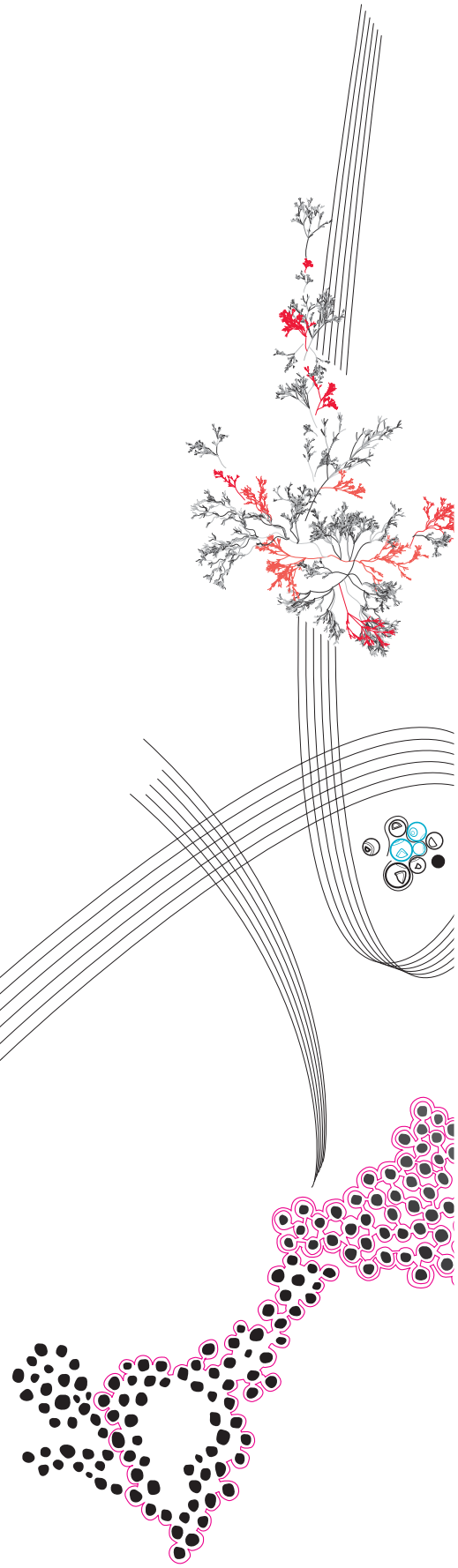


MSc Computer Science
Master Thesis

Evaluating Ordering Strategies For State Elimination

Pelle Cornelis Abraham de Greeuw

Supervisor: dr.ing. E.M. Hahn
External Assessor: dr.ir. A. Chiumento
Assessor: dr. A. Hartmanns

June 11, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

State elimination is an algorithm which reduces the state space of a Markov chain to a minimum size, without affecting the probabilities, when calculating unbounded reachability properties. The resource consumption in terms of memory and processing is greatly affected by the order in which states are eliminated while the result stays the same. The goal is to identify an order which performs better than other orders before the elimination process starts. This thesis lists several orders for elimination which are present in existing research or implementations. These orders are evaluated against one another on several Markov Chains from other existing research using two metrics. Using the data gathered from this evaluation several novel elimination orders are presented and also evaluated. One of these novel orders, $\overrightarrow{Heuristic2}$, has the best result on one metric for eleven out of fourteen models and second best for the remaining three. This result leads to the conclusion that there is one single best elimination order for all models which have been evaluated and likely any other model as well.

Keywords: State Elimination, Markov Chain, Elimination Order, Heuristic, Unbounded Reachability

Acknowledgements

My final project was not without problems or bumps in the road. With two failed attempts this is the third subject which finally stuck around. I would not have been able to continue on that journey without the continued support of my supervisor. Many thanks for the rest of the committee taking the time to assess this work and providing feedback during the process.

I am very grateful for my friends and family for believing in me throughout the last chapter of my master even when I myself did not see the finish line clearly. Special thanks go out to Henk, my cat, who tried to contribute many characters to this thesis and provided emotional support at all times of day. Mauw!

I am deeply indebted to my partner, Janiek, who has pushed me to work on this master project, provided emotional support throughout and lots of feedback on the final thesis. Without her, I would not be where I am now.

Contents

Chapter 1

Introduction

Designing protocols and writing software are processes which are prone to errors or bugs, which lead to real-world effects ranging from non-existent to life-threatening. If the game you are playing on your smartphone suddenly exits with an error message you are merely inconvenienced. On the other hand, if the road barriers of a railroad crossing do not close and a train is still allowed to enter the crossing, a disaster may follow. The operator of a railroad crossing needs a measure of certainty that such a situation will never occur with the control software they are using. A messaging protocol also benefits from certainty measures, providing certainty a message arrives at the destination can increase user trust. Model checking is a technique which can provide such measures.

Model checking uses a graph model of software, or any system, to reason about any properties it should have. Textual representations of such properties are: if a train is on the crossing, are the barriers closed; and what is the probability a message eventually arrives at the destination. The probability the last property calculates is called unbounded reachability, the probability that eventually something will be true. This probability can be calculated efficiently using the state elimination algorithm, which eliminates every state which is not initial and not final. After the elimination process, the requested probability can be read off the remaining model.

The algorithm is not always efficient, the order in which the states are eliminated is a great influence. This efficiency affects the computation time and the memory consumption of the algorithm, which can prevent larger models from being checked on smaller hardware if a worse order is selected. The goal of this thesis is to choose an elimination order which maximizes the efficiency and minimizes the computational and memory resource consumption.

The structure of this thesis is as follows. Chapter 2 introduces model checking, the state elimination algorithm and discusses existing research on the topic. This information is used in Chapter 3 to formally define the problem and research questions. Chapter 4 specifies the existing and novel elimination orders; implementation details; models; and tools used. Chapter 5 discusses the results from the evaluations and explains how the novel elimination orders are created. The thesis concludes in Chapter 6 with answers to the research questions, remarks and future work.

Chapter 2

Background

This chapter presents the background knowledge necessary for this thesis. Section 2.1 explains Markov Chains, Section 2.2 model checking and the state elimination algorithm, and Section 2.3 the related work.

2.1 Markov chains

This section will explain the concept of Markov Chains from the following definition:

> A Markov chain is a stochastic process for which the Markov property holds.

2.1.1 Stochastic process

A stochastic (or random) process [1] is a family of random variables, the state space, which is either indexed discretely or continuous by a parameter set. This parameter set can be interpreted as time passing in individual small increments or time flowing continuously respectively. Stochastic processes are often used as mathematical models of systems and phenomena that vary randomly.

When both the state space $\mathbb{S}$ and parameter space $\mathbb{T}$ of a stochastic process are discrete, the process is classified as a discrete parameter chain. Intuitively this means that for every time increment, $n \in \mathbb{T}$, the process can advance to either another state or the same state. Through the time parameter set the process can be viewed as a *chain* of states.

2.1.2 Markov property

A process has the Markov Property or is said to be Markov if and only if the next state of a process is only determined by the current state for all time increments. Mathematically this is expressed as follows: the process $(Z_n)_{n\in\mathbb{N}}$ which is taking values in state space $\mathbb{S} \in \mathbb{Z}$ has the Markov Property iff [2]

$$\mathbb{P}(Z_{n+1} = j | Z_n = i_n, Z_{n-1} = i_{n-1}, ..., Z_0 = i_o) = \mathbb{P}(Z_{n+1} = j | Z_n = i_n) \tag{2.1}$$

A stochastic process which is indexed by a discrete parameter set and holds the Markov Property is called a Discrete-Time Markov Chain (DTMC). This can be shortened to Markov Chain (MC). A MC is time-homogeneous if the parameter space has no effect

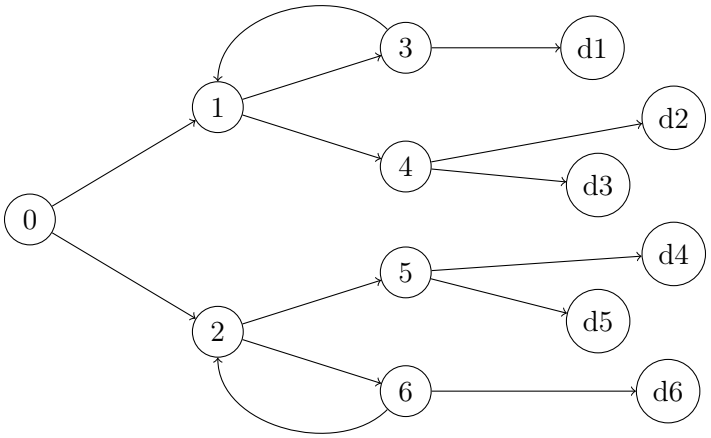


FIGURE 2.1: Die roll simulated by tossing a coin: upper branch corresponds to head, lower branch to tail. dx indicates the die rolled number x.

on the transition probabilities. In other words, if time does not affect the transition probabilities. The time-homogeneous property is expressed as follows:

$$\mathbb{P}(Z_{s+t}|Z_s = i) = \mathbb{P}(Z_t|Z_0 = i) \tag{2.2}$$

The process has to be well-defined: for every state, the sum of probabilities of the outgoing transitions equals 1. This is defined as:

$$\sum_{j \in S} \mathbb{P}(Z_1 = j|Z_0 = i), \quad i \in S \tag{2.3}$$

Since the process is also time-homogeneous this holds for any time index.

2.1.3 Representations

A Markov process can be modelled as a graph of edges and vertices. Every vertex represents a state of the process and every edge a transition to another state with an accompanying probability. Such a finite system where the process is in at most one state at a time is also called a Finite-State Automaton (FSA), often shortened to Finite Automaton (FA). Figure 2.1 shows a FA representation of a probabilistic program due to Knuth and Yao [3] which models a die using coin flips. The FA starts in state 0 and whenever heads is tossed, the process takes the upper branch. When tails is tossed, it takes the lower branch. This continues until one of the final states, labels starting with d, has been reached. As Daws [4] shows, this FA generates a uniform distribution where each die roll has a probability of $\frac{1}{6}$.

Only when a process has a finite state space, such as the die example, can the transition graph be made, otherwise it would also be of infinite size. The same holds for the transition probability matrix. This is a matrix which consists of all the probabilities of transitions between states, the edges in the graph. This is a $P_{ij} = \mathbb{S} \times \mathbb{S}$ matrix where each row number i is the originating state and each column number j the target state. The value stored at that point, P_{ij} is the probability of that transition.

$$P_{ij} = \mathbb{P}(Z_{n+1} = j|Z_n = i) \tag{2.4}$$

	0	1	2	3	4	5	6	$d1$	$d2$	$d3$	$d4$	$d5$	$d6$
0		0.5	0.5										
1			0.5	0.5									
2						0.5	0.5						
3		0.5						0.5					
4									0.5	0.5			
5											0.5	0.5	
6			0.5										0.5

TABLE 2.1: Transition probability matrix of a fair coin modeling a die. Row is the originating state, column the target state.

Table 2.1 shows the transition probability matrix for the die example. Every empty space indicates a probability of zero. The final states are absorbing, meaning they have a probability of one going to themselves. These are omitted from the matrix to improve its readability.

2.2 State elimination

This section explains how a system modeled as a Markov Chain can be used by model checking. It introduces the property of interest for this thesis and the algorithm used to calculate the probabilities of this property.

2.2.1 Model checking

Model checking is a verification technique that explores all possible system states in a brute-force manner [5]. A model checker can verify that a system truly satisfies a certain property using the underlying Markov chains. Properties which can be verified this way are, for example: is the generated result of a program OK; does the system eventually gracefully terminate; and can an error occur in the system within 1 hour?

These properties can be stated using Probabilistic Computation Tree Logic (PCTL) [6] which extends Computation Tree Logic (CTL) with time and probabilities. For the die model $P =?\ [F(\texttt{d1}|\texttt{d2})]$ specifies the probability P that the model eventually (or finally, F) reaches state d1 or d2.

2.2.2 RE/FA state elimination

The property of interest for this thesis is unbounded reachability: the probability that the system will eventually end up in a given set of states. For example, if a program will enter into an error state or with what probability a number of participants will eventually come to an agreement. The exact path taken to a state is therefore not of interest but only the probability that it will eventually be visited.

States which are neither the start of the chain nor are a subject in the requested property are only interesting because they influence the probabilities of the final result. If these probabilities are preserved these states can be removed from the chain. Daws [4] presents a way to use regular expressions (RE) as the transition probabilities. The resulting FA

can then be converted to a single RE using the state elimination method described by Hopcroft [7].

For simplicity the same notation is used for getting the character of an edge in a RE automaton as with the probability of an edge in an FA. $P_{ij} = C(Z_{n+1} = j | Z_n = i)$ where C labels the edge from state i to state j with a RE.

Let $s_k \in \mathbb{S}$ be a state which is to be eliminated. This state has predecessor states $A = \{a_i \in \mathbb{S} | P_{ik} \neq \emptyset\}$ and successor states $B = \{b_j \in \mathbb{S} | P_{kj} \neq \emptyset\}$. For each combination of predecessor and successor states, the respective edges to state s and the optional self-loop of s are combined. The result of this combination is added to the edge going from the predecessor state directly to the successor state. If such an edge is not present it will be created. The RE on this edge is now $P_{ij} = P_{ij} \cup P_{ik}(P_{kk})^* P_{kj}$, where P_{kk} is the self loop from s to s which can occur any number of times. Finally, state s_k and all its incoming and outgoing transitions are removed from the graph.

The RE result of the elimination process is read off the edges once all applicable states have been eliminated. This resulting RE is a function of the starting probabilities of the model. The elements in the RE are replaced with their original probabilities and the resulting expression is then solved to obtain the requested probabilities.

Elimination order

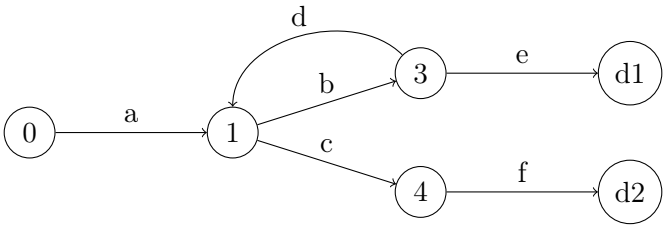
Going back to the die example of Figure 2.1, suppose we want to know the probability of throwing a 1 or a 2. That corresponds to the unbounded reachability of d1 and d2, the probability of eventually reaching those states. Once states 2 and d3 are visited, d1 and d2 can never be reached. Therefore, these states, and every state further down the path, can be merged into one single error state. For the purpose of this example the error state has been left out. The remaining part of the DTMC is turned into a FA with letters as edge labels, Figure 2.2a shows this starting state. The elimination algorithm cannot be used on an entire FA but rather on a single state of a FA. Therefore, we have to choose which states to eliminate and in what order. In this situation we are interested in states d1 and d2 and start in state 0, every other state can be eliminated. Figure 2.2 shows two possible orders for eliminating the states, the final resulting RE is both edges combined with the *or* operator.

While the language accepted by the resulting REs is the same, they differ in length: **a(bd)*be** + **a(bd)*cf** of length 10 compared to **ab(db)*e** + **acf**+**ab(db)*bcf** of length 15. This size difference increases with the size of the automaton as shown by Maneth [8], which makes the elimination order a candidate for optimization.

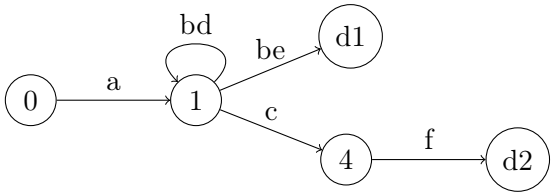
2.2.3 Markov chain state elimination

Another way to perform state elimination on a MC was proposed by Hahn et al. [9]. When performing the elimination algorithm the transition probabilities are solved directly. This results in smaller edge labels and completely prevents the resulting probability from exploding in size. The steps for elimination are very similar to state elimination in a FA as described in Section 2.2.2. The only difference is the formula used for combining the edges. This formula is:

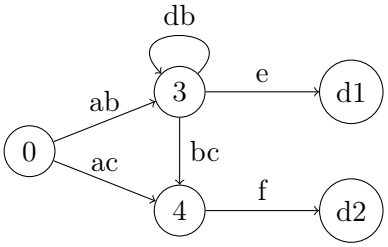
$$P_{ij} = P_{ij} + P_{ik} \frac{1}{1 - P_{kk}} P_{kj}$$



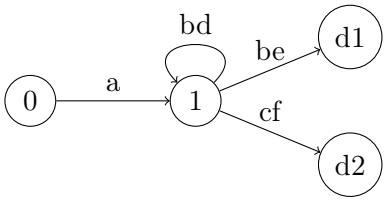
(A) Starting FA.



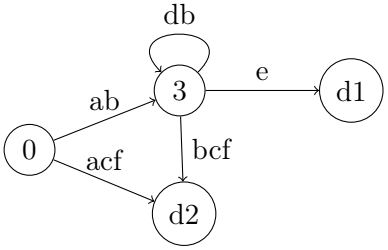
(B) State 3 eliminated.



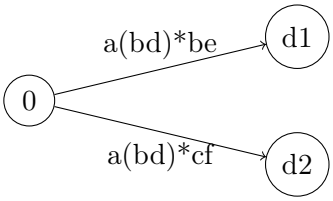
(C) State 1 eliminated.



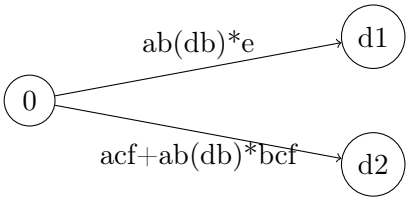
(D) State 4 eliminated.



(E) State 4 eliminated.

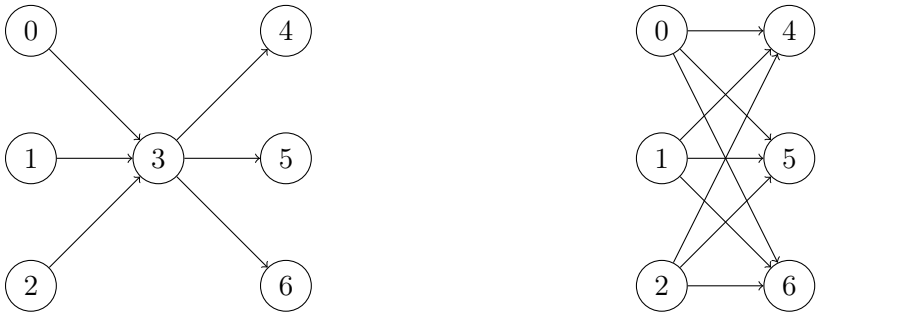


(F) State 1 eliminated.



(G) State 3 eliminated.

FIGURE 2.2: 2 different orders of state elimination. Both are starting in (A) then order 1 on the left eliminates $3 \to 4 \to 1$ and order 2 on the right eliminates $1 \to 4 \to 3$.



(A) Starting FA - 6 edges. (B) State 3 eliminated - 9 edges.

FIGURE 2.3: Example of state elimination generating more edges than it removes when eliminating a state.

where now P_{ij} is once again the probability of the transition occurring from state s_i to s_j.

Because these formulas can be mathematically solved or simplified during the elimination process this algorithm does not suffer as much from a different order as the basic FA algorithm. The number of calculations which need to be performed still has a difference: six calculations for the left order in Figure 2.2 compared to nine for the right order.

The memory usage of this algorithm can still blow up with the number of states because the elimination algorithm can generate new edges. Figure 2.3 shows an example of how eliminating a state can introduce more edges than it eliminates.

2.3 Related work

Most research related to the state elimination algorithm is in the field of RE generation from FAs. Since the algorithm is the same except for the edge labels I expect it to be applicable for the model checking use case as well.

2.3.1 Performance evaluations

Moreira et al. [10] compares the performance of several elimination orders based on the resulting alphabetical size of the RE. There is no best elimination order for every model tested, each order has the best performance for some of the models.

Han [11] introduces different orders and also performs a performance comparison. Concluding that the performance impact introduced on the elimination algorithm is negated by the reduced size of the resulting RE.

Gruber and Holzer [12] present a literature study on the conversion of RE to FA and back. They list several developments on the state elimination algorithm including on the order of elimination. Ordering states on their degree presented by McNaughton et al. [13] and ordering on how many new edges are generated by the elimination of a state by Lombardy et al. [14]. They also report comparable results between elimination orders.

2.3.2 Vertical decomposition

Bridge states are defined by Han [15] to be states where, if the chain passes through this state, it cannot visit any of the previous states except the bridge state itself. Effectively

they make it possible to split the automaton at this state, where the bridge state is the final state of the first half and the starting state of the second half.

These states are then used to chop the automaton in (several) sub-automata which are then separately calculated, and summarily concatenated for the final resulting RE. These states are rare according to Moreira [10], however, random automata have been used for these experiments which might not reflect real world automata.

2.3.3 State weight

Another option is to calculate the weight of each state and use that to eliminate either the heaviest or lightest state first. There are different definitions of this weight heuristic. Delgado and Morais [16] are performing state elimination on RE's and define the weight of a state as the weight added to the automaton when the state would be eliminated. The weight of the automaton is defined as the sum of the sizes of all regular expressions labeling the edges of the automaton.

When using probabilities, the weight of the automaton with this definition would only decrease when eliminating a state or when we are performing state elimination on a PMC with a small amount of parameters.

2.3.4 Counting cycles

Moreira et al. [10] propose 2 variations of cycle counting orderings: statically count the amount of cycles a state is present in the automaton; and dynamically count this for each elimination step. The latter being much more computationally intensive than the first; $\mathbb{O}(n^2)$ versus $\mathbb{O}(n^3)$ respectively. The states are ordered from the least cycles participated in to most.

Chapter 3

Research Questions

This chapter defines and explains the research questions for this thesis.

Section 2.2.3 explained the state elimination algorithm and the edge explosion problem which can occur during this elimination process. Different orderings result in different performance gains. Without a clear insight into why this is the case for particular DTMCs or automatons, an ordering cannot be chosen beforehand with certainty of a gain in performance. This thesis aims to provide this insight and use it to positively impact the performance of the state elimination algorithm.

This thesis focuses on the practical implementation of orders in existing research into an existing codebase and using that implementation to gather results. A smaller theoretical framework is presented for the reasoning behind the elimination orders.

RQ1

How to choose an elimination order which has a positive impact on performance?

The performance consists of two parts, compute and memory usage. They are closely related, more transitions means more is needed to store them and more computations are needed to eliminate them. This choice is made with a model as input, just before the elimination process starts. This question is separated into sub-questions below.

RQ1.1

Which state elimination orders exist?

To evaluate elimination orders against one another they have to be identified first.

RQ1.2

What is the difference in performance between elimination orders?

This question relates to the resources used by just the elimination algorithm itself. The computational cost of determining what that order is, is excluded.

RQ1.3

Which relations exist between characteristics of MCs and the performance of an elimination order?

The hypothesis is that MCs exhibit certain characteristics in terms of how states are interconnected, which can be correlated to a performance difference in elimination orders.

RQ1.4

When does the performance gained by choosing a better elimination order overtake the computational cost of determining that order?

The entire elimination process includes the determination of the elimination order. Detecting certain characteristics might prove computationally expensive to the point that it overtakes the performance gained by providing a better order.

Chapter 4

Methodology

The general approach for this thesis is as follows:

- Evaluate the ordering strategies against a compiled list of models.
- Create orders which have a better result for all models or, find characteristics in a model which can indicate a certain order should be used for that model.
- Implement these orders and characteristics such that the program selects an order and evaluate this choice.

This chapter first lays out all elimination orders and how they work in Section 4.1, Section 4.2 does this for the heuristic orders. Section 4.3 explains the metrics which quantity the elimination orders and Section 4.4 explains how they are stored. Section 4.5 lists the tools used. Section 4.6 discusses the implementation of both elimination orders and metrics. Section 4.7 explains the preprocessing steps in place. Section 4.8 lists the models used for evaluation the elimination orders. Section 4.9 lists utility scripts created during the thesis. Section 4.10 explains how the metrics are visualized for the evaluation.

4.1 Elimination orders

This section defines the elimination orders which have been evaluated. Why these have been chosen or created is discussed in Chapter 5. For clarity elimination orders are noted as $\overrightarrow{EliminationOrder}$ from now on.

Forward

$\overrightarrow{Forward}$ collects states starting from the initial set of states. The model is traversed breadth first and all states visited are collected in that order. Going **forward** from the initial state.

Forward reversed

$\overrightarrow{ForwardReversed}$ takes the order from $\overrightarrow{Forward}$ and reverses it.

Backward

$\overrightarrow{Backward}$ collects states similarly to $\overrightarrow{Forward}$ but is starting from the target set of states. Going **backward** from the target states to the initial state. Not all states can reach the target set of states, these are missed by this collection method. Any missing states which should be eliminated are added afterwards, ordered by state number. Ordering by state number is effectively random.

Backward reversed

$\overrightarrow{BackwardReversed}$ takes the order from $\overrightarrow{Backward}$ and reverses it.

Degree

The degree of a state is defined as the sum of edges incident to the state. $\overrightarrow{Degree}$ orders states from least to greatest degree.

Degree mult

Similar to $\overrightarrow{Degree}$ but instead of summing the in- and out-degree, they are multiplied. If a self loop is present, one is subtracted from the larger of the in- and out-degree. See Algorithm 1. $\overrightarrow{DegreeMult}$ orders states from least to greatest multiplicative degree.

Algorithm 1 Calculation of the multiplicative degree of a state.

```
function DEGREEMULT(indegree, outdegree, selfloop)
    if selfloop then
        if indegree ≥ outdegree then
            indegree ← indegree - 1
        else
            outdegree ← outdegree - 1
        end if
    end if
    return indegree × outdegree
end function
```

Simple cycle count

$\overrightarrow{SimpleCycleCount}$ counts all unique cycles a state is participating in and orders the states from the least cycles to the most cycles.

Cycle size 3 count

$\overrightarrow{CycleSize3Count}$ counts all unique cycles which have exactly 3 states participating. It orders the states from least to most participations in such cycles.

4.2 Heuristics

Heuristics are popularly known as rules of thumb, intuitive judgements or simply common sense, as defined by Pearl [17]. A heuristic function is a function that ranks alternatives at each step based on available information to decide which next step to take.

Heuristic elimination orders make use of any number of heuristic functions in any order. Each function gives a ranking to the given states. The best ranking states, the ones which should be eliminated next, are output as a set. This set is given to the next heuristic if present. Eventually a single state should remain, which is to be eliminated next. Heuristics are shown as $\overleftarrow{Heuristic}$ for clarity.

Degree mult

$\overleftarrow{DegreeMult}$ is a heuristic version of $\overrightarrow{DegreeMult}$, it returns the lowest degree states.

Neighbour degree change

Take all neighbours of a state and calculate the multiplicative degree of all of them. Then eliminate the state and recalculate the degrees of all previously selected neighbours. For each neighbour calculate the change in this degree and sum all changes. $\overleftarrow{NeighbourDegreeC}$ returns the states which have the lowest change. See Algorithm 2.

Algorithm 2 Calculation of change in multiplicative degree of the neighbours of a state.

```
function NeighbourDegreeChange(state, model)
    before ← NEIGHBOURDEGREEMULT(state, model)
    model.eliminate(state)
    after ← NEIGHBOURDEGREEMULT(state, model)
    model.revertElimination(state)
    return after - before
end function
```

Minimal neighbour degree change

$\overleftarrow{MinNeighbourDegreeC}$ takes the same initial approach as $\overleftarrow{NeighbourDegreeC}$ by calculating the degree change for every neighbour. Then it takes the minimal degree of all neighbours for every state and returns the state which have the lowest minimal value.

Indegree

$\overleftarrow{Indegree}$ returns the states with the lowest number of incoming edges, also called the indegree.

4.2.1 Heuristic orders

These are numbered heuristic elimination orders which consist of a number of ordered heuristics.

Heuristic 1

$\overleftarrow{DegreeMult} \rightarrow \overleftarrow{NeighbourDegreeC}$

Heuristic 2

$\overleftarrow{DegreeMult} \rightarrow \overleftarrow{NeighbourDegreeC} \rightarrow \overleftarrow{Indegree}$

Heuristic 3

$\overleftarrow{DegreeMult} \rightarrow \overleftarrow{NeighbourDegreeC} \rightarrow \overleftarrow{MinNeighbourDegreeC}$

Heuristic 4

$\overleftarrow{DegreeMult} \rightarrow \overleftarrow{MinNeighbourDegreeC} \rightarrow \overleftarrow{NeighbourDegreeC}$

4.3 Metrics

Number of edges

The performance of a model needs to be quantized in order to evaluate it. The goal of this thesis is to reduce memory and compute usage which is directly influenced by the size of the model, both int the number of states and the number of edges. During the elimination process the number of states with edges will only ever decrease. However, the number of edges can fluctuate greatly. This metric can be plotted on a graph, where the x-axis is the number of states eliminated and the y-axis the number of edges at the moment of elimination. Problematic orders can be spotted with ease in such a graph in the form of a large spike. It is not very suitable for comparison between elimination orders because a single large spike can have the same performance impact as many smaller ones. This difference is hard to spot from a graph as a human.

Calculations

A better metric for performance is the number of calculations made by the elimination algorithm. A calculation is defined as a combination of edges. In the elimination algorithm, every incoming edge is combined with every outgoing edge, optionally with the self-loop. When a state is eliminated the number of calculations is therefore, $\texttt{indegree} \times \texttt{outdegree}$ if no self loop is present and $\texttt{min}((\texttt{indegree}+1) \times \texttt{outdegree}, \texttt{indegree} \times (\texttt{outdegree}+1))$ if there is a self-loop. If a self-loop is present it is multiplied with either every incoming edge or every outgoing edge. A small optimization is to multiply it with the smaller set of the two. This also counts as a calculation, therefore, one is added to either the indegree or outdegree.

Every edge which is created also must be eliminated eventually, therefore, the calculations metric will shoot up similarly to the graph of the number of edges. Single numbers can be easily compared to one another and the best option can be determined as well. More edges need more memory to be stored, and every created edge also needs to be eliminated eventually. A higher calculation metric therefore indicates more memory and processor usage. The calculations metric will be used to quantify the performance of the elimination orders in this thesis.

Time

Calculating the order to be eliminated is a non-trivial task and can take significant computation time itself. Preferably a metric such as the calculations is used for quantifying the time orders take to calculate as well. Because of the number of different orders and complexity of calculations they perform this process would take too much time during this thesis. The calculation of the order is therefore simply timed using this metric thus depends on the hardware on which it is being executed.

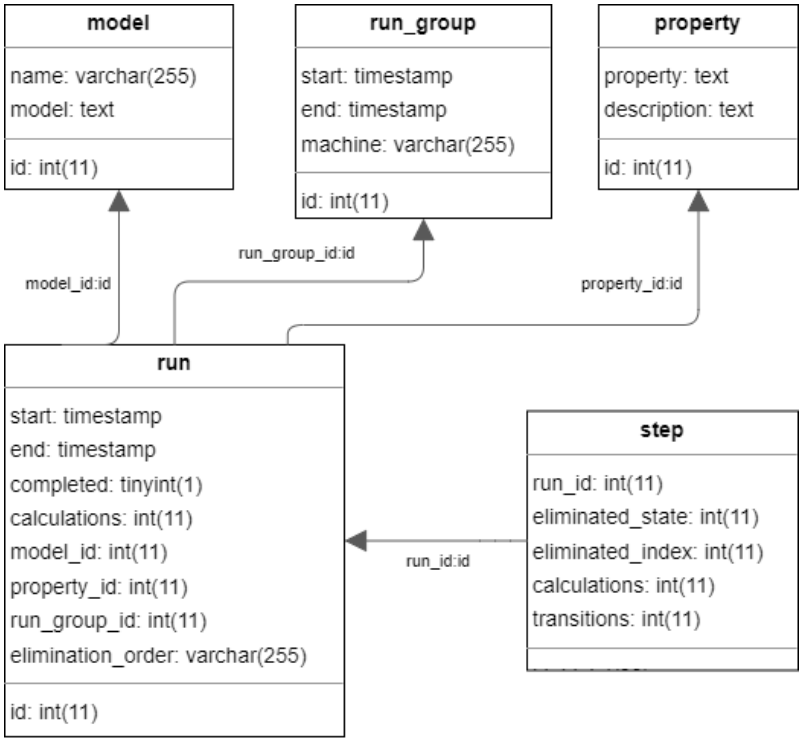


FIGURE 4.1: Database structure.

4.4 Data structure

A SQL database has been created to store the collected metrics, the metrics can then be queried manually or by another application for visualization and evaluation. The structure of this SQL database is show in figure 4.1 and an explanation of every field in table 4.1. For every *elimination run* (afterwards just *run*) every step is recorded: which state has been eliminated; how many calculations have been done; and how many edges are left after the state has been eliminated. These steps are linked to a run row which is in turn linked to a model and a property row. This allows the user to query the metrics for a certain model, elimination order or run. The elimination order does not have its own table since there is no extra data besides the name of the order, compared to the model which also stores the entire model definition.

4.5 Tools

The tools used in this thesis are open source if available to allow the reader to reproduce any results without having to acquire proprietary software. Many smaller tools and programs have been used throughout the thesis, but the most important ones are listed below.

Eclipse[1]

IDE with support for many languages. Used for the implementation and debugging of the algorithms presented.

[1] https://www.eclipse.org/downloads/

step	
eliminated_state	Number of the eliminated state
eliminated_index	Order index of the eliminated state
calculations	Number of calculations performed
transitions	Number of transitions after elimination
run	
start	Start of the run
end	End of the run
completed	Run has completed
calculations	Total number of calculations
elimination_order	Elimination order used

run_group	
start	Start of the benchmark
end	End of the benchmark
machine	Identifier of the machine the benchmark has been run on
model	
name	Name of the model
model	Model input text file
property	
name	Name of the property
description	Property description

TABLE 4.1: Explanation of data structure variables.

MariaDB[2]

SQL database server. Used to store the metrics from the elimination runs.

Apache Superset[3]

Data exploration and visualization platform. Used to visualize and explore the generated metrics.

4.6 Implementation

This section explains all details related to the implementation of the elimination orders and metrics into the PRISM codebase. First the location of the code is determined in Section 4.6.1 and how this code can be executed in Section 4.6.2. Section 4.6.3 discusses all new code which was written and Section 4.6.4 discusses all modifications to include this new code.

4.6.1 Existing codebase

PRISM has a few different engines which can be used for model checking, each having its own strengths and weaknesses. The exact engine is the only engine which has the state elimination algorithm already implemented and is therefore the engine which will be used. This engine stores the model explicitly in a graph format which makes it intuitive to manipulate it for the elimination process and inspect it for characteristics.

[2] https://mariadb.org/
[3] https://superset.apache.org/

With multiple engines comes quite an extensive codebase and not all parts originate from the PRISM project but have been incorporated into it at a later time. This is also the case with the exact engine which originates from PARAM [18]. Most of the engine has been placed in the `param` package which greatly decreases the required knowledge for adapting the code.

4.6.2 Compilation

The source code of PRISM is written in Java,but there are some dependencies on libraries which are outside the Java ecosystem. It makes use of a modified version of the CU Decision Diagram (CUDD) package[4], which is written in C and compiled using Make[5]. Make is not readily available for Windows and most installation instructions for PRISM also pertain to a Linux machine. Development on a Windows machine is achieved through WSL (Windows Subsystem Linux) where Eclipse is installed and run from to compile, run and debug PRISM inside the WSL container.

Compiling the Java source code does not result in an instantly runnable jar-file. The program relies on several environment variables which contain the location of external libraries. The project source root folder contains a Makefile which takes care of compiling both Java and C libraries. It creates a runnable shell file which sets all the applicable environment variables to run the jar-file and then calls the Java executable. By inspecting this file, the necessary environment variables are identified and copied to the run configurations in Eclipse. This run configuration is used as a template to create run configurations for each model. The run configurations not only allow for running PRISM from eclipse but also enables the use of the debugger, essential for debugging during implementation.

4.6.3 New code

Implementing the orders and extracting metrics has introduced several new classes into the PRISM codebase. This subsection describes all created classes and their functionality. Packages of the classes will be aligned to the right of the page.

EliminationOrderIterator **param.elimination**

The elimination orders have been abstracted to a single class `EliminationOrderIterator`. Other parts of the program do not need to know what the order is doing or when, they just need to know if there are more states to eliminate and what state to eliminate next. These 2 requirements are very much akin to the `Iterator` and therefore this class implements that interface. An elimination order is defined by having 2 functions, `hasNext()`, which returns true if there are more states to eliminate and `next()`, which returns the next state to eliminate.

The next function is supposed to be called only after the elimination of the previous state. This allows the implemented order to decide if it calculates the next state on instantiation of the class all at once, or incrementally after each elimination.

EliminationOrder **param.elimination**

It is possible to get all inheritors of a class and as such compute all available elimination orders at runtime, but this process reduces the readability of the code greatly. Therefore,

[4] https://github.com/ivmai/cudd
[5] https://www.gnu.org/software/make/

an `enum` has been created which holds all available orders. When a new elimination order is created by implementing the `EliminationOrderIterator` it should get a unique value in this enum as well.

Trivial elimination orders **param.elimination**

All elimination orders implement the `EliminationOrderIterator` either directly, or indirectly through another class. $\overrightarrow{Forward}$, $\overrightarrow{ForwardReversed}$, $\overrightarrow{Backward}$ and $\overrightarrow{BackwardReversed}$ are re-implemented from exising code in the `StateEliminator`. These orders are calculated before the elimination process starts, the order is stored in a `List`. A list already provides the ability to create an `Iterator` on it, `next()` and `hasNext()` on the order call the respective functions on this iterator. $\overrightarrow{Reversed}$ versions make use of the builtin reverse function of Java to reverse the list and the created iterator.
$\overrightarrow{Degree}$ and $\overrightarrow{DegreeMult}$ are trivially implemented but calculate the next state to be eliminated only when `next()` is called. The `MutablePMC` provides access to collections of incoming and outgoing edges for every state. The sizes of these collections are summed for $\overrightarrow{Degree}$ and multiplied for $\overrightarrow{DegreeMult}$, taking into account the extra logic in case of a self loop.

Cycle elimination orders **param.elimination**

The next two orders, order states based on how many cycles they participate in. This requires the detection of cycles, which is not an easy task. Johnsons' [19] algorithm is used to find all simple cycles in a graph. The algorithm takes Strongly Connected Components (SCCs) as input and requires the states to be ordered. The only requirement on this order of states is that it does not change while running the algorithm, therefore the existing state numbers can be used.

SCC computation has already been implemented in PRISM, albeit in a different engine. The `MutablePMC` has to be transformed into a `DTMCSimple` from the `explicit` engine, for which the algorithm has been implemented. To keep things simple all probabilities have been set to 1. These probabilities do not affect the result of the algorithm, smaller functions on the edges theoretically increase the memory usage the least. The resulting SCCSs are then fed to Johnsons' algorithm, which outputs every simple cycle in the graph.

The cycles output by Johnsons algorithm are used to keep a count for every state how many cycles it is participating in. The $\overrightarrow{SimpleCycleCount}$ order counts every cycle whereas the $\overrightarrow{CycleSize3Count}$ order counts only cycles of size 3. Any number of cycles can be implemented but 3 was chosen to experiment with.

Heuristic orders **param.heuristic**

A `Heuristic` is an abstract class implemented by all heuristics which provides functionality to return the set of states with the lowest weight calculated from a given set of states. The individual heuristics only need to implement the `getWeight()` function which returns the weight of a given state.

Degree Mult **param.heuristic**

$\overleftarrow{DegreeMult}$ is a simple heuristic which bases its implementation on $\overrightarrow{DegreeMult}$.

Neighbour degree change heuristic **param.heuristic**

When eliminating a state s, the edges are combined and the new probabilities on those edges are computed. This process is only reversible with great effort so a straightforward implement of Algorithm 2 is not possible. The probabilities on the edges are not used in that algorithm, only if the edge exists. If a neighbour is in s' outgoing state set, the new indegree of that neighbour can be calculated by taking the union of its incoming state set with s' incoming state set. Similarly, if the neighbour is in the incoming state set of s, the outgoing state set of the neighbour is unioned with the outgoing state set of s. s Is then removed from those sets and now the multiplicative degree can be calculated. Algorithm 3 shows the pseudocode implementation.

Algorithm 3 Implemented neighbour degree change.

```
function GETNEIGHBOURDEGREECHANGE(targetstate)
    change ← 0
    for neighbour ∈ (state.incomingstates ⋃ state.outgoingstates) do
        before ← DEGREEMULT(
            neighbour.indegree,neighbour.outdegree. neighbour.hasselfloop)
        incoming ← neighbour.incomingstates \ {state}
        outgoing ← neighbour.outgoingstates \ {state}
        if neighbour ∈ state.incomingstates then
            outgoing ← (outgoing ⋃ state.outgoingstates)
        end if
        if neighbour ∈ state.outgoingstates then
            incoming ← (incoming ⋃ state.incomingstates)
        end if
        after ← DEGREEMULT(
            incoming.size,outgoing.size, neighbour ∈ incoming)
        change ← change + (after - before)
    end for
    return change
end function
```

Minimal Neighbour Degree Change Heuristic **param.heuristic**

This heuristic copies the code from the previous Neighbour Degree Change Heuristic and adapts it slightly to get the minimal change from the neighbours instead of the sum.

Minimal Indegree Heuristic **param.heuristic**

Simply returns the indegree of a state as weight.

DOTExport **param.benchmark**

Helper class for exporting a `MutablePMC` to a dot representation.

Elimination-Step and -Run **param.benchmark**

`EliminationStep` and `EliminationRun` are Java records of the similarly named elimination step and run in Figure 4.1. The `EliminationRun` class contains all steps for a single

elimination run, which it exports upon completion of the run. This export is done because memory usage is one of the key problems identified in this thesis, therefore, all metrics are kept in memory only for as long necessary.

EliminationRunGroup **param.benchmark**

The state elimination process is split between a number of different classes. To limit changes to these classes as much as possible, the `EliminationRunGroup` class has been created to statically keep track of elimination runs. After all runs have been completed it exports the data to a JSON file and compresses this file along with the step file of each run to a single archive.

4.6.4 Modifications

Besides new code some classes have to be adapted as well. This subsection describes the important changes made to the existing codebase of PRISM.

PRISMSettings **prism**

This class holds the setting definitions for PRISM, including the available elimination orders. The orders setting has been adapted to be generated from the `EliminationOrder` enum. This removes the additional step of adapting the settings when adding a new elimination order.

ValueComputer **param**

This class is called by PRISM to verify properties on a model. It is the entry-point into the `param` package and takes care of delegating calls to the appropriate other classes for relevant properties and models. It constructs the `MutablePMC` from the input model, which is used as input for the elimination process.

The `ValueComputer` also performs preprocessing on the input model to which one more step is added, removal of unreachable states, described in section 4.7.2.

The elimination order was originally given to the `StateEliminator` which contained all order logic. Since that was abstracted away to `EliminationOrderItator`, the elimination orders must now be constructed in this class. The iterator is then passed on to the `StateEliminator`.

The elimination order is defined by the `EliminationOrder` enum throughout the calls until the program arrives at the elimination step. If the elimination order chosen is not `Benchmark`, the `ValueComputer` will instantiate a `StateEliminator` with the constructed `EliminationOrderIterator` and `MutablePMC`. After the state elimination has been performed the results are retrieved and mapped back onto the original model. Multiple preprocessing steps can change the number of states in the model, which can change the number of a state which is used to keep track of the states. A mapping is kept from the original model all the way to the model where the state elimination is performed on. If the elimination order is `Benchmark` the elimination process is repeated for all elimination orders defined in the `EliminationOrder` enum.

Logging calls to the `EliminationRunGroup` class are introduced at the appropriate points: starting, stopping and exporting of single runs.

StateEliminator **param**

The `StateEliminator` contains the elimination algorithm and helper functions related to the state elimination algorithm. It takes a pre-processed model and elimination order as input for instantiation. The `eliminate()` function is called to perform the algorithm and retrieve the resulting probabilities.

This class calculated the elimination order and stored the result in an array which was consumed during elimination. This has all been removed in favor of the `EliminationOrderItator`. The elimination algorithm is slightly adapted to call the `next()` and `hasNext()` functions.

Model exports with the `DOTExport` class have been placed at strategic points to allow inspection of the model at any point in the elimination process. These are disabled by default because large models can generate a lot of data by exporting the model at every step.

The PRISM implementation of the state elimination algorithm has support for rewards and time. This not only required extra calculations for each step but added more steps to the algorithm. Instead of removing all transitions when a state is eliminated, only the incoming edges and the self-loop are removed, the outgoing edges remain and add a significant number of calculations to the process. In essence PRISM is calculating the reachability probability for every state at once, this is necessary to get consistent results for properties referencing rewards.

In the case that the property which is verified does not require rewards, these extra steps are unnecessary but still performed. A copy of the elimination algorithm has been created which removes all outgoing edges. The version of the algorithm is selected based on if the model uses time, rewards or both. Removing all edges introduces a new problem related to the elimination of the initial state. Since the original algorithm does not delete outgoing edges, eliminating this state is not a problem. The adapted algorithm does remove these and thus leaves the initial state without outgoing transitions. This causes the `ValueComputer` to return incorrect results. All orders have been adapted to exclude the initial and target states, effectively preventing these states from being eliminated, which solves the problem of incorrect results.

4.7 Preprocessing

Before reaching the state elimination algorithm there are some preprocessing steps in place, some which happen for each engine and some which are specific to the state elimination situation. One step has been purposefully turned off sometimes and another has been added. They are listed in the following subsections.

4.7.1 Bisimulation

The `Lumper` is a class that performs a preprocessing step which, depending on settings, performs weak or strong bisimulation on the model. This bisimulation can group states together and usually results in a smaller model without changing the probabilities for the requested properties. Weak or strong simulation is mainly decided by the type of property used, but can be overridden through settings and also completely disabled.

This preprocessing step has mixed results depending on the input model. In some cases it can completely reduce the model to only 2 states while in other cases it does not remove

any states at all. This is taken into account when selecting models, a 2 state MC does not have a lot to order and a 2 million state MC cannot be calculated in a reasonable amount of time.

4.7.2 Unreachable states

PRISM performs a deadlock analysis which removes some unreachable and deadlocked states from each model. When performing state elimination there is an initial state, a set of target states and all states which are neither. For the latter category of states it is possible that they do not connect to the target and initial states. Therefore, they are completely unreachable and are removed from the model.

Another definition of unreachable is in relation to the target states. This is the same problem which presented itself with the $\overrightarrow{Backward}$ orders. States can exist which are reachable from the initial state, but once visited the model can never reach a target state. A preprocessing step has been introduced right before the bisimulation where all of these unreachable states are replaced by a single unreachable state. The states cannot simply be removed because that would leave other states with a total outgoing probability of less than one which violates the Markov property.

4.8 Dataset

Comparison of the orders is done by running all of them on a model. Running on only one model could result in the orders begin overly focussed on the patterns in that model. Therefore, more models are needed for the evaluation. There are a few sources identified which could yield models for running. The size of the model is of importance as the exact engine cannot handle the largest ones. Experimental runs suggest the limit is somewhere around the 500,000 states. Every model will be tested if it finishes before being added to the dataset. This section lists all models used in this thesis.

4.8.1 PRISM

PRISM itself has a number of examples for a number of supported model types, among them DTMC's. These models are sometimes used to explain a part of the PRISM program or are part of case studies done with PRISM.

BRP [20] Bounded Retransmission Protocol, a protocol which sends a file in a number of chunks but allows only a bounded number of retransmissions of each chunk.

Herman Herman's self stabilizing algorithm. A self-stabilizing protocol for a network of processes is a protocol which, when started from some possibly illegal start configuration, returns to a legal/stable configuration without any outside intervention within some finite number of steps. The original protocol is from Herman [21] and the model reference from Kwiatkowska et al. [22].

Leader Sync [23] Given a synchronous ring of N processors design a protocol such that they will be able to elect a leader (a uniquely designated processor) by sending messages around the ring.

Dice [3] Model of a die using only fair coins.

Besides the examples there is also a benchmark suite[6], this suite holds models which have a larger amount of states in them when compiled. These are used to benchmark PRISM and therefore can also be used to benchmark the elimination orders. Because of the engines size limitations not all available models can be used. The usable models are:

EGL A randomized protocol for signing contracts. The protocol is from Even et al. [24] and the model reference from Norman and Shmatikov [25].

Crowds A system for protecting users' anonymity by blending them into a crowd. The system is from Reiter and Rubin [26] and the model reference from Shmatikov [27].

NAND NAND multiplexing, a technique for constructing reliable computation from unreliable devices. The technique is from Neumann [28] and the model reference from Norman et al. [29].

4.8.2 QComp

The Comparison of Tools for the Analysis of Quantitative Formal Models or QComp[7] for short is a friendly competition among verification and analysis tools including PRISM. The competition also has a benchmark set, the Quantitative Verification Benchmark Set or QVBS[8] for short. Which includes benchmarks from different tools in different formats, some have been converted to be supported in other tools. The PRISM benchmark set is also included in this set and also has the most submissions in it. Nonetheless, there are 2 more models which can be used in the dataset:

Coupon This model describes a contest where the goal is to collect all coupons from a given set. It can be abstracted as drawing from an urn of N different coupons with replacement where drawing each coupon is equally likely. The model is first presented by Flajolet et al. [30] and used in Jansen et al. [31]

Haddad Haddad-Monmege is an adversarial example that highlights the problems of the traditional convergence criteria in value iteration. The model is first presented by Haddad and Monmege [32]

4.8.3 Other research

The models which are listed above all come from some form of research, protocol or technique. This search is extended to general libraries by looking for papers which reference DTMCs, MCs and probabilistic verification. There are a number of papers which make use of these techniques in different research domains. The main domains are computer science and biology. The research in the biology field usually pertains to either the inner systems, cells for example, or the outer systems, humans and societies.

The methodology of finding other research for the dataset is as follows. First search the papers and categorize them lightly on the size of the model which is used. After a certain amount has been collected try to categorize those papers more specifically on the type of DTMC used. Finally extract the DTMCs from useful categories and enter them into the dataset. This all with a boundary that a single paper may not take too much time to categorize or implement as that would in turn limit the time available for results.

[6] https://www.prismmodelchecker.org/benchmarks/models.php#dtmcs
[7] https://qcomp.org
[8] https://qcomp.org/benchmarks/

After the first step about 40 papers have been imported and lightly categorized into the categories small, large, queue, prism and unknown. Where small is less than 50 states, large greater than 50 states, prism has a prism model present, and the queue category is a single line of states.

When trying to categorize the papers more thoroughly, it became increasingly more obvious with each paper that they are not useful for this thesis. The smaller DTMC's do not provide useful results because they are too small for elimination orders to have an effect on the metrics. The larger models turned out to be queues or grids which are already present in the dataset. The implementation cost was also too high for the larger models. The models are described by many complex formulas, which take a lot of time to understand and convert to a representation suitable for PRISM.

Other research has therefore not been incorporated in the dataset.

4.9 Utility scripts

Some utility scripts have been created to automate parts of the processes which are not already done by the tools used. This section describes the scripts which have been created.

4.9.1 Data importer

During implementation single invocations of PRISM through eclipse are enough, the results print to the console or the debugger is used to inspect the process while running. When comparing multiple elimination orders against one another the data from multiple runs has to be used. This data is exported as described in Section 4.6.3, however this file still has to be imported into the database.

A Python script has been created which monitors a folder in which PRISM should deposit the data files. After a few seconds this script will then import this data-file into the database. The delay is added to allow PRISM to fully write the file before trying to read it back in.

4.9.2 Runner

Running every model from Eclipse manually would be a very time-consuming process. Another Python script has been created to run all models against all elimination orders consistently. It takes an Excel[9] file as input which contains the definition of the models. Where the model files are stored, which PRISM options should be added and if the model should be run at all. After all runs have concluded it calls the data importer script to import all generated data into the database.

4.9.3 DOTConverter

During elimination runs the process may output every step as a DOT file, if this is enabled in the code. This representation is efficient but not very useful when trying to inspect the graph. Another small Python script has been created to convert all DOT files inside a folder to PNG images which can be easily viewed on any device.

[9] https://www.microsoft.com/microsoft-365/excel

4.10 Visualization

All metrics gathered from the elimination runs are now inside the database. It provides a lot of functionality to query the information, but inspecting the data in this form is not intuitive for humans. As mentioned in section 4.5, Apache Superset has been chosen to visualize the data from the database. This section discusses the setup of that tool.

4.10.1 Dataset

First the database is connected to Superset, it inspects the tables which are then available for use in the *Dataset* tab. Datasets are an abstraction layer in Superset which hide the database implementation from the rest of the tool. Allowing a lot of different data storage solutions to be connected. For each table a dataset is created with some additional properties besides the existing columns. These properties are references to other tables using the defined relationships in the SQL database. When looking at Figure 4.1 the *run_group* dataset, has the property `run_group_id` added. All extra properties are listed in Table 4.2, why these are needed is explained in Section 4.10.2

step	
property	Full property name
elimination_order	Full elimination order name
run_group_id	Id of the run group this run is part of
run	
property	Full property name
model	Full model name
idorder	Concatenation of id and order run
order_id	id
run_group	
run_group_id	id

TABLE 4.2: Extra properties for Superset datasets

4.10.2 Dashboard

With all relationships between tables in place, a dashboard can be created to show all data in one place. A dashboard consists of a number of graphs which have the ability to cross-filter. When selecting a value from a table in a dashboard, Superset will try to filter every other graph on the information selected. This is completely done on property names, which is why the extra properties on the datasets are necessary. If the `id` property of the *run_group* table is clicked it will also incorrectly filter the *run* table on its `id`.

The following tables have been created for cross-filtering the results: run groups, run, models, properties and elimination orders.

The visualization of the data is done through graphs. The data in the graphs is affected by the cross-filtering of the tables. The following graphs have been added to the dashboard:

- line graph of the number of transitions over the number of states eliminated,
- bar graph of the number of calculations made at each elimination step,

- bar graph of the total number of calculations made and
- large table of the total number of calculations for each run.

These data graphs all have dimensions on the elimination order and run itself. Such that, for example, for each elimination order and run there is an individual line for the number of transitions over the number of states eliminated.

Chapter 5

Results

This chapter presents the results of all the elimination runs performed. It goes into detail per relevant set of elimination orders. The metrics of a set of orders and any observations made about them are discussed. With those observations conclusions can be drawn which lead to the next set of elimination orders. The chapter starts with the existing elimination orders in Section 5.1. Section 5.2 discusses the $\overrightarrow{Degree}$ orders which lead into the neighbour degree heuristic in Section 5.3. Finally, the remaining heuristics are discussed in Section 5.4.

5.1 Existing and cycle elimination orders

The first elimination orders which have been evaluated are existing elimination orders. The existing orders from PRISM are: $\overrightarrow{Forward}$ (F), $\overrightarrow{ForwardReversed}$ (FR), $\overrightarrow{Backward}$ (B) and $\overrightarrow{BackwardReversed}$ (BR). The cycle orders are from existing research: $\overrightarrow{SimpleCycleCount}$ (SCC) and $\overrightarrow{CycleSize3Count}$ (CS3C). Table 5.1 shows the calculation metric for every model when these orders are chosen.

Inspecting Table 5.1 it becomes clear that the cycle elimination orders do not have great results. They only have the lowest calculation metric for the Haddad-Monmege model, but every order shares that same lowest result. The weak performance of cycle elimination orders can be explained for some of the models by the fact that those models do not contain any cycles. The cycle orders have no defined order for these models and are completely random. Therefore, the results on these models do not provide much information on this order. Other models do contain cycles but still do not yield much better results. Inspecting the models further gives a little more insight into why this might be the case. Cycles are present, but there are still not many of them. The cycles which are present do not intersect with each other much. The number of cycles a state participates in therefore does not vary much, most states end up with the same number of cycles. This again leads to the order being mostly undefined and random.

The other elimination orders are more interesting to inspect. $\overrightarrow{Backward}$ never yields the best calculation metric but is not always the worst metric. Even while excluding the cycle orders this remains true. $\overrightarrow{Forward}$ and $\overrightarrow{ForwardReversed}$ split the best results on models with respectively four and five best calculation results. $\overrightarrow{BackwardReversed}$ has one best calculation result with the NAND model and 2 ties on the Leader Sync and EGL models.

Model	F	FR	B	BR	SCC	CS3C
Knuth	69	39	53	40	53	53
NAND	15273	10740	8294	4761	37057	37057
Leader4-6	3960	5184	5184	3960	5119	5184
Haddad-Monmege	1196	1196	1196	1196	1196	1196
EGL	7043	10727	10717	7046	10716	10716
(L) EGL	12	11	12	11	12	12
Crowds	375	217	337	282	314	314
(L) Crowds	42	22	31	28	23	23
Coupon	525	1850	3266	1605	3500	3500
(L) Coupon	38	62	79	73	83	83
BRP	1161	8803	7651	23024	7512	7512
(L) BRP	379	278	377	2779	747	747

TABLE 5.1: Number of calculations per model per elimination order for existing and cycle orders. Lower is better, lowest is marked per row.

5.1.1 Observations

Edge explosion

The results of the BRP model with $\overrightarrow{BackwardReversed}$ illustrates the edge explosion problem. The transition graphs for some of the elimination orders for this model are shown in Figure 5.1. Figure 5.1a shows a great increase in the amount of states which starts around 150 states eliminated. The graph grows in size until it has about double the number of edges compared to the starting graph. This is reflected in the calculation metric, $\overrightarrow{BackwardReversed}$ has more than two times as many calculations as the next worst elimination order.

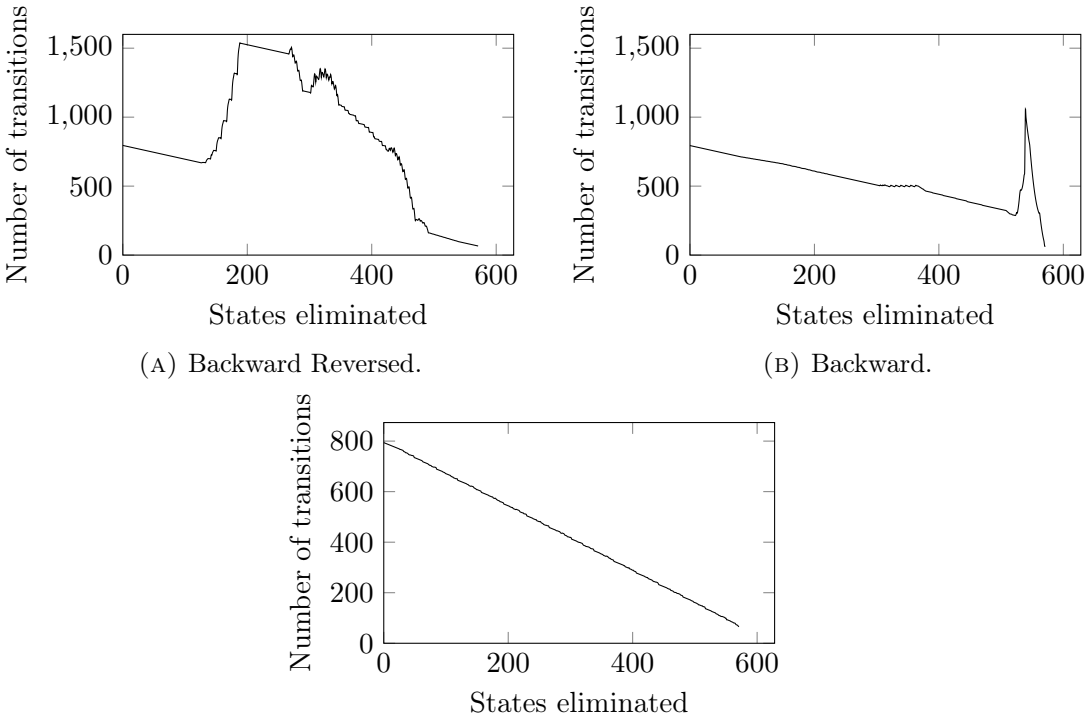
The transition graph of $\overrightarrow{Backward}$ in Figure 5.1b still has this explosion in edges albeit much smaller. The number of edges more than doubles compared to the number of edges a few steps before but that peak is gone a few steps after. This is an indication that there are some problem states or patterns in a model which can cause this blowup. Moving them around in the order can increase or decrease the effect they have. If ordered properly the effect can be nearly negligible as is shown by $\overrightarrow{Forward}$ in Figure 5.1c. No peak is visible in this last graph.

Lumped compared with not lumped

Most of the models have two entries in the results table, one where the lumper is enabled and one where it is not. This is mostly visible in the reduced amount of states for the lumped versions of the model. The most interesting results related to this are with the EGL and BRP models. These models have a different best order for the lumped and not lumped versions, the orders are respectively $\overrightarrow{Forward}$ and $\overrightarrow{ForwardReversed}$. This indicates that either the lumper changes the characteristics of a model or that a size difference in certain characteristics affects the metrics more than other characteristics.

5.1.2 Conclusions

These results confirm the results of the existing research. Among these orders and models there is no clear best order for all models. Individual models have a best order, but there



(A) Backward Reversed.

(B) Backward.

(C) Forward.

FIGURE 5.1: Transition graph of the BRP model.

is no clear indication for why that order is best.

From here on the Haddad-Monmege model will be excluded from the result. This is done because the model yields the same result for every elimination order and thus offers little value. Getting the same result across all orders is the result of the model structure, every state is only connected to the initial and target states. They are identical for the elimination algorithm and any change in order does not result in a change in the calculation metric.

5.2 Degree orders

Section 2.2.3 indicates the memory usage can increase explosively when state elimination is performed. This is due to the amount of edges which is generated in such a case. Two intuitive elimination orders were implemented based on this observation and existing research [13][14], $\overrightarrow{Degree}$ and $\overrightarrow{DegreeMult}$. $\overrightarrow{Degree}$ was implemented first. Inspecting the state elimination algorithm a little better shows that the amount of calculations is almost exactly the multiplicative degree of a state, $\overrightarrow{DegreeMult}$ was implemented shortly thereafter.

5.2.1 Observations

Validating assumption

The results validate the assumption that using the amount of edges as an input for ordering decreases the number of calculations overall. Table 5.2 shows that $\overrightarrow{Degree}$ takes over two best result spots (compared with the results of Table 5.1). $\overrightarrow{DegreeMult}$ matches those results and takes one additional best result spot for the BRP model. Besides the best

Model	Degree	Degree Mult
Knuth	38	38
NAND	8666	6701
Leader4-6	3960	3960
EGL	8308	8308
(L) EGL	12	12
Crowds	236	222
(L) Crowds	23	23
Coupon	1332	1080
(L) Coupon	67	61
BRP	1042	1027
(L) BRP	192	192

TABLE 5.2: Number of calculations per model per elimination order for degree orders. Lowest compared with previous metric table is marked.

result these two orders never yield the worst results out of all elimination orders shown so far.

Pattern in Coupon model

With these two elimination orders added to the results, there are more calculation metric results which are close to one another. Specifically in the lower numbers of the metric closeness is very useful. These lower numbers come from smaller models which in turn have smaller `MutablePMC`s. The amount of states which has to be eliminated is very small, therefore there is not much room for variation and choice. These models can be exported for each step in the elimination algorithm and then compared against another elimination order which has a similar calculation metric. During this comparison attention is given to the difference in order and which elimination step incurred the difference in calculation metric.

The calculation bar graph generated by Superset is a great asset during this inspection. Figure 5.2 shows these bar graphs for the Coupon model with $\overrightarrow{Forward}$ and $\overrightarrow{DegreeMult}$. The difference in the calculation metric is significant, 38 and 61 respectively, but the model is small enough to inspect by hand. The bar graph for $\overrightarrow{Forward}$ is consistently at two calculations per eliminated state. $\overrightarrow{DegreeMult}$ diverges from this consistency at five eliminated states and continues with four calculations per eliminated state. With this information the graph is inspected for both orders at this divergence to investigate a possible cause.

Figure 5.3 shows the graph representation of the coupon model. The probabilities have been removed from the edges for clarity. The model starts off as a pyramid where each state has two outgoing edges to the two states below it. After reaching its maximum width it starts to reduce in the same way except that the outer states also have an edge to a state which has only a self loop. Such a state is also called an absorbing state.

$\overrightarrow{Forward}$ starts to eliminate states breadth first starting from the initial state, $14 \to 17 \to 2 \to 4 \to 11$. This state is shown in Figure 5.4a. The result is that the top two rows have been removed and the initial state now has edges to states 1, 10, 0 and 8. $\overrightarrow{DegreeMult}$ on the other hand has eliminated $1 \to 2 \to 8 \to 14 \to 19$. Up until this step this has resulted

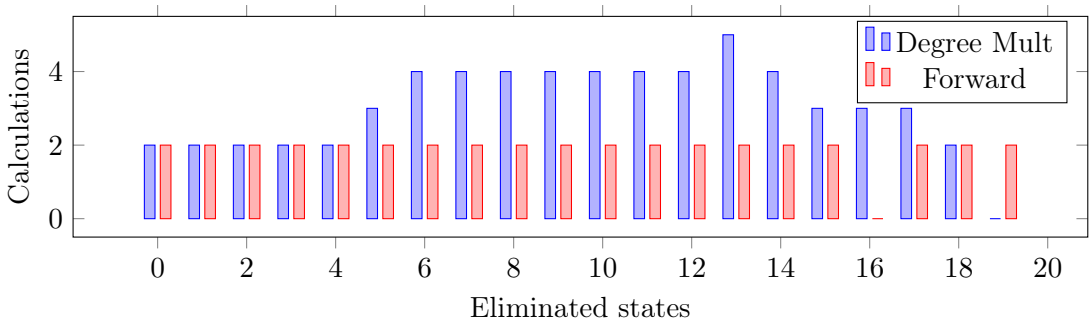


FIGURE 5.2: Bargraph of calculations per elimination step of the Coupon model.

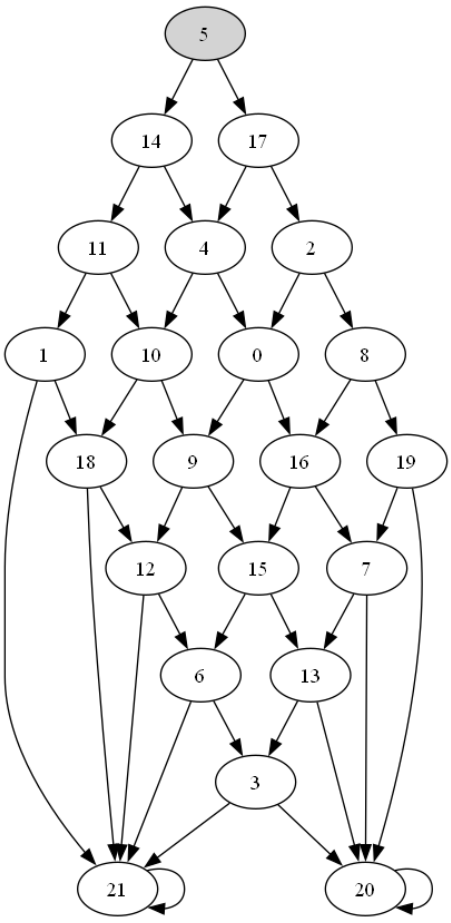
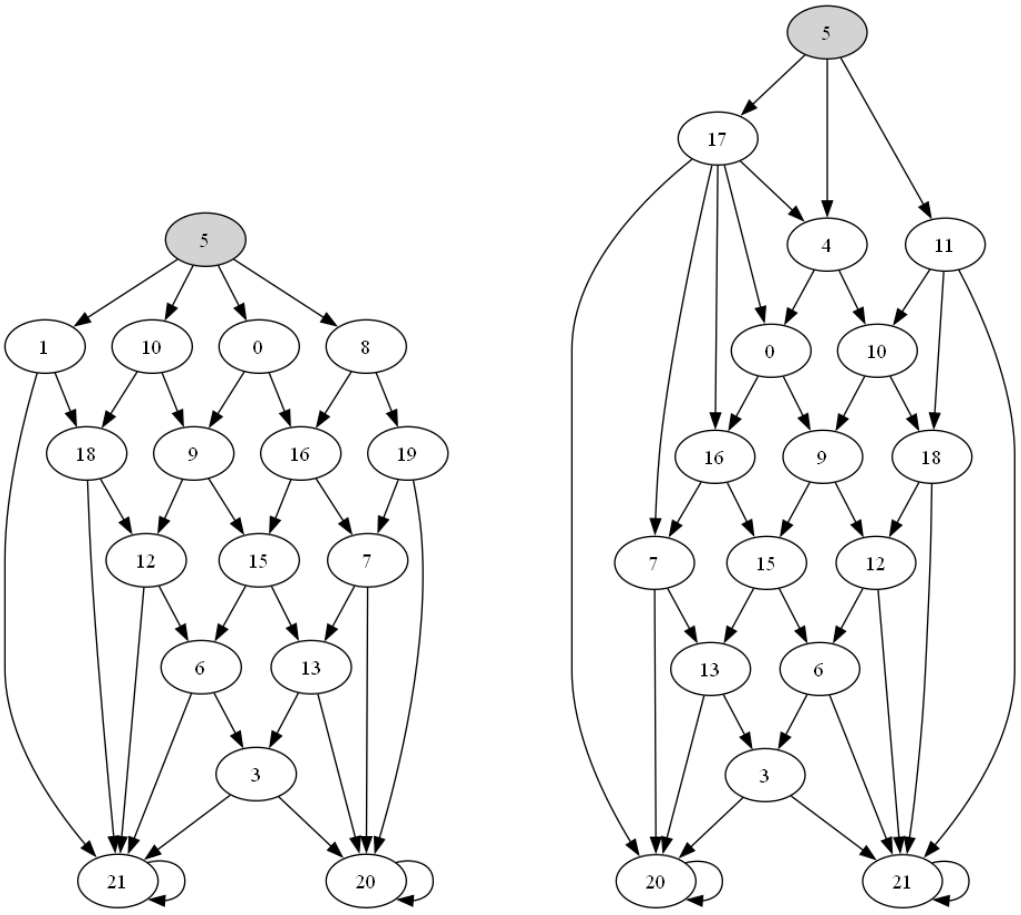


FIGURE 5.3: Graph representation of the Coupon model after lumping. Initial state is gray, target states have only a self loop.



(A) Forward order, will eliminate state 8 next.

(B) Degree Mult order, will eliminate state 11 next.

FIGURE 5.4: Coupon model after 5 states have been eliminated.

in the same amount of calculations being performed as $\overrightarrow{Forward}$. Figure 5.4b shows the graph at this point in the elimination process.

The issue $\overrightarrow{DegreeMult}$ faces now is that there are no more states which have a multiplicative degree of two. It can only choose a state which will result in a calculation cost of more than two. $\overrightarrow{Forward}$ still has states available with multiplicative degree two and will continue to have these until it finishes. Effectively $\overrightarrow{DegreeMult}$ has undermined itself by choosing a not optimal order from the states it deemed best to eliminate. The set of states this order deems fit to be eliminated at the start of the elimination process is $\{14, 17, 11, 1, 2, 8, 19\}$. This set contains the states which $\overrightarrow{Forward}$ selected to eliminate. $\overrightarrow{DegreeMult}$ could have selected those same states from the of set of available states but at this point all states are equal to one another. The only option it has left taking a random state out of those options.

This random choice is only a valid option if it does not affect the resulting calculation metric. In this case it does. The calculation metric can give the same result for many graphs but to illustrate that sometimes a random choice is necessary one can think of graph automorphisms [33]. A graph can be subdivided into multiple sub-graphs, if these sub-graphs are equal to one another in every way shape and form they are isomorph. The

Model	H1	H2	H3	H4
Knuth	39	39	39	38
NAND	3220	3219	3220	5659
Leader4-6	3960	3960	3960	3960
EGL	7043	7043	7043	7043
(L) EGL	12	11	12	12
Crowds	222	222	222	222
(L) Crowds	22	22	22	22
Coupon	525	525	525	525
(L) Coupon	38	38	38	38
BRP	796	761	796	1027
(L) BRP	191	191	191	191

TABLE 5.3: Number of calculations per model per elimination order for heuristic orders. Lowest compared with previous metric tables is marked.

parent graph then contains automorphisms. Take a graph which consists of an initial state, target state and exactly two sub-graphs which are isomorph. Any state selected will have at least one completely equal state in the other sub-graph. Therefore, a random choice between these states has to be made.

5.2.2 Conclusion

When taking a closer look at the two graphs some states stand out. States 17 and 11 which started as states with a multiplicative degree of two now have five and three respectively. Inspecting these states on earlier elimination steps reveals the steps where their multiplicative degree changed. State 17 increased its multiplicative degree at step one (steps start counting at zero) from two to three. This is due to the elimination of state 2, which introduced outgoing edges from state 17 to states 0 and 8. The conclusion drawn from this observation is that this order can be improved by taking the neighbours of a state into account. Specifically the effect eliminating a state has on its neighbours.

5.3 Neighbour degree

The states chosen by $\overrightarrow{Forward}$, which yield a better calculation metric, are already in the set generated by $\overrightarrow{DegreeMult}$. This set of states can be used as a starting point for more expensive computations. This is the basis for the $\overrightarrow{Heuristic}$ orders, refine a set of states with increasingly expensive computations until a single best state or set of states is left. $\overleftarrow{DegreeMult}$ is created from $\overrightarrow{DegreeMult}$ and $\overleftarrow{NeighbourDegreeC}$ is created to take the neighbours of a state into account. Together these two heuristics form $\overrightarrow{Heuristic1}$.

Table 5.3 shows that $\overrightarrow{Heuristic1}$ performs significantly better than $\overrightarrow{DegreeMult}$ and is on par with $\overrightarrow{Forward}$ for the Coupon model.

5.3.1 Observations

Initial and absorbing states

From the order calculated by $\overrightarrow{Heuristic1}$ and $\overrightarrow{Forward}$ we can observe the following: good states to eliminate can often be found near initial or absorbing states. The reason for this is

probably the unique property of those states, they either have no incoming or no outgoing states. When they are eliminated, if they are allowed to, no calculations are necessary. This is also visible from the calculation of the calculation metric. In these states one of the multiplicands is zero, any change on the other side therefore has no effect on the outcome.

Indegree

Overall the calculation metric of $\overrightarrow{Heuristic1}$ is a lot better than previous elimination orders, but there are a few models where it performs worse. Interestingly these are the smaller models, for which the order only performs a little worse. Knuth 38 calculations compared to 39, EGL 11 and 12 and the biggest difference in crowds with 217 compared to 222. The same method as before is used to inspect the points in the elimination process which could be improved.

The EGL model is inspected and Figure 5.5 shows the initial model and two options for the last elimination step. The calculation costs have again been equal until his point, where Figure 5.5b incurs a cost of one and Figure 5.5c a cost of two. This is due to the premature elimination of state four, it eventually reduced its multiplicative degree to one. Extending $\overleftarrow{NeighbourDegreeC}$ will catch this pattern but at an exponential cost for every further step looked ahead. There is another feature of state four which makes it uniquely distinguishable from the other states in this model. It has two incoming edges where the other states only have one. $\overleftarrow{InDegree}$ has been created as a result from this observation.

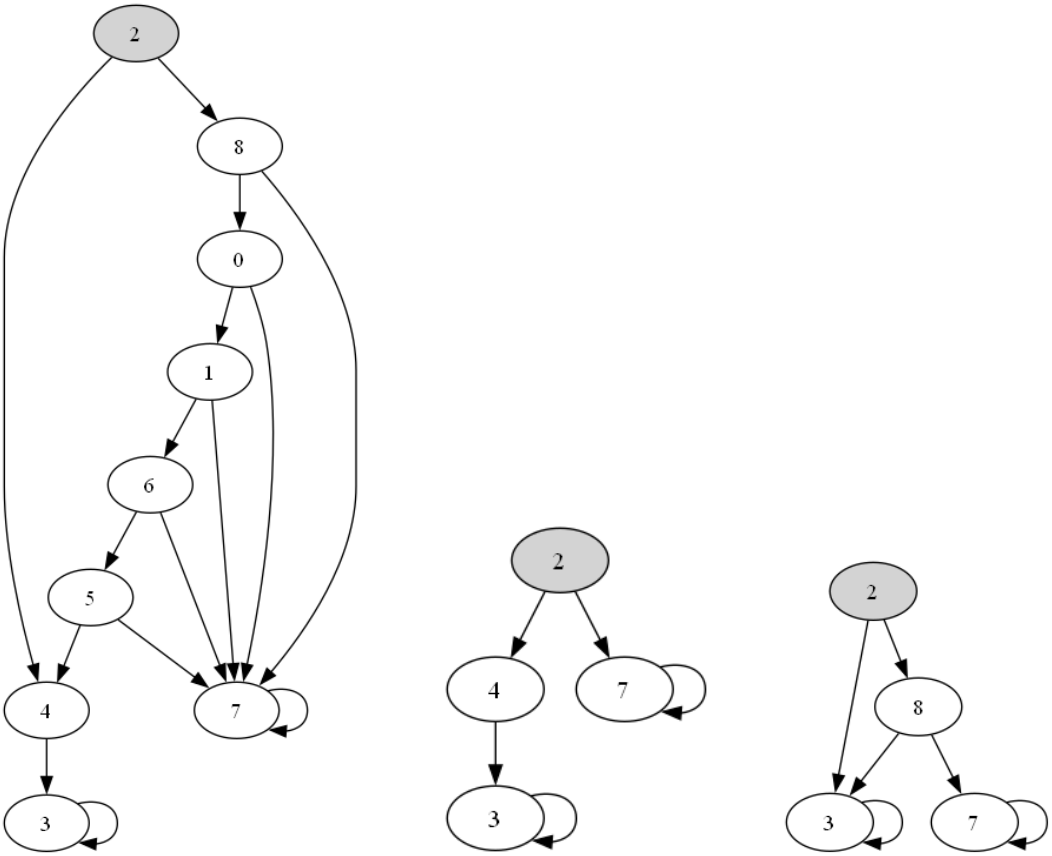
Minimum neighbour degree change

The Knuth model provides the last observation for $\overrightarrow{Heuristic1}$. At the end of the elimination process there is one extra step with a calculation cost of three, as shown in Figure 5.6. Figure 5.7 shows the where this increase takes place. States 15 and 0 are equal for $\overleftarrow{DegreeMult}$ but not for $\overleftarrow{NeighbourDegreeC}$. Because of an error when manually calculating this number they were deemed equal during inspection. Assume for now that this is indeed the case.

States 13 and 0 are the selected candidates for elimination. Both have the same multiplicative degree, neighbour degree change and indegree. State 13 was chosen by the elimination order, state 0 was therefore eliminated manually to compare the effects. Figure 5.7b shows the order $\overrightarrow{Heuristic1}$ has chosen where state 0 has been eliminated. With this option the lowest calculation cost of any state is three. Figure 5.7c shows the other option, where state 13 is eliminated. In this option there exists a state with a calculation cost of two. For both options the next two eliminations have a calculation cost of three. The last option yields the same (best) result as $\overrightarrow{DegreeMult}$ and would therefore be preferable over the first option.

The defining feature of the last option is the resulting state of the model. It has a minimum multiplicative degree of two, while the summation of all multiplicative degrees is equal to the other option. For option one the neighbouring degrees are calculated at three for state 0 and five[1] for state 3. For option two the calculation yields six for state 3 and two for state 13. Intuitively it makes sense that the option with the lower minimum multiplicative degree is the better choice. The higher degree state can be, and in this scenario is, influenced by the elimination of the lower degree state at a later elimination step. This is the cause of $\overleftarrow{MinNeighbourDegreeC}$.

[1] This is the incorrect calculation but assume it is



(A) Initial model. (B) Forward reversed. (C) Heuristic 1.

FIGURE 5.5: Elimination of EGL model at the last step.

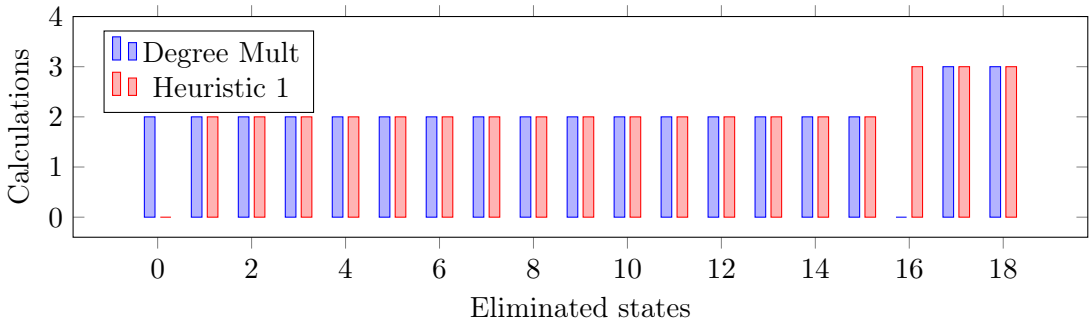
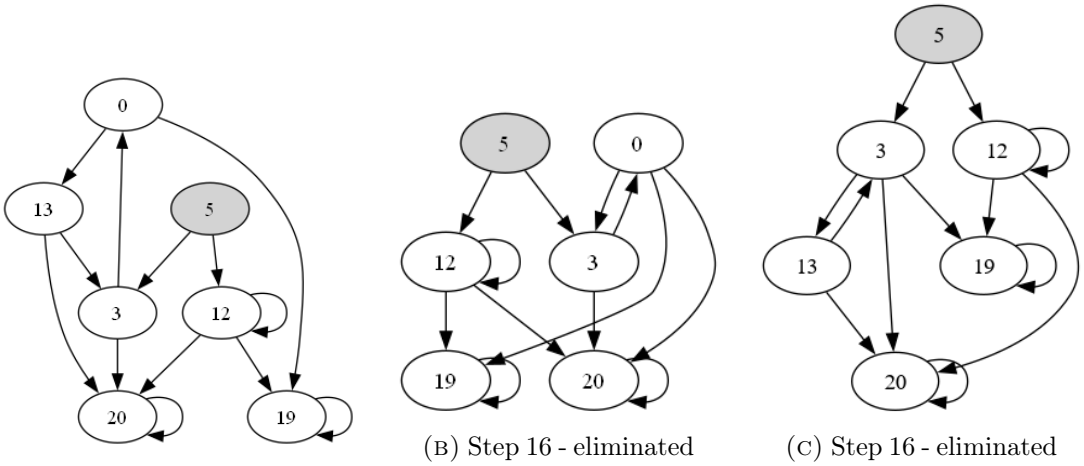


FIGURE 5.6: Bargraph of calculations per elimination step of the Knuth model.



(A) Step 15.

(B) Step 16 - eliminated state 13.

(C) Step 16 - eliminated state 0.

FIGURE 5.7: Bottleneck in the elimination process of the Knuth model with the Heuristic 1 order.

The calculation error was uncovered only after running the heuristic. The correct multiplicative degree of state 3 in Figure 5.7b is four, bringing the total degree to seven. This is lower than the other option which has a total degree of eight. This option would therefore already be disregarded by $\overleftarrow{NeighbourDegreeC}$.

5.4 Other heuristics

$\overrightarrow{Heuristic2}$ follows $\overrightarrow{Heuristic1}$ closely with an added step of $\overleftarrow{InDegree}$. Table 5.3 shows that this improves the EGL model results to the best result similar to $\overrightarrow{ForwardReversed}$. The pattern observed in Section 5.3.1 must also be present in the NAND and BRP models. These models also have an improved calculation metric, for both this is the best overall result. The BRP model is boasting a much larger improvement of 35 calculations compared to just 1 for the NAND model. Only for the Knuth and Crowds model does $\overrightarrow{Heuristic2}$ not yield the best result.

Adding $\overleftrightarrow{MinNeighbourDegreeC}$ in $\overrightarrow{Heuristic3}$ does not yield positive results. For most models the numbers stay the same but for NAND and BRP they get worse. Putting this heuristic before $\overleftarrow{NeighbourDegreeC}$ in $\overrightarrow{Heuristic4}$ does give the best result for the Knuth model however this is the only positive change. The other models stay the same or are much worse as seen again in the NAND and BRP cases.

The only model which does not have a best result with the heuristics models is now the Crowds model. Knuth does not have a best result for $\overrightarrow{Heuristic2}$ but has been investigated already. Crowds is a model which can be resized with the input parameters, crowd size `CrowdSize` and protocol runs to analyze `TotalRuns`. When running the benchmark with different input numbers an interesting result is shown. Table 5.4 shows only for the Crowds(2,5) model, which is the setup for the overall benchmark, a better calculation metric for $\overrightarrow{ForwardReversed}$.

The model has been inspected with TotalRuns = 2 and CrowdSize = 4. Figure 5.8 shows both elimination orders have the same two peaks which are followed by a drop in calculation cost. $\overrightarrow{Heuristic2}$ has a higher peak, which is most likely due to some non-optimal step taken

Model	Forward Reversed	H2
Crowds(3,5)	1539	1513
Crowds(2,5)	217	222
Crowds(2,4)	135	126
Crowds(2,3)	45	36

TABLE 5.4: Number of calculations per elimination order for the Crowds(TotalRuns, CrowdSize) model.

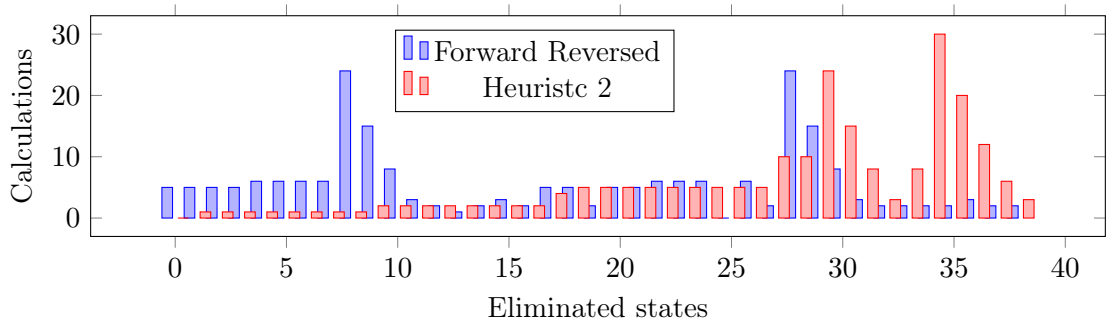


FIGURE 5.8: Bargraph of calculations per elimination step of the Coupon model.

earlier, since both peaks are at the end of the process. This model is quite a bit larger than the previous models when at this stage of the inspection. Making it smaller or enabling the Lumper will result in the elimination orders either switching place or having the same calculation metric.

The exported graph at the states did not yield a clear pattern which was causing the difference in calculation metric in the time spent on inspection. A pattern could still be present and is therefore left as future work.

5.5 Computation time

Table 5.5 shows all computation times for every elimination order against every model. Some abbreviations have been applied to model and order names. These results are only relevant relative to one another. This number depends highly on the hardware on which is it run, better hardware will result lower numbers. System load at the time of running also introduces a margin of error into these timing metrics.

The timing results show a very clear picture. The heuristics completely lose their advantage from the calculation metric. They take orders of magnitude more time to compute their orders compared to the other elimination orders. This is also seen with $\overrightarrow{Degree}$ and $\overrightarrow{DegreeMult}$ because they perform calculations at each step in comparison with for example $\overrightarrow{Forward}$.

Between $\overrightarrow{Forward}$, $\overrightarrow{ForwardReversed}$, $\overrightarrow{Backward}$ and $\overrightarrow{BackwardReversed}$ there still is a visible difference in computation time which cannot be attributed to a margin of error alone. This is due to the increased amount of calculations the elimination algorithm needs to do when an order is inefficient. Comparing these results with Table 5.1 for the NAND model shows a similar increase in calculation metric as in computation time for the $\overrightarrow{Forward}$ and $\overrightarrow{BackwardReversed}$ orders.

Model	F	FR	B	BR	SCC	CS3	D	DM	H1	H2	H3	H4
Knuth	22	15	12	10	9	2	7	5	11	6	5	6
NAND	1691	865	939	339	2935	41	724	444	3341	3056	4068	2510
Leader4-6	276	150	108	210	143	113	463	437	133s	136s	271s	239s
H-M	332	150	6	136	465	622	157	76	2260	2346	4388	4380
EGL	452	300	279	322	449	413	1268	1297	34s	36s	55s	56s
(L) EGL	6	4	0	1	6	0	4	2	4	2	3	1
Crowds	70	16	39	22	49	9	38	28	30	17	9	9
(L) Crowds	23	8	7	5	10	2	9	4	5	3	4	3
Coupon	179	78	85	55	41	9	65	92	63	27	20	15
(L) Coupon	29	32	17	17	13	5	13	12	15	3	5	7
BRP	243	206	911	842	88	9	70	70	288	214	263	290
(L) BRP	155	66	43	374	99	5	17	14	52	21	31	31

TABLE 5.5: Milliseconds (or seconds s) spent on the elimination algorithm per model per elimination order.

A clear win for $\overrightarrow{DegreeMult}$ is with the NAND model as well. It has a better calculation metric compared to the existing orders and a better computation time metric. Overall $\overrightarrow{DegreeMult}$ performs very well. It does not yield the best calculation metric but is a close competitor for most models. But more impressively it does not take a great amount of computation time to get to that result.

Chapter 6

Conclusion

In this thesis several elimination orders of the state elimination algorithm have been evaluated against one another on two metrics, calculation cost and computation time. The evaluation has been performed by implementing the elimination orders in an open source model checker called PRISM. Several models from existing research have been used to evaluate the elimination orders. The resulting data was analyzed and has been used to create several novel elimination orders. The elimination order $\overrightarrow{Heuristic2}$ has the best performance in all but two of the models, where it has been beaten by a very small margin. This indicates that an elimination order that performs best for any model does exist. This order does not perform well in the computation time metric, but this might be remediated by a more efficient implementation. The elimination order $\overrightarrow{DegreeMult}$ provides a consistent result in the calculation metric but not the best. The added computation time of calculating this order is very small. Therefore, this is the elimination order which should be chosen for the elimination process regardless of the input model.

The remainder of this chapter answers the individual research questions and discusses the possible future work on this topic.

6.1 Research Questions

RQ1.1

What state elimination orders exist?

Several elimination orders have been found, from existing research or in existing implementations. These are: $\overrightarrow{Forward}$, $\overrightarrow{ForwardReversed}$, $\overrightarrow{Backward}$, $\overrightarrow{BackwardReversed}$, $\overrightarrow{Degree}$, $\overrightarrow{DegreeMult}$ and $\overrightarrow{SimpleCycleCount}$

Several other orders have been created and implemented during this thesis. These are: $\overrightarrow{CycleSize3Count}$, $\overrightarrow{Heuristic1}$, $\overrightarrow{Heuristic2}$, $\overrightarrow{Heuristic3}$ and $\overrightarrow{Heuristic4}$.

RQ1.2

What is the difference in performance between elimination orders?

The number of calculations can vary greatly between elimination orders. Depending on the size and complexity of the graph this can range from single points in the calculation metric to factors of ten. This highlights the need for a consistent elimination order.

The calculation time is closely tied to the calculation metric and shows similar results of performance difference.

RQ1.3

Which relations exist between characteristics of MCs and the performance of an elimination order?

Several patterns have been identified in Chapter 5 which have been used to create heuristics to select the best order for those patterns.

Inspecting these patterns, the following method of ordering was determined to work best overall:

1. The state with the lowest multiplicative degree
2. The state which causes the least growth in multiplicative degree in its neighbours
3. The state with the least incoming transitions

This best overall result is $\overrightarrow{Heuristic2}$. For the models it was tested on it yielded the best calculation metric in all but 2.

RQ1.4

When does the performance gained by choosing a better elimination order overtake the computational cost of determining that order?

$\overrightarrow{DegreeMult}$ gives the best results of the computation time and calculation cost metrics together. Heuristics give a better calculation cost but currently at a computation time cost which is too high. The other orders give a computation time cost in the same order of magnitude as $\overrightarrow{DegreeMult}$ but do not give consistent calculation metric results.

RQ1

How to choose an elimination order which has a positive impact on performance?

Take $\overrightarrow{DegreeMult}$ if memory usage is less of a concern than computation time. Take $\overrightarrow{Heuristic2}$ if memory usage is more important than computation time.

6.2 Future work

6.2.1 More models

At the start of this thesis I made an assumption that plenty existing research using Markov Chains would be available. During the thesis this assumption turned out to be partially correct. There is plenty of research making use of Markov Chains however it is not interesting for this use case. Therefore, the models on which the orders have been evaluated is lower than desired. Evaluating the elimination orders on more models would be preferable to determine if there are more or other patterns which are not accounted for by these orders.

6.2.2 Unsolved pattern

The final best order in regard to the calculation metric is unfortunately not the best for each and every model. One identified pattern has no solution at the end of this thesis. More time could be spent on this pattern to identify a metric which can overcome it.

6.2.3 Efficiency

The orders created in this thesis are in some ways more of a proof of concept. Especially the heuristic orders, no time has been spent on efficiency or those orders. I have some intuitions that $\overrightarrow{NeighbourDegreeC}$ can benefit greatly by taking initial and absorbing states into account. If any substantial efficiency gains can be made the conclusion of this thesis might be adapted to $\overrightarrow{Heuristic2}$ being the best at all times.

6.2.4 Updated calculation metric

When more efficient algorithms are created the calculation metric should also be updated. This has already been done with the efficiency step of the self loop as described in Section 4.3. The combination of edges or creation is not taken into account currently. Completely new edges should be a cheaper operation since no functions have to be calculated, currently these are treated the same as when 2 edges are combined. This can be reflected in the calculation metric.

Bibliography

1. Jazwinski, A. H. en. in *Mathematics in Science and Engineering* 47–92 (Elsevier, 1970). ISBN: 978-0-12-381550-7. doi:10.1016/S0076-5392(09)60372-6.
2. Privault, N. *Understanding Markov Chains* ISBN: 978-981-4451-50-5 978-981-4451-51-2. doi:10.1007/978-981-4451-51-2 (Springer, Singapore, 2013).
3. Knuth, D. & Yao, A. in. Section: The complexity of nonuniform random number generation (Academic Press, 1976).
4. Daws, C. *Symbolic and Parametric Model Checking of Discrete-Time Markov Chains* en. in *Theoretical Aspects of Computing - ICTAC 2004* (eds Liu, Z. & Araki, K.) (Springer, Berlin, Heidelberg, 2005), 280–294. ISBN: 978-3-540-31862-0. doi:10.1007/978-3-540-31862-0_21.
5. Christel Baier & Joost-Pieter Katoen. *Principles of Model Checking* English. ISBN: 978-0-262-02649-9 (The MIT Press, Cambridge, Mass, 2008).
6. Hansson, H. & Jonsson, B. A logic for reasoning about time and reliability. en. *Formal Aspects of Computing* **6,** 512–535. ISSN: 1433-299X. doi:10.1007/BF01211866 (Sept. 1994).
7. Hopcroft, J. E., Motwani, R. & Ullman, J. D. *Introduction to automata theory, languages, and computation* 3rd ed. Pearson new international ed. English. Section: 488 pages : illustrations ; 28 cm. ISBN: 978-1-292-03905-3 (Pearson Education, Harlow, Essex, 2014).
8. *Implementation and Application of Automata: 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings* en (ed Maneth, S.) ISBN: 978-3-642-02978-3 978-3-642-02979-0. doi:10.1007/978-3-642-02979-0 (Springer, Berlin, Heidelberg, 2009).
9. Hahn, E. M., Hermanns, H. & Zhang, L. Probabilistic reachability for parametric Markov models. en. *International Journal on Software Tools for Technology Transfer* **13,** 3–19. ISSN: 1433-2779, 1433-2787. doi:10.1007/s10009-010-0146-x (Jan. 2011).
10. Moreira, N., Nabais, D. & Reis, R. State Elimination Ordering Strategies: Some Experimental Results. en. *Electronic Proceedings in Theoretical Computer Science* **31.** Publisher: Open Publishing Association, 139–148. ISSN: 2075-2180. doi:10.4204/EPTCS.31.16 (Aug. 2010).
11. Han, Y.-S. State Elimination Heuristics for Short Regular Expressions. en. *Fundamenta Informaticae* **128.** Publisher: IOS Press, 445–462. ISSN: 0169-2968. doi:10.3233/FI-2013-952 (Jan. 2013).
12. Gruber, H. & Holzer, M. From Finite Automata to Regular Expressions and Back—A Summary on Descriptional Complexity. en. *Electronic Proceedings in Theoretical Computer Science* **151.** Publisher: Open Publishing Association, 25–48. ISSN: 2075-2180. doi:10.4204/EPTCS.151.2 (May 2014).
13. McNaughton, R. & Yamada, H. Regular Expressions and State Graphs for Automata. *IRE Transactions on Electronic Computers* **EC-9.** Conference Name: IRE Transac-

tions on Electronic Computers, 39–47. ISSN: 0367-9950. doi:10.1109/TEC.1960.5221603 (Mar. 1960).
14. Lombardy, S., Régis-Gianas, Y. & Sakarovitch, J. Introducing VAUCANSON. *Theoretical Computer Science. Implementation and Application of Automata* **328,** 77–96. ISSN: 0304-3975. doi:10.1016/j.tcs.2004.07.007 (Nov. 2004).
15. Han, Y.-S. & Wood, D. Obtaining shorter regular expressions from finite-state automata. *Theoretical Computer Science* **370,** 110–120. ISSN: 0304-3975. doi:10.1016/j.tcs.2006.09.025 (Feb. 2007).
16. Delgado, M. & Morais, J. *Approximation to the Smallest Regular Expression for a Given Regular Language* en. in *Implementation and Application of Automata* (eds Domaratzki, M., Okhotin, A., Salomaa, K. & Yu, S.) (Springer, Berlin, Heidelberg, 2005), 312–314. ISBN: 978-3-540-30500-2. doi:10.1007/978-3-540-30500-2_31.
17. Pearl, J. *Heuristics: Intelligent search strategies for computer problem solving* (Addison-Wesley Pub. Co. Inc Reading, MA, Jan. 1984).
18. Hahn, E. M., Hermanns, H., Wachter, B. & Zhang, L. *PARAM: A Model Checker for Parametric Markov Models* en. in *Computer Aided Verification* ISSN: 1611-3349 (Springer, Berlin, Heidelberg, 2010), 660–664. ISBN: 978-3-642-14295-6. doi:10.1007/978-3-642-14295-6_56.
19. Johnson, D. B. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing* **4.** Publisher: Society for Industrial and Applied Mathematics, 77–84. ISSN: 0097-5397. doi:10.1137/0204007 (Mar. 1975).
20. Helmink, L., Sellink, M. P. A. & Vaandrager, F. W. *Proof-checking a data link protocol* en. in *Types for Proofs and Programs* ISSN: 1611-3349 (Springer, Berlin, Heidelberg, 1994), 127–165. ISBN: 978-3-540-48440-0. doi:10.1007/3-540-58085-9_75.
21. Herman, T. Probabilistic Self-stabilization. *Information Processing Letters* **35,** 63–67 (1990).
22. Kwiatkowska, M., Norman, G. & Parker, D. Probabilistic verification of Herman's self-stabilisation algorithm. *Form. Asp. Comput.* **24,** 661–670. ISSN: 0934-5043. doi:10.1007/s00165-012-0227-6 (July 2012).
23. Itai, A. & Rodeh, M. Symmetry breaking in distributed networks. *Information and Computation* **88,** 60–87. ISSN: 0890-5401. doi:10.1016/0890-5401(90)90004-2 (Sept. 1990).
24. Even, S., Goldreich, O. & Lempel, A. A randomized protocol for signing contracts. *Commun. ACM* **28,** 637–647. ISSN: 0001-0782. doi:10.1145/3812.3818 (June 1985).
25. Norman, G. & Shmatikov, V. Analysis of Probabilistic Contract Signing. *Journal of Computer Security* **14,** 561–589 (2006).
26. Reiter, M. K. & Rubin, A. D. Crowds: anonymity for Web transactions. *ACM Trans. Inf. Syst. Secur.* **1,** 66–92. ISSN: 1094-9224. doi:10.1145/290163.290168 (Nov. 1998).
27. Shmatikov, V. Probabilistic Model Checking of an Anonymity System. *Journal of Computer Security* **12,** 355–377 (2004).
28. Neumann, J. v. *Probabilistic logics and synthesis of reliable organisms from unreliable components* in *Automata Studies* (eds Shannon, C. & McCarthy, J.) (Princeton University Press, 1956), 43–98.
29. Norman, G., Parker, D., Kwiatkowska, M. & Shukla, S. Evaluating the Reliability of NAND Multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **24,** 1629–1637 (2005).
30. Flajolet, P., Gardy, D. & Thimonier, L. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics* **39,** 207–229. ISSN: 0166-218X. doi:10.1016/0166-218X(92)90177-C (Nov. 1992).

31. Jansen, N., Dehnert, C., Kaminski, B. L., Katoen, J.-P. & Westhofen, L. *Bounded Model Checking for Probabilistic Programs* en. in *Automated Technology for Verification and Analysis* ISSN: 1611-3349 (Springer, Cham, 2016), 68–85. ISBN: 978-3-319-46520-3. doi:10.1007/978-3-319-46520-3_5.
32. Haddad, S. & Monmege, B. Interval iteration algorithm for MDPs and IMDPs. *Theoretical Computer Science. Reachability Problems 2014: Special Issue* **735,** 111–131. ISSN: 0304-3975. doi:10.1016/j.tcs.2016.12.003 (July 2018).
33. Pahl, P. J. & Damrath, R. *Mathematical Foundations of Computational Engineering: A Handbook* en. Google-Books-ID: kvoaoWOfqd8C. ISBN: 978-3-540-67995-0 (Springer Science & Business Media, July 2001).