Bachelor's Thesis Computing Science

Efficiently calculating reachability probabilities in Markov chains

Quinten Kock
s1032724

August 28, 2023

First supervisor/assessor:
dr. Sebastian Junges

Second assessor:
dr. Nils Jansen

Radboud University

Abstract

Markov chains describe stochastic system behaviour and are suitable to describe the behaviour of a plethora of systems that are subject to randomness. Reachability probabilities reflect the probabilities that within these Markov chains, certain events such as failures occur. By answering reachability queries, we can therefore compute the probability of e.g. a packet being lost in a network protocol. Existing approaches for resolving these queries tend to suffer from scalability issues as the number of states increases. In this thesis, we propose a new method which is based on the same ideas, but requires significantly less memory by not storing a transition matrix for the Markov chain. A prototype implementation of this method achieves competitive results with existing tools like Prism, Storm and Rubicon, suggesting that matrix-free methods are worth further investigation.

Contents

Chapter 1

Introduction

Model checking Model checking is an important tool for software verification. However, many systems deal with uncertainty, including for example distributed systems (which are subject to potentially unreliable networks), or any system that relies on randomness. One example of such a system is the Ethernet protocol, which relies on randomized back-off to deal with collisions [14]. Because of this, it is often impossible to fully guarantee that a system is absolutely always behaving as desired, meaning that classic model checkers are not able to meaningfully verify at least some properties (such as availability) of such a system.

Probabilistic model checking However, we still expect systems like Ethernet to be reliable. For example, packet loss due to collisions should be extremely rare on most network configurations, even if it is not possible to fully avoid it. To verify that this is indeed the case, we can use probabilistic model checking to determine the probability of e.g. losing a packet when n devices all want to transmit simultaneously. Probabilistic model checking works on Markov chains which represent the system [7].

Problem One big concern for probabilistic model checking is scalability: the ability for model checking tools to deal with very large state spaces. Conventional probabilistic model checking tools such as Storm [8] store the full transition matrix for the Markov chain that is being checked, which has a high cost, both in terms of memory and CPU time. Other tools such as Rubicon use a different approach based on paths through the Markov chain, which can be more efficient depending on the problem.

Our solution In this thesis, we propose an alternative method that is similar to conventional probabilistic model checkers, but based on computation. Instead of storing the transition matrix in memory, we calculate its values

on-the-fly using a *successor function*. This has the potential to greatly reduce memory consumption, and in a system with limited memory bandwidth also has the potential to be faster.

Outline In Chapter 2, we give precise definitions of Markov chains and the reachability problems solved by probabilistic model checking tools. Then, in Chapter 3 we discuss two approaches that we implemented, which we compare against existing work in Chapter 4. In Chapter 5 we compare a prototype implementation of our ideas with existing probabilistic model checkers, and finally in Chapter 6 we conclude with future work.

Chapter 2

Preliminaries

2.1 Markov chains

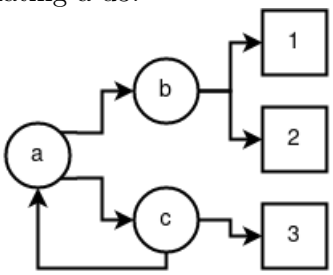
A Markov chain is a way of modeling a stochastic process.

Definition 2.1 (Markov chain) *A Markov chain is a tuple $\mathcal{M} = (S, P, \iota_0)$, where S is a set of states, $P : S \times S \to [0, 1]$ is the transition probability between two states such that for all states $s \in S$ it holds that $\sum_{s' \in S} P(s, s') = 1$, and ι_0 represents the initial probability distribution such that $\sum_{s \in S} \iota_0(s) = 1$ [2].*

Transition probability The transition probability $P(s, s')$ is the chance that, if we are currently in state s, that in the next step we will be in state s'.

Transition matrix P is often represented as a matrix (a transition matrix), such that multiplying it with a probability distribution yields the probability distribution in the next time step.

Visual representation Markov chains can be represented visually in a directed graph, by using vertices for states and edges for transitions. A good illustrative example is the Knuth-Yao die [11], which simulates a fair die with only fair coin flips. The illustration below shows a simplified version simulating a d3:



2.1.1 Absorbing Markov chains

A state which cannot be left after it is entered is called an absorbing state. If every state can reach an absorbing state, then the entire Markov chain is absorbing. The dice example above is absorbing, as states a to c all enable us to reach the absorbing states represented by 1 to 3. In an absorbing Markov chain, the probability of being in an absorbed state approaches (or even becomes) 1 as time passes, since the other states all have a nonzero probability of going to one of the absorbing states.

2.1.2 Higher-level Markov chains

Semantic states Markov chains as introduced here do not give any information about states $s \in S$. As such, we could assign each state a number, making $S = \{1, 2, \ldots, |S|\}$. However, in many cases, each state carries some sort of semantic meaning. It can therefore be useful to define states not as simple numbers, but instead to define a set of variables V, with each state being a mapping $V \to Z$. For example, when representing a board game as a Markov chain one might take $V = \{Turn, Player1, Player2\}$ where the variables represent whose turn it is and where each player is located on the board.

Higher-level successor function P is often dependent on the values of these variables. Continuing on the board game example, if in a given game a player can only move if it is their turn, then if we have two states s and s', where $s(Turn) = 1$ and $s(Player2) \neq s'(Player2)$, we can determine that $P(s, s') = 0$ (depending on the game). This allows us to give P in the form of a program.

Markov chain description languages There exist languages that are intended to describe Markov chains in such a way, for example the PRISM language, which is associated with the PRISM model checker [12] but also supported by other model checkers such as Storm [8]. In this language, one defines modules which contain variables (which are bounded integers or booleans), and *commands*, which consist of a *guard* (condition, e.g. `Turn = 1`) and one or more *updates* (combinations of probabilities and changed variables, e.g. `0.25 : Player1' = Player1 + 4` to have a 25 percent chance to move player 1 ahead by 4 squares).

2.2 Reachability probability

The reachability probability in a Markov chain is the chance that a certain state or set of states is reached, either within a number of time steps or at all. This is essential for verifying properties about a probabilistic process, such

as the aforementioned dice model. In this thesis, we focus on two variations of the reachability probability problem.

2.2.1 Indefinite horizon

The reachability probability with an indefinite horizon is the probability that a state is ever reached. In the die example, this is useful to determine that the die is fair: 1 through 3 should all have a probability of exactly $\frac{1}{3}$. We can represent an indefinite horizon reachability problem as follows:

Definition 2.2 (Indefinite horizon reachability problem) *An indefinite horizon reachability problem can be described as a tuple $(\mathcal{M}, T)$ where $\mathcal{M}$ represents a Markov chain and $T \subseteq S$ represents the set of target states. A solution to the problem consists of a probability p that we ever reach one of the states in T.*

Note that our definition uses a set of states instead of a single state, which is not strictly necessary, since if we wish to generalize to multiple target states we can also change the Markov chain to accomplish this. For this, we could take $S' = S \setminus T \cup \{t\}$ and $P' = P$ except that $\forall s \in S$ we set $P'(s,t) = \sum_{s' \in T} P(s,s')$. Similarly, $\iota_0' = \iota_0$ except $\iota_0'(t) = \sum_{s \in T} \iota_0(s)$.

2.2.2 Fixed horizon

The reachability with a fixed horizon is the probability that a state is reached after n timesteps. This is useful for time-sensitive properties. For example it can be used to determine the probability that the die model terminates within 5 coinflips.

Definition 2.3 (Fixed horizon reachability problem) *A fixed horizon reachability problem consists of a tuple $(\mathcal{M}, T, n)$ where $\langle I, \lambda, P \rangle$ represent a Markov chain, $T \subseteq S$ represents the set of target states, and n represents the horizon. A solution consists of the probability p that we are in one of the states in T after n timesteps.*

Definition 2.4 (Bounded horizon reachability problem) *A bounded horizon reachability problem consists of a tuple $(\mathcal{M}, T, l, u)$ where $\mathcal{M}$ is a Markov chain, $T \subseteq S$ is the set of target states, l is the lower bound, and u is the upper bound. A solution consists of the probability p that we are in one of the states in T after at least l but no more than u timesteps.*

Note that the bounded horizon problem generalizes both the fixed horizon problem as well as the indefinite horizon one. For the first we need to set $l = u$, and for the latter we need $l = 0$ and $u = \infty$.

Chapter 3

Approaches

This thesis focuses on two approaches to calculate the reachability probability in a discrete time Markov chain. We describe these approaches in this chapter.

3.1 Forward iteration

Idea Forward iteration is an approach that is very suitable for fixed-horizon problems. It works by recursively calculating probability distributions. Say that we need to calculate the probability of being in some state s after exactly five timesteps in a Markov chain with an initial distribution ι_0 and a transition matrix P. We can then calculate $iota_1$, the probability distribution after one step, by calculating $\iota_1 = P\iota_0$ (and in general, $\iota_{t+1} = P\iota_t$). The probability of being in state s after five timesteps can then be looked up in ι_5 by taking the value of the corresponding element, as described in Algorithm 1. This is then equivalent to the mass of all paths where the 5th state is s, because Markov chains are not dependent on previous states.

Algorithm 1 Forward iteration method

```
1:  procedure FORWARD(ι_0, T, n)          ▷ n is the number of timesteps
2:      x ← ι_0                           ▷ initialize with initial distribution
3:      for i ← 1, n do
4:          x ← ITERATE(x)                ▷ equivalent to x ← Px, see Algorithm 2
5:      end for
6:      p ← 0
7:      for all s ← T do                  ▷ for every target state
8:          p ← p + x_s                   ▷ add its probability to p
9:      end for
10:     return p
11: end procedure
```

Upper bounds Should we wish to calculate the probability of reaching s after u timesteps or earlier, all that needs to be done is to make the target state absorbing: when in state s, the only possible successor is s. We can then take $n = u$. This works because if we have a Markov chain M and apply the described transformation to get another Markov chain M', and we have a path through M where state s is first reached after $i \leq u$ transitions, then following the same path through M' for the first i transitions will bring us to state s as well, and because it is absorbing we will still be in state s at time n.

Lower bounds In addition, calculating the probability of reaching s after at least l timesteps is also possible. For this, we begin by doing l steps in Markov chain M, and then using the obtained distribution ι_l to continue using the probability function from M'.

Multiple target states One advantage of this approach is that if we have multiple target states, we can look up their values individually without doing the iteration multiple times. This means it is possible to compare the probabilities for various outcomes, which is useful e.g. if the goal is to determine fairness.

Reach-avoid properties Furthermore, if states need to be avoided in the path leading up to a target state, it is possible to exclude these paths from the solution by making these so-called 'bad states' absorbing. Then, if we have an original Markov chain M and one with absorbing bad states M_B, and a path p through M such that $p_n = s$ and $p_i = b$ where $i < n$, the path p^b is identical to p until i, after which it diverges because it stays in b and as such will not reach s at time n.

Implementation choices There are many possible implementations of this approach. For example, it is possible to calculate and store the transition matrix (e.g. as a sparse matrix or MTBDD [4]) and do standard matrix-vector multiplication. However, doing this is very memory intensive due to needing to store the entire matrix. Hence, we have chosen to try a novel approach where we make use of a successor function in the form of a program, taking in a state and returning a list of possible successor states, essentially calculating the values in the matrix P on-the-fly.

Variable mapping Doing this does require some way of finding the variables corresponding to a state. This can be done by using a hashmap or similar associative data structure, although this does require storing detailed states in memory. It is also possible to use an invertible function to map between integers and 'full' states, which prevents us from needing to store

each possible state in detail. We have chosen to do the latter, by creating functions stateToIndex and indexToState such that if we have integer variables $a \in [0 \,..\, A), b \in [0 \,..\, B)$ we associate these with the unique integer $s = aB + b$. We can then obtain back $a = \lfloor s/B \rfloor$ and $b = s \mod B$. We can then use the integer s as an index into a vector which represents the probability distribution.

Algorithm 2 Single iteration

```
1:  procedure ITERATE(ι_t)
2:      ι_{t+1} ← 0⃗                          ▷ initialize result vector with zeroes
3:      for i ← 1, n do                        ▷ n is the size of the vector
4:          if ι_{t,i} = 0 then
5:              continue                       ▷ if ι_{t,i} is 0 then line 10 has no effect
6:          end if
7:          state ← INDEXTOSTATE(i)
8:          for all (s, P) ← SUCCESSORS(state) do
9:              j ← STATETOINDEX(s)
10:             ι_{t+1,j} ← ι_{t+1,j} + P · ι_{t,i}
11:         end for
12:     end for
13:     return ι_{t+1}
14: end procedure
```

3.1.1 Approximating infinite horizon problems

For absorbing Markov chains that terminate but do not have a fixed horizon, it is possible to use this method to approximate the probability of terminating by iterating until the chance of absorption is sufficiently high (e.g. 99.99%). Each outcome is then at least as likely as it is in the final obtained distribution, and can only be a small amount higher (for example, 0.01% when terminating at 99.99%).

Error When we terminate at 99.99%, we can be sure that if we found a probability p so far, the true probability is somewhere between p and $p + 0.01\%$. As such we can minimize the potential error by guessing the value $p + 0.005\%$, as the error in either direction can be at most $\epsilon = 0.005\%$ in that case. However, in practice it is likely that $\frac{p}{99.99\%}$ will be a closer approximation as the remaining paths will most likely have a similar distribution as the absorbed paths.

Implementing this method To implement this method, we need to adjust Algorithm 1. We remove the parameter n and instead pass the desired ϵ. On line 3 we can then change the loop to a **while** which terminates if

the absorption probability $p_a = \sum_{a \leftarrow A} x_a$ (where A is the set of absorbing states) is larger than or equal to $100\% - 2\epsilon$. Additionally, as described in the previous paragraph, it is possible to return a closer approximation than p, such as $p + 0.5(1 - p_a)$ or $\frac{p}{p_a}$ on line 10.

Probability bounds As this approach gives us probability bounds at every iteration $[p \downarrow, p \uparrow] = [p, p + (1 - p_a)]$, this approach is quite suitable for verifying properties that themselves involve bounds (like "there is at least a 90% chance of X" or "there is at most a 30% chance of Y"), as the termination condition can be changed to the correct bound. In general, if the property says that the probability should be between (or outside of) the range $[b \downarrow, b \uparrow]$ then we can terminate if $[p \downarrow, p \uparrow]$ is either entirely inside or entirely outside of $[b \downarrow, b \uparrow]$, as we are then know for sure if for our true probability p_t it holds that $p_t \in [b \downarrow, b \uparrow]$.

3.2 Linear system solving

Idea Linear system solving is an approach that is suitable for indefinite-horizon problems. It works by creating a linear system of equations that describe the probability of reaching the target state. For example, if we have a target state s_t we know that the probability of reaching it from itself P_t equals 1. If we have another state s_a with successors s_b and s_c, each with probability 0.5, we know that $P_a = 0.5P_b + 0.5P_c$.

Matrix generation By enumerating the possible states, we can then generate a linear system of the form $Ax = b$, and use any linear system solver to calculate x, which then represents our probabilities. For a Markov chain as described in the example above, this system would look something like

$$\begin{bmatrix} -1 & 0.5 & 0.5 & 0 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_a \\ P_b \\ P_c \\ P_t \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

To generate this matrix, it is necessary to create a mapping from states to indices, so that we know what column in A corresponds to a state.

Implementation choices Similarly to the forward iteration method, it is possible to store this matrix in some form. We can then use a traditional linear system solver (either direct or iterative) to obtain the values for P. Just like with forward iteration, we have chosen to forego storing the matrix, instead calculating its values on the fly using a successor function. This does restrict us to only using iterative solvers, as we cannot directly manipulate

the values in the matrix anymore. We have chosen to use the EIGEN [6] library which natively supports this approach, called matrix-free solving. To do this, one has to create a custom matrix type and implement a C++ function that performs matrix-vector multiplication for it, which we have done as in Algorithm 3. From the iterative solvers available in EIGEN, we have chosen to use BiCGSTAB [16]. We did not evaluate in detail other available solvers such as IDR(s) [17].

Algorithm 3 Pseudo matrix-vector multiplication

```
1: procedure MATVEC(rhs: Vector⟨n⟩)
2:     result ← 0⃗
3:     for i ← 1, n do
4:         state ← INDEXTOSTATE(i)
5:         for all (s, P) ← SUCCESSORS(state) do
6:             j ← STATETOINDEX(s)
7:             result_i ← result_i + P· rhs_j
8:         end for
9:         if state is not absorbing then
10:            result_i ← result_i− rhs_i
11:        end if
12:    end for
13: end procedure
```

Related work Using linear system solving to calculate reachability probabilities is not new, and has been done before, for example in an analysis of the Game of the Goose [5], and is also used in STORM [8]. However, these use explicit representations for the matrix, while in our implementation we have chosen to not directly represent the matrix, and use a matrix-free approach instead.

Chapter 4

Related Work

4.1 Storm

Overview Storm [8] is a probabilistic model checker that represents the current 'state-of-the-art'. It is implemented in a modular fashion, which means that it can use different approaches and libraries depending on the problem at hand. For example, Storm provides different *engines*. However, most of these either require storing the model in some form (e.g. as a matrix or BDD). Its other engines such as `expl` or `abs` are highly limited in the properties they are able to check.

Difference In contrast, this thesis focuses on ways to avoid storing the model numerically at all, and contains methods centered around a successor function instead, which give our methods the potential to have large reductions in memory usage compared to any of the engines available in Storm. For example, Storm also supports linear system solving, but not doing it in a matrix-free manner.

4.2 Rubicon

Overview Rubicon [10] represents an alternative method of probabilistic model checking, which uses the probabilistic inference engine Dice [9]. This approach is currently limited to fixed-horizon reachability problems. The way it works is by building up a BDD that represents the paths through a Markov chain, and then calculating a weighted sum.

Differences This is a completely different approach to what is outlined in this thesis, but provides an interesting comparison, because Rubicon does not directly store a numerical representation of the model either, instead choosing to represent paths. As such it provides another way to avoid storing the transition matrix.

4.3 Sound Value Iteration

Overview Sound value iteration [15] is one of the possible approaches to iteration as done in probabilistic model checkers. It is supported in Storm [8], and, like other traditional probabilistic model checking techniques [2], works backwards. However, like the forward iteration method described in this thesis, it has reliable lower and upper bounds. As such it is quite similar.

Differences Of course, the main difference is the iteration direction. This difference has consequences in some areas, for example in our iteration step described in Algorithm 2 the optimization in line 4 is only possible with a forward approach, as a backward method (like in Algorithm 3) does not allow us to know in advance if the weight will be zero. Furthermore, our approach only keeps track of a single vector, while sound value iteration keeps track of two vectors, one representing the lower bound and the other representing the difference between the upper and lower bound. On the other hand, sound value iteration supports Markov decision processes, and we do not know if it is possible to extend forward iteration to work with this kind of model.

Chapter 5

Experimental results

In this chapter we will compare a prototype of our described approaches with some existing probabilistic model checking tools on two problems we believe are reasonably representative.

5.1 Experimental setup

We will be comparing our prototype (written in C++ and using the Eigen library [6]) with the following existing model checking tools:

- Storm 1.8.0 (as available on Docker Hub)
- Prism 4.8
- A custom build of Rubicon in combination with Dice as available on Docker Hub

All testing was performed on a desktop computer with an 8-core 16-thread AMD Ryzen processor and 32 GiB of system memory running Linux. We note that Docker containers were run in Podman, which should not affect performance. For measuring total run-time, we used Hyperfine [13]. This tool runs a benchmark multiple times and outputs the mean used time (which is more accurate than a single run), and also outputs CPU time (which can, for example, be used to determine if there is significant use of multithreading). Other run-times are extracted from standard output, which is less precise but gives a global indication of how a tool like Storm spends its time. Memory measurements are done by looking at the peak resident set size of the process, which should be reliable as long as swap is not used by the measured process. Source code and results are also available at `https://github.com/ColonelPhantom/thesis-prototype`.

5.2 Problem A: Factories

The Factories example was introduced in the RUBICON paper [10] as a motivating example. It involves a probability matrix that is fully dense (which means storing it sparsely does not save any memory) and highly irregular (meaning that techniques based on decision diagrams are also not effective). Because our approaches do not store the matrix, we are hopeful that our approach performs reasonably well by avoiding the memory bottleneck.

5.2.1 Problem description

The model consists of n factories, that have two possible states: striking or working. Furthermore, each factory has a unique probability p_i to start striking, as well as a probability q_i to go back to work. Because every factory can go to either state each day, all states can reach all other states which makes the transition matrix fully dense. The irregularity is caused by all the values for p_i and q_i being different. The property for this model we will be investigating is "what is the probability that all factories are striking at the same time, within h days" (given as `P=? [ F <= h allStrike ]`).

5.2.2 Baseline results

As a baseline, we have chosen to use a model with $f = 12$ and $h = 10$. We can then evaluate how different tools perform with this configuration and how they scale with varying the f and h parameters. Here are the results for the various tools:

Tool	Wall time	Memory
STORM (dd)	Error (`BDD Unique table full`)	
STORM (hybrid)		
STORM (sparse)	5.2 seconds	1020 MB
PRISM (sparse, `-cuddmaxmem 4g`)	49.9 seconds	2403 MB
RUBICON	0.5 seconds	141 MB
Prototype (forward iteration)	0.5 seconds	4 MB

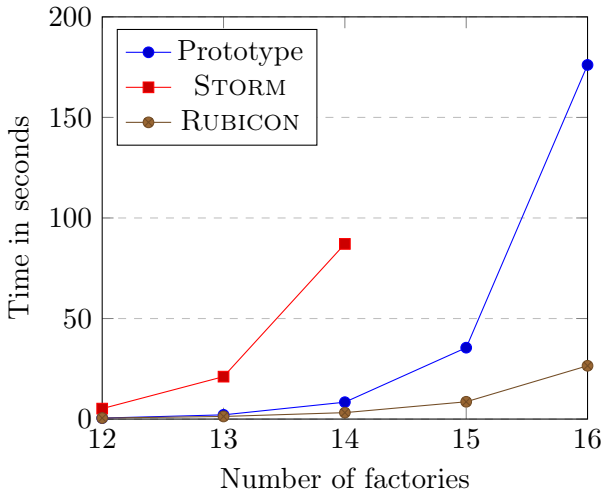
We note that our prototype has the model compiled into it. A practical implementation would need to compile the model to machine code at runtime, which would introduce additional overhead. Compiling the prototype takes around 4 seconds on this machine, but we believe most of that time is spent compiling code from the Eigen dependency. Additionally we did not include the time it takes for Rubicon to generate the Dice program. We have decided to do a more in-depth comparison between our prototype, STORM's sparse engine, and RUBICON, to observe how these different approaches scale with the f and h parameters.

5.2.3 Varying model size

We will now analyze how our implementation scales with models where f varies, while keeping a fixed $h = 10$. f affects the number of states ($= 2^f$) and transitions ($= (2^f)^2$). With $f = 15$, STORM ran out of memory on this computer during the model checking phase, despite the presence of a sizable swap partition. It was killed by the operating system and as such there are no results. By contrast, the memory usage of our prototype did not exceed 5 MB.

	STORM				RUBICON		Prototype
f	Construct	Check	Total	Memory	Time	Memory	Time
12	4.7 s	0.6 s	5.2 s	1020 MB	0.5 s	142 MB	0.5 s
13	18.6 s	2.4 s	21.1 s	3900 MB	1.3 s	272 MB	2.1 s
14	77.1 s	9.6 s	87.1 s	15405 MB	3.2 s	511 MB	8.4 s
15	321.0 s	nan s	nan s	$>$ 32000 MB	8.6 s	766 MB	35.5 s
16	nan s	nan s	nan s	nan MB	26.5 s	1906 MB	176.1 s

Speed for model checking factories problem with $h = 10$

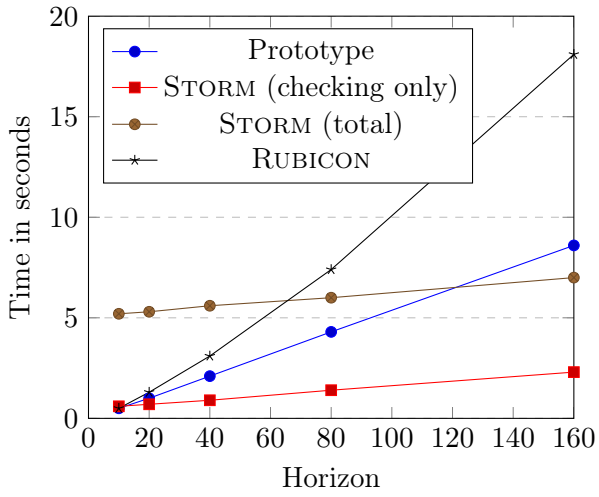


5.2.4 Varying horizon size - small model

Keeping $f = 12$ as in the baseline but increasing the horizon to higher values, the memory usage stays the same for STORM and our prototype stay but computation time increases due to needing more iterations. RUBICON does need more memory because the paths it needs to represent become bigger. We have tried values $h \in \{10, 20, 40, 80, 160\}$.

	Storm			Rubicon		Prototype
h	Construction	Checking	Total time	Time	Memory	Time
10	4.7 s	0.6 s	5.2 s	0.5 s	141MB	0.5 s
20	4.5 s	0.7 s	5.3 s	1.3 s	289MB	1.0 s
40	4.7 s	0.9 s	5.6 s	3.1 s	510MB	2.1 s
80	4.6 s	1.4 s	6 s	7.4 s	781MB	4.3 s
160	4.5 s	2.3 s	7 s	18.1 s	1557MB	8.6 s

Speed for model checking factories problem with $f = 12$



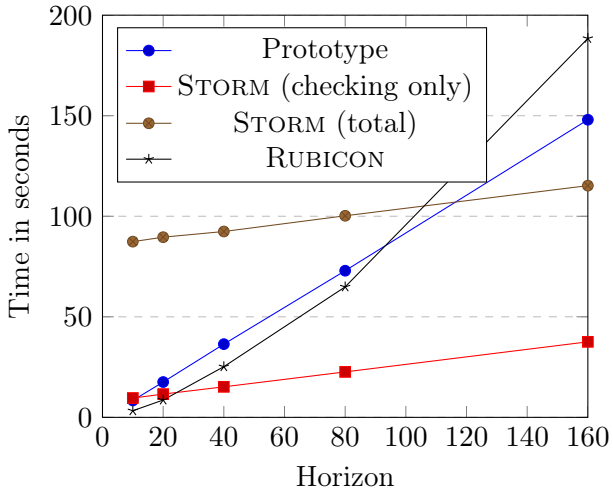
We observe that our prototype slows down more than Storm when adding more iterations, being slower than Storm's sparse engine in the case where horizon $h = 160$ despite the overhead of model construction. In addition, Rubicon sees by far the biggest negative impact, being the slowest option above $h = 80$ and also being the only tool for which memory consumption grows with h.

5.2.5 Varying horizon size - bigger model

We also tried the above test, but with f increased to 14, as this is the biggest model that Storm can evaluate on our test machine.

	Storm			Rubicon		Prototype
h	Construction	Checking	Total time	Time	Memory	Time
10	77.0 s	9.6 s	87.4 s	3.3 s	510MB	8.4 s
20	78.5 s	11.6 s	89.6 s	8.6 s	933MB	17.6 s
40	77.1 s	15.2 s	92.4 s	25.2 s	2023MB	36.4 s
80	77.1 s	22.6 s	100.2 s	64.9 s	3726MB	72.9 s
160	78.9 s	37.6 s	115.2 s	188.4 s	7669MB	148.0 s

Speed for model checking factories problem with $f = 14$



We observe that RUBICON still scales the worst of all tools with horizon size, but it compares less unfavourably than with $f = 12$. At $h = 80$ it is now the fastest tool, beating our prototype. In addition, even at $h = 160$ its memory usage (around 8 GB) is now below STORM (which uses nearly 16 GB).

5.2.6 Discussion

We have observed that STORM uses a very large amount of memory for this problem, severely limiting the maximum model size for this problem. In addition, its model construction phase also takes a large amount of time. However, STORM does proceed through model checking quite quickly, making it a suitable tool for problems with a horizon that is far away. RUBICON is quite fast with nearby horizons, but still uses significant amounts of memory.

For this problem, our implementation therefore has a strong benefit in that it uses only a miniscule amount of memory. Runtime is also competitive, being generally worse than RUBICON for small horizons and worse than STORM for big horizons, but also being the fastest in at least some cases (such as $f = 12, h = 40$).

5.3 Problem B: Snakes & Ladders - Termination

Snakes & Ladders is a widely-known board game, for which the amount of turns has been analyzed before [1]. With probabilistic model checking, it is also possible to analyze games like this, calculating the probability of the game being finished after a certain number of turns. Compared to the Factories problem, storing the transition matrix for this problem is quite easy, since there are at most six outgoing transitions per state (meaning it is

very sparse) and only a small amount of possible values, most being either 0 or $\frac{1}{6}$.

5.3.1 Problem description

There is a board with n players, which all start on square 0. The goal is to reach square 100, the game ends as soon as one player does so, as that player then wins. Players take turns, and when it is their turn they roll a 6-sided die. After this, they move ahead by the number of squares indicated. If they land at the bottom of a ladder after moving, they can climb it, which brings them to a square closer to 100. If they land at the head of a snake, they slide down it, which brings them to a square closer to 0. If moving would bring the player to a square beyond 100, they do not move. For this section, we calculate the probability of the game ending in h turns: `P=? [ F<=h finished ]`.

5.3.2 Baseline results

As a baseline, we have chosen the parameters $n = 3, h = 80$.

Tool	Wall time	CPU time	Memory
Storm (dd)	137.2 s	2024.2 s	3884 MB
Storm (hybrid)	4.6 s	71.4 s	2938 MB
Storm (sparse)	8.3 s	8.2 s	652 MB
Prism (hybrid)	1.9 s	2.4 s	176 MB
Rubicon	Out of memory		$>$ 32000 MB
Prototype (forward iteration)	1.2 s	1.2 s	28 MB

Prism We observe that Prism performs very well in this benchmark, being faster than Storm. Storm also puts a much heavier load on the CPU, making use of all 16 hardware threads for the hybrid and dd engines.

Rubicon Rubicon runs out of memory, likely due to the relatively high value for h, which we have observed to increase its memory requirements for the Factories problem.

Prototype Our prototype performs well for this problem, although we do note that the hand-written successor function is quite well optimized. It is an open question whether or not an algorithmically generated function (based on the Prism program we used for the other tools) would perform equally well. In addition, the memory usage in the prototype is based on a close-to-optimal index $\leftrightarrow$ state mapping, where impossible values for variables are eliminated. A more naive mapping could result in a higher memory usage.

In-depth In the coming subsections, we will compare our prototype with PRISM and STORM in more detail, to compare their scaling with n and h.

5.3.3 Extra player

Keeping $h = 80$ but increasing n to 4 increases memory usage significantly. PRISM and our prototype now use similar amounts of memory, and STORM's sparse engine runs out of memory. The hybrid engine does not, but does get close to the system's memory limit.

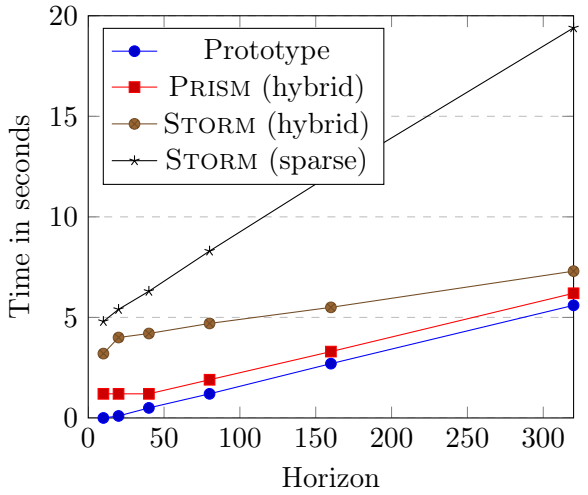
Tool	Wall time	CPU time	Memory
STORM (sparse)	Out of memory		$>$ 32000 MB
STORM (hybrid)	423.8 s	6763 s	30424 MB
PRISM (hybrid)	212.5 s	212.6 s	2651 MB
Prototype (forward iteration)	109.9 s	104.4 s	2693 MB

5.3.4 Varying horizon

For $n = 3$ we ran some more tests, trying $h \in \{10, 20, 40, 80, 160, 320\}$. Memory usage did not change from the baseline, so it is not included here.

h	STORM (sparse)	STORM (hybrid)	PRISM (hybrid)	Prototype
10	4.8 s	3.2 s	1.2 s	0.0 s
20	5.4 s	4.0 s	1.2 s	0.1 s
40	6.3 s	4.2 s	1.2 s	0.5 s
80	8.3 s	4.7 s	1.9 s	1.2 s
160	12.1 s	5.5 s	3.3 s	2.7 s
320	19.4 s	7.3 s	6.2 s	5.6 s

Speed for model checking Snakes & Ladders problem with $n = 3$



All tools scale acceptably, although STORM's sparse engine scales somewhat worse than the others, while its hybrid engine scales slightly better.

5.4 Problem C: Snakes & Ladders - Winning

With the same Snakes & Ladders game, we can also investigate the winning probabilities of various players. For this we will be using the property `P=? [ F "win0" ]`, which is the probability of the first player winning.

5.4.1 Baseline results

For the baseline, we have picked $n = 3$, and set the tools to iterate until a relative error of $1e-6$, and for forward iteration an absolute error of $1e-6$ due to not knowing what

Tool	Wall time	CPU time	Memory
STORM (dd)	610 s	8960 s	3884 MB
STORM (hybrid)	5.3 s	82.7 s	3824 MB
STORM (sparse)	6.1 s	6.0 s	1518 MB
PRISM (hybrid) `-gs`	6.2 s	6.6 s	156 MB
Prototype (linear system solving)	11.5 s	11.5 s	72 MB
Prototype (forward iteration)	6.3 s	6.3 s	28 MB

We observe that our prototype has the upper hand in memory usage, and is competitive in time usage. Forward iteration also performs very well, despite not being ideal for indefinite horizon problems like this.

5.4.2 Snakes & Snakes

It is possible to reverse all the ladders in the game to turn them into snakes. This has the effect of making all approaches take more iterations, by slowing down their convergence. A good linear system solver has the upper hand here as its number of iterations is not determined by the amount of timesteps until absorption like with an iterative approach.

Tool	Wall time	CPU time	Memory
STORM (hybrid)	6.6 s	103.5 s	3992 MB
STORM (sparse)	7.4 s	7.4 s	1729 MB
PRISM (hybrid) `-gs`	204 s	204 s	169 MB
Prototype (linear system solving)	13.8 s	13.8 s	72 MB
Prototype (forward iteration)	320.6 s	320.0 s	28 MB

We can see that PRISM and forward iteration both suffer massively from

this adjustment, taking multiple minutes whereas STORM and the solver-based prototype take just a few more seconds than for the regular game.

5.5 Discussion

In the Factories problem, we have observed a great memory advantage for our prototype, reducing memory usage by sometimes more than a thousand times compared to STORM and RUBICON, while remaining competitive in run-time. For Snakes & Ladders, the memory benefit is less pronounced, as PRISM also achieves a reasonable memory usage. STORM and RUBICON do however use much more memory. Time usage is once again competitive with other tools. These good results were however achieved with a reasonably optimized successor function, so it remains an open question if an automatically translated function will also have competitive run-time.

Chapter 6

Conclusions

In this thesis we have introduced a new approach to probabilistic model checking using matrix-free approaches, relying on on-the-fly computation of successor states instead of storing the transition matrix in some way. We also introduced an approach we call forward iteration, as opposed to the backward iteration usually employed by model checking tools. Finally, we implemented and benchmarked a prototype that uses such matrix-free versions of both forward iteration and linear system solving. We have observed an acceptable run-time, as well as significant reductions in memory usage, which indicates that it can be worthwhile to use a computation-oriented approach like this.

6.1 Future work

Prototype to production For this thesis, only a prototype has been implemented. In order to more fairly and more comprehensively compare the approaches outlined in this thesis with other methods, it would be necessary to implement them either as part of an existing model checker (e.g. as an engine in Storm [8]), or by writing the necessary parts of one (for example parsing models in an existing format such as JANI [3]). An important part of this would be translating the model to executable code automatically.

Multithreading The prototype implementation only runs on a single CPU thread. It should be possible to create a significant speedup, given today's near-ubiquitous many-core processors. The main loop of the simulated matrix-vector multiplication for the linear system method should trivially parallelize, and the forward iteration method would require some use of atomics or locking to safely update the result vector but should still be quite easy to parallelize. Because parallelization is done over states, of which there are often very many, it could be interesting to look into GPU acceleration using tools such as CUDA, OpenCL or SYCL, as GPUs excel at such massively parallel problems.

Asynchronous forward iteration The forward iteration method currently iterates all possible states at the same time. An asynchronous variant that instead only moves forward (some of) the most likely state(s) could be more efficient for the indefinite horizon case, as it means less work is done for unlikely states.

Parallel chain isolation Markov chains often work in parallel. In the given factories example, each factory is introduced as a separate 2-state Markov chain. The final problem is then taken by combining these into one big state machine, taking a product of the various states. Similarly, in the Snakes and Ladders problem, it is possible to view each player as a separate Markov chain.

However, as these examples do not have interaction between different actors (Markov chains), it should suffice to model each actor separately, at least for the forward iteration method. This should lead us to a model space that is only $|A| + |B| + |C|$ states large, rather than $|A| \cdot |B| \cdot |C|$.

Real-world problems aren't so nice, and tend to have interaction between different actors. This happens for example in more complex board games, like Game of the Goose, which is fully probabilistic just like Snakes and Ladders, but where players can interact in various ways. However, using the forward iteration method, we believe that it may be possible to keep some separation in some cases.

Bibliography

[1] S. C. Althoen, L. King, and K. Schilling. How long is a game of snakes and ladders? *The Mathematical Gazette*, 77(478):71–76, 1993.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[3] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: quantitative model and tool interaction. In *TACAS (2)*, volume 10206 of *Lecture Notes in Computer Science*, pages 151–168, 2017.

[4] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods Syst. Des.*, 10(2/3):149–169, 1997.

[5] Jan Friso Groote, Freek Wiedijk, and Hans Zantema. A probabilistic analysis of the game of the goose. *SIAM Review*, 58(1):143–155, 2016.

[6] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[7] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *[1989] Proceedings. Real-Time Systems Symposium*, pages 102–111, 1989.

[8] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker storm. *International Journal on Software Tools for Technology Transfer*, 24(4):589–610, 2022.

[9] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. October 2020.

[10] Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd Millstein, Sanjit A. Seshia, and Guy Van den Broeck. Model checking finite-horizon markov chains with probabilistic inference. pages 577–601. Springer International Publishing, 2021.

[11] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[12] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[13] David Peter. hyperfine, March 2023.

[14] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers.

[15] Tim Quatmann and Joost-Pieter Katoen. Sound value iteration. *CoRR*, abs/1804.05001, 2018.

[16] Diederik R. Sleijpen, Gerard L.G.; Fokkema. Bicgstab() for linear equations involving unsymmetric matrices with complex spectrum. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 1:11–32, 1993.

[17] Peter Sonneveld and Martin B. van Gijzen. Idr(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM J. Sci. Comput.*, 31(2):1035–1062, 2008.