that the individual bisimulation partition blocks of a graph vary greatly in size, due to the nature of real-world graphs. In Section 4.2 this is discussed in more detail, and examples can be seen in Figure A.1. Due to this skew it might be more interesting to the user or application to consider weighted versions of the correctness and coverage metrics. This weighted version is based on counting nodes rather than partition blocks.

$$WP(G_1, G_2) = \frac{\left|\left\{v \in V^2 \middle| v \in P \in \mathcal{P}_1 \cap \mathcal{P}_2\right\}\right|}{|V_2|}$$

$$WR(G_1, G_2) = \frac{\left|\left\{v \in V^1 \middle| v \in P \in \mathcal{P}_1 \cap \mathcal{P}_2\right\}\right|}{|V_1|}$$

With very few exceptions, the weighted metrics show often much higher values than the standard metrics. These observations will be further discussed in Section 5.1.2 and we will explain the reason why these metrics are often much higher in value.

In Figure 2.4 we see our example graph (indicated by circular nodes) merged with a smaller graph (indicated by square nodes). If we were to compute the weighted correctness and weighted coverage of the "square" graph onto the original "circle" graph, we would get $WP(circle, square) = \frac{4}{4} = 100\%$ and $WR(circle, square) = \frac{6}{9} \approx 67\%$.

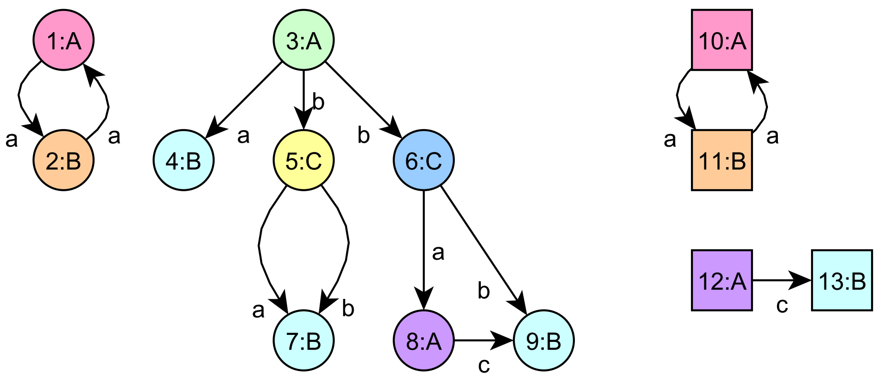


Figure 2.4: Example of two merged labeled directed multigraphs, where the nodes of one graph are shown using circles, and the nodes of the other graph are shown using squares

2.6 Performance metrics for distributed algorithms

As we will be dealing with analyzing the performance of distributed algorithms, we need to introduce some metrics here. As will be explained in Section 3.2 we use a message passing model to define our distributed algorithms. We will use the same metrics as used by Ma et al. [22]. These are:

1. Visit times: the total number of messages sent. The visit times of a machine is the number of messages sent to that machine. This metric indicates the complexity of interactions.
2. Makespan: the total running time of the distributed algorithm, from the moment the first machine starts working on the problem until the moment the last machine finishes working on the problem.
3. Data shipment: the total size of all the messages exchanged between machines. The data shipment of a machine is the total size of messages which have been sent to and from that machine.

All three of these metrics would ideally be as low as possible. However, one has to balance between the three, as efforts to decrease one will increase one or two of the others. For example: one could simply ship all of the graph segments directly to a single machine and do the sequential computation on that machine; this would minimize the data shipment but maximize the makespan. Another example: one could ship the graph segments in a chain towards a single machine and do the sequential computation on that machine; this would minimize the visit times but maximize the makespan.

Chapter 3

Algorithms

In this chapter we introduce the algorithms relevant to our studies. In Section 3.1 we will define and describe the sampling algorithms. In Section 3.2 we will define and describe the partition algorithms. All of the algorithms are also provided with an analytically derived worst-case performance asymptotic complexity bound.

Example source code, written in C#, of all algorithms can be found at `http://bit.ly/RARtDl`. In some cases the implementation of some algorithms may look slightly different from the actual algorithms presented here, because it lead to better optimization in the implementation or because the algorithms would be easier to read. The functionality of the implementation and the algorithms presented here, however, are exactly the same.

3.1 Sampling algorithms

In this section we will describe the sampling algorithms which we used. There are three variations of the Queued Traversal Algorithm, which will be discussed in Section 3.1.5, making for a total of eight sampling algorithms. All of the sampling algorithms return a set of nodes S (the sample) which is then used to induce a subgraph G_S of the original graph G. For the induced subgraph $G_S = (S, E', s', t', \Sigma_S, \Sigma_{E'}, \lambda_S, \lambda_{E'})$ we have the following construction. Note that the function signatures of the subgraph are $s' : E' \to S$, $t' : E' \to S$, $\lambda_S : S \to \Sigma_S$ and $\lambda_{E'} : E' \to \Sigma_{E'}$.

1. $E' = \{e \in E \big| s(e) \in S \wedge t(e) \in S\}$
2. $\forall e \in E' : s'(e) = s(e) \wedge t'(e) = t(e) \wedge \lambda_{E'}(e) = \lambda_E(e)$
3. $\Sigma_S = \{\lambda_V(v) \big| v \in S\}$
4. $\Sigma_{E'} = \{\lambda_E(e) \big| e \in E'\}$
5. $\forall v \in S : \lambda_S(v) = \lambda_V(v)$

Put simply: once we obtain our sample set of nodes S we keep only those edges that connect nodes within the sampled set. The labels of the nodes that are kept and the edges that are kept, remain the same. For example let's look at our example graph in Figure 2.1. Suppose we sample three nodes ($S = \{3, 5, 7\}$), then we obtain the induced subgraph shown in Figure 3.1. To illustrate the functionality of the sampling algorithms, we shall look at examples of subgraphs of the graph shown Figure 2.1 which the algorithms can produce.

There are many scenarios in which graph sampling might be applied. Sometimes the graph fits in memory, and sometimes it doesn't. Sometimes it's stored on a single machine and sometime it isn't. There are even cases where random access is prohibited or graph data can only be streamed. For example: the World Wide Web needs to be crawled by exploration, and one cannot simply request for a random node (web page). For our studies we stick to the simple scenario where we have a graph which fits in memory, and we can access arbitrary nodes and edges, and access neighboring nodes and edges given a node.

It is important that the sampling algorithms run in low time complexities, otherwise it would defy the purpose of sampling. The most ideal sampling algorithm would always give the exact solution to the bisimulation partition problem, but this would take too much computation time, unless a more efficient solution than the current algorithms is found. Thus we find ourselves with the following paradox - we need to sample to analyze huge graphs, but a perfect sampler would need to know at least those properties we're trying to preserve.

For a survey on common sampling techniques there is a nice work in progress by Hu and Lau [13].

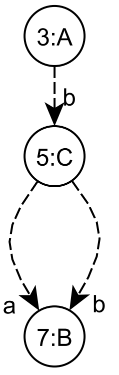


Figure 3.1: Example of an induced subgraph, derived from a sample set of three nodes, where the edges we keep are indicated by dashed lines

3.1.1 Random Node (RN)

Description This is the simplest sampling algorithm and is commonly studied in the field of graph sampling. Given a set of nodes we shall take up to N nodes, randomly uniformly chosen. This sampling method gives a good baseline for what happens when we put no thought into the way we sample nodes. Figure 3.2 shows an example of what the RN sampling method can give as output.

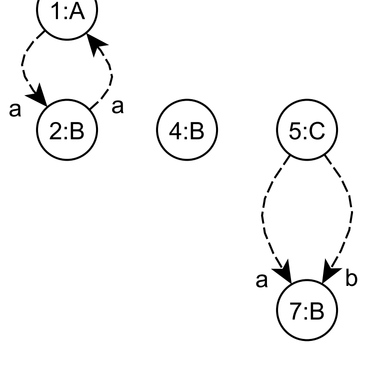


Figure 3.2: Example of an induced subgraph, where our set of sample nodes $S = \{1, 2, 4, 5, 7\}$ was randomly chosen by the RN sampling method

```
Algorithm $RN(V, N)$
1.  $N \leftarrow \min(N, |V|)$
2.  $V' \leftarrow$ random permutation of $V$
3.  return $V'[1..N]$
```

Complexity The list of nodes can simply be shuffled to obtain a random selection of nodes. Because we want to select nodes without replacement, this method has better worst-case complexity than selecting random nodes and then checking if we have already picked them or not. This method takes $O(|V|)$ time.

3.1.2 Random Edge (RE)

Description This algorithm is based on selecting edges before nodes. First we randomly uniformly choose up to N edges. Then the endpoints (source and target nodes) of those edges are selected as our sample nodes. Like the RN sampling method, this is a pretty simple way of sampling, but typically preserves many (structural) properties of the original graph as mentioned by Al Hasan et al. [1]. Figure 3.3 shows an example of what the RE sampling method can give as output.

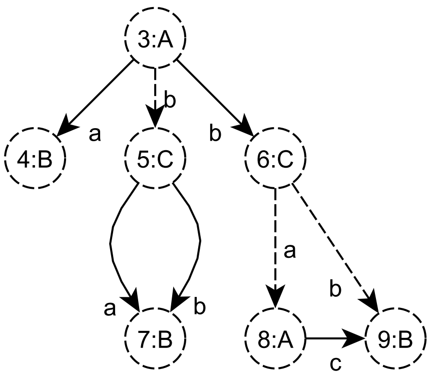


Figure 3.3: Example of an induced subgraph, where our set of sample nodes $S = \{3, 4, 5, 6, 7, 8, 9\}$ was randomly chosen by selecting the edges indicated by the solid arrows in the RE sampling method

```
Algorithm $RE(E, N, s, t)$
1.  $N \leftarrow \min(N, |E|)$
2.  $E' \leftarrow$ random permutation of $E$
3.  $S \leftarrow \emptyset$
4.  for $i = 1$ to $N$
5.      do $S \leftarrow S \cup \{s(E'[i]), t(E'[i])\}$
6.  return $S$
```

Complexity The list of edges can simply be shuffled to obtain a random selection of edges. Because we want to select edges without replacement, this method has better worst-case complexity than selecting random edges and then checking if we have already picked them or not. This method takes $O(|E|)$ time.

3.1.3 Low Degree First (LDF)

Description This algorithm selects those nodes with the lowest degree first. We sort the nodes by their degree in ascending order, and return the first up to N nodes. The idea here is that for low-degree nodes it is easier to get the partition block in our sampled graph. Therefore we should get high correctness in our sampled graph, but our coverage may suffer, since we only pay attention to a specific group of nodes. Figure 3.4 shows an example of what the LDF sampling method can give as output.

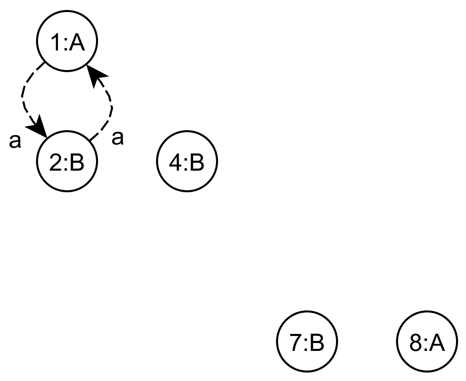


Figure 3.4: Example of an induced subgraph, where our set of sample nodes $S = \{1, 2, 4, 7, 8\}$ was chosen by selecting the nodes with lowest degree first

Algorithm *LDF*(V, E, N, s, t)
```
1. ▷ Let $deg(x)$ denote $\left|\{e \in E | x = s(e) \vee x = t(e)\}\right|$
2. $N \leftarrow \min(N, |V|)$
3. $V' \leftarrow$sort $V$ by $(x, y) \rightarrow deg(x) < deg(y)$
4. return $V'[1..N]$
```

Complexity We assume that querying the degree of a node takes $O(1)$ time. Then the sorting will take $O(|V| \log|V|)$ time.

3.1.4 Greedy Labels (GL)

Description This algorithm provides a sense of equal opportunity for every node label. Each node label can, in turn, add one of its associated nodes to the sample. If a label has relatively few nodes associated with it then the remaining spots are divided equally among the remaining labels. This is repeated until the number of desired nodes N has been reached or until there are no more remaining nodes. Since we first shuffle each group of nodes, if there is only one node label in the entire graph, this algorithm should have exactly the same effect as RN sampling. For graphs with multiple node labels, the idea is to improve upon RN by ensuring more diversity in our sampled graph. Figure 3.5 shows an example of what the GL sampling method can give as output.

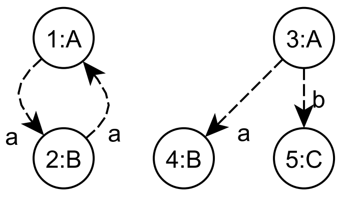


Figure 3.5: Example of an induced subgraph, where our set of sample nodes $S = \{1, 2, 3, 4, 5\}$ was chosen by repeatedly allowing each node label (A, B and C) to add one of its nodes to the sample

```
Algorithm $GL(V, N, \Sigma_V, \lambda_V)$
1.  for $l \in \Sigma_V$
2.      do $group[l] \leftarrow$random permutation of $\{v \in V | \lambda_V(v) = l\}$
3.         $count[l] \leftarrow 0$
4.  $L \leftarrow \Sigma_V$
5.  $i \leftarrow 0$, $M \leftarrow |L|$, $S \leftarrow \emptyset$
6.  while $N > 0 \wedge M > 0$
7.      do $l \leftarrow L[i+1]$
8.         if $|group[l]| > count[l]$
9.           then $v \leftarrow group[l][count[l]+1]$
10.               $S \leftarrow S \cup \{v\}$
11.               $count[l] \leftarrow count[l] + 1$
12.               $N \leftarrow N - 1$
13.               $i \leftarrow i + 1$
14.          else Remove item from $L$ at position $i+1$
15.               $M \leftarrow M - 1$
16.        if $M > 0$
17.          then $i \leftarrow i \mod M$
18. return $S$
```

Complexity Using a hash table to store the lists of nodes per node label, we can achieve a time complexity of $O(|V|)$. This assumes that $\Sigma_V = \{\lambda_V(v) | v \in V\}$, i.e. there are no unused node labels in Σ_V.

3.1.5 Queued Traversal (QT: BFS, DFS, RFS)

Description This algorithm explores the graph starting from a seed node (actually, multiple seed nodes, one from each connected component of the graph). It does so by keeping a queue of nodes that have yet to be visited. When a node is visited, its neighboring nodes are added to the queue. Because bisimilarity depends very much on the outgoing edges of a node, we suspect the exploration-based algorithms are good at sampling the partition blocks of nodes that are visited during the exploration. We have used three kinds of queues throughout our experiments:

1. FIFO (First In, First Out) queue, simulating a breadth-first traversal (BFS).
2. LIFO (Last In, First Out) queue, simulating a depth-first traversal (DFS).
3. AIRO (Anything In, Random Out) queue, simulating the algorithm by Kashtan et al. [15] (RFS). In their algorithm they pick random edges which are adjacent to edges which have been explored so far, using a seed edge. However, their algorithm does not guarantee termination (for example if the sampling fraction is very high and there are nodes without

edges, their algorithm does not terminate). The idea of selecting random neighbors from the explored part of the graph is what we've kept in our RFS algorithm.

The order in which seed nodes are selected is randomized by line 2 of the algorithm. For some types of graphs, a possible improvement to this strategy is to select seed nodes by selecting the highest degree nodes first. We have no experiments for this strategy thus far, but future work may test this proposal. Figure 3.6, Figure 3.7 and Figure 3.8 show examples of what the BFS, DFS and RFS sampling methods respectively can give as output.

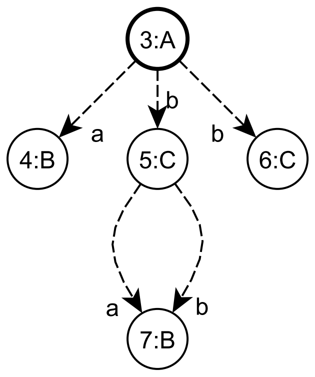


Figure 3.6: Example of an induced subgraph, where our set of sample nodes $S = \{3, 4, 5, 6, 7\}$ was chosen by exploring in a breadth-first manner, starting at node 3

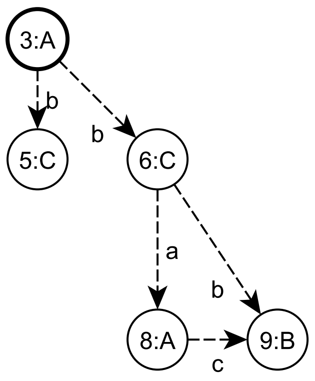


Figure 3.7: Example of an induced subgraph, where our set of sample nodes $S = \{3, 5, 6, 8, 9\}$ was chosen by exploring in a depth-first manner, starting at node 3

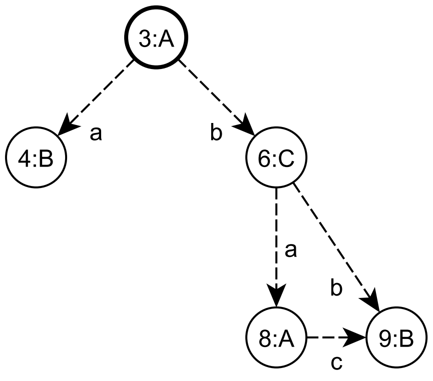


Figure 3.8: Example of an induced subgraph, where our set of sample nodes $S = \{3, 4, 6, 8, 9\}$ was chosen by exploring in a random-first manner, starting at node 3

```
Algorithm $QT(V, E, N, s, t)$
1.  $S \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \emptyset$, $i \leftarrow 1$
2.  $V' \leftarrow$ random permutation of $V$
3.  while $N > 0$
4.     do if $|\mathcal{Q}| = 0$
5.           then while $V'[i] \in S$
6.                   do $i \leftarrow i + 1$
7.                      if $i > |V'|$
8.                        then return $S$
9.                $v \leftarrow V'[i]$
10.               Enqueue $v$ onto $\mathcal{Q}$
11.               $S \leftarrow S \cup \{v\}$
12.               $N \leftarrow N - 1$
13.        Dequeue $u$ from $\mathcal{Q}$
14.        $\mathcal{N} \leftarrow \left\{ u' \in V \middle| \exists e \in E : u \xrightarrow{e} u' \right\}$
15.        for $v \in \mathcal{N}$
16.           do if $v \notin S \land N > 0$
17.                then Enqueue $v$ onto $\mathcal{Q}$
18.                     $S \leftarrow S \cup \{v\}$
19.                     $N \leftarrow N - 1$
20. return $S$
```

Complexity We will assume that the outgoing neighbors of a node v can be obtained in at most $O(deg(v))$ time. While exploring, we may have to process each edge once. Additionally, we explore the entire graph and not just one connected component. Therefore the time complexity of this sampling algorithm is $O(|V| + |E|)$.

3.1.6 Distinct Label Breadth-First (DLBF)

Description This algorithm is much like the previous algorithm (Section 3.1.5). It too explores the graph based on seed nodes. The difference lies in the way neighboring nodes are selected and added to the queue. Instead of taking every neighbor of a node, we only take one neighboring node per edge label. For example, say node 3 has the following three outgoing edges: $3 \xrightarrow{e} 4$,

$3 \xrightarrow{f} 5$, $3 \xrightarrow{g} 6$ where $\lambda_E(e) = a$, $\lambda_E(f) = b$ and $\lambda_E(g) = b$. Then 4 is selected along with either 5 or 6. Figure 3.9 shows an example of what the DLBF sampling method can give as output. The reason we think this may improve on the previous algorithm is because bisimilarity does not care about the multiplicity of outgoing edges with the same label and bisimilar target nodes. We don't know the bisimulation partition block of a target node as we're sampling, but we do know the edge label. Our effort is therefore focused on selecting only one node per outgoing edge label. Here we stick to one type of queue (FIFO) throughout our experiments.

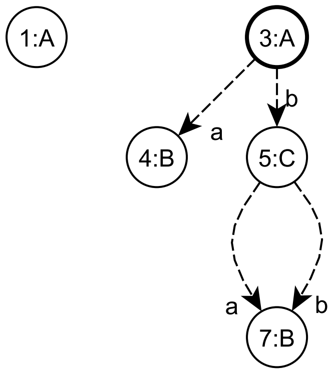


Figure 3.9: Example of an induced subgraph, where our set of sample nodes $S = \{1, 3, 4, 5, 7\}$ was chosen by exploring in a breadth-first manner, adding only one node per outgoing edge label to the queue, starting at node 3; after exhausting the queue, the seed node 1 was chosen randomly

```
Algorithm $DLBF(V, E, N, s, t, \Sigma_E, \lambda_E)$
1.  $S \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \emptyset$, $i \leftarrow 1$
2.  $V' \leftarrow$ random permutation of $V$
3.  while $N > 0$
4.     do if $|\mathcal{Q}| = 0$
5.           then while $V'[i] \in S$
6.                   do $i \leftarrow i + 1$
7.                      if $i > |V'|$
8.                        then return $S$
9.                $v \leftarrow V'[i]$
10.               Enqueue $v$ onto $\mathcal{Q}$
11.               $S \leftarrow S \cup \{v\}$
12.               $N \leftarrow N - 1$
13.        Dequeue $u$ from $\mathcal{Q}$
14.        $L \leftarrow \{l \in \Sigma_E | \exists e \in E : \lambda_E(e) = l \wedge s(e) = u\}$
15.        for $l \in L$
16.            do Choose $v \in \left\{u' \in V \middle| \exists e \in E : u \xrightarrow{e} u' \wedge \lambda_E(e) = l\right\}$ arbitrarily
17.               if $v \notin S \wedge N > 0$
18.                 then Enqueue $v$ onto $\mathcal{Q}$
19.                      $S \leftarrow S \cup \{v\}$
20.                      $N \leftarrow N - 1$
21. return $S$
```

A possible improvement is to select neighbors not just by distinct edge label, but also by distinct target node label. This should improve the quality of the sample in graphs where both nodes and edges are labeled.

Complexity We will make the same assumption made in Section 3.1.5. This means we can easily obtain the outgoing edge labels of a node too. Using the same reasoning as in Section 3.1.5 we obtain the same time complexity of $O(|V| + |E|)$.

3.2 Partition algorithms

In this section we will describe the partition algorithms. There are four such algorithms: the sequential signature-based algorithm, the sequential hash-based algorithm, the distributed signature-based algorithm, and the distributed hash-based algorithm. The signature-based algorithms are well-known, and compute the exact solution to the bisimulation partition problem, whereas our new hash-based algorithms have no correctness guarantees but give good results (see Section 5.2.1). For each algorithm we derive analytical bounds in terms of time complexity. For the distributed algorithms, we additionally discuss the data shipment and visit times complexities. These metrics are the same as those used by Ma et al. [22]. For completeness sake we also give definitions of these metrics in Section 2.6. For an empirical evaluation of the performance metrics for these algorithms, see Section 5.2.2.

Both distributed algorithms use a message passing model and assume synchronous (TCP-like) message passing. Due to the nature of the message passing model, the distributed algorithms are written in an event-driven style. We model our distributed algorithms around having a single coordinator and W worker machines. More details about the role of each type of machine can be found in Section 3.2.3 and Section 3.2.4 where we discuss the distributed algorithms. The actual graph itself is split and stored in segments on the worker machines. The nodes of a graph are segmented in such a way that they are evenly distributed among the worker machines. How the graphs were segmented in our experiments is explained in Section 5.2.2. The worker machines also require knowledge about which other machine has a node which is connected by an outgoing edge to a local node.

The output of each algorithm is an associative array mapping each node to the identifier of the partition block it was assigned to.

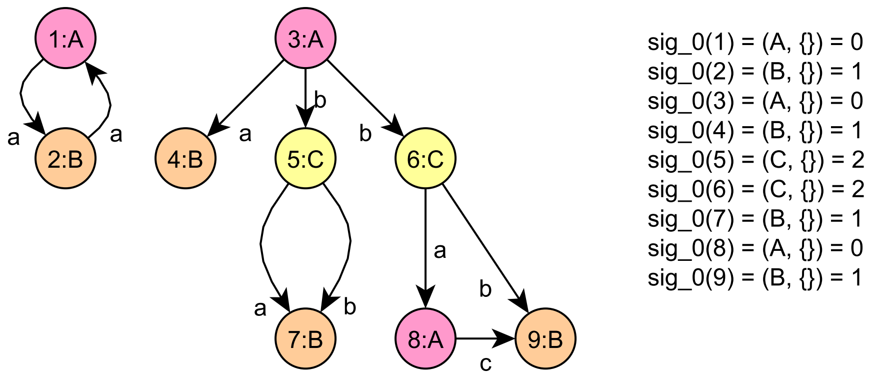
The best in-memory bisimulation partition algorithm to date is that of Paige and Tarjan [26]. Their algorithm runs in $O(|E| \log |V|)$ time complexity. The best external memory algorithm to date has I/O complexity $O(k \cdot \mathit{sort}(|E|) + k \cdot \mathit{scan}(|V|) + \mathit{sort}(|V|))$ by Luo et al. [21]. Here, $\mathit{scan}(n)$ and $\mathit{sort}(n)$ are the cost of scanning and sorting, respectively, a file occupying n pages in external memory. However, if the graph is a Directed Acyclic Graph (DAG) then there is an even more efficient external memory algorithm with I/O complexity $O(\mathit{sort}(|V| + |E|))$ by Hellings et al. [12]. We implement the partition algorithm based on signatures, as described for example by Blom and Orzan [6] and Blom and Pol [7]. This algorithm is asymptotically slower than that of Paige and Tarjan, running in $O(k \cdot (|E| + |V|^2))$ worst-case time complexity. However, in practice it performs quite well and it is easily extendable to parallel algorithms. It can also be easily extended to compute partitions of different equivalence relations, as long as a fitting signature can be constructed. For example: one could find signature definitions which determine structural equivalence, or even branching bisimulation equivalence.

3.2.1 Sequential signature-based algorithm

Description This is the sequential algorithm which computes the bisimulation partition of a graph using inductively defined signatures. It is described for example by Blom and Orzan [6] and Blom and Pol [7]. Correctness follows from Proposition 1 in Appendix B where we prove that signature equality is equivalent to bisimilarity. In our algorithm we compute these inductively defined signatures. In order to reduce complexity, we identify signatures by a unique identifier.

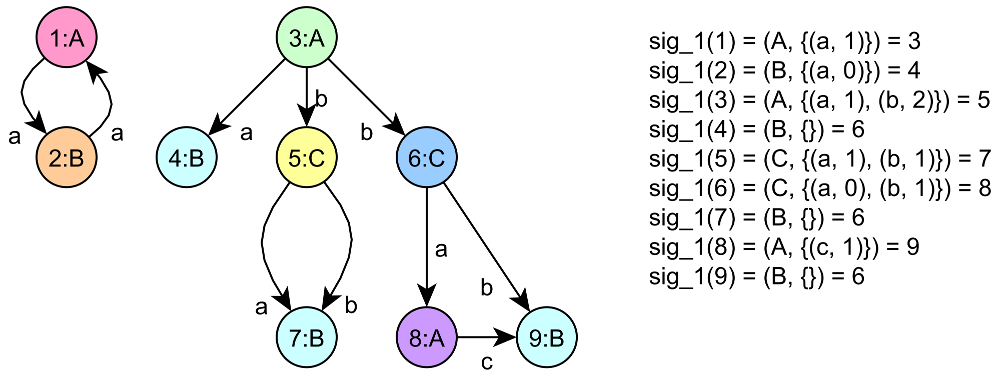
We ensure uniqueness of signature identifiers by keeping a counter at line 2 of the algorithm. In lines 3-7 we introduce a new identifier for each node label, and assign the $k = 0$ partition signature identifier to each node. This works because the $k = 0$ partition only depends on the node labels. The loop of lines 8-21 is there to iteratively refine the partition as we increase k. We do this until the partition is no longer refined, i.e. we have reached a stable partition. The correctness of this termination condition is proven by Corollary 2 in Appendix B. We use a hash table *signatureId* to keep track of signature identifiers, which we can discard after each iteration because we do not reset the counter to 0.

Figure 3.10 and Figure 3.11 show an example of the signatures and the identifiers which are computed by this algorithm. As it turns out, $k = 1 = k_{max}$ for our example graph, and therefore in the third iteration of lines 8-21 (where we compute the $k = 2$ partition) the number of partition blocks does not increase.



sig_0(1) = (A, {}) = 0
sig_0(2) = (B, {}) = 1
sig_0(3) = (A, {}) = 0
sig_0(4) = (B, {}) = 1
sig_0(5) = (C, {}) = 2
sig_0(6) = (C, {}) = 2
sig_0(7) = (B, {}) = 1
sig_0(8) = (A, {}) = 0
sig_0(9) = (B, {}) = 1

Figure 3.10: Example of a labeled directed multigraph with the 0-bisimulation partition illustrated by a coloring of the partition blocks; additionally the signatures computed by the sequential signature-based algorithm are shown



sig_1(1) = (A, {(a, 1)}) = 3
sig_1(2) = (B, {(a, 0)}) = 4
sig_1(3) = (A, {(a, 1), (b, 2)}) = 5
sig_1(4) = (B, {}) = 6
sig_1(5) = (C, {(a, 1), (b, 1)}) = 7
sig_1(6) = (C, {(a, 0), (b, 1)}) = 8
sig_1(7) = (B, {}) = 6
sig_1(8) = (A, {(c, 1)}) = 9
sig_1(9) = (B, {}) = 6

Figure 3.11: Example of a labeled directed multigraph with the 1-bisimulation partition illustrated by a coloring of the partition blocks; additionally the signatures computed by the sequential signature-based algorithm are shown

```
Algorithm SequentialSignatureBasedBisimulationPartition$(V, E, s, t, \Sigma_V, \lambda_V, \lambda_E)$
1.  $k \leftarrow 0$
2.  $counter \leftarrow 0$
3.  for $l \in \Sigma_V$
4.      do $labelId[l] \leftarrow counter$
5.          $counter \leftarrow counter + 1$
6.  for $v \in V$
7.      do $P_k[v] \leftarrow labelId[\lambda_V(v)]$
8.  repeat
9.      $\triangleright$ Let $signatureId[S] =$ nil for all signatures $S$
10.     $k \leftarrow k + 1$
11.     for $u \in V$
12.         do $L \leftarrow \emptyset$
13.             for $e \in E$ such that $s(e) = u$
14.                 do $\triangleright$ Let $v$ denote $t(e)$
15.                     $L \leftarrow L \cup \{(\lambda_E(e), P_{k-1}[v])\}$
16.             $\triangleright$ Let $S = (\lambda_V(u), L)$ denote the signature of $u$
17.             if $signatureId[S] =$ nil
18.                 then $signatureId[S] \leftarrow counter$
19.                     $counter \leftarrow counter + 1$
20.             $P_k[u] \leftarrow signatureId[S]$
21. until $|P_k^{\rightarrow}| = |P_{k-1}^{\rightarrow}|$
22. return $P_k$
```

Complexity We will now analyze the time complexity of the algorithm. The loop of lines 3-5 of the algorithm has complexity $O(|\Sigma_V|)$. It makes sense that there are at most as many node labels as nodes, hence we will assume that $|\Sigma_V| \leq |V|$. The loop of lines 6-7 of the algorithm has complexity $O(|V|)$. Suppose the loop of lines 8-21 executes k times. This value k is at least 1 and at most $k_{max} + 1$ as proven by Lemma 2. Within one round of the loop of lines 8-21, line 15 of the algorithm is executed exactly $|E|$ times. If we implement L using a hash table we can achieve a total complexity of $O(k \cdot |E|)$ for line 15. If we implement *signatureId* using a hash table then lines 17, 18 and 20 each cost $O(|V|)$ time for a single execution. This is because a signature is of size $O(|V|)$[1] and hash collisions of *signatureId* will often occur if signatures are recurrent. A single execution of line 21 can be implemented in constant time if the counter is used, or linear time in $|V|$ if a hash table is used to compare the range of the two partitions. In total we end up with a time complexity of $O(k \cdot (|E| + |V|^2))$.

3.2.2 Sequential hash-based algorithm

Description The hash-based sampling algorithm gives up correctness in return for efficiency. This allows fast estimation of a graph's bisimulation partition, which could then be used for example to hint a sampling algorithm about which nodes to sample. It is inspired by the notion of signatures, like the exact algorithm of Blom and Orzan (see Section 3.2.1).

On its own, it can be used for various applications where 100% accuracy of the bisimulation partition is not required. It attempts to compute the unbounded bisimulation partition of a graph as accurately as it can. As such, the hash-based partition algorithm is a best-effort algorithm. There are two operators used in the algorithm which need further defining:

1. $\hbar(...)$: A hash function which takes a number of objects as input and produces some hash value as output. For example: it could simply evaluate to 0 no matter what the input is, but

[1]This is an overestimation. It is more accurate to say the signature is of size $O(\max_{v \in V} deg(v))$.

this wouldn't make for a good function. We have tested a number of hash functions and for our experiments we settled on using the CRC32 algorithm [8], which gave excellent results.

2. $\diamond(H)$: A function which combines the hash values (H) returned by $\hbar$. If for two nodes the set of hash values H are the same, then this function should give the same value, hence it must compute its value in an associative and commutative way, unless an ordering on the set H is applied. For example: one could choose the function which simply evaluates to 0, or the exclusive disjunction ($\oplus$) if $\hbar$ returns sequences of bits. From our experience the combine operator which gave the best results is simple integer addition. That is: $\diamond(H) \equiv \sum_{h \in H} h$.

The quality of the hash-based partition algorithm depends very much on the quality of the functions chosen. In our experiments we dealt with 32-bit integers which allows for over 4 billion possible hash values. The randomness of how the hash function maps the input to one of the possible values is also of importance. With the functions we chose and the data sets we tested, the quality was sufficient enough to achieve a 100% accuracy.

In lines 2-3 of the algorithm we compute an estimation of the $k = 0$ bisimulation partition. We do this by simply making a partition block for each unique (hashed) node label, which involves applying the $\diamond$ function on the set which contains the single element $\hbar(\lambda_V(v))$. Then in lines 4-12 we iteratively refine the previous partition until the partition node longer is refined. For each node we compute a hash value representation of its signature. We do this by combining the set of hash values H. Each hash value in H is computed by some hash function $\hbar$ which is given the node's label and the tuples of outgoing-edge labels and the identifier of the edge's target node's previous partition block.

Algorithm *SequentialHashBasedBisimulationPartition*$(V, E, s, t, \lambda_V, \lambda_E)$

```
1.  $k \leftarrow 0$
2.  **for** $v \in V$
3.      **do** $P_k[v] \leftarrow \diamond(\{\hbar(\lambda_V(v))\})$
4.  **repeat**
5.      $k \leftarrow k + 1$
6.      **for** $u \in V$
7.          **do** $H \leftarrow \{\hbar(\lambda_V(u))\}$
8.              **for** $e \in E$ such that $s(e) = u$
9.                  **do** $\triangleright$ Let $v$ denote $t(e)$
10.                     $H \leftarrow H \cup \{\hbar(\lambda_E(e), P_{k-1}[v])\}$
11.             $P_k[u] \leftarrow \diamond(H)$
12. **until** $\left|P_k^{\rightarrow}\right| = \left|P_{k-1}^{\rightarrow}\right|$
13. **return** $P_k$
```

Although we cannot argue the correctness of the hash-based algorithm, we can argue why it works well. Following Proposition 1 we know that signatures can be used to represent the partition blocks, as we make use of for our algorithm in Section 3.2.1. Indeed, Blom and Orzan [6], and Blom and Pol [7] use the same notion of signatures to argue the correctness of the algorithm which computes the signatures iteratively. The traditional algorithm terminates because at some point the partition will not change any longer, as argued by Corollary 2. We terminate when the size of the partition does not change which is justified by Lemma 2. Because comparing signatures can be costly ($O(|V|)$ for each comparison, if using a hash table) we can sacrifice correctness for efficiency. Instead of comparing signatures, we compute a hash value representation of the signatures and compare those instead.

Complexity For the complexity analysis of the hash-based partition algorithm we shall assume that the computation of $\diamond(H)$ runs in $O(H)$ time. We also assume that $\{e \in E | s(e) = u\}$ is easily accessible for line 8 of the algorithm. That is: we can iterate the outgoing edges of a node u in linear ($O(deg(u))$) time. Again we cannot formally argue the complexity of the algorithm because it depends very much on the functions $\hbar$ and $\diamond$. However it makes sense that the number of executions of the loop of lines 4-12 is roughly k_{max}, analogous to the signature-based exact algorithm. From our experience this has always been the case. Once again this depends very much on the quality of the hash functions used. Thus we get running time of $O(k \cdot (|V| + |E|))$, where k is the number of executions of the loop of lines 4-12.

3.2.3 Distributed signature-based algorithm

Description For the distributed signature-based algorithm we have implemented a slight variant of the distributed algorithm by Blom and Orzan [6]. In our model we consider W worker machines and a single coordinator. Administrative work is done on the coordinator, while the workers are responsible for the actual computation of the signatures of the nodes. Since the computations of the workers and the administrative work of the coordinator is always done separately (mutually exclusive with respect to time), one could easily run the coordinator in a thread on one of the worker machines without negatively impacting the performance.

Figure 3.12 depicts a sequence diagram of the coordinator and two worker machines. In essence, the distributed algorithm repeatedly refines the partition locally (on the nodes located on a machine), and then shares the partition block identifier of each node between machines. In each round of sharing, only those partition block identifiers that are of interested for another machine are shared with it. The unique partition block identifiers are computed on the coordinator and are then given to the workers as Globally Unique Identifiers (GUIDs).

Event-driven *DistributedSignatureBasedBisimulationPartition-Coordinator*

```
{ Event 1: Initialize coordinator }
1.  $counter \leftarrow 0$
2.  Let $signatureId[S] =$ nil for all signatures $S$
3.  Set $workerState[w] = Refining$ for each worker $w$
4.  send Clear to each worker

{ Event 2: upon receive pairs $M$ of signatures and local identifiers from worker $w$ }
1.  $idMap \leftarrow \emptyset$
2.  for $(S, id) \in M$
3.      do if $signatureId[S] =$ nil
4.            then $signatureId[S] \leftarrow counter$
5.                 $counter \leftarrow counter + 1$
6.          $idMap \leftarrow idMap \cup (id, signatureId[S])$
7.  send GUID mapping $idMap$ to worker $w$
8.  Set $workerState[w] = FinishedRefining$

{ Event 3: upon $workerState[w] = FinishedRefining$ for all workers $w$ }
1.  if Number of partition blocks has increased
2.     then Set $workerState[w] = Sharing$ for each worker $w$
3.          send Share to each worker
4.     else Set $workerState[w] = Collecting$ for each worker $w$
5.          send Collect to each worker

{ Event 4: upon receive FinishedSharing from worker $w$ }
1.  Set $workerState[w] = FinishedSharing$

{ Event 5: upon $workerState[w] = FinishedSharing$ for all workers $w$ }
1.  Let $signatureId[S] =$ nil for all signatures $S$
2.  Set $workerState[w] = Refining$ for each worker $w$
3.  send Refine to each worker

{ Event 6: upon receive local partition $P_w$ from worker $w$ }
1.  $P \leftarrow P \cup P_w$
2.  Set $workerState[w] = FinishedCollecting$

{ Event 7: upon $workerState[w] = FinishedCollecting$ for all workers $w$ }
1.  return $P$
```

Event-driven *DistributedSignatureBasedBisimulationPartition-Worker*

```
{ Event 1: upon receive Clear from coordinator $c$ }
1.  $counter \leftarrow 0$
2.  Let $signatureId[S] = \textbf{nil}$ for all signatures $S$
3.  for $v \in V$ such that $v$ is a node on this machine
4.      do $S \leftarrow (\lambda_V(v), \emptyset)$
5.         if $signatureId[S] = \textbf{nil}$
6.           then $signatureId[S] \leftarrow counter$
7.                $counter \leftarrow counter + 1$
8.         $P[v] \leftarrow signatureId[S]$
9.  send Pairs of signatures and local identifiers $signatureId$ to coordinator $c$

{ Event 2: upon receive GUID mapping $idMap$ from coordinator $c$ }
1.  for $(old\_id, new\_id) \in idMap$
2.      do for $v \in V$ such that $v$ is a node on this machine and $P[v] = old\_id$
3.             do $P[v] \leftarrow new\_id$

{ Event 3: upon receive Share from coordinator $c$ }
1.  ▷ Let $boundary(w)$ denote the nodes on this machine which have an incoming edge from a
    node on machine $w$
2.  for each worker $w$ except this machine
3.      do $update \leftarrow \{(v, P[v]) | v \in boundary(w)\}$
4.         send Partition update $update$ to worker $w$
5.  send FinishedSharing to coordinator $c$

{ Event 4: upon receive Partition update $update$ from worker $w$ }
1.  for $(v, p\_id) \in update$
2.      do $P[v] \leftarrow p\_id$

{ Event 5: upon receive Refine from coordinator $c$ }
1.  $counter \leftarrow 0$
2.  Let $signatureId[S] = \textbf{nil}$ for all signatures $S$
3.  for $u \in V$ such that $u$ is a node on this machine
4.      do $L \leftarrow \emptyset$
5.         for $e \in E$ such that $s(e) = u$
6.             do ▷ Let $v$ denote $t(e)$
7.                $L \leftarrow L \cup \{(\lambda_E(e), P[v])\}$
8.         ▷ Let $S = (\lambda_V(u), L)$ denote the signature of $u$
9.         if $signatureId[S] = \textbf{nil}$
10.          then $signatureId[S] \leftarrow counter$
11.               $counter \leftarrow counter + 1$
12.        $P'[u] \leftarrow signatureId[S]$
13. $P \leftarrow P'$
14. send Pairs of signatures and local identifiers $signatureId$ to coordinator $c$

{ Event 6: upon receive Collect from coordinator $c$ }
1.  send Local partition $P$ (only for local nodes) to coordinator $c$
```

Complexity For analyzing the bounds we look at the sequence diagram in Figure 3.12. We assume that nodes are evenly distributed among workers, and that the number of workers is at most the number of nodes. Let W be the number of workers.

Visit times First we look at the visit times complexity. In the initial phase we have W clear messages followed by W signature mappings and W GUID mappings. In the second phase we have

W share messages followed by $O(W^2)$ shares of signature identifiers and W messages indicating that the worker is finished sharing. We also get W refine messages followed by W signature mappings and W GUID mappings. This is repeated $k = k_{max} + 1$ times. In the final phase we have W collect messages and W subsequent partition segment messages. The visit times for the coordinator is then $O(k \cdot W)$ and for a single worker it is also $O(k \cdot W)$. Thus in total we have a visit times complexity of $O(k \cdot W^2)$.

Data shipment Now we shall look at the data shipment complexity. A clear message is simply a command without any additional information. Likewise, a collect message, share message, and a message indicating that a worker is finished sharing carry no additional information. A message shipping signatures from a worker to the coordinator contains all unique signatures for nodes located on that machine, and the identifiers the worker has assigned to them. A node's signature is of size $O(|V|)$; this means that $O(\frac{|V|^2}{W})$ data is shipped from each worker to the coordinator. The coordinator responds to each worker with a GUID mapping, which is of size $O(\frac{|V|}{W})$, because there are at most as many partition blocks as there are nodes. Workers sharing partition identifiers with each other send messages, each of size $O(\frac{|V|}{W})$. A partition segment collection message is also of size $O(\frac{|V|}{W})$. In total we get $O(k \cdot |V|^2)$ data shipment for the coordinator, as well as $O(k \cdot (|V| + \frac{|V|^2}{W}))$ for each worker. The coordinator and workers combined then have a total of $O(k \cdot |V|^2)$ data shipment.

Makespan To analyze the time complexity we will have to think about the work being done at each event of a machine. First we shall look at the complexity of the coordinator.

Event 1. This event is fired once. We set the state of each worker and send a message to each worker, taking $O(W)$ operations.

Event 2. This event is fired k times for each worker. From each worker, we receive roughly $\frac{|V|}{W}$ signatures. In total, comparing signatures takes $O(k \cdot |V|^2)$ time, including every round of W events of the k rounds.

Event 3. This event is fired k times, and we can simply count the number of unique signatures we saw during Event 2 of this round. Therefore it takes $O(k \cdot W)$ work, including all k rounds.

Event 4. This event is fired k times for each worker and takes $O(1)$ every time it is fired, giving a total of $O(k \cdot W)$ operations.

Event 5. This event is fired k times. Each time we simply clear the associative array $signatureId$ and write to each worker's state. In total we get $O(k \cdot W)$ operations.

Event 6. This event is fired once per worker. For each worker we do work linear in the number of its local nodes, hence we get a total complexity of $O(|V|)$.

Event 7. This event is fired once and we simply return the partition, taking $O(1)$ time.

All events. Combining all the previous events, we get a total running time of $O(k \cdot |V|^2)$ for the coordinator.

Next we look at the complexity of a single worker. As mentioned before, we assume that nodes are evenly distributed among the workers. Since the workers always run in parallel, we will keep the factor of W^{-1} in our complexities.

Event 1. This event is fired once. We simply set the signature of each (local) node v to $sig_0(v)$, which only looks at the node label, taking a total of $O(\frac{|V|}{W})$ time.

Event 2. This event is fired k times. The GUID mapping contains at most $\frac{|V|}{W}$ pairs. Using an associative array we can achieve $O(\frac{|V|}{W})$ complexity per round, giving a total of $O(k \cdot \frac{|V|}{W})$.

Event 3. This event is fired k times. We assume that $boundary(w)$ is readily available. Since $boundary(w)$ is a subset of the local nodes, we get a complexity of $O(k \cdot W \cdot \frac{|V|}{W}) = O(k \cdot |V|)$.

Event 4. This event is fired k times for each worker. Each update is of size $O(\frac{|V|}{W})$ so here we also get a complexity of $O(k \cdot |V|)$.

Event 5. This event is fired k times. We do work for each local node, which involves comparing its signature with other local node signatures, taking $O(\frac{|V|^2}{W^2})$ comparisons for all nodes. We also do work for each edge, but we do not assume that edges are spread evenly among workers. Thus in total we get $O(k \cdot (|E| + \frac{|V|^2}{W^2}))$.

Event 6. This event is fired once and we simply ship the local partition to the coordinator, taking $O(1)$ time.

All events. Combining all the previous events, we get a total worst-case running time of $O(k \cdot (|E| + |V| + \frac{|V|^2}{W^2}))$ for a worker.

Combining the coordinator and all of the workers, we get a total running time of $O(k \cdot (|E| + |V|^2))$. Note that (potentially) much of work is done by the coordinator, during the computation of the GUIDs (Event 2). This could be partially solved by using a distributed hash table. The hash-based algorithm eliminates the need for this computation.

3.2.4 Distributed hash-based algorithm

Description Like the distributed signature-based algorithm from Section 3.2.3 we consider the model of W worker machines and a single coordinator. Administrative work in done on the coordinator, while the workers are responsible from the actual computation of the hash values of the nodes. Since the computations of the workers and the administrative work of the coordinator is always done separately (mutually exclusive with respect to time), one could easily run the coordinator in a thread on one of the worker machines without negatively impacting the performance.

Figure 3.13 depicts a sequence diagram of the coordinator and two worker machines. Like the signature-based algorithm, the distributed hash-based algorithm repeatedly refines the partition locally (on the nodes located on a machine), and then shares the partition block hash value of each node between machines. In each round of sharing, only those partition block hash values that are of interested for another machine are shared with it. In contrast to the signature-based distributed algorithm, there is no need to compute Globally Unique Identifiers (GUIDs), which saves the coordinator from a great deal of work.

Like the sequential hash-based algorithm of Section 3.2.2 we give up correctness for efficiency. The functions $\hbar$ and $\diamond$ used in the algorithm are exactly the same as the ones for the sequential variant. For a discussion on those functions see Section 3.2.2.

Event-driven *DistributedHashBasedBisimulationPartition-Coordinator*

```
{ Event 1: Initialize coordinator }
1.  $H \leftarrow \emptyset$
2.  Set $workerState[w] = Refining$ for each worker $w$
3.  send Clear to each worker

{ Event 2: upon receive Set of Hash values $H_w$ from worker $w$ }
1.  $H \leftarrow H \cup H_w$
2.  Set $workerState[w] = FinishedRefining$

{ Event 3: upon $workerState[w] = FinishedRefining$ for all workers $w$ }
1.  if Number of partition blocks ($|H|$) has increased
2.     then Set $workerState[w] = Sharing$ for each worker $w$
3.          send Share to each worker
4.     else Set $workerState[w] = Collecting$ for each worker $w$
5.          send Collect to each worker

{ Event 4: upon receive FinishedSharing from worker $w$ }
1.  Set $workerState[w] = FinishedSharing$

{ Event 5: upon $workerState[w] = FinishedSharing$ for all workers $w$ }
1.  $H \leftarrow \emptyset$
2.  Set $workerState[w] = Refining$ for each worker $w$
3.  send Refine to each worker

{ Event 6: upon receive local partition $P_w$ from worker $w$ }
1.  $P \leftarrow P \cup P_w$
2.  Set $workerState[w] = FinishedCollecting$

{ Event 7: upon $workerState[w] = FinishedCollecting$ for all workers $w$ }
1.  return $P$
```

Event-driven *DistributedHashBasedBisimulationPartition-Worker*

```
{ Event 1: upon receive Clear from coordinator $c$ }
1.  for $v \in V$ such that $v$ is a node on this machine
2.      do $P[v] \leftarrow \diamond(\{\hbar(\lambda_V(v))\})$
3.  send Set of Hash values $\{P[v] | v$ is a node on this machine$\}$ to coordinator $c$

{ Event 2: upon receive Share from coordinator $c$ }
1.  ▷ Let $boundary(w)$ denote the nodes on this machine which have an incoming edge from a
    node on machine $w$
2.  for each worker $w$ except this machine
3.      do $update \leftarrow \{(v, P[v]) | v \in boundary(w)\}$
4.          send Partition update $update$ to worker $w$
5.  send FinishedSharing to coordinator $c$

{ Event 3: upon receive Partition update $update$ from worker $w$ }
1.  for $(v, p\_id) \in update$
2.      do $P[v] \leftarrow p\_id$

{ Event 4: upon receive Refine from coordinator $c$ }
1.  for $u \in V$ such that $u$ is a node on this machine
2.      do $H \leftarrow \{\hbar(\lambda_V(u))\}$
3.          for $e \in E$ such that $s(e) = u$
4.              do ▷ Let $v$ denote $t(e)$
5.                  $H \leftarrow H \cup \{\hbar(\lambda_E(e), P[v])\}$
6.          $P'[u] \leftarrow \diamond(H)$
7.  $P \leftarrow P'$
8.  send Set of Hash values $\{P[v] | v$ is a node on this machine$\}$ to coordinator $c$

{ Event 5: upon receive Collect from coordinator $c$ }
1.  send Local partition $P$ (only for local nodes) to coordinator $c$
```

Complexity For analyzing the bounds we look at the sequence diagram in Figure 3.13. We assume that nodes are evenly distributed among workers, and that the number of workers is at most the number of nodes. Let W be the number of workers.

Visit times First we look at the visit times complexity. In the initial phase we have W clear messages followed by W sets of hash values. In the second phase we have W share messages followed by $O(W^2)$ shares of block identifiers and W messages indicating that the worker is finished sharing. We also get W refine messages followed by W sets of hash values. This is repeated $k = k_{max} + 1$ times. In the final phase we have W collect messages and W subsequent partition segment messages. The visit times for the coordinator is then $O(k \cdot W)$ and for a single worker it is also $O(k \cdot W)$. Thus in total we have a visit times complexity of $O(k \cdot W^2)$.

Data shipment Now we shall look at the data shipment complexity. A clear message is simply a command without any additional information. Likewise, a collect message, share message, and a message indicating that a worker is finished sharing carry no additional information. A message shipping hash values from a worker to the coordinator contains all unique partition block hash values for nodes located on that machine. A hash value (block identifier) is of size $O(1)$; this means that $O(\frac{|V|}{W})$ data is shipped from each worker to the coordinator. In contrast to the signature-based algorithm, the coordinator does not have to send a remapping of some sort. Workers sharing partition block hash values with each other send message, each of size $O(\frac{|V|}{W})$. A partition segment collection message is also of size $O(\frac{|V|}{W})$. In total we get $O(k \cdot |V|)$ data shipment for the coordinator, as well as $O(k \cdot |V|)$ for each worker. The coordinator and the workers combined then have a total of $O(k \cdot W \cdot |V|)$ data shipment.

Makespan To analyze the time complexity we will have to think about the work being done at each event of a machine. First we shall look at the complexity of the coordinator.

Event 1. This event is fired once. We set the state of each worker and send a message to each worker, taking $O(W)$ operations.

Event 2. This event is fired k times for each worker. From each worker, we receive roughly $\frac{|V|}{W}$ block identifiers. In total, comparing these identifiers takes $O(k \cdot |V|)$ time, including every round of W events of the k rounds.

Event 3. This event is fired k times, and we can simply count the number of unique hash values we saw during Event 2 of this round. Therefore it takes $O(k \cdot W)$ work, including all k rounds.

Event 4. This event is fired k times for each worker and takes $O(1)$ every time it is fired, giving a total of $O(k \cdot W)$ operations.

Event 5. This event is fired k times. Each time we simply clear the hash table H and write to each worker's state. In total we get $O(k \cdot W)$ operations.

Event 6. This event is fired once per worker. For each worker we do work linear in the number of its local nodes, hence we get a total complexity of $O(|V|)$.

Event 7. This event is fired once and we simply return the partition, taking $O(1)$ time.

All events. Combining all the previous events, we get a total running time of $O(k \cdot |V|)$ for the coordinator.

Next we look at the complexity of a single worker. As mentioned before, we assume that nodes are evenly distributed among the workers. Since the workers always run in parallel, we will keep the factor of W^{-1} in our complexities.

Event 1. This event is fired once. We simply set the block identifier of each (local) node v to the hash value of its label, taking a total of $O(\frac{|V|}{W})$ time.

Event 2. This event is fired k times. We assume that *boundary*(w) is readily available. Since *boundary*(w) is a subset of the local nodes, we get a complexity of $O(k \cdot W \cdot \frac{|V|}{W}) = O(k \cdot |V|)$.

Event 3. This event is fired k times for each worker. Each update is of size $O(\frac{|V|}{W})$ so here we get a complexity of $O(k \cdot |V|)$.

Event 4. This event is fired k times. We do work for each local node, which involves computing the new hash value based on its outgoing edges. As such, we do work for each edge, and we do not assume that edges are spread evenly among workers. Thus in total we get $O(k \cdot (|E| + \frac{|V|}{W}))$.

Event 5. This event is fired once and we simply ship the local partition to the coordinator, taking $O(1)$ time.

All events. Combining all the previous events, we get a total worst-case running time of $O(k \cdot (|E| + |V|))$ for a worker.

Combining the coordinator and all of the workers, we get a total running time of $O(k \cdot (|E| + |V|))$.

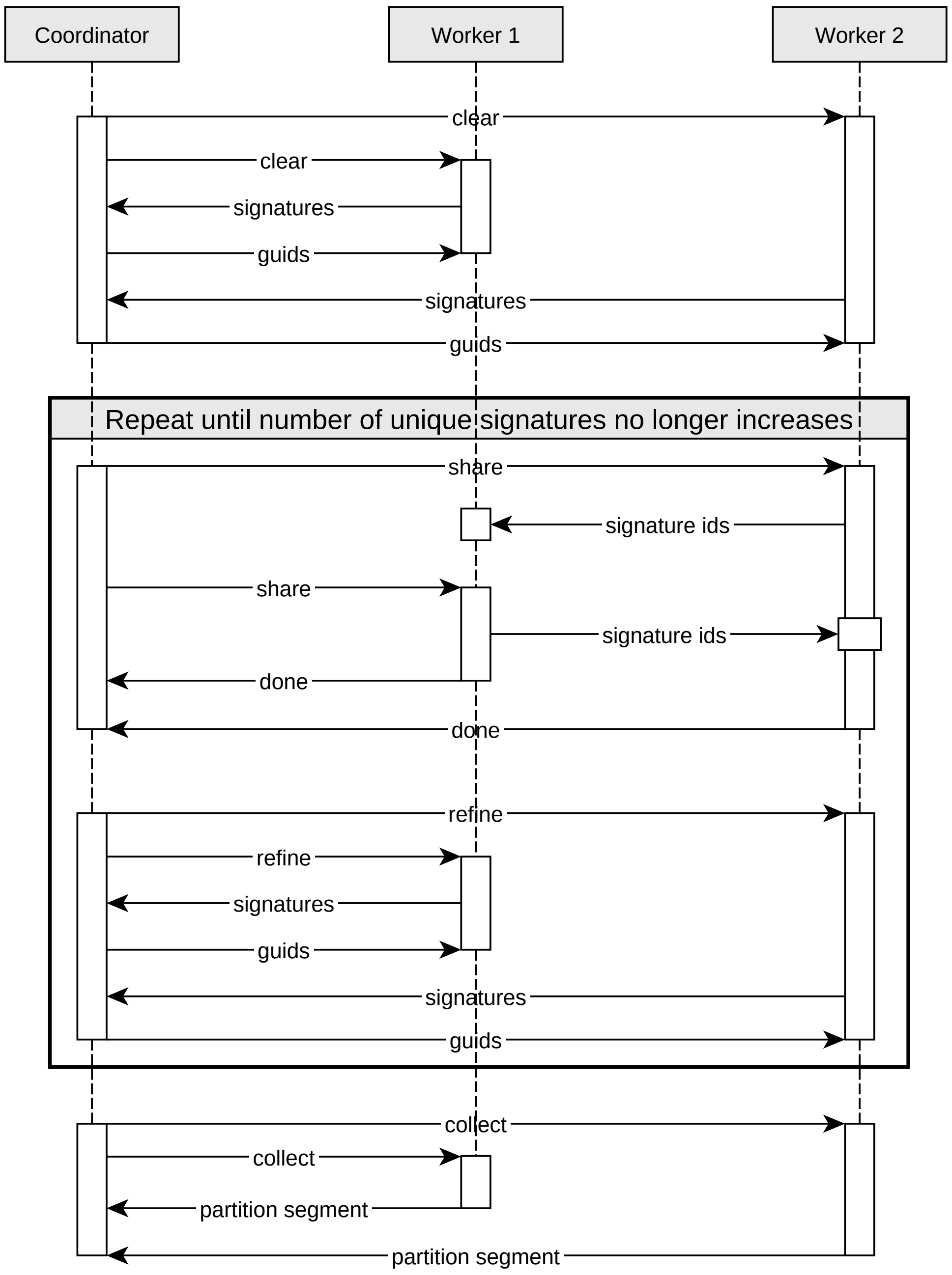


Figure 3.12: Example sequence diagram of the distributed signature-based algorithm with two workers

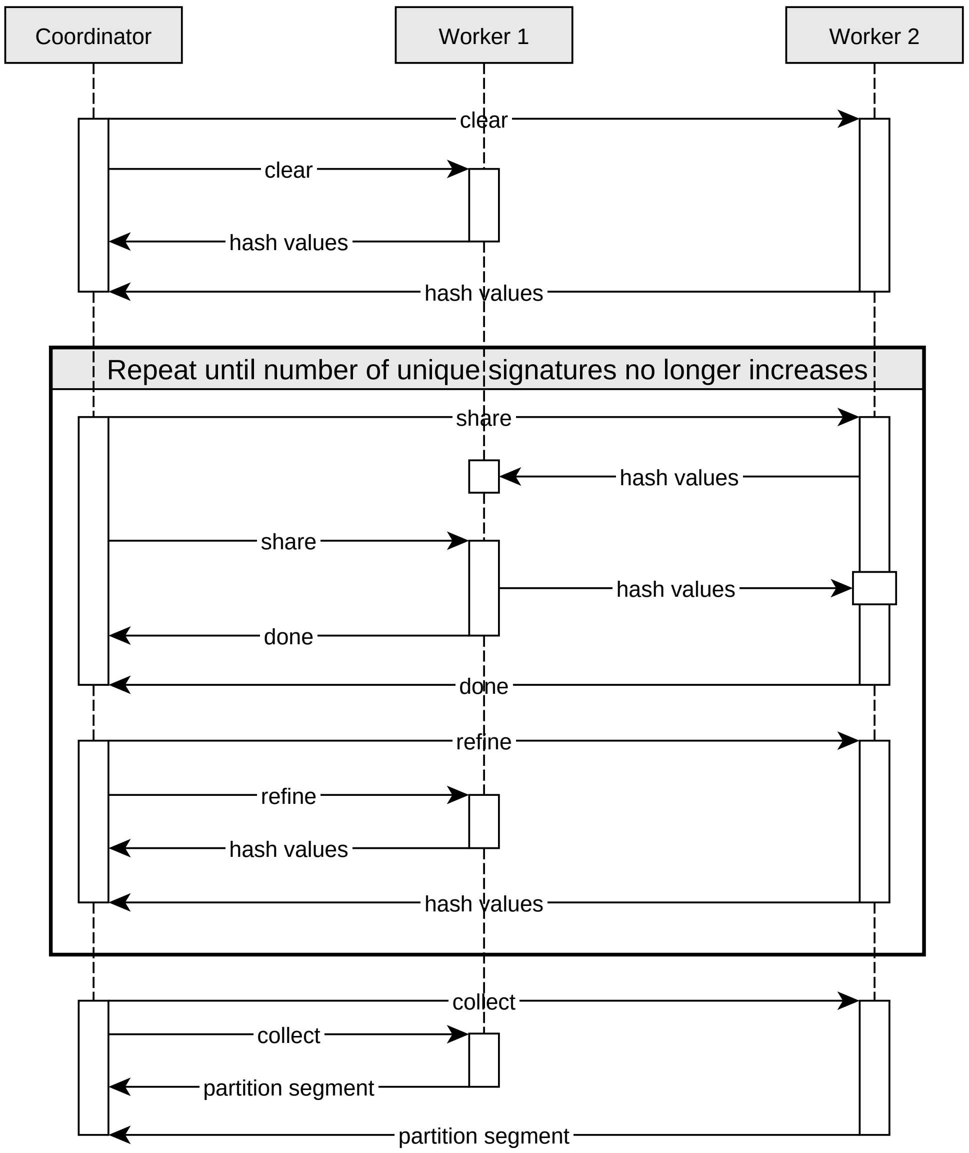


Figure 3.13: Example sequence diagram of the distributed hash-based algorithm with two workers

Chapter 4

Data sets

In this chapter we will introduce and discuss the data sets which were used in our empirical study. Our experiments were set up to investigate the quality of the sampling algorithms, and the quality and performance of the partition algorithms. It makes sense that some data sets are better suited to be used for bisimulation reduction than others. We have a number of data sets which we have categorized, so as to investigate how well graphs are suited for bisimulation, as well as how well bisimulation preserving sampling applies to different categories (or types) of graphs. For some categories the number of data sets representing it is low, and we propose that future work can be done to better understand which types of graphs are well-suited to sampling while preserving the bisimulation structure.

A number of synthetic data sets were generated to investigate how well the bisimulation structure of graphs with a known structure is preserved. We also have a few real world data sets; this is to verify the applicability of bisimulation preserving sampling for different types of graphs, and to test the performance of our algorithms on larger data sets which do not have a very well-defined structure.

4.1 Overview

Table 4.1 shows an overview of the data sets which were used. Here follows an explanation of the properties shown in the table.

1. Type: an informal description of the type of graph. We follow the categorization used by [16] as much as possible.
2. $|V|$: The size of the graph in number of nodes.
3. $|E|$: The size of the graph in number of edges.
4. k_{max}: The largest value for k-bisimulation which refines the $(k-1)$-bisimulation partition.
5. $|\mathcal{P}|$: The number of partition blocks modulo k_{max}-bisimulation.
6. D: The diameter of the graph.
7. Labels on: information about which entities are labeled.

[1]To be found at http://konect.uni-koblenz.de/networks/advogato
[2]To be found at http://dbtune.org/jamendo/
[3]To be found at http://grouplens.org/datasets/movielens/
[4]To be found at http://snap.stanford.edu/data/wiki-Elec.html
[5]To be found at http://staffweb.cms.gre.ac.uk/~wc06/partition/
[6]To be found at http://campuscurico.utalca.cl/~rangles/gdbench/

Table 4.1: Data sets used in experiments

Name	Type	$\|V\|$	$\|E\|$	k_{max}	$\|\mathcal{P}\|$	D	Labels on
Real graphs							
Advogato[1]	Social	7,422	56,507	4	3,850	11	Edges
Jamendo[2]	RDF, features	484,610	1,047,921	3	228	4	Edges
MovieLens100k[3]	Ratings, bipartite	2,625	100,000	1	12	1	Edges
WikiElec[4]	Contact	7,220	112,139	4	4,444	10	Nodes and edges
WingNodal[5]	Geometry	10,937	150,976	23	10,937	26	Nodes
Synthetic graphs							
Chains	Structure	9,000	7,500	5	60	5	Nodes and edges
DAG	Structure, random	261	728	4	9	5	Neither
ErdosRenyi	Random	10,000	9,956	15	2,204	91	Neither
GDGenerator[6]	Social, ratings, random	1,000	11,751	8	745	30	Nodes and edges
L-DAG	Structure, random	608	1,828	3	19	5	Edges
Stars	Structure	9,000	7,500	1	60	1	Nodes
Trees	Structure	9,009	8,866	1	60	5	Nodes

4.2 Discussion of data sets

4.2.1 Advogato

Advogato is a an online community of developers. Developers can indicate how much they trust other developers. This forms a graph where nodes are people and edges between them are ratings of how much one person (source node) trusts another person (target node). The number of partition blocks is relatively high compared to the number of nodes, as seen in Table 4.1. Meaning that if we hope to achieve 100% accuracy with our sample we must sample at least about 51% of the nodes in graph.

In Figure A.1 we can see that there are a few partition blocks which cover about a third of the nodes of the graph. About 50.5% of the nodes of the graph have their own partition block, resulting in the long tail of Figure A.1. This indicates that social networks are often diverse with respect to bisimilarity of their nodes. As seen in Figure A.3 it may be more useful to perform bounded bisimulation ($k = 0, 1$) on such graphs.

This power law behavior seems to be common in many of our data sets. Luo et al. in [20] also observe this power law and analyze it in-depth.

4.2.2 Jamendo

This data set is a collection of Resource Description Framework (RDF) triples. It relates musicians with records, albums, etc. The graph which this data was transformed into exists of subjects and objects as nodes, and predicates as labeled edges. Because of faulty Uniform Resource Identifiers (URIs) in the data, it was sanitized before we used it for our experiments. This meant the removal of triples with invalid URIs. The number of partition blocks is relatively low compared to the number of nodes, as seen in Table 4.1. This is reflected with good results from our experiments.

Again we observe a power law behavior in the size of the partition blocks (see Figure A.1). The largest partition most likely consists of dangling objects.

4.2.3 MovieLens100k

This is a bipartite graph with users and movies. Users (nodes) of the MovieLens data set have given ratings (edges) to movies (nodes). Each rating lies between 1 and 5 as indicated by the edge labels.

Yet again we observe a power law behavior when looking at the number of nodes per partition block in Figure A.1. This is due to the combinations of ratings users can give and the frequency of those combinations. For example: there may be many users who give a rating of 5 to all the movies they have rated, whereas there may be only a handful of users who have dished out all possible ratings (1, 2, 3, 4 and 5). Apparently, out of all possible combinations of ratings a user can give, only 11 $(12 - 1)$ of those combinations are actually used.

4.2.4 WikiElec

The WikiElec data set consists of some Wikipedia users as nodes. Users can elect another user to become an administrator of Wikipedia. Other users can then vote for this administrator to become an administrator. The node labels are determined by three things: (1) did the user get elected? (2) did the user elect someone else? (3) did the user vote in an election? Based on these three observations the nodes get a label out of eight unique possibilities. The edge labels are determined by whether a user is electing another user, or voting in an election.

With the labels set up like this, the number of partition blocks is relatively large compared to the number of nodes. Once again we observe a power law behavior in Figure A.1. A large portion of users have never cast a vote or elected another user, and thus aggregate in the largest partition block where nodes have no outgoing edge. The tail of the plot in Figure A.1 is most likely due to the complex paths of votes and elections that can occur in the graph. Like the Advogato data set, it may also make sense for this data set to only look at bounded $(k = 0, 1)$ bisimulation as Figure A.3 shows there are much fewer partition blocks in those cases.

4.2.5 WingNodal

This data set is a graph of topological nodes connected to form a mesh. Originally the graph is undirected with 75,488 edges. Because we only work with directed graphs all edges were copied in the opposite direction. The graph is entirely connected and does not have edge labels. Therefore it is required that at least one pair of nodes has distinct node labels, otherwise all nodes would be in the same partition block. To achieve this, a single node (the one with the highest degree) has been given a distinct label, whereas all the other nodes have the same node label. Due to the complex connections in the mesh, this means that each node has its own partition block. As such, it seems that (undirected) meshes are not suited for bisimulation partitioning, but we have analyzed this data set regardless.

An interesting observation for this data set is that the Distance PMF of Figure A.2 and the partition size of Figure A.3 seem to be related. In fact, upon closer inspection of the other data sets, it is often the case that the partition size (as k grows) looks like the cumulative sum of the distance probabilities, though this is not always the case.

4.2.6 Chains

This synthetic data set contains a number of chains. Each chain is of length 6 (nodes) thus enforcing that $k_{max} = 5$. There are 10 types of chains: one type of chain with all node labeled “1”, one with all node labeled “2”, etc. This ensures that there are exactly 60 $(6 \cdot 10)$ partition blocks in the graph. Each of the 10 types of chains is repeated 150 times such that there are 9,000 nodes in total. Figure 4.1 illustrates the first three types of chains.

As shown in Figure A.1 the way this graph was generated ensures that each partition block covers exactly 150 nodes.

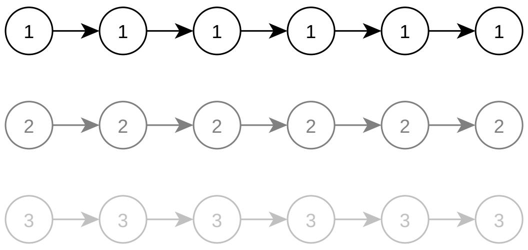


Figure 4.1: Illustration of Chains data set generation

4.2.7 DAG

For this data set a DAG (Directed Acyclic Graph) is generated. First a tree is generated using two parameters: α and β. The parameter α regulates the depth of the tree, whereas the β regulates the breadth of the tree. This process is described in pseudocode in *TreeGeneration*. The tree has unequal partition block sizes, as further discussed in secion 4.2.12. To remedy this we shall make copies of nodes in inferior partition blocks until all partition blocks are of equal size. This process is described in pseudocode in *DirectedAcyclicGraphGeneration*. An illustration of the latter algorithm is shown in Figure 4.2. In this figure, the original nodes from T of the algorithm are filled with white, and the copied nodes from G of the algorithm are filled with a shade of gray. This does not represent the actual labels of those nodes in the graph, but is merely there to help the reader understand what is happening.

```
Algorithm TreeGeneration($\alpha, \beta$)
1.  ▷ Let $T$ be the empty graph with nodes $V$
2.  Add a root node to $T$
3.  while $\max_{v \in V} depth(v) < \alpha$
4.      do for $v \in V$ such that $depth(v) = \max_{v \in V} depth(v)$
5.             do Choose $n$ in $[0, \beta]$ randomly uniformly
6.                Add $n$ new child nodes of $v$ to the graph $T$
7.  return $T$
```

```
Algorithm DirectedAcyclicGraphGeneration($\alpha, \beta$)
1.  $T \leftarrow TreeGeneration(\alpha, \beta)$
2.  ▷ Let $\mathcal{P}$ be the bisimulation partition of $T$
3.  ▷ Let $|P|$ denote the number of nodes in partition block $P \in \mathcal{P}$
4.  ▷ Let $G$ be a copy of $T$
5.  for $P \in \mathcal{P}$
6.      do for $i = |P|$ to $\max_{P \in \mathcal{P}} |P| - 1$
7.             do Select a node $v$ from partition $P$ randomly
8.                Add a copy $u$ of $v$ to $G$
9.                Replicate all incoming and outgoing edges of $v$ in $T$ for $u$ in $G$
10. return $G$
```

For our experiments we used a DAG data set generated with parameters $\alpha = 5$ and $\beta = 3$. The number of partition blocks, as shown in Table 4.1, is relatively low compared to the number of nodes. This was chosen in such a way as to make the data set well-suited for bisimulation partitioning.

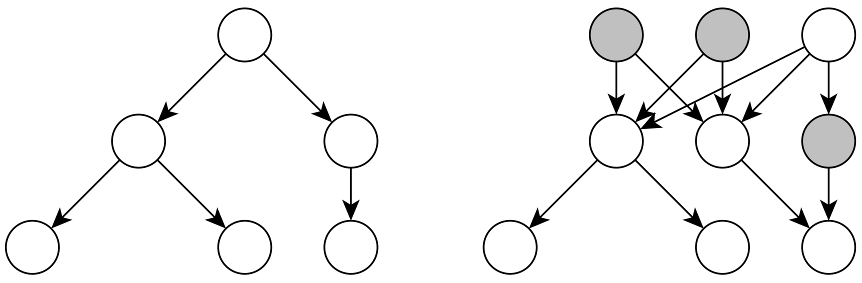


Figure 4.2: Illustration of DAG data set generation

4.2.8 ErdosRenyi

This synthetic data set was generated using the well-known Erdős-Rényi random graph model [10]. For our experiments we used $n = 10,000$ nodes and an independent probability of $p = \frac{1}{10,000}$ for an edge to exist. It is interesting to see that the partition block size distribution of this random graph shows a power law behavior, just like many other data sets (Figure A.1). Its k_{max} value is surprisingly high compared to other data sets. We suspect this is because of long chains which are often observed in Erdős-Rényi random graphs where $p = \frac{1}{n}$. The distance probability mass function of this graph (shown in Figure A.2) also looks rather unique among our data sets.

4.2.9 GDGenerator

This data set was generated using the application provided by the authors of [3]. It was used because we also wanted a synthetic social network data set. It consists of people and web pages as nodes. Edges between people indicate friendship; an edge from a person to a web page indicates that the person likes that web page.

Unfortunately not all properties of the synthetic graph correspond to properties we observe for real social networks. For example, the diameter is too large and the distance probability mass function (Figure A.2) is nothing like that of the Advogato or WikiElec data sets. However, its block size distribution (Figure A.1) is still close to that of the real data sets, exhibiting a strong power law behavior. As with other social network data sets is it not very well suited for unbounded bisimulation partitioning, but may be well-suited for bounded bisimulation.

4.2.10 L-DAG

The Labeled Directed Acyclic Graph (L-DAG) was generated in almost the exact the same way as the DAG data set of Section 4.2.7. This only difference is that in line 6 of the *TreeGeneration* algorithm each edge is assigned a label. The labels are such that the edge to the first child of v gets label “1”, the edge to the second child gets label “2”, and so on. By adding some variety to the edge labels, sampling algorithms which make use of this information may be more effective on this data set compared to others. Figure 4.3 illustrates an example L-DAG generated in the same way as the example of the DAG data set. Once again, the original nodes from T of the algorithm are filled with white, and the copied nodes from G of the algorithm are filled with a shade of gray. Note that the graphs generated are significantly different, in spite of the same structure generated by the tree generation algorithm. This is because the two nodes of the tree at depth 1 are in distinct partition blocks now.

For our experiments we used an L-DAG data set generated with parameters $\alpha = 5$ and $\beta = 3$.

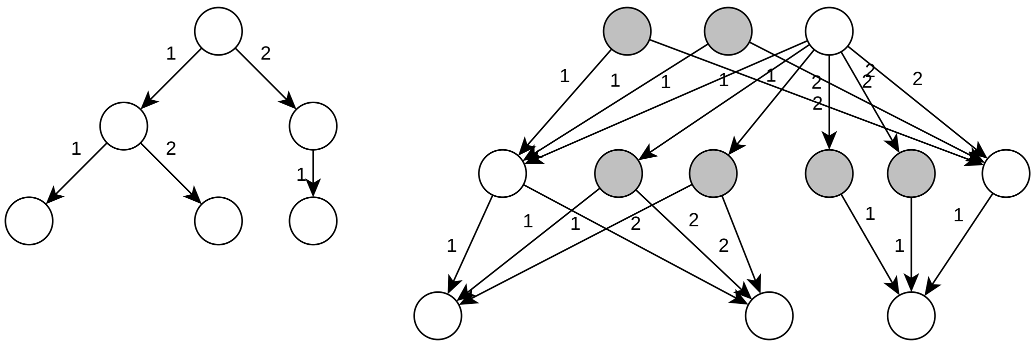


Figure 4.3: Illustration of L-DAG data set generation

4.2.11 Stars

This synthetic data set contains a number of stars. Each star has 1 center node and 5 leaf nodes, making it of size 6. The center node has outgoing edges to the leaf nodes, but not vice versa. This means that $k_{max} = 1$ for this data set. Much like the chains data set, there are 10 types of stars: each center node has label "1", but the leaf nodes have all distinct labels across all 10 types. Figure 4.4 demonstrates what this looks like for the first few stars. By repeating each type of star 150 times we get a total of 9,000 nodes, and each partition block covers 150 nodes.

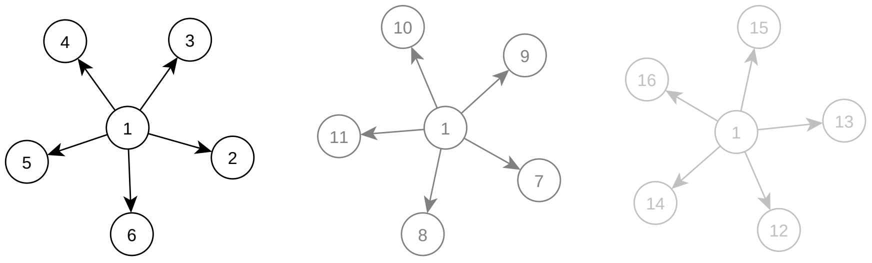


Figure 4.4: Illustration of Stars data set generation

4.2.12 Trees

This data set was generated by replicating a given tree structure. The skeleton tree structure contains 63 nodes. All the internal nodes have exactly two child nodes, and there is one root node. Figure 4.5 illustrates the skeleton tree structure but without labels. Labels are added to nodes from top-to-bottom such that the tree structure contains exactly 60 partition blocks. This means that the majority of nodes in the tree skeleton have distinct labels, and only a few of them have the same label. The tree structure is replicated 143 times to achieve a total graph size of 9,009 nodes.

As shown in Figure A.1 the nodes are distributed relatively equally among partition blocks. The exception is the partition block which contains the nodes from the skeleton structure which were not labeled to introduce new partition blocks. A single tree generally suffers from the problem that there will be one larger partition block, unless all of its nodes are labeled uniquely. This is because the nodes towards the bottom of the will be easier to group into a partition block than nodes higher up. Leaf nodes, for example, will group by their node labels, whereas their parents will group by their own node labels and also the labels of the leaves. This problem is remedied by the DAG data set (Section 4.2.7).

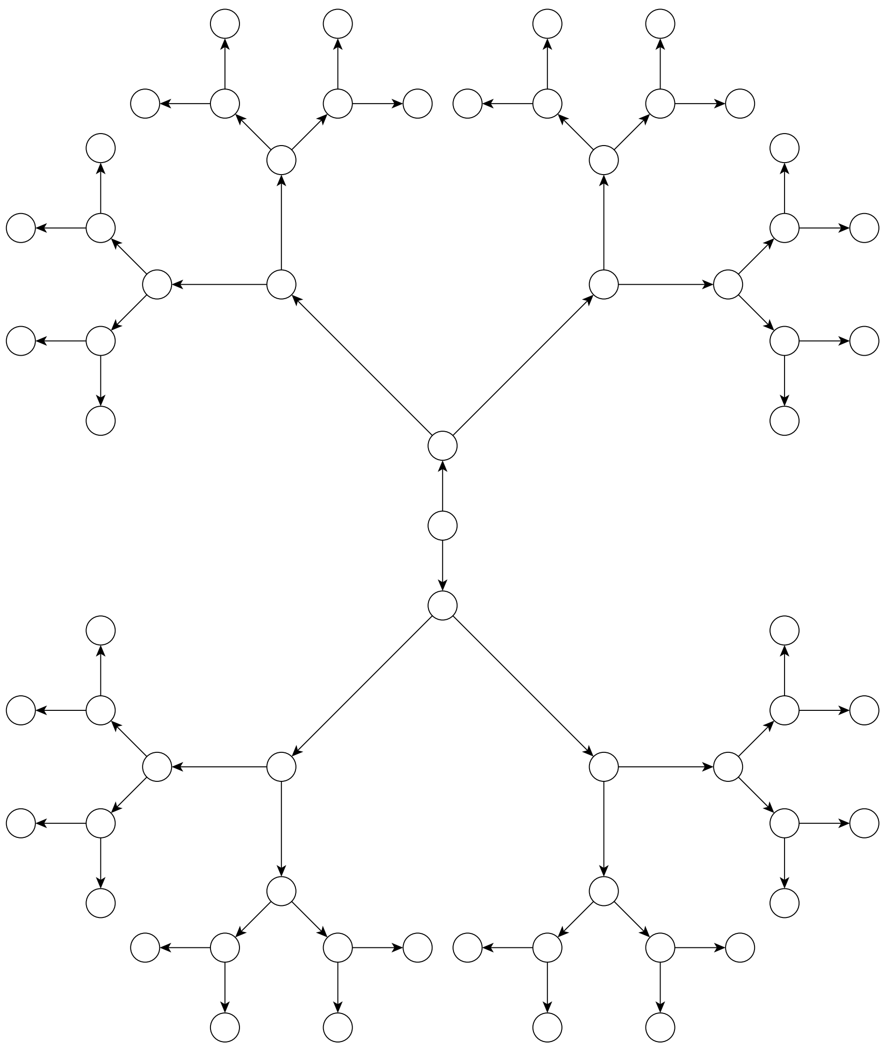


Figure 4.5: Illustration of Trees data set generation

Chapter 5

Results

In this chapter we will showcase all of the results we obtained from doing experiments. There are two sections: one for evaluating the quality of the sampling algorithms (Section 5.1), and one for evaluating the quality and performance of the different partition algorithms (Section 5.2). In each section we will also discuss the results and state some conclusions from our observations.

5.1 Sampling algorithms

The purpose of the experiments with sampling algorithms is to find patterns in the quality of certain sampling algorithms for different types of graphs. In the next section we will explain our setup and the variables we varied to obtain our results. Then we will move on to discussing the results and empirically evaluating the sampling algorithms using our metrics defined in Section 2.5. All of the sampling algorithms are described and analyzed in Section 3.1.

5.1.1 Experimental setup

All of the sampling algorithms require a number which tells them how large the sample can be. In the case of RE sampling this is the maximum number of edges, in all other cases it is the maximum number of nodes. Because the data sets greatly vary in size, it makes more sense to introduce a sampling fraction. We denote by $\alpha \in [0..1]$ the fraction of nodes (or edges) sampled from the original graph.

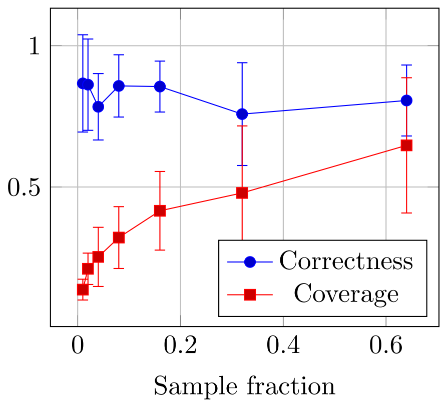
For each data set we are interested in measuring the correctness (P) and coverage (R), as well as the weighted correctness (WP) and weighted coverage (WR) for a number of scenarios. See Section 2.5 for a definition of these four metrics. In our experiments we observed the average (and standard deviation) over 10 runs of these four metrics, for each sampling algorithm, while varying the following parameters:

- α with the values 1%, 2%, 4%, 8%, 16%, 32% and 64%, and
- k with the values $0, ..., k_{max}$ where k_{max} depends on the data set.

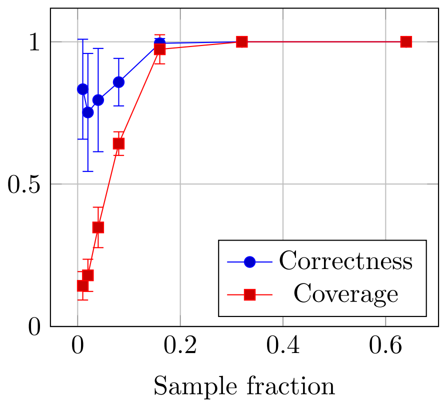
5.1.2 Correctness and coverage

Our result space is so large that we cannot possibly put it into a single document. Therefore we shall highlight some results that we found particularly interesting and discuss them in this section. However, if the reader is interested in exploring the entire result space, we have developed an interactive exploratory tool available on the following web page: `http://bit.ly/1iUXXoA`.

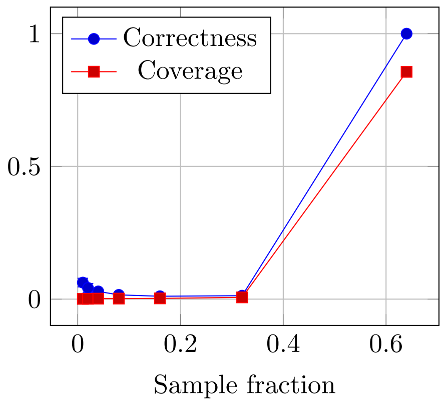
Standard metrics We know for many of data sets that there is a minimum number of nodes which need to be sampled in order to achieve 100% accuracy. For example: the L-DAG data set has 608 nodes and 19 partition blocks. This means that the sampling fraction must be at least $\frac{19}{608} \approx 3.1\%$, otherwise there simply can't be enough nodes in the sample to cover each partition block. For the Advogato data set this minimum is $\frac{3{,}850}{7{,}422} \approx 52\%$. In Figure 5.1b and Figure 5.1c we see the standard correctness and coverage for the L-DAG and Advogato data sets as α varies. In both cases k_{max} bisimulation was used to determine the metrics. From these pictures we can see that the point at which the samplers start to lose their excellent performance is when α is around the aforementioned ratios.



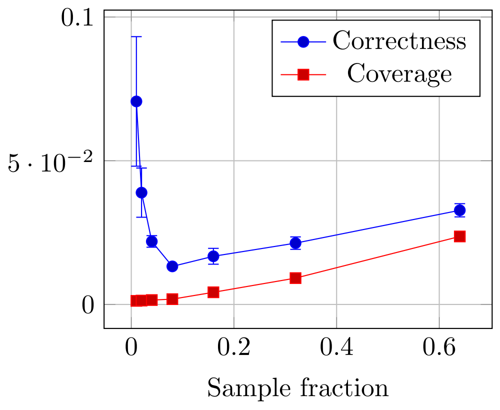
(a) L-DAG data set with BFS sampling



(b) L-DAG data set with DLBF sampling



(c) Advogato data set with BFS sampling



(d) Advogato data set with DLBF sampling

Figure 5.1: Correctness P and coverage R (with respect to the k_{max} bisimulation partition) of the sampled graph as α changes for four scenarios, average of 10 runs with standard deviation shown as error bars

It should not come as a surprise that the DLBF sampler performs well on the L-DAG data set. This sampling method was specifically designed to deal with the recurring edge labels that do not introduce new partition blocks, which we see very often of in the L-DAG data set. When we look at Figure 5.1a and Figure 5.1b we can see that DLBF outperforms BFS. BFS is the second best sampler for this data set, so the lead DLBF has here applies to all contenders.

Heat maps So far we have seen in Figure 5.1 that a particular sampling method may work well for a certain type of data set, and may not be so good for other types. To get a better overview of which samplers work well for which kind of data sets, we can have look at Table 5.1 and Table 5.2. In these tables we fix the sampling fraction to $\alpha = 4\%$ and display the correctness and coverage

Table 5.1: Heat map showing the average correctness (P) over 10 runs of the sampling algorithms at k_{max}-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0.03	0.03	0.02	0.27	1	0.01	0.02	0.23
Jamendo	0.94	0.96	0.99	0.02	1	0	0.93	0.02
MovieLens100k	1	1	1	0.48	1	1	1	0.47
WikiElec	0.03	0.02	0.04	0.06	0	0.01	0.03	0.12
WingNodal	0	0	0	0	0	0	0	0
GDGenerator	0.5	0.37	0.05	0.46	1	0.01	0.37	0.72
Stars	0.99	0.99	0.75	0.95	1	0.41	0.99	0.82
Chains	1	1	1	1	1	1	1	1
DAG	0.98	1	0.96	1	1	0.57	1	1
ErdosRenyi	0.87	0.79	0.73	1	1	0.99	0.77	1
L-DAG	0.78	0.84	0.8	0.7	1	0.19	0.83	0.73
Trees	0.94	0.98	0.56	0.43	1	0.34	0.97	0.44

for all combinations of data sets and samplers.

We can observe a number of interesting facts from Table 5.1 and Table 5.2. For instance: the LDF sampling method achieves very good correctness, but it sacrifices coverage to do so. It makes sense when we realize what LDF does: satisfy the partition blocks with the least amount of outgoing edges first. These low-degree partition blocks are typically easy to satisfy, as the number of nodes in their transitive closure tends to be smaller than that of high-degree nodes.

We can also observe that the BFS, DFS and RFS sampling algorithms generally perform equally well. These three are the three types of Queued Traversal discussed in Section 3.1.5. Thus, it seems that the method of exploration does not matter that much. When coverage is a concern, the exploration algorithms have good results compared to other sampling methods.

Overall, we see that for a sampling fraction as low as $\alpha = 4\%$ the coverage for most data sets is poor. As mentioned before in Secion 4.2, for some applications it may make more sense to restrict k to lower values. For this reason we show in Table 5.3 the coverage when we restrict ourselves to 1-bisimulation with samples of $\alpha = 4\%$.

Lower k values A noticeable jump in values when we compare Table 5.2 with Table 5.3 occurs for the Advogato data set. In Figure A.3 we see that the number of blocks goes down from 3850 at k_{max} to 16 at $k = 1$. From this we conclude that although social networks are too diverse and interconnected to be suitable for k_{max} bisimulation, their structure modulo $k = 1$ bisimulation may still be of use. From the heat map we also conclude that the samplers can be used to retrieve a good estimation for the bisimulation partition at $k = 1$.

In Table 5.4 we increase k up to 2. Note that for the MovieLens100k, Stars, and Trees data set we have $k_{max} = 1$, and therefore their partitions at $k = 1$ and $k = 2$ are the same: $\mathcal{P}_1 = \mathcal{P}_2$. For the remaining data sets we generally see lower values for $k = 2$ than for $k = 1$, and lower values yet for $k = k_{max}$ as expected.

Weighted metrics We have only looked at our standard (unweighted) metrics so far. As we know from Figure A.1 is it often the case that there are many small partition blocks, and relatively few large blocks. We discussed this power law behavior in the size of partition blocks in more detail in Chapter 4. Our example graph in Figure 2.1 can help us understand why this is often the case. Low degree nodes are typically easier to group into bisimulation partition blocks, simply because they have less fan-out. In many data sets we also observe a power law behavior in the degree distribution of the graph, as also discussed in more detail in Chapter 4. This means that as low degree nodes are sampled more often, we often sample nodes that belong to a large partition block. In turn this means that we can easily satisfy sampling the partition block of a large amount of (low degree) nodes, but leave the partition blocks of high degree nodes under-sampled. Therefore the weighted metrics that take into account the number of nodes per block will yield higher values. So when we are only covering 50% of the partition blocks of a graph, it may still be the case that those blocks cover 99% of the nodes.

In Table 5.5 we show the weighted coverage for k_{max}-bisimulation with sampling fraction $\alpha = 4\%$. Comparing this to the standard coverage of Table 5.2 we notice some major increases in values. If it is in the interest of the application to correctly cover as many nodes as possible, rather than as many partition blocks as possible, the weighted metrics give a better indication than the unweighted metrics. When considering lower k values we observe the same trend we did previously for lower k values. For example in Table 5.6 we see that for the weighted coverage we get higher values for $k = 1$ than what we saw for $k = k_{max}$. In Table 5.7 we also show the weighted coverage for $k = 2$ for comparison. Once again, note that for $k = 1$ the values are generally higher than for $k = 2$, and for $k = 2$ higher than for $k = k_{max}$.

Table 5.2: Heat map showing the average coverage (R) over 10 runs of the sampling algorithms at k_{max}-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0	0	0	0	0	0	0	0
Jamendo	0.2	0.22	0.22	0.01	0	0.04	0.2	0.01
MovieLens100k	0.2	0.19	0.32	0.64	0.08	0.87	0.22	0.62
WikiElec	0	0	0	0	0	0	0	0
WingNodal	0	0	0	0	0	0	0	0
GDGenerator	0	0	0	0	0	0	0	0
Stars	1	1	0.83	0.83	0.83	0.83	0.99	0.83
Chains	0.96	0.97	0.97	0.29	0.17	0.56	0.96	0.28
DAG	0.36	0.37	0.49	0.19	0.11	0.56	0.32	0.22
ErdosRenyi	0.05	0.05	0.01	0	0	0	0.04	0
L-DAG	0.25	0.28	0.35	0.09	0.05	0.21	0.22	0.08
Trees	0.99	1	0.55	0.49	0.48	0.71	0.99	0.49

Table 5.3: Heat map showing the average coverage (R) over 10 runs of the sampling algorithms at 1-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0.96	0.99	0.98	0.79	0.26	1	0.96	0.76
Jamendo	0.65	0.64	0.7	0.09	0.03	0.53	0.64	0.08
MovieLens100k	0.2	0.19	0.32	0.64	0.08	0.87	0.22	0.62
WikiElec	0.24	0.17	0.33	0.25	0	0.81	0.22	0.14
WingNodal	0.33	0.33	0.6	0.67	0.33	1	0.33	0.37
GDGenerator	0.7	0.7	0.8	0.83	0.5	0.75	0.75	0.93
Stars	1	1	0.83	0.83	0.83	0.83	0.99	0.83
Chains	1	1	1	0.83	0.5	1	1	0.84
DAG	1	1	1	0.75	0.5	1	1	0.75
ErdosRenyi	1	1	1	1	0.5	1	1	1
L-DAG	0.93	0.85	1	0.35	0.25	1	0.75	0.35
Trees	0.99	1	0.55	0.49	0.48	0.71	0.99	0.49

Table 5.4: Heat map showing the average coverage (R) over 10 runs of the sampling algorithms at 2-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0.02	0.02	0.03	0.01	0	0.14	0.02	0.01
Jamendo	0.24	0.25	0.3	0.01	0.01	0.06	0.24	0.01
MovieLens100k	0.2	0.19	0.32	0.64	0.08	0.87	0.22	0.62
WikiElec	0.01	0.01	0.02	0	0	0.28	0.01	0
WingNodal	0.25	0.25	0.3	0.33	0.25	1	0.33	0.25
GDGenerator	0.51	0.49	0.9	0.54	0.25	0.83	0.53	0.55
Stars	1	1	0.83	0.83	0.83	0.83	0.99	0.83
Chains	1	1	1	0.58	0.33	0.93	1	0.57
DAG	0.78	0.75	0.8	0.4	0.25	1	0.78	0.35
ErdosRenyi	1	1	0.98	0.57	0.25	1	1	0.6
L-DAG	0.31	0.21	0.38	0.08	0.06	0.44	0.27	0.08
Trees	0.99	1	0.55	0.49	0.48	0.71	0.99	0.49

Table 5.5: Heat map showing the average weighted coverage (WR) over 10 runs of the sampling algorithms at k_{max}-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0.44	0.44	0.45	0.46	0.11	0.46	0.43	0.46
Jamendo	1	1	1	0.31	0.31	0.91	1	0.33
MovieLens100k	0.89	0.93	0.97	0.94	0.64	1	0.88	0.83
WikiElec	0.16	0.15	0.24	0.21	0	0.25	0.16	0.23
WingNodal	0	0	0	0	0	0	0	0
GDGenerator	0.14	0.21	0.2	0.22	0.22	0.21	0.14	0.22
Stars	0.99	0.99	0.83	0.83	0.83	0.83	0.99	0.83
Chains	0.97	0.96	0.98	0.3	0.17	0.56	0.96	0.29
DAG	0.34	0.36	0.51	0.21	0.11	0.5	0.36	0.16
ErdosRenyi	0.74	0.74	0.67	0.55	0.37	0.69	0.73	0.57
L-DAG	0.21	0.25	0.36	0.09	0.05	0.19	0.19	0.07
Trees	0.99	0.99	0.56	0.51	0.51	0.71	0.99	0.52

Table 5.6: Heat map showing the average weighted coverage (WR) over 10 runs of the sampling algorithms at 1-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	1	1	1	0.92	0.41	1	1	0.93
Jamendo	1	1	1	0.42	0.31	0.99	1	0.4
MovieLens100k	0.89	0.93	0.97	0.94	0.64	1	0.88	0.83
WikiElec	0.4	0.4	0.88	0.6	0	0.99	0.41	0.82
WingNodal	1	1	1	1	1	1	1	1
GDGenerator	0.88	0.79	0.99	0.74	0.22	0.99	0.79	0.83
Stars	0.99	0.99	0.83	0.83	0.83	0.83	0.99	0.83
Chains	1	1	1	0.74	0.17	1	1	0.73
DAG	1	1	1	0.29	0.11	1	1	0.38
ErdosRenyi	1	1	1	1	0.37	1	1	1
L-DAG	0.9	0.78	1	0.14	0.07	1	0.84	0.15
Trees	0.99	0.99	0.56	0.51	0.51	0.71	0.99	0.52

Table 5.7: Heat map showing the average weighted coverage (WR) over 10 runs of the sampling algorithms at 2-bisimulation with sampling fraction $\alpha = 4\%$

Data set	BFS	DFS	DLBF	GL	LDF	RE	RFS	RN
Advogato	0.5	0.5	0.51	0.48	0.11	0.65	0.52	0.48
Jamendo	1	1	1	0.31	0.31	0.96	1	0.31
MovieLens100k	0.89	0.93	0.97	0.94	0.64	1	0.88	0.83
WikiElec	0.18	0.2	0.43	0.23	0	0.67	0.18	0.3
WingNodal	0.99	0.99	0.99	0.99	0.99	1	0.99	0.99
GDGenerator	0.67	0.72	0.98	0.39	0.22	0.97	0.78	0.59
Stars	0.99	0.99	0.83	0.83	0.83	0.83	0.99	0.83
Chains	1	1	1	0.31	0.17	0.82	1	0.28
DAG	0.56	0.61	0.73	0.18	0.11	1	0.57	0.14
ErdosRenyi	1	1	0.98	0.63	0.37	1	1	0.68
L-DAG	0.22	0.25	0.38	0.08	0.05	0.4	0.28	0.11
Trees	0.99	0.99	0.56	0.51	0.51	0.71	0.99	0.52

5.2 Partition algorithms

In this section we will first briefly demonstrate the effect of choosing different hash functions for the hash-based partition algorithm. Then we will move on to empirically evaluating the performance of the four different partition algorithms. The partition algorithms are described and analyzed in Section 3.2.

5.2.1 Quality

We have tested a number of hash functions in our experiments. As mentioned before in Section 3.2.2, the best choices we found were CRC32 for the $\hbar$ function, and $\diamond(H) \equiv \sum_{h \in H} h$. With these function, the accuracy of the hash-based algorithm is 100% for our data sets. That is: the hash-based algorithm with these hash functions produces the exact bisimulation partition, for each of our data sets.

To demonstrate the effectiveness of choosing the right hash-functions we will show some examples. In Figure 5.2 and Figure 5.3 we compare, for two data sets, the reduced graphs computed using the exact bisimulation partition and the estimated partition produced by the hash-based algorithm. In the latter case, the operator used to combine hash values is the exclusive disjunction. As mentioned before, when using the addition operator the estimated partition has been the same as the exact solution for all our data sets. However, even when we use the exclusive disjunction operator, we still obtain reasonable results. In total, for 5 out of 12 data sets the estimated solution using this operator is not equal to the exact solution. The two most severely wrong results are shown in the figures, which are for the L-DAG and MovieLens100k data sets.

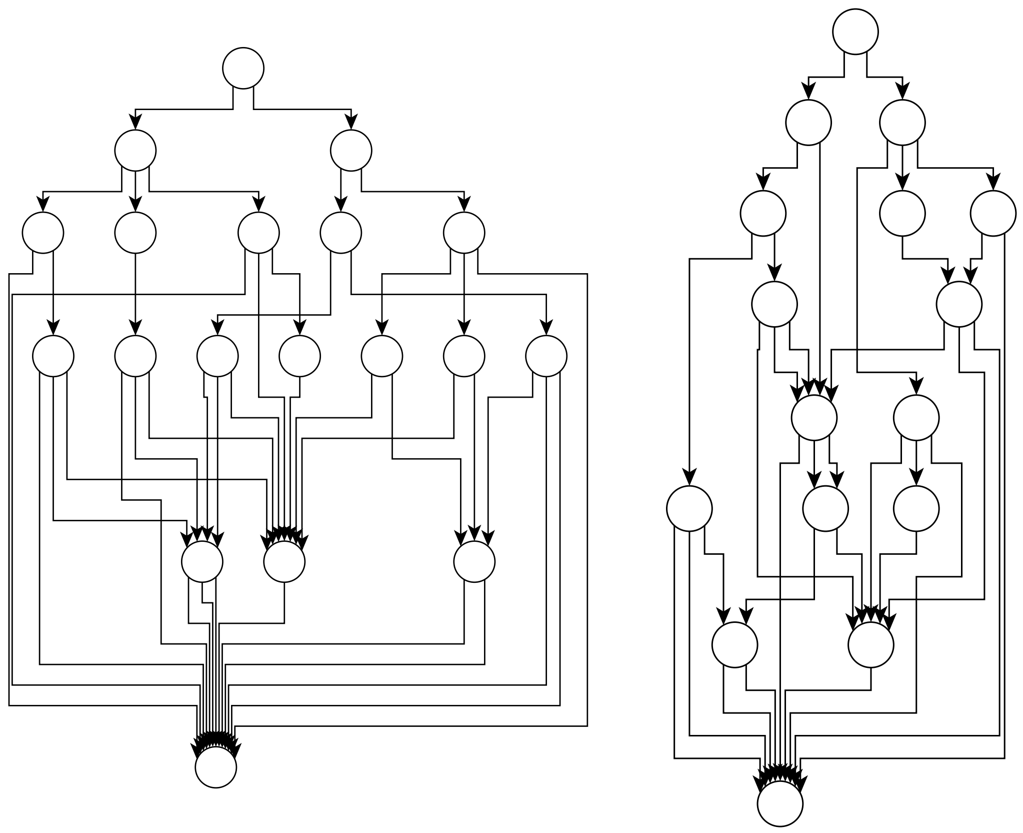


Figure 5.2: Two reduced graphs of the L-DAG data set with (left) the exact bisimulation solution and (right) the estimated solution of the hash-based algorithm using $\diamond(H) \equiv \oplus_{h \in H} h$

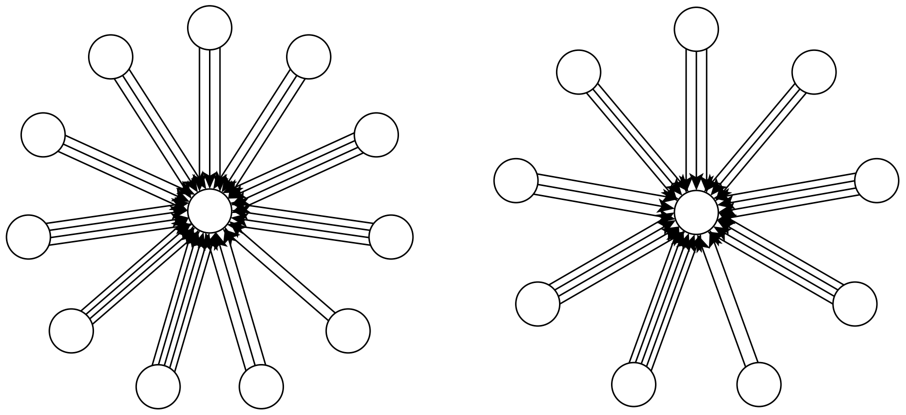


Figure 5.3: Two reduced graphs of the MovieLens100k data set with (left) the exact bisimulation solution and (right) the estimated solution of the hash-based algorithm using $\diamond(H) \equiv \oplus_{h \in H} h$

Still, the structure of the reduced graphs of said data sets seems to be preserved reasonably well by the estimation. As we can see in the figures, the DAG structure of the L-DAG data set is preserved. Likewise the star structure of the MovieLens100k data set is also preserved. For some applications, for example visualization of the data, it may therefore still be feasible to use the exclusive disjunction operator.

5.2.2 Performance

To analyze the performance of the partition algorithms, we measured the total time it took for each algorithm to compute the k_{max}-bisimulation partition for each data set. We call this measure the makespan. For the distributed algorithms we have two additional metrics which we have measured: the total data shipment and visit times. A more detailed discussion about these metrics can be found in Section 2.6.

Another important aspect which will have an impact on the performance of the distributed algorithms is the way we segment the graph (i.e. how we split the nodes among the worker machines). One can imagine that when we split the graph in such a way that there are very few edges between nodes on different machines, then the amount of necessary communication between machines will be far less, compared to when there are many cross-machine edges. One could also imagine that if we have two worker machines, where one machine has 1 node, and the other machine has $|V| - 1$ nodes, then the potential positive effect of distributing the work is completely negated. In our experiments we have used two methods for splitting the graph onto the worker machines. We required that the methods for segmenting the graph are such that the amount of nodes per machine is roughly the same (evenly distributed). The objective is then to minimize the number of edges crossing node segments (finding the minimum cut as described by Andreev and Räcke [2]). We settled on using the following two methods, although there are better methods out there (see for example a recent paper by Wang et al. [29]).

- Random split: this is a very simple splitting method, which simply assigns nodes to worker machines randomly. This method is used as a baseline for comparison.
- Exploration-based split: this method explores the graph starting at some seed node. As the graph is being explored, the visited nodes are added to one machine, until we reach that machine's capacity (roughly $\frac{|V|}{W}$ where W is the number of workers). Then, subsequent nodes which are visited are added to the next machine until we use its capacity, and so on. The idea is that a new node which we discover is connected to some node which we have visited before, thus they are connected by an edge. By assigning these two nodes to the same worker, we reduce the number of cross-machine edges. We have tested this method

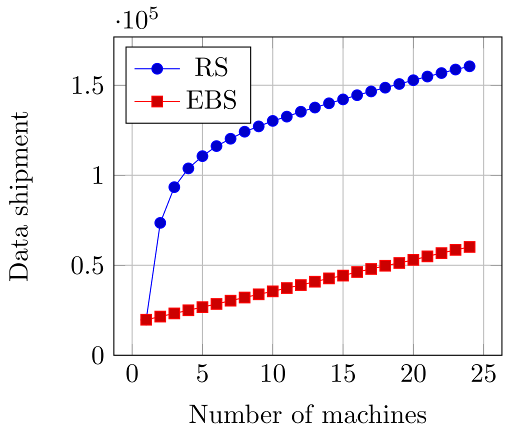
with our three familiar queues (FIFO, LIFO, AIRO, see Section 3.1.5) and settled on using the AIRO queue which gave the lowest minimum cut.

Our experiments were run on a system with two Intel(R) Xeon(R) E5-2630 CPUs at 2.30 GHz with 64.0 GB memory. Each CPU has 6 physical cores with Hyperthreading enabled by Windows 7 Enterprise Service Pack 1, amounting to a total of 24 virtual cores. This allowed us to vary the number of worker machines W for the distributed algorithms from 1 to 24.

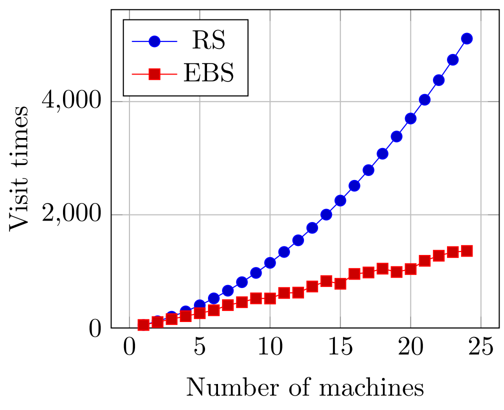
Every measurement we present is from an average of 10 runs where we show the standard deviation with error bars. Like the results from Section 5.1.2, we also have quite a large result space for the performance measurements of the partition algorithms. We invite the reader to use another interactive tool we have developed, which allows for investigating the performance results, and can be found on the following web page: `http://bit.ly/1k3k9Au`. Out of these results we shall highlight some of the more important observations we have made.

Random split versus exploration-based split First we shall verify our claim that choosing the right segmentation method can have an impact on performance. In Figure 5.4a we compare the data shipment of the distributed signature-based algorithm for the two segmentation methods. When we compared the data shipment values for each data set, the exploration-based split always gave better results than the random split. We show this comparison for the Chains data set, because the difference is the most pronounced for this data set. Intuitively this makes sense, because the Chains data set benefits a lot from the placement of entire chains onto the same machine. The random split method can scatter pieces of chains all across different machines.

For the visit times metric we also see that the exploration-based segmentation method gives much better results than the random segmentation method. This holds true for all data sets. An example is shown in Figure 5.4b for the Chains data set. Once again we chose to show the difference for the Chains data set, because this data set benefits the most from choosing the right segmentation method.



(a) Data shipment



(b) Visit times

Figure 5.4: Data shipment and visit times of the distributed signature-based algorithm for the Chains data set; we compare the values for using Random split (RS) and Exploration-based split (EBS) to segment the graph onto the worker machines

The impact on the makespan metric is less obvious, but it is there as well. Typically, shipping data is not very computationally intensive and rather involves network latency. However, due to the computations that follow from data shipments there is still a slightly noticeable difference. In Figure 5.5 we see a comparison between the Random split and Exploration-based split and their impact on the makespan of the distributed signature-based algorithm, using the WikiElec data set. For this data set we see the largest difference in performance.

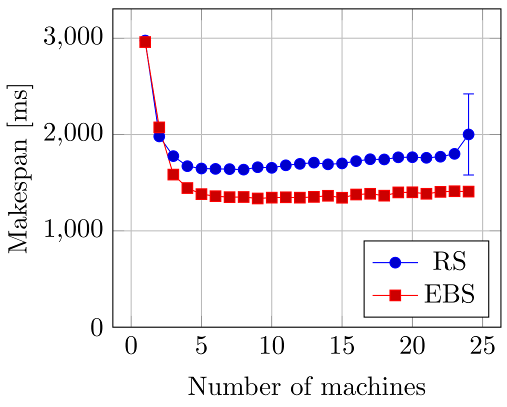


Figure 5.5: Makespan (in milliseconds) of the distributed signature-based algorithm for the Wiki-Elec data set; we compare the values for using Random split (RS) and Exploration-based split (EBS) to segment the graph onto the worker machines

Signature-based versus hash-based The most important question we wanted to answer is: does the hash-based algorithm improve upon the performance of the signature-based algorithm? The premise of the hash-based algorithm was to give up correctness for efficiency, so it is vital to its success that it does so. In Table 5.8 we see the makespan measurements for the sequential partition algorithms for all data sets. For the MovieLens100k data set we actually see a worse performance for the hash-based algorithm than for the signature-based algorithm. For the all the other data sets we that the opposite is true, and we see a maximum speedup for the hash-based algorithm of about a factor of 57 for the ErdosRenyi data set. If we think about what happens during the computation of the partition and the structure of the data sets, it makes sense that the hash-based algorithm performs worse for the MovieLens100k data set. The structure of the MovieLens100k data set is visible in Figure 5.3 (left), and we know that it is a bipartite graph with $k_{max} = 1$. The diversity in signatures of nodes in this data set is therefore really quite limited. As such, the overhead of computing a hash value for the signature of a node, as done by the hash-based algorithm, has lead to an increase in makespan for the hash-based algorithm. For the data sets we have tested, MovieLens100k is the only data set for which the sequential hash-based algorithm performs worse than the sequential signature-based algorithm. Therefore it is relatively successful, but we note that if we expect very small (and very few) distinct signatures, then the signature-based algorithm can have better performance.

We also want to know how the hash-based algorithm compares to the signature-based algorithm in a distributed setting. In Figure 5.6 we see the makespan of the sequential and distributed signature-based (left) and hash-based (right) algorithms, for the Jamendo data set. The distributed algorithms were run of a graph split by the exploration-based method. We can see that, although that the sequential hash-based algorithm performs about 3 times better than the sequential signature-based algorithm, the distributed hash-based algorithm suffers from too large overhead when we increase the number of machines too much. At around $W = 8$ the distributed signature-based algorithm overtakes the distributed hash-based algorithm in performance. We see this happening for 4 out of the 12 data sets we have tested. For the remaining 8 data sets, we observe a consistent improvement in performance of the distributed hash-based algorithm compared to the distributed signature-based algorithm. An example of this is shown in Figure 5.7 for the WingNodal data set.

We also want to consider what happens for the data shipment and visit times metrics. However, from analyzing the algorithms (see Figure 3.12 and Figure 3.13) it is easy to see that the visit times of the hash-based algorithm never exceed that of the signature-based algorithm, as long as the graph is segmented exactly the same way. Our results confirm this, and an example is shown in Figure 5.8a for the GDGenerator data set. For the data shipment metric, we also see a consistent

Table 5.8: Makespan of the sequential partition algorithms; the speedup indicates how much faster the hash-based algorithm is compared to the signature-based algorithm

	Sequential signature-based		Sequential hash-based		
Data set	Average [ms]	σ	Average [ms]	σ	Speedup
Real graphs					
Advogato	$1.99 \cdot 10^3$	$3.73 \cdot 10^1$	$1.91 \cdot 10^2$	$3.34 \cdot 10^0$	10.45
Jamendo	$1.25 \cdot 10^4$	$9.86 \cdot 10^1$	$4.23 \cdot 10^3$	$8.98 \cdot 10^1$	2.95
MovieLens100k	$8.75 \cdot 10^1$	$2.87 \cdot 10^0$	$1.08 \cdot 10^2$	$2.56 \cdot 10^0$	0.81
WikiElec	$1.84 \cdot 10^3$	$1.44 \cdot 10^1$	$3.41 \cdot 10^2$	$4.35 \cdot 10^0$	5.39
WingNodal	$2.68 \cdot 10^4$	$1.39 \cdot 10^2$	$2.26 \cdot 10^3$	$1.41 \cdot 10^1$	11.9
Synthetic graphs					
Chains	$1.02 \cdot 10^2$	$6.52 \cdot 10^0$	$7.62 \cdot 10^1$	$2.71 \cdot 10^0$	1.34
DAG	$3.10 \cdot 10^0$	$3.00 \cdot 10^{-1}$	$2.70 \cdot 10^0$	$6.40 \cdot 10^{-1}$	1.15
ErdosRenyi	$1.37 \cdot 10^4$	$4.22 \cdot 10^1$	$2.40 \cdot 10^2$	$1.17 \cdot 10^1$	57.19
GDGenerator	$1.15 \cdot 10^2$	$4.21 \cdot 10^0$	$6.33 \cdot 10^1$	$2.93 \cdot 10^0$	1.82
L-DAG	$9.20 \cdot 10^0$	$6.00 \cdot 10^{-1}$	$6.00 \cdot 10^0$	$0.00 \cdot 10^0$	1.53
Stars	$3.24 \cdot 10^1$	$4.25 \cdot 10^0$	$2.92 \cdot 10^1$	$1.54 \cdot 10^0$	1.11
Trees	$3.40 \cdot 10^1$	$1.11 \cdot 10^1$	$2.94 \cdot 10^1$	$1.20 \cdot 10^0$	1.16

improvement for the hash-based algorithm compared to the signature-based algorithm. That is: the amount of data shipped by the hash-based algorithm is consistently less than the amount shipped by the signature-based algorithm. An example of our results, shown in Figure 5.8b confirms this. What we also observe is that the amount of data shipped by the hash-based algorithm is often an order of magnitude less than the amount shipped by the signature-based algorithm.

Sequential versus distributed We have already seen some of the differences between the sequential and distributed algorithms in Figure 5.6 and Figure 5.7. The most important observation is that while the distributed algorithm can speed up work by allowing for parallel computations, it does not always necessarily do that. It very much depends on the data set and how compute intensive it is to get the signature IDs. Usually the reason to use distributed algorithms on a graph data set is because the graph does not fit on a single machine. In this situation it is often a luxury to have a distributed algorithm that is more efficient than the sequential variant.

One thing that is universal among the data sets we have tested, is that at some point of increasing W it starts to have a diminishing returns effect on the makespan for the hash-based algorithm. We can clearly see that it is around $W = 5$ for the Jamendo and WingNodal data set in Figure 5.6b and Figure 5.7b. For the WikiElec data set this value is around $W = 12$ as shown in Figure 5.9b. This does not hold true for the signature-based algorithm. For the signature-based algorithm we almost always observe a flatlining as we increase W, for example in Figure 5.9a we see this for the WikiElec data set. The only exception to this behavior we observed was for the DAG data set. The flatlining of the distributed signature-based algorithm can occur either above, at, or below the sequential makespan. It usually happens below the sequential makespan, with the DAG and WingNodal data sets being the only exceptions to this rule.

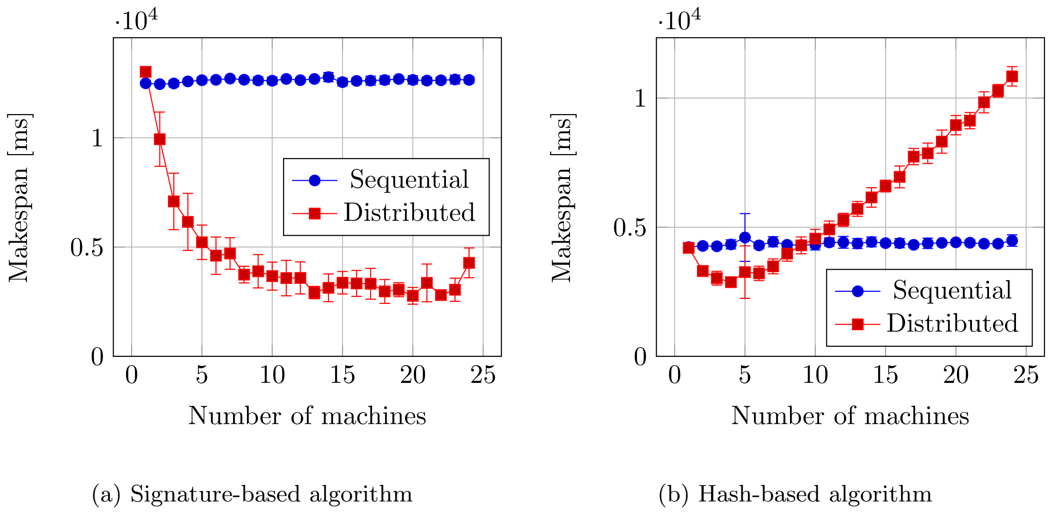


Figure 5.6: Makespan (in milliseconds) of the sequential and distributed signature-based and hash-based algorithms for the Jamendo data set; the distributed algorithms used the graph as split by the exploration-based method

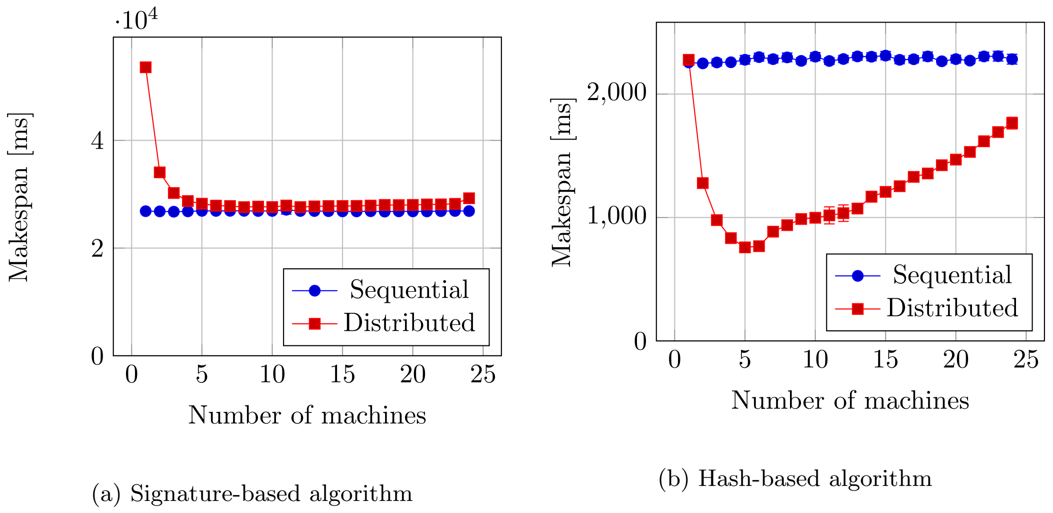
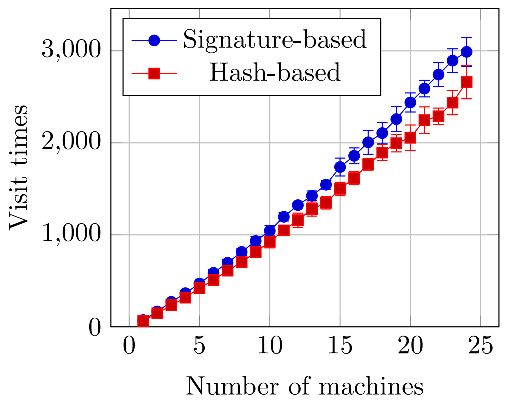
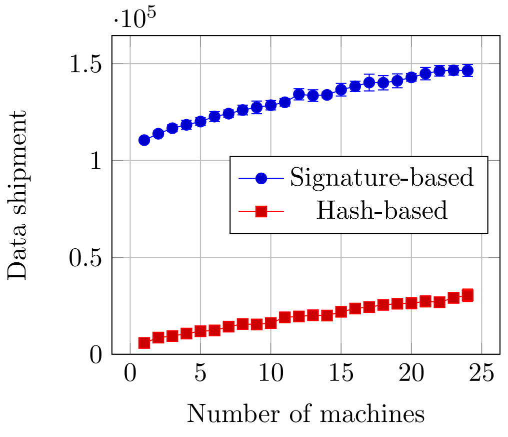


Figure 5.7: Makespan (in milliseconds) of the sequential and distributed signature-based and hash-based algorithms for the WingNodal data set; the distributed algorithms used the graph as split by the exploration-based method

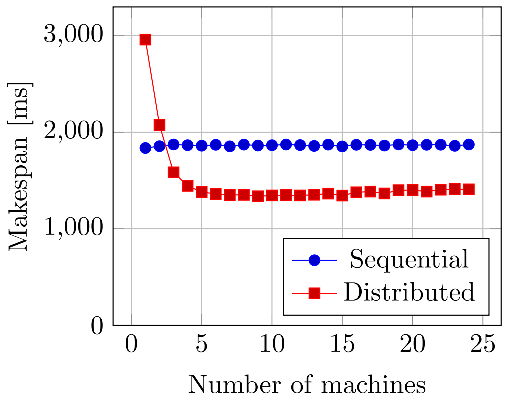


(a) Visit times

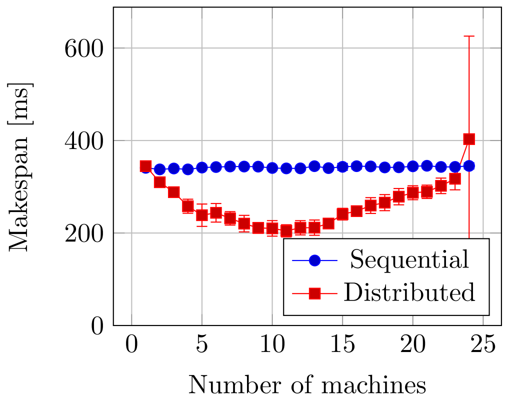


(b) Data shipment

Figure 5.8: Visit times and data shipment of the distributed signature-based and hash-based algorithms for the GDGenerator data set; the distributed algorithms used the graph as split by the exploration-based method



(a) Signature-based algorithm



(b) Hash-based algorithm

Figure 5.9: Makespan (in milliseconds) of the sequential and distributed hash-based algorithm for the WikiElec data set; the distributed algorithm used the graph as split by the exploration-based method

Chapter 6

Conclusions

Our main contribution in this thesis was to provide insight into the ability of graph sampling algorithms to preserve structural properties. Additionally we presented a new hash-based algorithm for computing the bisimulation of a graph, which sacrifices correctness for efficiency. In this chapter we summarize some of the major observations and conclusions we have made in the results chapter (Chapter 5). We will also mention some ideas for future work related to our studies.

Observations on sampling algorithms We have seen that for some data sets it makes sense to look at the bisimulation partition of a graph, while for others it does not. The WingNodal data set, for example, has as many partition blocks as it has nodes. As such, it makes no sense to look at the k_{max}-bisimulation partition for this data set, because there is no bisimulation equivalence between any of the nodes. We note that for a particular data set it may make sense to look at $k = 0$, $k = 1$, $k = 2$, $k = k_{max}$, or any other k value.

When we sample a graph, a number of things may affect the quality of the sampled graph as a representative of the bisimulation partition. Not only do we get higher values for lower k. We've also seen that it is generally easier to get a high value for correctness than for coverage. It is also easier to get higher values for the weighted metrics (WP, WR) than for the standard metrics (P, R). Some sampling methods work very well for some data sets, especially if they were designed to deal with some property that those graphs exhibit. For example: the DLBF sampler was designed to deal with graphs like the L-DAG data set, and indeed in Figure 5.1b we see that it works really well. We note that sampling methods can be tailored to work well for the graph which they are intended to sample, given that we have knowledge of its structure. This knowledge may be obtained by a domain expert, but perhaps there is an automated way to estimate the expected accuracy of a sampler for a given data set. In general we find that the exploration-based samplers have good quality in terms of our metrics.

Observations on partition algorithms We observe that (our new) hash-based partition algorithm generally performs better than the signature-based algorithm, for both the sequential and distributed variants, with respect to all three metrics (makespan, visit times, and data shipment). The highest speedup we saw for the ErdosRenyi data set where the sequential hash-based algorithm was 57 times faster than the signature-based algorithm. Important to note is that the chosen hash functions matter a lot in the quality of the hash-based algorithm. For example, we found a hash function which works well enough to achieve 100% accuracy for our data sets. Generally, the larger the data set, the larger the output space for the hash values has to be.

Our implementations of the distributed algorithms generally perform as well as, or even better, than the sequential algorithms. However, increasing the number of worker machines W too much can have a diminishing returns effect on the performance, or even make the distributed algorithms perform worse.

Future work Throughout this document we have left a number of hints at future work. We will recap on those here.

The sampling algorithms we have studied most likely leave a lot of room for improvement. The exploration-based samplers can possibly be improved by choosing better seed nodes. The DLBF sampling method in particular can be improved by not only looking at unique edges labels, but also unique target node labels. We would also like to see a sampling algorithm which uses knowledge from the hash-based partition algorithm, on large data sets, where the hash-based partition algorithm does not give a completely accurate partition.

How the hash functions can impact the quality of the hash-based algorithm is shown by a comparison of two functions. However, it might be interesting to do a more extensive study about the relation between the size of the graph and the number partition blocks it has, the randomness and output space of the hash functions, and the quality of the hash-based algorithm. The method of computing the hash value for each node may possibly also be improved, by changing on what information the hash values are computed.

Our aim was to get an overview of what kind of sampling method works well with what kind of graph. To strengthen our observations, it can be use to test the quality of even more sampling algorithm with even more data sets. Preferably, more data sets within a bounded set of categories would be used for this kind of study.

As mentioned earlier, perhaps there is an automated way of selecting (or designing) a good sampling algorithm given a data set. Future work can be done to discover relations between other (easily computable) graph properties and the quality of some sampling method.

Bibliography

[1] Mohammad Al Hasan, Nesreen K. Ahmed, and Jennifer Neville. Network sampling: Methods and applications. *SIGKDD Tutorial*, 2013. 11

[2] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, Barcelona, Spain, 2004. ACM. 46

[3] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM. 35

[4] Roberto Barbuti, Andrea MaggioloSchettini, and Paolo Milazzo. Extending the calculus of looping sequences to model protein interaction at the domain level. In Ion Mndoiu and Alexander Zelikovsky, editors, *Bioinformatics Research and Applications*, volume 4463 of *Lecture Notes in Computer Science*, pages 638–649. Springer Berlin Heidelberg, 2007. 1

[5] J.A. Bergstra and J.W. Klop. Process theory based on bisimulation semantics. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 50–122. Springer Berlin Heidelberg, 1989. 1

[6] Stefan Blom and Simona Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *STTT*, 7(1):74–86, 2005. 17, 20, 21

[7] Stefan Blom and Jaco van de Pol. Distributed branching bisimulation minimization by inductive signatures. In *PDMC*, pages 32–46, 2009. 17, 20

[8] G. Castagnoli, S. Bräuer, and M. Herrmann. Optimization of cyclic redundancy-codes with 24 and 32 parity bits. *IEEE Trans. on Communications*, 41(6), 1993. 20

[9] Emrah Cem, M.E. Tozal, and K. Sarac. Impact of sampling design in estimation of graph characteristics. In *Performance Computing and Communications Conference (IPCCC), 2013 IEEE 32nd International*, pages 1–10, Dec 2013. 1

[10] P. Erdős and A. Rényi. On random graphs, I. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959. 35

[11] Jan Friso Groote and Frank van Ham. Interactive visualization of large state spaces. *International Journal on Software Tools for Technology Transfer*, 8(1):77–91, 2006. 1

[12] Jelle Hellings, George H. L. Fletcher, and Herman J. Haverkort. Efficient external-memory bisimulation on dags. In *SIGMOD Conference*, pages 553–564, 2012. 17

[13] Pili Hu and Wing Cheong Lau. A survey and taxonomy of graph sampling. *CoRR*, abs/1308.5865, 2013. 10

[14] Ruoming Jin, Victor E. Lee, and Hui Hong. Axiomatic ranking of network role similarity. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 922–930, San Diego, California, USA, 2011. ACM. 1

[15] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. *Bioinformatics*, 20(11):1746–1758, July 2004. 13

[16] Jérôme Kunegis. KONECT networks @ONLINE. `http://konect.uni-koblenz.de/networks/`, March 2014. 31

[17] Maciej Kurant, Minas Gjoka, Yan Wang, Zack W. Almquist, Carter T. Butts, and Athina Markopoulou. Coarse-grained topology estimation via graph sampling. *CoRR*, abs/1105.5488, 2011. 1

[18] Sang Hoon Lee, Pan-Jun Kim, and Hawoong Jeong. Statistical properties of sampled networks. *Phys. Rev. E*, 73:016102, Jan 2006. 1

[19] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 631–636, Philadelphia, PA, USA, 2006. ACM. 1

[20] Yongming Luo, George H. L. Fletcher, Jan Hidders, Paul De Bra, and Yuqing Wu. Regularities and dynamics in bisimulation reductions of big graphs. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 13:1–13:6, New York, NY, USA, 2013. ACM. 32

[21] Yongming Luo, George H.L. Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. External memory k-bisimulation reduction of big graphs. In *Proceedings of the 22nd ACM international conference on information & knowledge management*, CIKM '13, pages 919–928, San Francisco, California, USA, 2013. ACM. 17

[22] Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 949–958, Lyon, France, 2012. ACM. 8, 17

[23] V.H. Moll. *Numbers and Functions: From a Classical-experimental Mathematician's Point of View.* Student Mathematical Library. American Mathematical Society, 2012. 67

[24] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998. 1

[25] Committee on the Analysis of Massive Data; Committee on Applied, Theoretical Statistics; Board on Mathematical Sciences, Their Applications; Division on Engineering, and Physical Sciences; National Research Council. *Frontiers in Massive Data Analysis.* The National Academies Press, 2013. 1

[26] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, December 1987. 17, 68

[27] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, New York, NY, USA, 2011. 1

[28] Davide Sangiorgi and Jan Rutten. *Advanced Topics in Bisimulation and Coinduction.* Cambridge University Press, New York, NY, USA, 1st edition, 2011. 1

[29] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 568–579, March 2014. 46

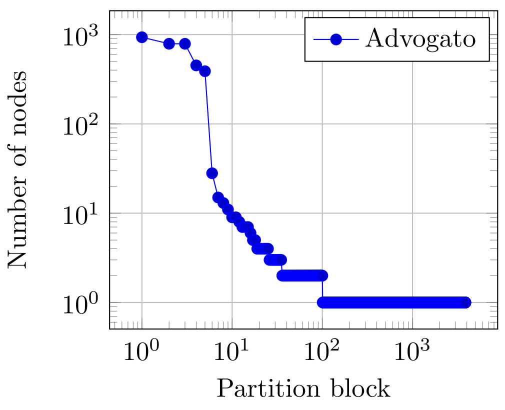
[30] Himanshu Zade, Santosh Arvind Adimoolam, Sai Gollapudi, Anind K. Dey, and Venkatesh Choppella. Edit distance modulo bisimulation: A quantitative measure to study evolution of user models. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1757–1766, Toronto, Ontario, Canada, 2014. ACM. 1

Appendix A

Data sets characteristics

A.1 Partition block sizes

These are the partition block sizes for each data set. The partitions are modulo k_{max} bisimulation.



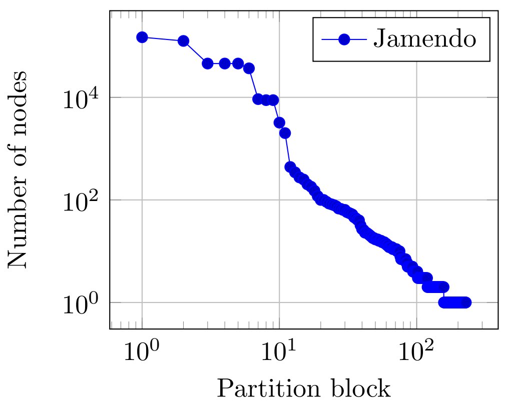
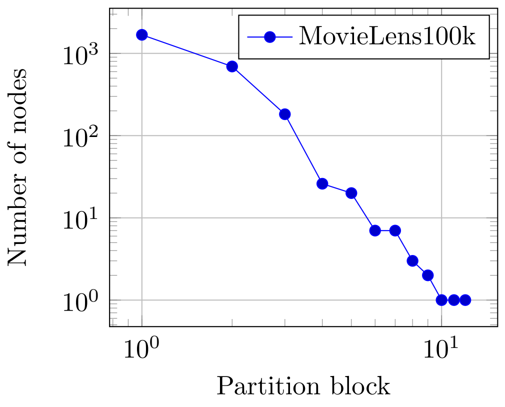


Figure A.1: Number of nodes per partition block for the Advogato and Jamendo data sets



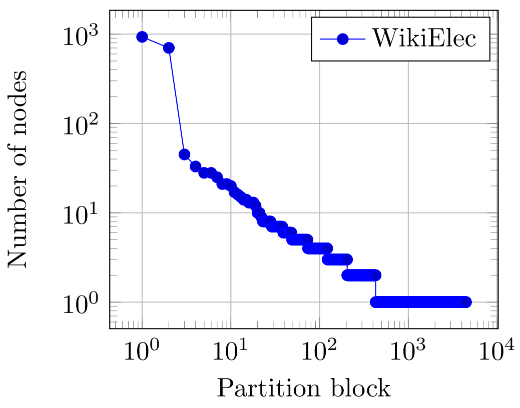
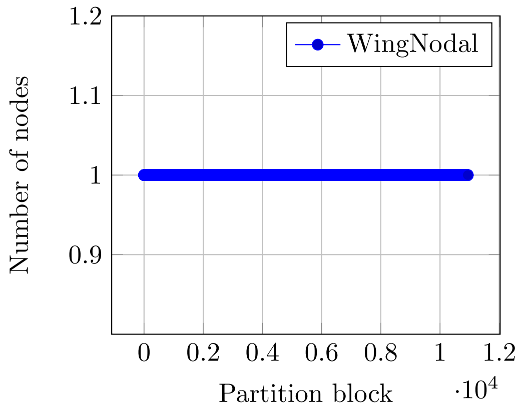


Figure A.1: Number of nodes per partition block for the MovieLens100k and WikiElec data sets



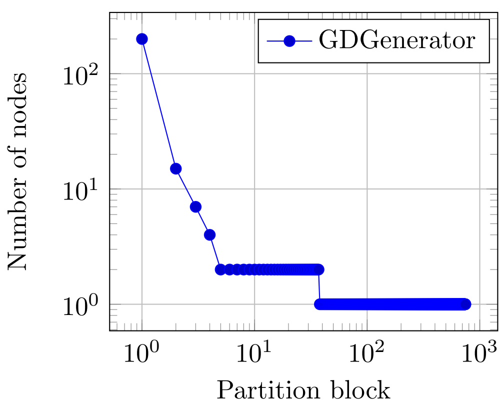
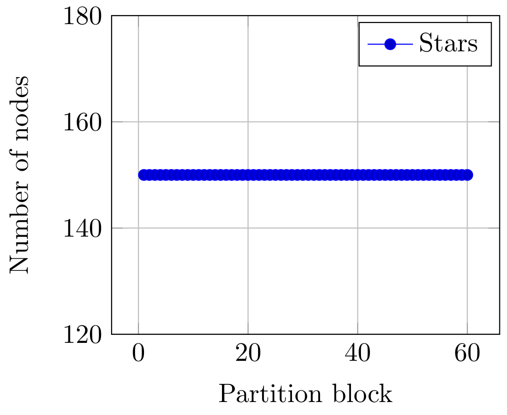


Figure A.1: Number of nodes per partition block for the WingNodal and GDGenerator data sets



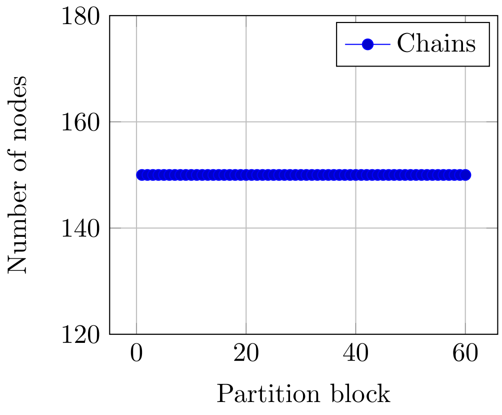
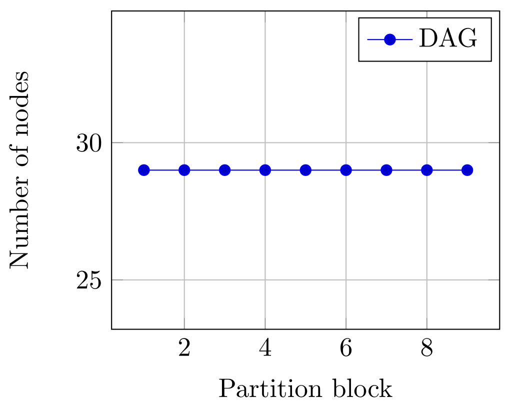


Figure A.1: Number of nodes per partition block for the Stars and Chains data sets



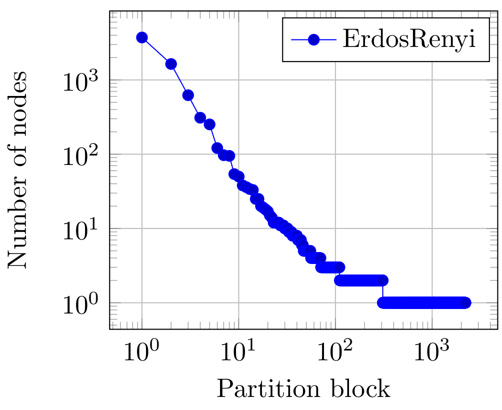


Figure A.1: Number of nodes per partition block for the DAG and ErdosRenyi data sets

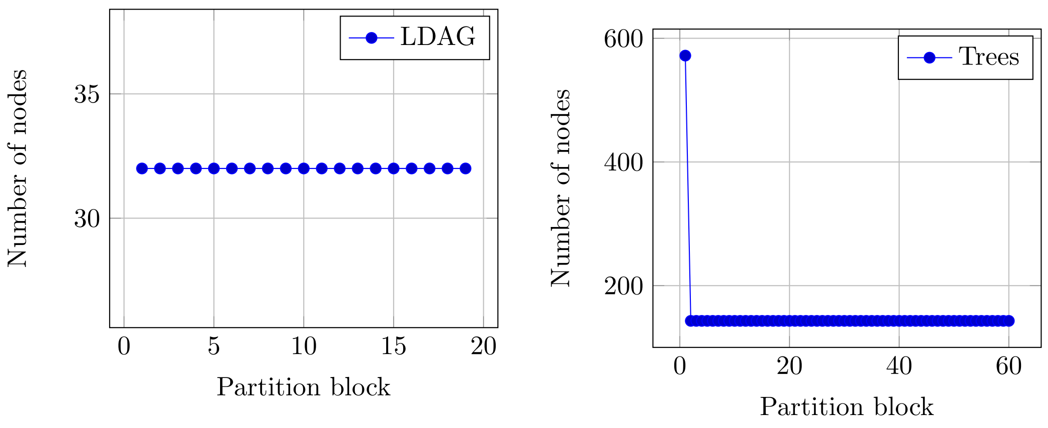
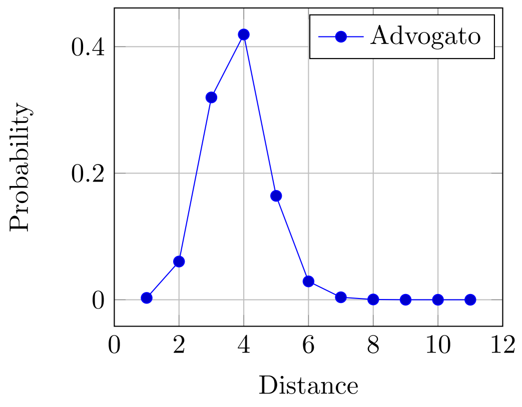


Figure A.1: Number of nodes per partition block for the L-DAG and Trees data sets

A.2 Distance Probability Mass Functions

For each data set, these are the distance probability mass functions for pairs of distinct connected nodes. Note that in a directed graph a pair of nodes (u, v) may be connected while the pair (v, u) is not.



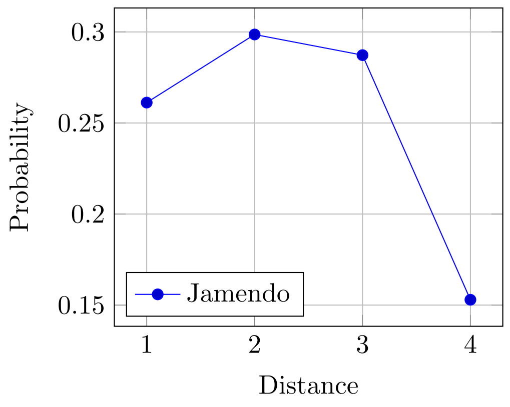
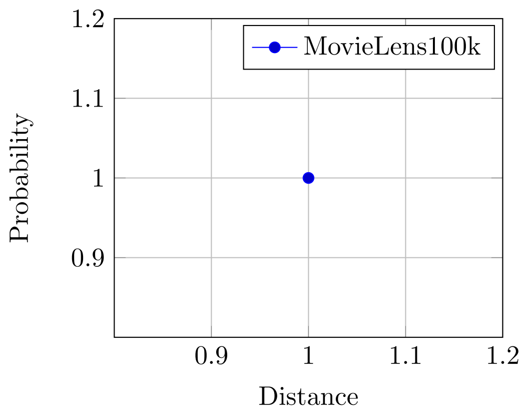


Figure A.2: Distance Probability Mass Function for the Advogato and Jamendo data sets



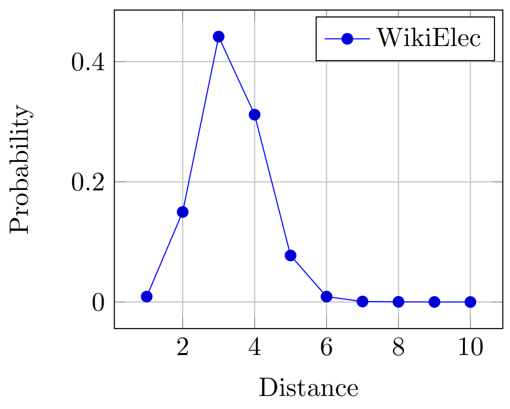
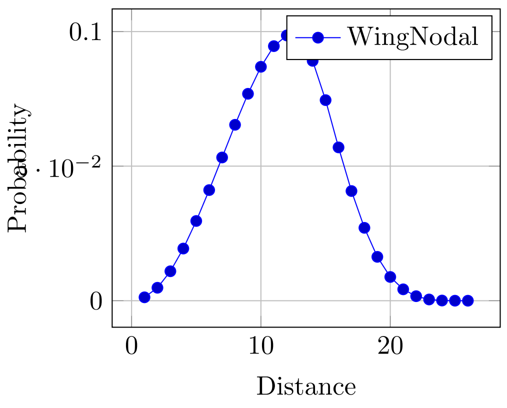


Figure A.2: Distance Probability Mass Function for the MovieLens100k and WikiElec data sets



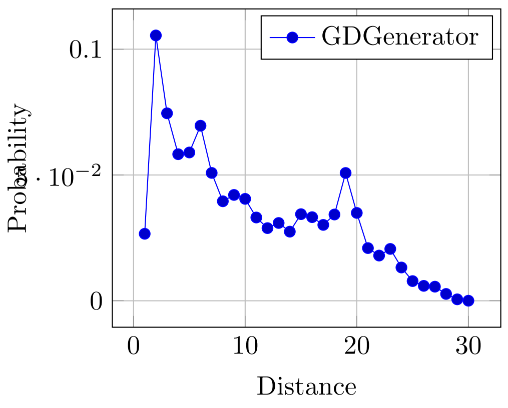
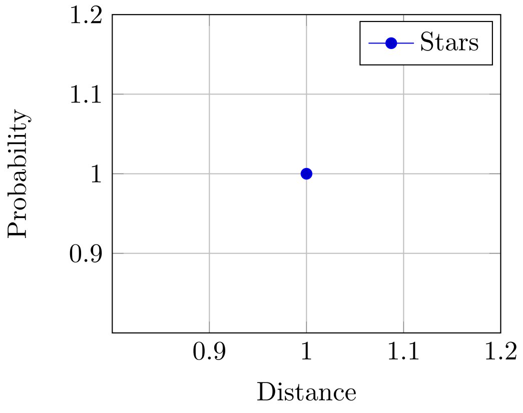


Figure A.2: Distance Probability Mass Function for the WingNodal and GDGenerator data sets



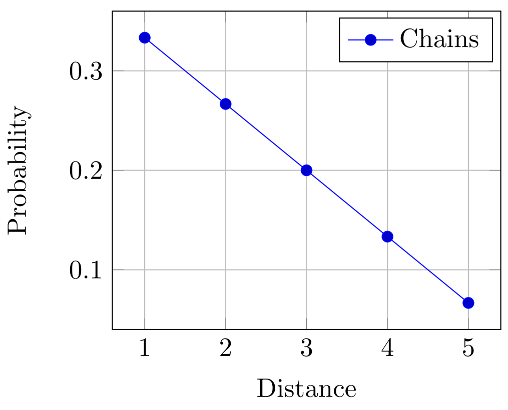
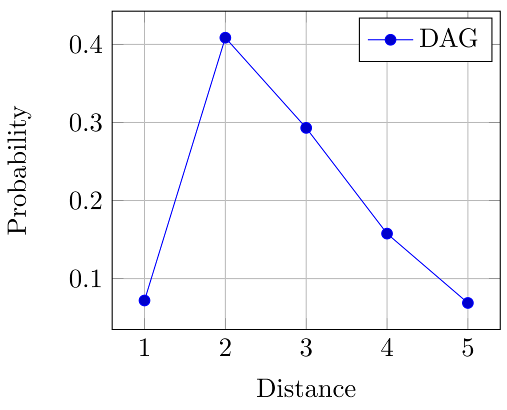


Figure A.2: Distance Probability Mass Function for the Stars and Chains data sets



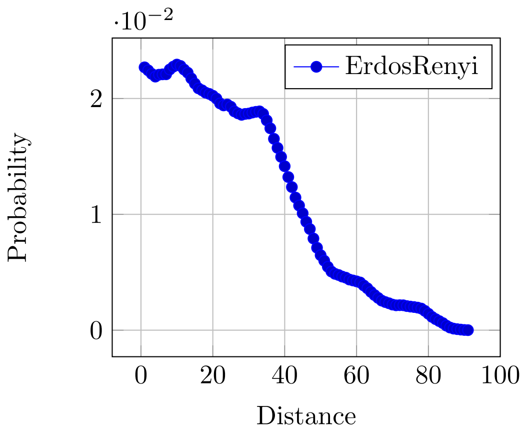


Figure A.2: Distance Probability Mass Function for the DAG and ErdosRenyi data sets

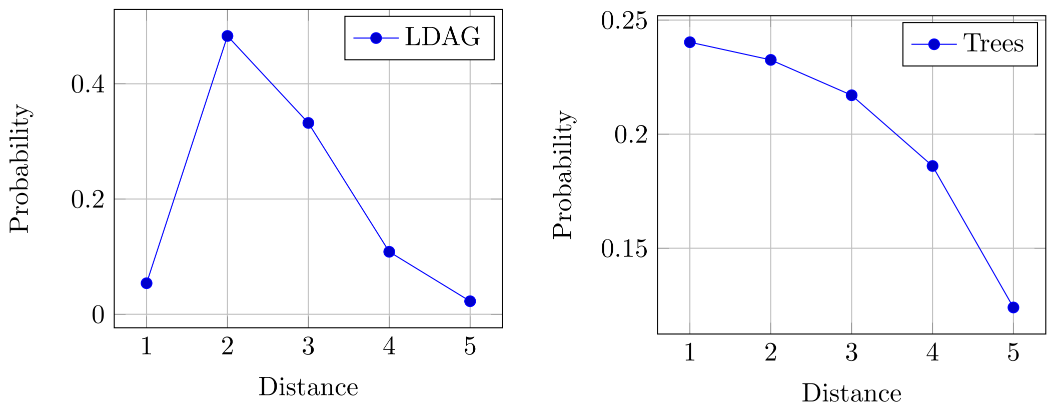
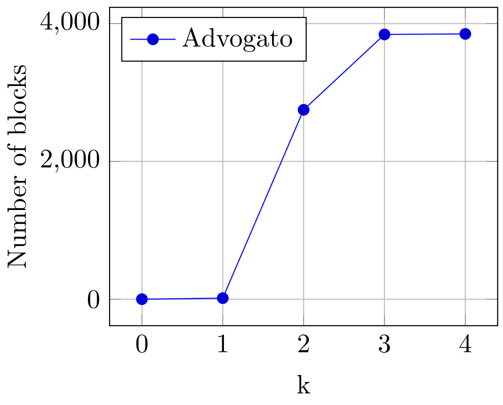


Figure A.2: Distance Probability Mass Function for the L-DAG and Trees data sets

A.3 Number of partitions for different localities

In these figures we observe the number of partition blocks (modulo k bisimulation) as $0 \leq k \leq k_{max}$ grows.



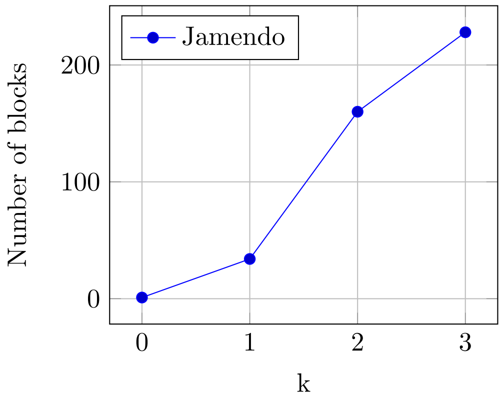
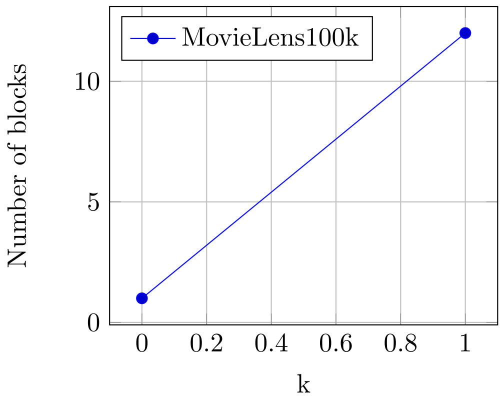


Figure A.3: Partition sizes for different k for the Advogato and Jamendo data sets



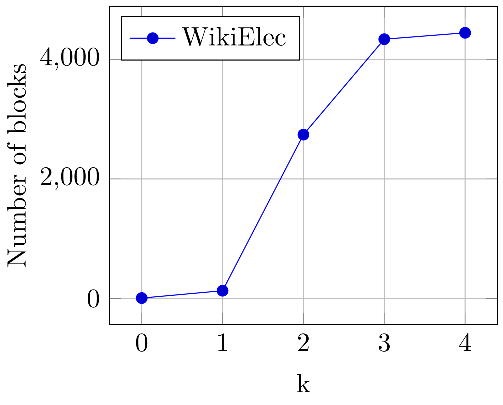
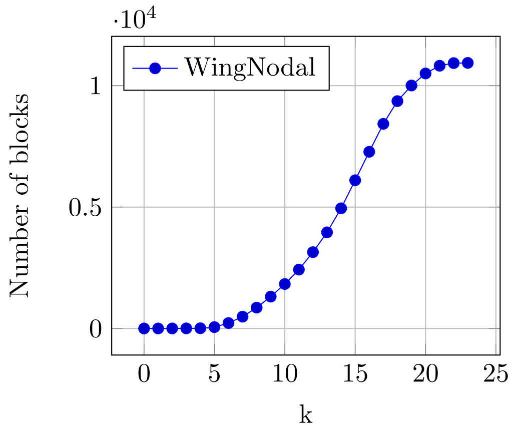


Figure A.3: Partition sizes for different k for the MovieLens100k and WikiElec data sets



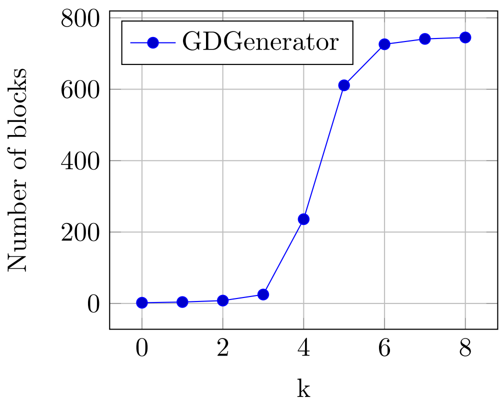
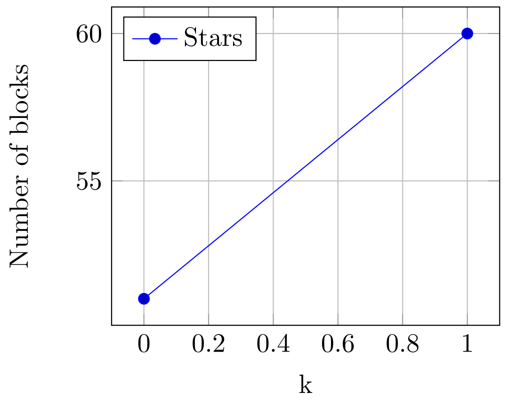


Figure A.3: Partition sizes for different k for the WingNodal and GDGenerator data sets



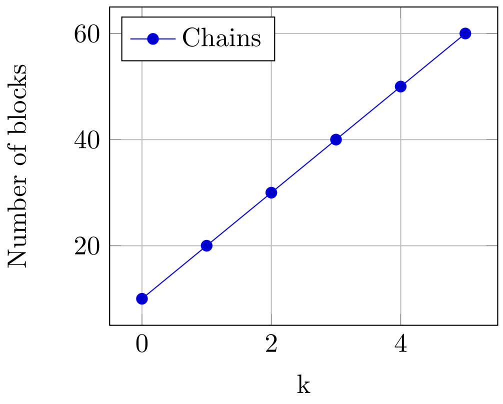
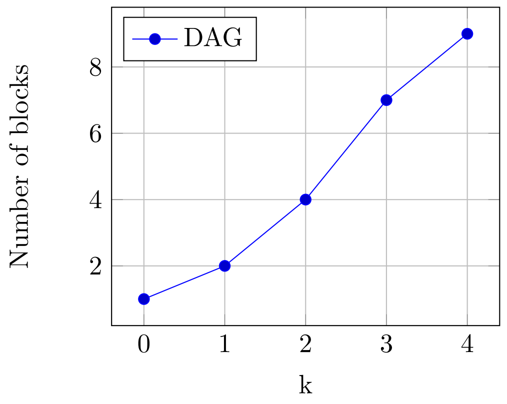


Figure A.3: Partition sizes for different k for the Stars and Chains data sets



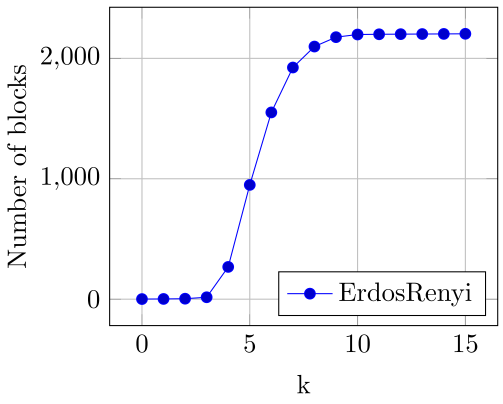
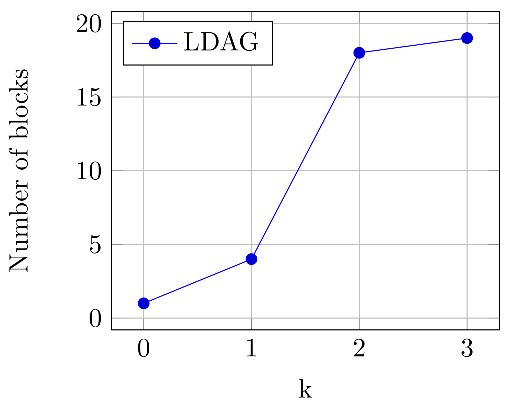


Figure A.3: Partition sizes for different k for the DAG and ErdosRenyi data sets



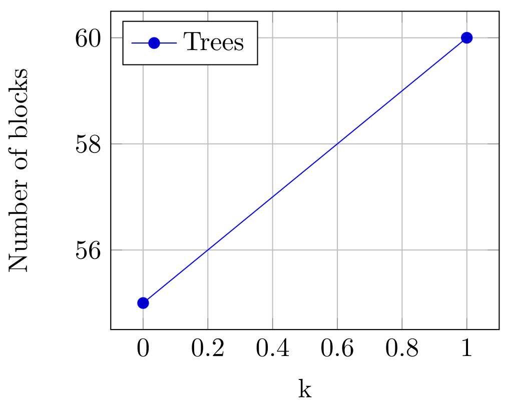


Figure A.3: Partition sizes for different k for the L-DAG and Trees data sets

Appendix B

Formal proofs

B.1 Introduction

In this appendix we provide formal proofs for the most important claims in earlier chapters. We stick to our usual definitions (see also Chapter 2) of the labeled directed multigraph G and the bisimulation relation:

$$G = (V, E, s, t, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$$

where V is the set of nodes, E the set of edges, s and t functions mappings edges to their source and target nodes, Σ_V and Σ_E the labels of the nodes and edges, and λ_V and λ_E functions mapping nodes and edges to their labels. We will often abbreviate "labeled directed multigraph" by simply using "graph" for readability reasons. Given such a labeled directed multigraph G and a non-negative integer k, nodes $u, v \in V$ are k-bisimilar (denoted $u \approx^k v$) if and only if the following three conditions hold:

1. $\lambda_V(u) = \lambda_V(v)$,
2. if $k > 0$ then $\forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow$
   $\exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{k-1} v' \wedge \lambda_E(e) = \lambda_E(e')$, and
3. if $k > 0$ then $\forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow$
   $\exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{k-1} u' \wedge \lambda_E(e) = \lambda_E(e')$.

To clarify the notation used in our proofs, by $[v]_{\approx^k}$ we denote the k-bisimulation partition block to which node v belongs.
In Lemma 1 together with Corollary 1 we prove that increasing k will always refine the previous partition. In Lemma 2 we prove that if we increase k but the number of blocks in the partitions remains equal, then the partitions are equal. In Lemma 3 we prove the notion of a stable partition. That is: if we increase k by 1 and the partition stays the same, then there won't be any changes to the partition for any higher values of k. In Corollary 2 we bring everything together and prove the notion of k_{max}. This is the smallest value k for which we reach a stable partition. In Proposition 1 we prove the correctness of the signature based algorithm, by proving that signatures can be used to exactly represent partition blocks. In Lemma 4 we claim that bisimilarity implies inter-bisimilarity. In fact, inter-bisimilarity is a generalized version of bisimilarity: bisimilarity applies to nodes of the same graph, whereas inter-bisimilarity does not have that requirement. In Lemma 5 we prove for a special graph R that it is bisimilar to the graph G used to construct it. In Proposition 2 we prove that our method of obtaining a minimally reduced graph (modulo bisimulation) indeed gives the smallest graph possible. By smallest we mean the least amount of nodes and edges.

B.2 Partition refinement and stability

Lemma 1. *For all $u, v \in V$ and $k \geq 0$ we have $u \approx^{k+1} v \Rightarrow u \approx^k v$.*

Proof. Take arbitrary $u, v \in V$ and $k \geq 0$. We prove the Lemma by induction on k.

- *Base case* ($k = 0$):

$$\begin{array}{ll} & u \approx^1 v \\ \Rightarrow & \{\text{Definition of } \approx^k\} \\ & \lambda_V(u) = \lambda_V(v) \\ \Leftrightarrow & \{\text{Definition of } \approx^k\} \\ & u \approx^0 v \end{array}$$

- *Step case* ($k \geq 1$):
Suppose that $x \approx^k y \Rightarrow x \approx^{k-1} y$ for all $x, y \in V$ (induction hypothesis).

$$\begin{array}{ll} & u \approx^{k+1} v \\ \Leftrightarrow & \{\text{Definition of } \approx^k\} \\ & \lambda_V(u) = \lambda_V(v), \\ & \forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow \\ & \exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^k v' \wedge \lambda_E(e) = \lambda_E(e'), \\ & \forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow \\ & \exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^k u' \wedge \lambda_E(e) = \lambda_E(e') \\ \Rightarrow & \{\text{Induction hypothesis + weaken conjunctions in the existential quantifiers}\} \\ & \lambda_V(u) = \lambda_V(v), \\ & \forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow \\ & \exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{k-1} v' \wedge \lambda_E(e) = \lambda_E(e'), \\ & \forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow \\ & \exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{k-1} u' \wedge \lambda_E(e) = \lambda_E(e') \\ \Leftrightarrow & \{\text{Definition of } \approx^k\} \\ & u \approx^k v \end{array}$$

□

Corollary 1. *As a consequence of Lemma 1 we have $[u]_{\approx^{k+1}} \subseteq [u]_{\approx^k}$ for all $u \in V$ and $k \geq 0$. In other words: partition $\mathcal{P}_{k+1}$ refines partition $\mathcal{P}_k$ for all $k \geq 0$.*

Lemma 2. *For all $k \geq 0$ we have $|\mathcal{P}_k| = |\mathcal{P}_{k+1}| \Rightarrow \mathcal{P}_k = \mathcal{P}_{k+1}$ where $\mathcal{P}_k$ and $\mathcal{P}_{k+1}$ are bisimulation partitions of graph G.*

Proof. Let $\mathcal{P}_k$ and $\mathcal{P}_{k+1}$ be bisimulation partitions of graph G. Assume that $|\mathcal{P}_k| = |\mathcal{P}_{k+1}|$. Assume (towards contradiction) that $[x]_{\approx^k} = [y]_{\approx^k}$ and $[x]_{\approx^{k+1}} \neq [y]_{\approx^{k+1}}$ for some $x, y \in V$. With Corollary 1 we also know that $[x]_{\approx^{k+1}} \subset [x]_{\approx^k}$ and $[y]_{\approx^{k+1}} \subset [x]_{\approx^k}$. Note that $\sum_{P \in \mathcal{P}_k} |P| = \sum_{P \in \mathcal{P}_{k+1}} |P|$ by the inclusion-exclusion principle [23]. Therefore there must be a $z \in V$ such that $\left|[z]_{\approx^{k+1}}\right| > \left|[z]_{\approx^k}\right|$. However, this violates our statement in Corollary 1, and therefore our assumption is false. Hence we conclude that for all $x, y \in V$ it holds that $[x]_{\approx^k} = [y]_{\approx^k} \Rightarrow [x]_{\approx^{k+1}} = [y]_{\approx^{k+1}}$. In terms of partitions this means that $\mathcal{P}_k = \mathcal{P}_{k+1}$. □

Lemma 3. *For $k \geq 0$ we have $\mathcal{P}_k = \mathcal{P}_{k+1} \Rightarrow \mathcal{P}_k = \mathcal{P}_l$ for all $l \geq k$.*

Proof. Let $k \geq 0$ and assume that $\mathcal{P}_k = \mathcal{P}_{k+1}$. Without loss of generality we shall prove that $\mathcal{P}_k = \mathcal{P}_l = \mathcal{P}_{l+1}$ by induction on l.

- *Base case* ($l = k$):
Trivially $\mathcal{P}_k = \mathcal{P}_l = \mathcal{P}_{l+1}$ is true by assumption.

- *Step case* ($l > k$):
  Suppose that $\mathcal{P}_k = \mathcal{P}_{l-1} = \mathcal{P}_l$ (induction hypothesis).
  We shall prove $\mathcal{P}_k = \mathcal{P}_{l+1}$ which is sufficient to prove the step case.

$$\begin{array}{ll}
 & \top \\
\Leftrightarrow & \{\text{Induction hypothesis}\} \\
 & \mathcal{P}_k = \mathcal{P}_l \\
\Leftrightarrow & \{\text{Definition of } \mathcal{P}_k\} \\
 & u \approx^k v \Leftrightarrow u \approx^l v \\
\Leftrightarrow & \{\text{Definition of } \approx^k\} \\
 & u \approx^k v \Leftrightarrow \\
 & \lambda_V(u) = \lambda_V(v), \\
 & \forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow \\
 & \exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{l-1} v' \wedge \lambda_E(e) = \lambda_E(e'), \\
 & \forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow \\
 & \exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{l-1} u' \wedge \lambda_E(e) = \lambda_E(e') \\
\Leftrightarrow & \{\text{Induction hypothesis}\} \\
 & u \approx^k v \Leftrightarrow \\
 & \lambda_V(u) = \lambda_V(v), \\
 & \forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow \\
 & \exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{l} v' \wedge \lambda_E(e) = \lambda_E(e'), \\
 & \forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow \\
 & \exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{l} u' \wedge \lambda_E(e) = \lambda_E(e') \\
\Leftrightarrow & \{\text{Definition of } \approx^k\} \\
 & u \approx^k v \Leftrightarrow u \approx^{l+1} v \\
\Leftrightarrow & \{\text{Definition of } \mathcal{P}_k\} \\
 & \mathcal{P}_k = \mathcal{P}_{l+1}
\end{array}$$

□

Corollary 2. *For all graphs there is a critical value $0 \leq k_{max} < |V|$ for which the following holds:*

1. $\forall k \in \mathbb{N} : k > k_{max} \Rightarrow \mathcal{P}_{k_{max}} = \mathcal{P}_k$, *and*
2. *if* $k_{max} \geq 1$ *then* $\mathcal{P}_{k_{max}} \neq \mathcal{P}_{k_{max}-1}$.

In other words: partition $\mathcal{P}_{k_{max}}$ is the coarsest partition of V which is stable with respect to bisimulation. For more information about the coarsest partition problem see [26].

Proof. Let k_{max} denote the smallest value k for which $|\mathcal{P}_k| = |\mathcal{P}_{k+1}|$. Then using Corollary 1 we know that $|\mathcal{P}_0| < ... < |\mathcal{P}_{k_{max}}|$. If $k_{max} \geq |V|$ then the partition $\mathcal{P}_{k_{max}}$ contains at least $|V| + 1$ blocks. However, there can only be at most $|V|$ partition blocks (in this case each node is contained in its own block). Therefore $k_{max} < |V|$. With Lemma 2 and Lemma 3 we can conclude the first statement of this Corollary. With Lemma 2 we can conclude the second statement of this Corollary. □

B.3 Signature equality

Definition 1. *For all $u \in V$ and $k \geq 0$ let $sig_k(u) = (\lambda_V(u), L_k(u))$ where*

$$L_k(u) = \begin{cases} \emptyset & k = 0 \\ \left\{ (\lambda_E(e), sig_{k-1}(v)) \middle| e \in E, v \in V, u \xrightarrow{e} v \right\} & k \geq 1. \end{cases}$$

Proposition 1. *For all $u, v \in V$ and $k \geq 0$ we have $u \approx^k v$ iff $sig_k(u) = sig_k(v)$.*

Proof. Take arbitrary $u, v \in V$ and $k \geq 0$. We prove the Lemma by induction on k.

- *Base case* ($k = 0$):
  $u \approx^0 v$
  $\Leftrightarrow$ {Definition of $\approx^k$}
  $\lambda_V(u) = \lambda_V(v)$
  $\Leftrightarrow$ {Definition of sig_k}
  $sig_0(u) = sig_0(v)$

- *Step case* ($k \geq 1$):
  Suppose that $x \approx^{k-1} y$ iff $sig_{k-1}(x) = sig_{k-1}(y)$ for all $x, y \in V$ (induction hypothesis).
  $u \approx^k v$
  $\Leftrightarrow$ {Definition of $\approx^k$}
  $\lambda_V(u) = \lambda_V(v)$,
  $\forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow$
  $\exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge u' \approx^{k-1} v' \wedge \lambda_E(e) = \lambda_E(e')$,
  $\forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow$
  $\exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge v' \approx^{k-1} u' \wedge \lambda_E(e) = \lambda_E(e')$
  $\Leftrightarrow$ {Induction hypothesis + tuple equality}
  $\lambda_V(u) = \lambda_V(v)$,
  $\forall u' \in V, e \in E : u \xrightarrow{e} u' \Rightarrow$
  $\exists v' \in V, e' \in E : v \xrightarrow{e'} v' \wedge (\lambda_E(e), sig_{k-1}(u')) = (\lambda_E(e'), sig_{k-1}(v'))$,
  $\forall v' \in V, e \in E : v \xrightarrow{e} v' \Rightarrow$
  $\exists u' \in V, e' \in E : u \xrightarrow{e'} u' \wedge (\lambda_E(e), sig_{k-1}(v')) = (\lambda_E(e'), sig_{k-1}(u'))$
  $\Leftrightarrow$ {Definition of L_k}
  $\lambda_V(u) = \lambda_V(v)$,
  $\forall u' \in V, e \in E : (\lambda_E(e), sig_{k-1}(u')) \in L_k(u) \Rightarrow (\lambda_E(e), sig_{k-1}(u')) \in L_k(v)$,
  $\forall v' \in V, e \in E : (\lambda_E(e), sig_{k-1}(v')) \in L_k(v) \Rightarrow (\lambda_E(e), sig_{k-1}(v')) \in L_k(u)$
  $\Leftrightarrow$ {Set equality}
  $\lambda_V(u) = \lambda_V(v)$,
  $L_k(u) = L_k(v)$
  $\Leftrightarrow$ {Definition of sig_k}
  $sig_k(u) = sig_k(v)$

□

B.4 Minimally reduced graphs

Definition 2. *For two graphs*

$$G_1 = (V_1, E_1, s_1, t_1, \Sigma_{V_1}, \Sigma_{E_1}, \lambda_{V_1}, \lambda_{E_1})$$

$$G_2 = (V_2, E_2, s_2, t_2, \Sigma_{V_2}, \Sigma_{E_2}, \lambda_{V_2}, \lambda_{E_2})$$

we say that nodes $u \in V_1$ and $v \in V_2$ are k-inter-bisimilar (denoted $u \simeq^k v$) if and only if the following three conditions hold:

1. $\lambda_{V_1}(u) = \lambda_{V_2}(v)$,
2. *if* $k > 0$ *then* $\forall u' \in V_1, e \in E_1 : u \xrightarrow{e} u' \Rightarrow$
   $\exists v' \in V_2, e' \in E_2 : v \xrightarrow{e'} v' \wedge u' \simeq^{k-1} v' \wedge \lambda_{E_1}(e) = \lambda_{E_2}(e')$, *and*
3. *if* $k > 0$ *then* $\forall v' \in V_2, e \in E_2 : v \xrightarrow{e} v' \Rightarrow$
   $\exists u' \in V_1, e' \in E_1 : u \xrightarrow{e'} u' \wedge v' \simeq^{k-1} u' \wedge \lambda_{E_2}(e) = \lambda_{E_1}(e')$.

We say that graphs G_1 and G_2 are k-bisimilar if and only if the following two conditions holds:

1. *for every $u \in V_1$ there exists a $v \in V_2$ such that $u \simeq^k v$, and*
2. *for every $v \in V_2$ there exists a $u \in V_1$ such that $v \simeq^k u$.*

Lemma 4. *For two graphs*

$$G_1 = (V_1, E_1, s_1, t_1, \Sigma_{V_1}, \Sigma_{E_1}, \lambda_{V_1}, \lambda_{E_1})$$

$$G_2 = (V_2, E_2, s_2, t_2, \Sigma_{V_2}, \Sigma_{E_2}, \lambda_{V_2}, \lambda_{E_2})$$

and nodes $u, v \in V_1$ and $w \in V_2$ it holds that if $u \approx^k v$ and $u \simeq^k w$ then $v \simeq^k w$.

Proof. Left as an exercise for the reader. □

Lemma 5. *Suppose we have a graph $G = (V, E, s, t, \Sigma_V, \Sigma_E, \lambda_V, \lambda_E)$. Let $K \geq 0$ and let $R = (\mathcal{P}, E', s', t', \Sigma_V, \Sigma_E, \lambda_{\mathcal{P}}, \lambda_{E'})$ be a graph where:*

1. *$\mathcal{P}$ is the K-bisimulation partition of G;*
2. *$\lambda_{\mathcal{P}}(P) = \lambda_V(v)$ for all $P \in \mathcal{P}$ and arbitrary $v \in P$;*
3. *$E' = \left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}$;*
4. *$s'(P, Q, l) = P$ for all $(P, Q, l) \in E'$;*
5. *$t'(P, Q, l) = Q$ for all $(P, Q, l) \in E'$;*
6. *$\lambda_{E'}(P, Q, l) = l$ for all $(P, Q, l) \in E'$.*

Then for graphs G and R, and for all $u \in V$ and $0 \leq k \leq K$ we have that $u \simeq^k [u]_{\approx^K}$.

Proof. Let G and R be the graphs described by the Lemma. Then we prove this Lemma by induction on k.

- *Base case* ($k = 0$):
  Since $\lambda_V(u) = \lambda_{\mathcal{P}}([u]_{\approx^K})$ is true by construction of R it is easy to see that $u \simeq^0 [u]_{\approx^K}$.
- *Step case* ($1 \leq k \leq K$):
  Suppose that $x \simeq^{k-1} [x]_{\approx^K}$ for all $x \in V$ (induction hypothesis). Take an arbitrary node $u \in V$.
  - Again, it is easy to see that $\lambda_V(u) = \lambda_{\mathcal{P}}([u]_{\approx^K})$ by construction of R.
  - Take arbitrary $u' \in V$ and $e \in E$ such that $u \xrightarrow{e} u'$. By construction of R we have the edge $e' \in E'$ such that $[u]_{\approx^K} \xrightarrow{e'} [u']_{\approx^K}$ and $\lambda_E(e) = \lambda_{E'}(e')$. By the induction hypothesis we also have $u' \simeq^{k-1} [u']_{\approx^K}$.
  - Take arbitrary $Q \in \mathcal{P}$ and $e \in E'$ such that $[u]_{\approx^K} \xrightarrow{e} Q$. By construction of R we know there must be nodes $v \in [u]_{\approx^K}, v' \in Q$ and an edge $e' \in E$ such that $v \xrightarrow{e'} v'$ and $\lambda_{E'}(e) = \lambda_E(e')$. Since $u \approx^K v$ we know that there must be $u' \in V$ and $e'' \in E$ such that $u \xrightarrow{e''} u'$ and $\lambda_E(e') = \lambda_E(e'')$ and $u' \approx^{K-1} v'$. By Lemma 1 we get $u' \approx^{k-1} v'$ and by the induction hypothesis we also have $Q \simeq^{k-1} v'$. Now we get $u' \simeq^{k-1} Q$ using Lemma 4.

  These three derivations fulfill the three requirements of k-inter-bisimilarity respectively, hence we conclude that u and $[u]_{\approx^K}$ are k-inter-bisimilar: $u \simeq^k [u]_{\approx^K}$.

□

Definition 3. *A graph G (with nodes V and edges E) is a minimally reduced graph if and only if for all graphs G' (with nodes V' and edges E') which are bisimilar to G, it holds that $|V| \leq |V'|$ and $|E| \leq |E'|$.*

Proposition 2. *For a graph G with bisimulation partition $\mathcal{P}$, G is a minimally reduced graph if and only if:*

1. *$|V| = |\mathcal{P}|$, and*
2. *$|E| = \left|\left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}\right|$.*

Proof. Let G be a graph and let $\mathcal{P}$ be its bisimulation partition. We shall prove the equivalence by proving both sides of the implications.

($\Rightarrow$) Suppose that G is a minimally reduced graph. This means that for all graphs G' which are bisimilar to G, G has at most as many nodes and at most as many edges. Let $R = (\mathcal{P}, E', s', t', \Sigma_V, \Sigma_E, \lambda_{\mathcal{P}}, \lambda_{E'})$ be a graph where:

1. $\mathcal{P}$ is the bisimulation partition of G;
2. $\lambda_{\mathcal{P}}(P) = \lambda_V(v)$ for all $P \in \mathcal{P}$ and arbitrary $v \in P$;
3. $E' = \left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}$;
4. $s'(P, Q, l) = P$ for all $(P, Q, l) \in E'$;
5. $t'(P, Q, l) = Q$ for all $(P, Q, l) \in E'$;
6. $\lambda_{E'}(P, Q, l) = l$ for all $(P, Q, l) \in E'$.

With Lemma 5 we know that R is bisimilar to G. This means that G has at most as many nodes and at most as many edges as R. There can be no empty partition blocks in $\mathcal{P}$ which means that $|V| \geq |\mathcal{P}|$. For any edge $e \in E$ it holds that $([s(e)]_{\approx}, [t(e)]_{\approx}, \lambda_E(e)) \in \left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}$. Therefore we also know that $|E| \geq \left|\left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}\right|$. Hence $|V| = |\mathcal{P}|$ and $|E| = \left|\left\{(P, Q, \lambda_E(e)) \middle| P, Q \in \mathcal{P}, u \in P, v \in Q, e \in E, u \xrightarrow{e} v\right\}\right|$.

($\Leftarrow$) Suppose that graph G has as many of nodes and edges as described in the Proposition. Suppose (towards contradiction) that G is not minimally reduced. Then there is a smaller graph G' with nodes V' and edges E' which is bisimilar to G, such that $|V| > |V'| \vee |E| > |E'|$. Note that the number of nodes of the smaller graph cannot be less than the number of partition blocks, because then there would be empty partition blocks. For any nodes $u, u' \in V$ and outgoing edge $u \xrightarrow{e} u'$ there must exist nodes $v, v' \in V'$ and an edge $v \xrightarrow{e'} v'$ with equal labels and $u' \simeq v'$. This means that for node v there are at least $\left|\{([u']_{\approx}, \lambda_E(e)) | u \xrightarrow{e} u'\}\right|$ edges in E'. Hence, in total the number of edges in E' must be at least $\left|\{(u, [u']_{\approx}, \lambda_E(e)) | u \in V, u \xrightarrow{e} u'\}\right|$. This means our earlier statement that $|V| > |V'| \vee |E| > |E'|$ is contradicted. We conclude that our assumption is false and therefore G is minimally reduced.

□