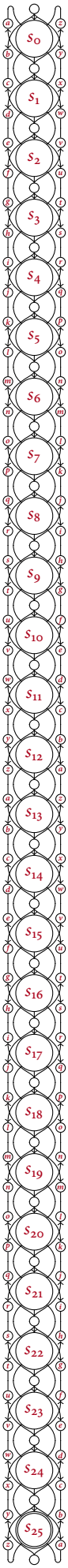


Representing PCTL Counterexamples

Master of Science Thesis

BERTEUN DAMMAN

Graduation Committee

Prof. Dr Ir Joost-Pieter Katoen
Tingting Han MEng (韩婷婷)
Dr Ir Arend Rensink

University of Twente
Enschede, The Netherlands

Hic labor extremus, longarum haec meta viarum.

Abstract

We discuss counterexamples for Probabilistic Computational Tree Logic (PCTL), and an algorithm based on the k shortest path algorithm to find these counterexamples. Also discussed are the hop-constrained variants of the algorithms, and way to reduce hop-constrained problems to unconstrained problems. The discussed algorithms are implemented and this implementation is used to analyse several case studies. The experiments and mathematical analysis show that in practice the number of paths needed for a counterexample may grow exponentially. In order to combat this size explosion we propose to use the existing technique of bisimulation minimisation on the one hand, and on the other hand we introduce regular expressions to represent sets of paths compactly. Experiments of these compactifications are also included, which show that very short expressive, and above all, intelligble, expressions can be obtained.

Contents

Preface

The last miles takes the longest: this proverb summarises the whole process. It has been a long time since I have commenced my graduation. However, at last, the final product has been completed.

Because the subject is quite theoretical, and uses many mathematical ideas not familiar to most computer science students, and on the other hand uses quite some ideas from computer science not familiar to students of mathematics, this thesis contains a substantial introduction to the techniques used. I hope that this will make the ideas clear, since I do believe that the combination of mathematics and computer science in this area is very worthwhile, even tough during my study I have seen how many mathematical subjects have disappeared from the curriculum.

Finally I would like to thank my friends, parents and my graduation committee; and especially Joost-Pieter Katoen & Tingting Han, who gave me the opportunity to work on this subject in Aachen. Even though the ride was not very smooth at times, they were confident and willing to continue the supervision, even when I had my doubts.

As it turned out, they were right, and I am very grateful for this.

Berteun Damman,
Enschede, 18th December 2008

Introduction

0

The Dutch computer scientist Edsger Dijkstra, famous for his shortest path algorithm, once remarked in an interview with the Dutch news paper *Trouw*: 'Software? By your leave, I think it's rubbish.'[1] He is clearly aggravated by the state of software and the number of times computers tends to crash.

And indeed, since the day the first computers have been built, and the first software was programmed, programmers and users alike have been plagued by bugs. Many strategies have surfaced to prevent, detect and repair these problems.

Some strategies rely on establishing 'best practices': rules of thumb which, if followed, avoid a number of common pitfalls; other strategies involve extensive testing of the written software against predetermined cases, in order to find the most common bugs. But as the same Dijkstra famously remarked: 'Program testing can be used to show the presence of bugs, but never to show their absence!'

The field in which this thesis is written, is the field of formal methods. In this field the main strategy to prevent software bugs is called 'model checking'. The model, a formal structure derived from an actual hardware or software design, is checked to see whether it obeys its formal specifications. This forms the core idea: verify a *formal* structure, against *formal* specifications. This verification is exhaustive. Every possible scenario should be verified. Verifying every possible scenario however, is very hard, and making it feasible to check ever larger and more complex systems is one of the challenges researchers in model checking face.

Because of the ability to mathematically verify software, and to guarantee the absence of errors, model checking is usually deployed in scenario's in which correct behaviour of the software is of paramount importance, such as software for medical apparatus, space probes, flood barriers, and such. A notorious example of software errors leading to the death of three people is given by the Therac-25 radiation therapy machine, which would under certain conditions give massive overdoses of radiation. This underlines the danger of software errors in certain applications, and the important part model checking can play in preventing these mistakes.

The foundations of model checking were laid in 1981, by Edmund Clarke and E. Allen Emerson, initially developing model checking for hardware. As computers became more powerful, it became possible to also verify software models. In addition

[1] The article *Software? Ik vind het, met permissie, puin.* can be read in Dutch at `http://www.trouw.nl/archief/article1322199.ece`.

to the scenario described above, where faulty software can endanger lives, a new need for ensuring the correctness of software has emerged because of the proliferation of embedded software in all kinds of devices. Much software is located in places where patching is very expensive, such as in cars, and where a recall due to a software bug would be prohibitively costly for the manufacturer.

This has lead to the widespread adoption of model checking in both the hardware and software industries, and for this the Clarke and Emerson have been awarded the Turing Award[2] in 2007. Nevertheless, new techniques and refinements of existing techniques are constantly being discovered and developed, and this thesis tries to explore a small area of this large field even further.

Model checking thus aims to automate the process of finding bugs in software. Although knowing that software does not function according to the specification is useful, it is even more useful to be able to give a demonstration when it goes exactly awry. Only reporting 'It doesn't work' will not aid the programmer much in ameliorating the problem. For this, *counterexamples* are much more useful. These provide a sequence of steps after which the problem occurs. Such a sequence usually gives a clear indication where the software goes wrong, and which parts of the source code need to be patched.

Earlier theories of model checking allowed the specification and checking of properties which stated that something bad would never happen, so called *safety properties*, or something good would always be the case, so called *liveness properties*. For many practical applications such specifications are too rigid. Even if you backup your data every day to different computers in different countries, it could in theory still happen that in the same night each of these buildings burns down, and you lose your data. Since this is a very improbable scenario, most people are willing to take this risk.

Usually, we would like to know what risk we are taking exactly. A server equipped with multiple hard disks in a RAID-configuration might fail irreparably if the rare situation occurs that after one disk fails a second one fails before the first one is replaced and rebuilt. Other questions could be that we want to guarantee a minimum level of service: if e-mail messages arrive randomly, but on average with a specific frequency, which might be higher during working hours, we would like to guarantee the mail server has enough capacity to always deliver nearly all messages within one minute, even if peak loads occur because by coincidence many messages are sent at the same time, in stead of evenly distributed. Finally, an example we shall see in this thesis too, in many distributed in which computers can join and leave a network, there has to be one computer that is designated the leader, electing such a leader occurs through some sort of voting process, but this can lead to a tie. If such a case, the computers have to vote again, by introducing a random factor in the vote, the election will be different next time, but we want to guarantee that ties do not occur too often, because as long as a leader has not been elected communication is usually suspended.

These are some very practical reasons why we are usually interested in questions such as 'is the system able to respond within ten seconds 98% of the time?' Or: 'Will the availability be at least 99%?' For this we need to be able to specify probabilities,

[2] The 'Nobel Prize for Computer Science'.

randomness and time-limits. A logic that allows us to specify this has been developed, and is called PCTL. Existing model checking algorithms for PCTL are heavily based on the theory of Markov chains, which captures random and probabilistic behaviour and forms the underlying formal structure, and they are able to report whether a property is violated or satisfied by the model. In case it is violated however, these methods cannot provide the programmer with a counterexample. Algorithms that fill this gap have been designed recently.

The current algorithm works by finding a set of paths, which together form a counterexample. Roughly speaking, every path describes some scenario in which a problem occurs, and the probability of these scenario's added together exceeds the specified limit.

This thesis elaborates on the theory behind the algorithms, and describes a practical implementation of these algorithm which was made in order to get acquainted with the behaviour in practice. The obtained results show that in practice counterexamples will consist of *many* paths. The number of paths will be so large that investigating them will be a daunting task.

The final chapters therefore will describe some preliminary ideas to reduce the size of these counterexamples, using regular expressions, or to present them in a form which can be more easily digested by the programmer.

Furthermore some remarks are made with respect to the algorithms used, and the possibility to achieve the same results by transforming the underlying transition structure instead of adapting the algorithms for certain verification problems involving bounded temporary operators.

The thesis will start with a chapter covering the preliminary notions, such as automata, regular expressions, Markov chains, Computational Tree Logic (CTL) and PCTL. The second chapter formalises what a counterexample is and describes the problem of finding counterexamples and which algorithms, mainly variants of the shortest path algorithm, can be used. These two chapters together form the theoretical basis for the description of the practical implementation, which is found in the third chapter. This describes the chosen algorithms and the data structures used in more detail, as well as the limits and possibilities of the algorithm. The next chapter presents the results of the practical experiments, and analysis the growth of the counterexample size. Finally chapter five proposes the use of regular expressions to combat counterexample size explosion.

Preliminaries

1

This chapter intends to serve as a reminder for those already familiar with the concepts used in this thesis, or as a primer for those unfamiliar with these concepts; the latter might want to refer to some of the textbooks mentioned, if they wish to acquire a more intimate knowledge of these subjects.

Those already knowledgeable of finite state automata, Markov chains or any of the other subjects introduced in this chapter might still want to glance over the material in order to get acquainted with the syntax used.

Although the reader can choose to only read those section wherewith he is not familiar, he or she should note that they were written to be read in sequential order, and will assume knowledge of the material in the previous sections.

A basic mathematical understanding, especially with respects to sets, is expected. Those unfamiliar with this should turn to a text book such as (Sudkamp, 1998) (especially the first chapter) or Wikipedia for an explanation of the concepts involved.

1.1 Words and languages

The term *language* describes several distinct, yet related concepts. First of all there are languages such as English, German or Dutch, so called *natural languages*. Furthermore there are *computer languages*, a rather specific set of symbols which can be used to write computer programs. Then there are also *sign languages* for the deaf in which words are represented by gestures.

Speaking in an abstract manner, these languages share a common concept of symbols which can be combined to form words. In the case of English or Latin these are simply the twenty-six letters of the Latin Alphabet: *a*, *b*, et cetera. German adds some other symbols such as *ä* and *ß*. Greek and Russian even use a completely different set of symbols, whereas sign language uses gestures. Chinese has an enormous collection of basic symbols.

These symbols can be combined to form words. In the case of English one might combine five characters to form the word *hello*, or in case of German *moin*. One could also form the word *xuqa*, which, although humans will not know what it means, is a valid word in the sense that it uses only characters from the English alphabet. It clearly is not a Greek or Russian word, but it is a word – a word without any meaning.

This general idea that an utterance in a language consists of words which are in turn formed by symbols is the way language is defined in a theoretic way.

Definition 1.1.1. An *alphabet* consists of a finite set of symbols, also called letters. We usually use Σ to denote the alphabet of a language.

Definition 1.1.2. A *word* (also called *string*) is formed by juxtaposing a finite sequence of letters from an alphabet Σ. That is: $w = a_1 a_2 \cdots a_n$ with $a_i \in \Sigma$. Its length is the number of symbols, usually denoted as $|w|$.

Example 1.1.3. For example, consider the alphabet $B = \{0, 1\}$. Examples of words over this alphabet are 001, 10001, 0, with respective lengths of three, five and one.

There also is an *empty word* which has a length of zero, and this is denoted by ε.

Although words are considered to be indivisible units in a language, just as the word 'symbol' cannot be split in 'sym' and 'bol', we will allow definitions such as $v = wa_1$ where w is a word, and a_1 is a letter from Σ. We also allow 'glueing' of two words together, which is more precisely defined in the following definition:

Definition 1.1.4. If v and w are words over an alphabet Σ, with:

$$v = a_1 a_2 \cdots a_n \qquad\qquad w = b_1 b_2 \cdots b_n$$

The notation vw is used as a shorthand for:

$$vw = a_1 a_2 \cdots a_n b_1 b_2 \cdots b_n$$

This is called the *concatenation* of v and w. If w is the empty word ε, then $vw = v\varepsilon = v$, and also if v is the empty word $vw = \varepsilon w = w$.

Definition 1.1.5. If a word w can be written as the concatenation of two words u and v, i.e. $w = uv$, (with possibly $u = \varepsilon$ or $v = \varepsilon$), we say that u is a *prefix* of w, and v is a suffix of w. If there exists some u and v such that $w = uxv$, we say that x is a *sub-word* of w. Note that ε and w itself are always a prefix, suffix and sub-word of w. A *proper prefix* u of w is a prefix of w, where $u \neq \varepsilon$ and $u \neq w$. Proper suffixes and proper sub-words are defined analogously.

Using the empty word and this notation of concatenation we can give a definition of the set of words over Σ, which henceforth will be denoted as Σ^*.

Definition 1.1.6. The set of words over Σ is recursively defined as follows:

1. $\varepsilon \in \Sigma^*$, this is the basis.
2. If $w \in \Sigma^*$ and $a \in \Sigma$, then also $wa \in \Sigma^*$; this forms the recursive step.
3. Only those words formed by a finite number of applications of the previous two steps are in Σ^*

Example 1.1.7. Let $\Sigma = \{a, b\}$. Using the previous definition we can list the elements of Σ^*, these are in increasing length (halting at length 3):

Length 0 ε;

Length 1 a, b;

Length 2 aa, ab, ba and bb;

Length 3 aaa, aab, aba, abb, baa, bab, bba, bbb.

We see that there will be 2^k words of length k for this alphabet. In general we see that for an alphabet consisting of n symbols there will be n^k words of length k.

Similar to the English language, not every possible word that can be formed is considered a valid English words (such as *xuqa*). A language will normally have some restrictions on the set of all possible words, thus restricting the language to a subset of all possible words. This is expressed in the following definition.

Definition 1.1.8. A *language* over an alphabet Σ is a subset of Σ^*.

This need not be a proper subset, for the language of non-negative numbers is formed by all words over the alphabet $\{0, 1, 2, \ldots, 9\}$. The language consisting of all prime numbers however, is a subset of the words over the alphabet $\{0, 1, 2, \ldots, 9\}$. The word 32 is not part of this language, but 13 and 71317 are. The latter would even be part of the more restricted language of palindromic primes.

One can also have more than one language over the same alphabet of course. One could for example define the language 'even numbers', and the language 'multiples of three' over the alphabet $\{0, 1, 2, \ldots, 9\}$.

Languages themselves, similar to words, can also be combined in different ways to create new languages. Two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ can be merged into a new language $\mathcal{L}$: $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$, which defines the language of all words either in $\mathcal{L}_1$ or in $\mathcal{L}_2$ or both.

Two languages can also be *concatenated*, this is written as

$$\mathcal{L}_1\mathcal{L}_2 = \{w_1w_2 \mid w_1 \in \mathcal{L}_1 \text{ and } w_2 \in \mathcal{L}_2\}$$

Example 1.1.9. Suppose we would have a language $\mathcal{L}_1 = \{over, under\}$ and another language with the words $\mathcal{L}_2 = \{achieve, coat, sea\}$, then the concatenation of these languages $\mathcal{L}_1\mathcal{L}_2$ would consist of the six (British) English words $\{overachieve, overcoat, oversea, underachieve, undercoat, undersea\}$.

Commonly though, this construct is used to expand a set of letters to a language consisting of all words that can be formed by these letters; by repeatedly concatenating the language to itself, which we shall see below.

This way, also the *power* of a language can be defined. The n-th power of a language $\mathcal{L}$ is written as $\mathcal{L}^n$, where n is a non-negative integer, and is defined as follows:

1. $\mathcal{L}^0 = \{\varepsilon\}$
2. $\mathcal{L}^n = L^{n-1}L$, for $n > 0$.

A final, important, operation, is the (Kleene) *star* or *Kleene closure* of a language, which is written as $\mathcal{L}^*$ and defined as:

$$\bigcup_{i=0}^{\infty} \mathcal{L}^i$$

Note that the notation Σ^* is consistent with this.

Now that we have stipulated what is meant by words and languages we can advance to the concept of a Finite State Automaton (FSA).

1.2 Finite State Automata

1.2.1 History

Finite state automata can be found at the heart of theoretical computer science. Despite this theoretical character, their practical applications are surprisingly wide ranging and manifold.

The roots of FSAs can be traced back to the work of Warren McCulloch and Walter Pitts, whose article, (McCulloch & Pitts, 1943), described a mathematical model of neurons, i.e. nerve cells in the brain. Even though Alan Turing had presented his Turing machine seven years before, and as such had introduced the idea of an abstract machine performing in a deterministic manner, his machine is way more powerful than a finite state automaton.

The work of McCulloch an Pitts was subsequently presented to Stephen Kleene for investigation by the RAND corporation. This research, although completed in 1951, was not published until 1956. His seminal work (Kleene, 1956) also provides a proof of what is now known as 'Kleene's theorem', a result linking regular sets and finite state automata intimately.

During the following years the theory behind FSAs has been greatly expanded and FSAs have been applied in many fields, partly because of their simplicity which gives them a large practical advantage over Turing-machines, whose properties are mostly studied in an abstract context. FSAs however are very useful in solving problems involving circuit design, lexical analysis and text processing and in recognising certain numbers or even biological sequences.

The interested reader, with a command of the French language is referred to Perrin (1995) for a more detailed exposition of the history of FSAs.

1.2.2 An abstract automaton

FSAs are usually used to determine whether a word belongs to a language, i.e. a subset of all possible words (or strings) over a given alphabet. The language of even numbers over the alphabet of digits will only consists of digits ending in 0, 2, 4, 6 or 8. Something like 13 is a valid word, but it is not part of the language of even numbers. It turns out that for certain languages one can construct an FSA, whereas one cannot for other. The languages for which one can construct an FSA are called *regular languages*.

There are several ways to depict automata, as a picture with circles and arrows or in a more mathematical way. We shall start out with the pictures, which give a more intuitive idea, and will then provide the mathematical definition.

A sample automaton is shown in figure 1.1. Its basic features are circles and arrows connecting them. Every circle is called a *state*, and this model is a schematic layout

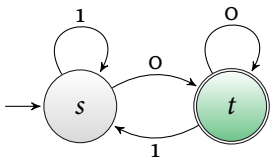


Figure 1.1 An automaton accepting words over $\{0, 1\}$, ending with 0.

of the states of the automaton. When the automaton 'runs' it will move from state to state, obeying the arrows in the figure. If there is an arrow from one state to another, that means the automaton can make a *transition* from the first state to the next one, provided the input matches the symbol near the arrow. For example, the automaton in figure 1.1 can make a transition from s to t provided the input is 0. It can return to s from t in case of a 1 in the input. Also, it can stay in t, if another 0 is input.

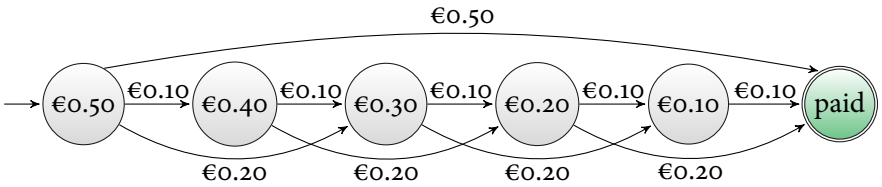
The input is usually a word, in the sense defined in definition 1.1.2. The automaton will consume this word symbol by symbol, and take the appropriate transition. It will start reading from the left, so if the word 001 was provided to our sample automaton at the moment it was in s, it would move to t, consuming the first 0, and leaving 01 to be processed. After that it would consume another 0, which means it will stay in t and finally it would see a 1, so it would return to s. If the automaton would reside in t at the moment it was given 001 it would also end up in s.

To make clear where an automaton starts, a special state, called an *initial state* exists, which is designated as $\twoheadrightarrow\bigcirc$. In figure 1.1 this is state s. Also, there is at least one *final state* in the automaton, which is indicated by a circle with a double border: ◎, which is state t in the example.

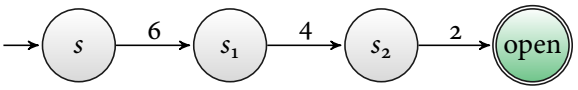
If the automaton ends in a final state after having consumed the final symbol of the input word, it is said to *accept* this word. Our example automaton will thus accept any word consisting of zeros and ones, ending with a zero. One could interpret these words as binary numbers and say that it will only accept those numbers ending with 0, or indeed the even numbers.[1]

In the real world, many automata also take some form of input. In the case of vending machines, this will consist of some coins, which need to add up to the required amount; or in case of a combination lock it should be a series of digits that have to be input in the right order. One can also model these automata as an FSA. In the first case the alphabet will for example consist of the symbols €0.10, €0.20 and €0.50, if the automaton only accepts ten, twenty and fifty cent coins. A 'word' is a sequence of these symbols. One can see in figure 1.2a that the words that are accepted are those that have a monetary value of €0.50. In case of the combination lock, shown in figure 1.2b the winning combination is 642, which has to be input in the right order.

[1] In case of the even numbers, one can easily construct an automaton that takes decimal numbers as input. It should go from s to t, or stay in t if an even digit was input, and otherwise go to s. In general however, certain problems can only be solved if the input is in the right number base. One can imagine that it is easy to recognize words that are a power of 10 in base 10 or a power of 2 in base 2. These words will have exactly a single one and end with a series of zeros in their respective bases. In other bases however, these numbers are not so easy to recognize.



a A model of a vending automaton, which accepts exactly €0.50 in coins of €0.10, €0.20 or €0.50.



b A model of a combination lock. This is like peering inside the lock – in reality one would only be able to try to input something and see whether the automaton accepts it.

Figure 1.2 Two examples of finite state automata, based on 'real world' automata.

One could wonder, what if someone inputs a symbol which is not defined for that state? For example, someone inputs a 3 in the combination lock. In fact this is negligence on our part, a truly deterministic FSA should have exactly one transition at every state for every possible input symbol. Usually it is made implicitly clear what to do if a symbol is not defined, in the case of the combination lock it should start over, in other cases the automaton might be amended to include one extra state, to which every 'undefined' transition will go. Once having entered this state, the automaton will stay there and make a transition to itself for every possible input symbol.

One can see how an automaton accepts a word, and we know that a language consists of words. Noting this, we can see how an automaton also describes a language, viz. the language of those words accepted by the automaton, which is usually a subset of all words over the (input) alphabet of the automaton. The 'language' of the vending machine consists of input sequences that added up are worth €0.50. The language of the combination lock are only those sequences that end in 642, out of all possible sequences of integers.

We will make this connection more precise and formal in the next section.

1.2.3 The mathematical automaton

The previous section presented a rather intuitive model of the automaton, and a way to visualise it, using circles and arrows. Now we shall provide a mathematical definition. This mathematical definition formalises and specifies what is meant exactly when we mean by accepting a word.

The previous section however, can always be used as an intuitive guidepost.

Definition 1.2.1. A *Finite State Automaton* is defined as a quintuple (Σ, Q, I, F, E), where Σ is the alphabet over which the automaton is defined, Q is a set of states, I is a set of initial states, with $I \subset Q$; the set of final states is given by F, also with $F \subset Q$, and

finally the edges are given by E such that $E \subset Q \times \Sigma \times Q$.

Definition 1.2.2. A deterministic FSA is an FSA with the following constraints:

- $|I| = 1$, i.e. there is a unique initial state.
- For each pair $(s, a) \in Q \times A$, there is at most one state $t \in Q$ such that $(s, a, t) \in E$, i.e. in every state there is at most one choice for a given input symbol.

The automata in the figure 1.1 and figures 1.2a and 1.2b are all deterministic. In the combination lock example however, we did not specify what should happen if the user tries to input 4 in state s. Automata like this one, which do not specify an action for every input symbol in every input state are called incomplete, a notion specified in the next definition.

Definition 1.2.3. An FSA is called *complete* if for each pair $(s, a) \in Q \times A$, there is at least one state $t \in Q$, such that $(s, a, t) \in E$. i.e. in every state there is at least one choice for a every given input symbol. If there is no option for some input symbol from the symbol, the automaton is called *incomplete*.

Now that we know what an automaton looks like mathematically speaking, we can make the concept of *accepting a word* more precise. But first we need to define how we associate words with automata.

Definition 1.2.4. A *path* in an automaton $\mathcal{A} = (\Sigma, Q, I, F, E)$ is a sequence of connected edges $p = e_1 \cdots e_n$, where $e_i = (s_i, a_i, t_i) \in E$, such that $t_i = s_{i+1}$, for $i < n$. The state s_i will be called the *source* of the path and the state t_n will be called the *end* of the path. The ordered concatenation of the symbols a_i, i.e. $a_1 \cdots a_n$ gives the *label* of the path. This label is also a word, compare with definition 1.1.2.

Having established how we associate a word with a path in a automaton the time is ripe to give a definition of accepting a word.

Definition 1.2.5. An automaton $\mathcal{A} = (\Sigma, Q, I, F, E)$ is said to *accept* a word w if there is a path in A with label w such that its source is an element of I and its target is an element of F. Such a path is called *successful*.

The previous definitions enable us to straightforwardly give a definition of the language of an automaton:

Definition 1.2.6. The *language* of an automaton A, written as $\mathcal{L}(A)$ is defined as the set of all words accepted by A.

The attentive reader might have noted that we have spoken about finite state automata, deterministic finite state automata, complete automata, thus implying that there should also be non-deterministic automata, or incomplete automata. The reader might also wonder whether there is a substantial difference between the two, can one type of automata accept words another type cannot, and what is the real difference?

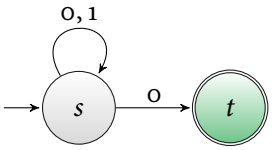


Figure 1.3 A non-deterministic automaton accepting words over $\{0, 1\}$, ending with 0.

The following theorem, which is given without its (short) proof states this is not the case – the interested reader is referred to (Sudkamp, 1998) or (Perrin, 1990) for details.

Theorem 1.2.7. *For each finite automaton, there exists a deterministic and complete finite automaton which accepts exactly the same language.*

In practice however, it is often much easier to specify an incomplete automaton (as we have done in the examples in figure 1.1, 1.2a and 1.2b), and at times a lot more concise to specify a Non-Deterministic Finite State Automaton (NFA). Although we have not paid much attention to NFAs, the main difference with a Deterministic Finite State Automaton (DFA) is that for an NFA which accepts a word w there might be other paths with the same label that are not successful. This does not matter as long as there is at least one path that is successful. An example of a slightly more compact specification of the automaton in figure 1.1 is given in figure 1.3. For the input symbol 0 there are two choices in state s. Either stay in s, or go to t. The 'right thing' to do is to only choose the transition to t if the 0 is the last symbol of the input. The automaton of course does not know this, it is not supposed to be clairvoyant. It has to decide in an ad-hoc manner what to do with the current input symbol. If one wishes, one could interpret it as always choosing the correct option, or as exploring every option in parallel. Our definition however, only states that there should exist some successful path for this word; how this path is to be found in practice is immaterial.

1.3 Regular expressions

The previous section described the way FSAs relate to languages and words in the sense we have defined them. As implied therein, there are many different ways to define which words can be part of a language, but one of the simplest ways, known to linguists as a 'Type 3 grammar' is given by regular expressions.[2]

It turns out that these regular expressions give another method to define regular languages. Because FSAs can also be used to define regular languages, these two formalisms are equivalent. Every language for which one can construct an automaton, one can also define by a regular expression.

[2] Some authors, especially from the Francophone parts of the world, prefer the term *rational expression*, because of the close analogy between these expressions and the rational power series (or fractions) of classical algebra. We shall follow the multitude however.

Definition 1.3.1. Let Σ be an alphabet. The set of regular expressions over Σ is recursively defined as follows:

1. ε and $\varnothing$ are regular expressions.
2. For every symbol $a \in \Sigma$, a is a regular expression.
3. If e_1 and e_2 are regular expression, then so are e_1e_2, $(e_1 + e_2)$ and (e_1^*).
4. Nothing else is a regular expression.

Normally, the parentheses are not always written down. For example, an expression like $a\varepsilon + bc^*$ should be read like $((a\varepsilon) + (b(c^*)))$. From this we can see concatenation has precedence over $+$, but $*$ has precedence over concatenation. A frequently encountered variant notation for $a + b$ is $a \mid b$.

The regular expressions a and ε represent the languages which just consist of that single symbol. An expression like $e_1 + e_2$ represents a choice, it represents the union of the languages defined by e_1 and e_2. The * represents repetition, and is a shorthand for all strings in which the previous symbol is repeated zero or more times.

Definition 1.3.2. The language $\mathcal{L}(e)$ defined by a regular expression e is as follows

$$\mathcal{L}(\varnothing) = \varnothing \qquad \mathcal{L}(\varepsilon) = \varepsilon$$
$$\mathcal{L}(a) = \{a\} \qquad \mathcal{L}(e_1e_2) = \mathcal{L}(e_1)\mathcal{L}(e_2)$$
$$\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \qquad \mathcal{L}(e^*) = \mathcal{L}(e)^*$$

Kleene (1956) first proved the equivalence of regular expressions and FSAs. There furthermore exist different methods to convert a regular expression to an FSA which accepts the same language as the regular expression defines, and vice versa. An overview of the different approaches can be found in the article by Yu (1997).

1.4 Markov chains

The field of Markov chains is mathematically very well developed, and knows many applications in the fields of biology, physics and also computer science. They are named after the Russian mathematician Andrei Markov (Андрей Марков).

A Markov chain, in a way, can be viewed as some sort of autonomous automaton, which starts in a certain state, and then moves with some probability to another state by itself. It hence differs from a normal automaton by not requiring input, like the vending machine example of figure 1.2a. Instead, it makes the transitions according to a certain probability. The defining property of the Markov chain is that this probability of going from a certain state s_1 to a state s_2, only depends on the current state the automaton is in, and not on any previous state it might have visited, this is known as the *Markov property*. Finally, the Markov chain does not have an accepting state.

If we take an early look at figure 1.5a on page 19, and assume it models a fair coin, we can assign probabilities to the transitions in this automaton. The transition

start→heads should have a probability of ½, and hence the transition to *tails* should also have probability ½. Indeed, every transition should have probability ½ in this automaton.

This expresses, that even though one might have flipped heads already ten times, the probability that heads will come up after the next flip does not change, as we expect, because the coin does not have any memory.

For example, many games, such as Monopoly, can be viewed as Markov chains. The square a player will land on will solely depend on the square its currently on and the outcome of the next throw.[3] Other games, such as Blackjack, cannot be modelled as a Markov chain, since the probability that you won't exceed 21 if you already have 17 depends on the cards already drawn during previous turns.

Mathematically, a Markov chain is usually represented as a transition matrix. That is, an entry p_{ij} in the matrix represents the probability of going from state i to state j. Furthermore, the probabilities in every row should add up to one, i.e. $\sum_j p_{ij} = 1$, because you always have to go to some other state (or stay where you are). A matrix in which every row adds up to 1 is called *stochastic*.

The name *chain* means the state space of the Markov process is finite. The process is defined as a sequence of random variables $X_1, X_2, \ldots$ which indicate the state for each moment in time, with the Markov property which states that the next state solely depends on the present state and not on the previous state, or formally:

$$\Pr\left(X_{n+1} = x \mid X_n = x_n, \ldots, X_1 = x_1\right) = \Pr\left(X_{n+1} = x \mid X_n = x_n\right).$$

Note that we have also tacitly assumed that $\Pr\left(X_{n+1} = x \mid X_n = x_n\right)$ (the one step probabilities) are independent of n, which means the transitions probabilities are *stationary*. This assumption allows us to arrange the transition probabilities as a matrix.

1.4.1 Discrete Time Markov Chains

To be precise, the Markov chains we use are Discrete Time Markov Chains (DTMCs)[4]. In the following definition, let AP denote a fixed, finite set of atomic propositions. We usually use $a, b, c, \ldots$ or a_i to denote these propositions. Their rôle is semantic. For example, the atomic proposition h might indicate an outcome 'heads' for the coin toss example. Or *success* might indicate some operation has succeeded. We use these propositions to annotate states; the next section will make use of these propositions.

Definition 1.4.1. A (labelled) Discrete Time Markov Chain (DTMC) is a triple $\mathcal{D} = (S, \mathbf{P}, L)$, where:

[3] After the throw, one might land on Community Chest, or Chance, thus moving again, but assuming you pick a random card from the pile in each case, this can be modelled very simply. Suppose you have to throw 5 to arrive at Chance. This has a probability of ⅑. Then, there is probability of 1/16 you pick the 'Advance to Go' card, and hence for that turn there is a probability of 1/144 of ending at Go, assuming you couldn't arrive at it directly.

[4] This name is actually historic and can be considered a bit of a misnomer, since we usually abstract away from the notion of time when using these automata. The transitions between states are discrete however, but they do not necessarily take a fixed amount of time.

- S is a finite set of states;
- $\mathbf{P} : S \times S \to [0,1]$ is a stochastic matrix;
- $L : S \to 2^{AP}$ is a labelling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in s.

Note that, from a computer scientist's point of view, a DTMC is a Kripke structure in which all transitions are equipped with discrete probabilities such that the sum of the outgoing transition probabilities of each state is equal to 1.

Note also, that we do not equip the DTMC with a starting distribution, in this thesis we always assume the DTMC has a unique initial state.

If $\mathbf{P}(s,s) = 1$ for $s \in S$, then we call that state *absorbing*. If we draw the DTMC as a picture, we do not draw the transitions between states s_1 and s_2 for which $\mathbf{P}(s_1,s_2) = 0$. The *size* of a DTMC $\mathcal{D}$ is denoted by $|\mathcal{D}|$, and is the number of non-zero entries in $\mathbf{P}$.

1.4.2 Paths in DTMCs

Having established what a DTMC is, and how to give certain properties to states using atomic propositions and a labelling function, we shall now make clear how to specify a sequence of movements 'through' a DTMC.

Definition 1.4.2. Let $\mathcal{D} = (S, \mathbf{P}, L)$ be a DTMC.

- An *infinite* path σ in $\mathcal{D}$ is an infinite sequence $s_0 s_1 s_2 \cdots$ of states, such that $\forall i \geq 0 : \mathbf{P}(s_i, s_{i+1}) > 0$.
- A *finite* path is a finite prefix of an infinite path.

Note that, because $\mathbf{P}$ is stochastic we can never get stuck in a state, every state has a successor. An infinite path is thus always possible. Because there is only a finite number of states in the structure, an infinite path needs to have some parts that are repeated infinitely often, we use the ω subscript for this, e.g. $s_1 s_2 (s_3 s_4)^\omega$ indicates a path where the suffix $s_3 s_4$ is repeated infinitely often at the end of the path.

We use $Paths^{\mathcal{D}}(s)$ to denote the set of all infinite paths in $\mathcal{D}$ that start in state s and we use $Paths^{\mathcal{D}}_{fin}(s)$ to denote the set of finite paths starting in s. If it is clear from the context which $\mathcal{D}$ is meant, we omit the superscript.

Let $\hat{\sigma} = s_0 \cdot s_1 \cdots s_n \in Paths^{\mathcal{D}}_{fin}(s_0)$. That is, $\hat{\sigma}$ is a finite path starting in s_0. If $\mathbf{P}(s_n, s) > 0$, then we can extend a path σ by s, which we denote as $\sigma \cdot s$. The length of a path σ is denoted as $|\sigma|$ and is measured by the number of *transitions* of a path. Hence, $|\hat{\sigma}| = |s_0 s_1 \cdots s_n| = n$, and $|s| = 0$. For an infinite path σ we have $|\sigma| = \infty$.

There are also some operators for obtaining a specific state in a path, and for obtaining the prefix or suffix of a path. Indexing starts at 0, so we write $\sigma[i]$ to obtain the $(i+1)$-th state in the path, with $0 \leq i \leq |\sigma|$. So, for example, $\hat{\sigma}[i] = s_i$. Let $\sigma[\leq i]$ denote the prefix of path σ truncated at length i, thus ending at the $(i+1)$-th state. That is $\sigma[\leq i] = \sigma[0]\sigma[1]\cdots\sigma[i]$. Dually, the suffix of a path, written as $\sigma[\geq i]$ is defined as $\sigma[\geq i] = \sigma[i]\sigma[i+1]\cdots$.

The set of all prefixes of a path σ is denoted as $Pref(\sigma)$ with:

$$Pref(\sigma) = \bigcup_{i=0}^{|\sigma|} \{\sigma[\leq i]\}$$

1.4.3 Probability of paths

Now that we have a way of specifying paths in a DTMC, where every transition has a certain probability associated with it, we would like to assign a probability to the whole path. That is, the probability that these transitions actually occur.

For this we make a small sidestep into measure theory, but this is only for the theoretical foundations, since the measure itself is as we would intuitively expect.

When looking from a probabilistic point of view, terms like *experiment*, *outcome* and *event* are used. For example, the experiment might be a throw of a dice, the outcomes would be 1 to 6, and an event could be throwing an odd number, which would be an outcome of 1, 3 or 5. So basically events consist of a set of outcomes.

With DTMC, the experiment is a *run* through the DTMC; that is to say: we start in our initial state, then go to a successor state, again, and again. The outcome of such a run will be a path, in the sense of definition 1.4.2. Because paths are infinite, the probably that a specific single path will occur can be zero, for example the probability of infinitely often throwing heads is zero.[5] This is why we are usually interested in events that consist of an (infinite) set of paths. We use the set of finite paths as as basis. The reason for this is that generally we are in something happening within a finite amount of time, and after that the course of the path is irrelevant for us.

This leads us to the idea of a (basic) cylinder: namely the set of all paths that start with a specified prefix.

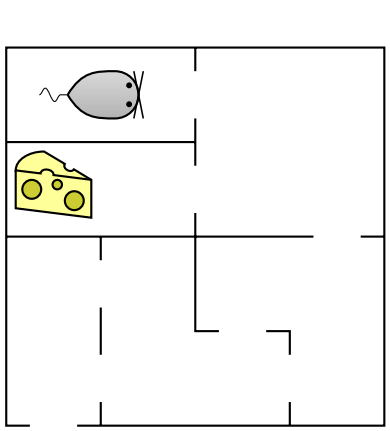
Definition 1.4.3. The *cylinder set* of a finite path $\hat{\sigma} = s_0 \cdot s_1 \cdots s_n \in Paths^{\mathcal{D}}_{fin}(s_0)$ is defined as:

$$Cyl(\hat{\sigma}) = \left\{\sigma \in Paths^{\mathcal{D}}(s_0) \mid \hat{\sigma} \in Pref(\sigma)\right\}$$

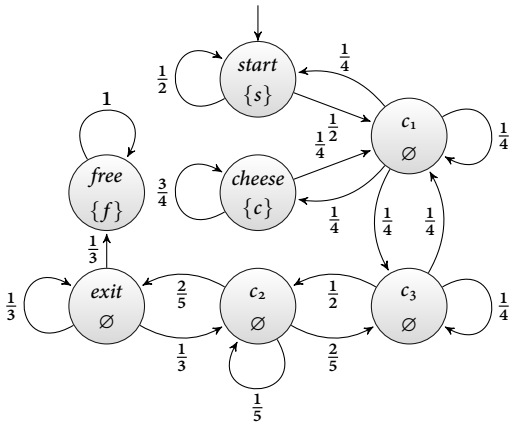
These cylinder sets play an important part in what we consider an event in the context of DTMC. The probability of an event comprising of a single cylinder set can now simply be found by multiplying the probabilities of the transactions in the prefix that generated the cylinder set. That this should indeed be a valid probability, can intuitively be seen by considering that the only thing that matters is that the prefix occurs, after that any continuation is good. The probability that something happens is 1. Hence, the probability for the whole set is given by the prefix.

We complete this interlude with a formal definition of the informal exposition above. The reader who is interested in a more thorough discussion of the ideas presented

[5] This indeed is one of the reasons we need more sophisticated tools such as measure theory to analyse the probabilities, because adding probabilities naïvely will not get us anywhere, because the basic outcomes have zero probability.



a A mouse in a maze with cheese and an exit.



b The corresponding transition sytem.

Figure 1.4 An example of a maze and the DTMC modelled after it.

here is referred to the article by Panangaden (2001) or the book of Baier & Katoen (2008, chapter 10). In addition, Capinski & Kopp (2004) provides a basic introduction into probability and measure theory, and explains for example what a σ-algebra is, which we use without further explanation in the following definition.

Definition 1.4.4. The probability measure $\Pr_{s_0}^{\mathcal{D}}$ (briefly Pr) induced by a DTMC $\mathcal{D}$ with initial state s_0 is the unique measure on its σ-algebra:

$$\Pr\{\underbrace{\sigma \in \mathit{Paths}(s_0) \mid s_0 s_1 \cdots s_n \in \mathit{Pref}(\sigma)}_{\mathit{Cyl}(s_0 \cdots s_n)}\} = \prod_{i=1}^{n} \mathbf{P}(s_{i-1}, s_i). \tag{1.1}$$

The σ-algebra should be the smallest σ-algebra containing all cylinder sets induced by the finite paths starting in s_0.

By slight abuse of notation we also write $\Pr\{\sigma\}$ as a shorthand for $\Pr\{\mathit{Cyl}(\sigma)\}$, where σ is a finite path.

1.4.4 Example DTMC

We shall now present an example of a situation which can be modelled by a DTMC to elucidate the terminology introduced in the previous sections.

A commonly encountered example is a mouse in the maze. An example maze is shown in figure 1.4. The mouse starts in the upper left of the maze, after which it is assumed to make a move to another cell every now and then. This move might also mean it stays in the same cell. We thus assume the mouse does not learn, which might be adequate considering the size of the maze. In the room next to the starting room,

there is a piece of cheese[6]. If the mouse discovers this, it will be less inclined to go to another room. If the mouse escapes, it will never come back in the maze.

In figure 1.4b, the corresponding transition system is shown. If a cell has two exits to another cell, the mouse is more likely to end up in this cell. We can see that 'free' is an absorbing state. This means that eventually, the mouse will always find the exit. Although, of course, in theory the mouse could decide to always stay in the first room, the probability of this is negligible.

A finite path in this maze could be $\sigma_1 = start{\cdot}c_1{\cdot}cheese{\cdot}cheese$. An infinite path might be $\sigma_2 = start{\cdot}c_1{\cdot}c_3{\cdot}c_2{\cdot}exit{\cdot}(free)^\omega$

The probability of the first path, $\Pr\{\sigma_1\}$, can be found by multiplying the probabilities on the transitions: $\Pr\{\sigma_1\} = \frac{1}{2} \cdot \frac{1}{4} \cdot \frac{3}{4} = \frac{3}{32}$. If we would formally define a DTMC for this transition system, say $\mathcal{D} = (S, \mathbf{P}, L)$, we would have:

- $S = \{start, cheese, exit, free, c_1, c_2, c_3\}$
- For the transition matrix (which uses the same order for the states as the previous item) we have:

$$\mathbf{P} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{3}{4} & 0 & 0 & \frac{1}{4} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & 0 & \frac{2}{5} & 0 & 0 & \frac{1}{5} & \frac{2}{5} \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix}$$

- And the labelling function is defined as: $L(start) = \{s\}$, $L(cheese) = \{c\}$, $L(free) = \{f\}$. For the other nodes we have $L(\cdot) = \varnothing$.

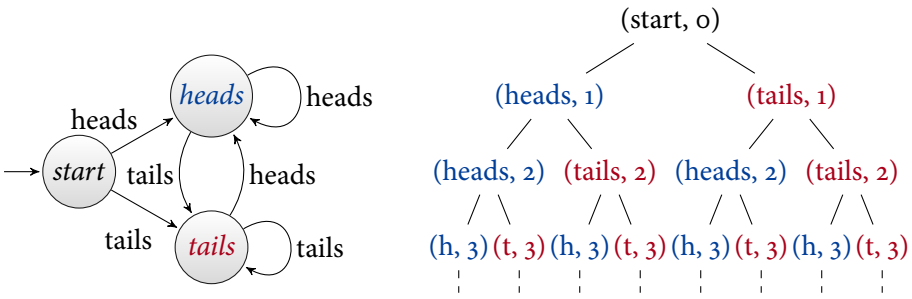
1.5 Computational Tree Logic

Computational Tree Logic (CTL) (Clarke & Emerson, 1981) is defined as a propositional branching-time temporal logic. As it is one form of temporal logic, it fits in the larger framework of modal logic; in this thesis however, we only concern ourselves with CTL as far as necessary for an understanding of the material involved here. Emerson (1990) provides a more detailed account on various temporal and modal logics.

Being a *temporal* logic, CTL allows us to propose things that will happen as time passes. For example, if someone is flipping a fair coin, one can state: until you flip heads, you flip tails. The previous will of course happen *always*, but you could also propose that 'you will eventually flip two heads and two tails' in a row. This need not happen, for one could, in theory, flip only heads, or flip heads and tails alternately.

Being a *branching* temporal logic, CTL allows us to view time, and in particular the future, as a tree of possible scenario's. That is, to continue the coin flipping example, first one might flip heads, and then one might flip heads again, and then tails. Or perhaps

[6] In reality, mice do not particularly like cheese, but we assume they do.



a An automaton representing an infinite sequence of coin flips.

b An infinite sequence of coin flips as an infinite tree (h = heads, t = tails)

Figure 1.5 A finite structure and the corresponding unwound infinite tree.

one would start with tails, et cetera. Viewed thusly, one could draw an infinite tree of scenario's. Such a tree can be derived by unwinding a finite structure, for example an automaton. An example of this idea is given in figure 1.5, where one sees an automaton and its corresponding tree.

Also, one can see in the figure 1.5b that time is represented as a discrete sequence of events. This allows us to speak about 'the next moment in time', or 'within five moments'; meaning we speak about the next level or the next five levels in the tree. Furthermore, we can clearly see how many scenario's there are. There are, for example four different ways to arrive at the second level in the tree, where each node represents a different sequence of coin flips.

The important aspect when employing temporal logic, is that one tries to reason about the behaviour of an underlying system which possibly never ceases running. These systems are typically *reactive*. They wait for some input, move to another state, and wait for input again. Other methods for program verification are often based on the *transformation* from one state to another, and one proves that this transformation is executed correctly, and after the computation the program is done. An example of this is a compiler which takes an input file and outputs an executable or a formatted document. An operating system however can keep on running and reacting to messages from the keyboard and the network.

The next section will define the syntax and semantics in a more thorough and mathematical way. Note that, although the semantics CTL can be defined more generally for any transition system (or semi-automaton), we use the DTMCs described in the previous section as the underlying structure.

1.5.1 Syntax and semantics

Since CTL is a branching logic, there is more than one scenario for the future. Its operators reflect this fact. One can quantify over these different scenario's – or paths. There are the basic temporal operator A (for all futures) and E (for some future); these are always followed by a usual linear time operator like F (sometime) G (always), X

(next moment), U (until) and W (unless). These linear time operators do not view time as branching, these are applied to one specific future scenario or path; the branching aspect of CTL is provided by the A and E operators. A combination like EF *win lottery* intuitively might have the meaning 'I might potentially win the lottery.' Sadly enough, this is not the guaranteed possible future, and it is not very probable it will be ours. The morbid AF *die* expresses that whatever the future will be, it is inevitable that we die some time. We furthermore allow for a bound on the maximum number of moments that can pass before something should hold, for example $\mathsf{AG}^{\leq 3}$ *rain* might be interpreted as a bad weather forecast which states that 'for the next three days it will invariantly rain'. If the bound is not mentioned, it is taken as being infinite and called *unbound*.

The inclusion of a time-limit on the operators was independently proposed by (Emerson *et al.*, 1991) and (Hansson & Jonsson, 1994). The latter also introduced a probabilistic operator, thus describing what is known as Probabilistic Computational Tree Logic (PCTL), which will be described in the next section.

1.5.1.1 Syntax

Definition 1.5.1. We again use an inductive definition. Let AP the set of propositions. First we give the state-formulae:

1. For every $a \in AP$ we have: a is a state-formula.
2. If Φ is state-formula, then $\neg\Phi$ is a state formula.
3. If Φ and Ψ are state-formulae, then $\Phi \vee \Psi$ is a state-formula.
4. If φ is a path-formula, then $\mathsf{E}\,\varphi$ and $\mathsf{A}\,\varphi$ are state-formulae.
5. Only those formulae formed by a finite number of applications of the previous four steps are state-formulae.

Path formulae are formed by the following rules:

1. If Φ is a state-formula, then $\mathsf{X}\,\Phi$ is a path formula.
2. If Φ and Ψ are state-formulae, then $\Phi\,\mathsf{U}^{\leq h}\,\Psi$, with $h \in \mathbb{N} \cup \{\infty\}$ is a path-formula.
3. If Φ and Ψ are state-formulae, then $\Phi\,\mathsf{W}^{\leq h}\,\Psi$, with $h \in \mathbb{N} \cup \{\infty\}$ is a path-formula.
4. Only those formulae formed by a finite number of applications of the previous three steps are path-formulae.

Remark 1.5.2. In stead of $\mathsf{U}^{\leq\infty}$ and $\mathsf{W}^{\leq\infty}$ we simply write U and W. These operators are also called *unbounded*, whereas $\mathsf{U}^{\leq h}$ and $\mathsf{W}^{\leq h}$ with $0 \leq h \leq \infty$ are called *bounded*.

Other logical connectives, such as $\Phi \wedge \Psi$ or other temporal operators, such as $\mathsf{F}\,\Phi$ can all be expressed in terms of the connectives and operators given above. The following

equivalences hold

$$\begin{aligned} \mathrm{tt} &\equiv \Phi \vee \neg\Phi & \mathsf{EF}^{\leq h}\,\Phi &\equiv \mathsf{E}(\mathrm{tt}\ \mathsf{U}^{\leq h}\ \Phi) \\ \mathrm{ff} &\equiv \neg\mathrm{tt} & \mathsf{AF}^{\leq h}\,\Phi &\equiv \mathsf{A}(\mathrm{tt}\ \mathsf{U}^{\leq h}\ \Phi) \\ \Phi \wedge \Psi &\equiv \neg(\neg\Phi \vee \neg\Psi) & \mathsf{EG}^{\leq h}\,\Phi &\equiv \mathsf{E}(\Phi\ \mathsf{W}^{\leq h}\ \mathrm{ff}) \\ \Phi \Rightarrow \Psi &\equiv \neg\Phi \vee \Psi & \mathsf{AG}^{\leq h}\,\Phi &\equiv \mathsf{A}(\Phi\ \mathsf{W}^{\leq h}\ \mathrm{ff}) \\ \Phi \Leftrightarrow \Psi &\equiv \Phi \Rightarrow \Psi \wedge \Psi \Rightarrow \Phi & \mathsf{AX}\,\Phi &\equiv \neg\,\mathsf{EX}\,\neg\Phi \end{aligned}$$

1.5.1.2 Semantics

In order to give the formulae some meaning, apart from the intuitive idea, we define two satisfaction relations (both denoted by $\models$). The first is for the path-formulae, and the second one for the state-formulae. Remember, we use the DTMCs described in the previous section as our underlying structure. So formally we would have $((\mathcal{D}, s), \Phi) \in \models$, but we simply write $s \models \Phi$ if $\mathcal{D}$ is clear from the context.

Definition 1.5.3. Let $\mathcal{D} = (S, \mathbf{P}, L)$, and $s \in S$, then for $\mathcal{D}$ and s we define $\models$ as the least relation satisfying:

$s \models \mathrm{tt}$		*True in every state*
$s \models a$	iff	$a \in L(s)$
$s \models \neg\Phi$	iff	not $(s \models \Phi)$
$s \models \Phi \vee \Psi$	iff	$s \models \Phi$ or $s \models \Psi$
$s \models \mathsf{E}\,\varphi$	iff	$s \models \varphi$ for some $\sigma \in \mathit{Paths}(s)$
$s \models \mathsf{A}\,\varphi$	iff	$s \models \varphi$ for all $\sigma \in \mathit{Paths}(s)$

For $\sigma \in \mathit{Paths}(s)$ we have:

$\sigma \models \mathsf{X}\,\Phi$	iff	$\sigma[1] \models \Phi$
$\sigma \models \Phi\ \mathsf{U}^{\leq h}\ \Psi$	iff	$\exists i \leq h : \big(\sigma[i] \models \Psi \wedge \forall o \leq j < i : (\sigma[j] \models \Phi)\big)$
$\sigma \models \Phi\ \mathsf{W}^{\leq h}\ \Psi$	iff	either $\sigma \models \Phi\ \mathsf{U}\ \Psi$ or $\forall\, i \leq h : (\sigma[i] \models \Phi)$

Remark 1.5.4. For $\sigma \in \mathit{Paths}_{fin}$ the semantics of the $\mathsf{U}^{\leq h}$ and $\mathsf{W}^{\leq h}$ operator need to be adapted slightly. We need to change the range of $i \leq h$ to $i \leq \min\{h, |\sigma|\}$. Note that in each case the validity of the U operator can be witnessed by a finite prefix of an infinite path.

Remark 1.5.5. More general versions of the bounded operators are possible of course. One could think of $\mathsf{U}^{\geq h}$ or $\mathsf{U}^{h_l \leq h \leq h_u}$ or $\mathsf{U}^{=h}$ which would give a lower bound, an interval or an exact number on the number of moments that can pass before the formula needs to hold. All these forms could be covered by one operator $\mathsf{U}^{[h_l, h_u]}$ where h_l specifies the lower bound, which can be zero, and h_u specifies the upper bound, which can be infinite. Again, this requires an adaptation to the range of $i \leq h$ to $h_l \leq i \leq h_u$.

In the above definition, the $\mathsf{W}^{\leq h}$ operator is mainly given for completeness, the thesis itself will primarily focus on the $\mathsf{U}^{\leq h}$ operator.

Note that if a finite path σ satisfies an until formula $\Phi \mathsf{U} \Psi$, so will any extension of σ, say $\sigma{\cdot}s$. In other words, it 'doesn't matter what happens after Ψ is valid in a state'. With this in mind we also define the minimal satisfaction relation of a path formula.

Definition 1.5.6. The minimal satisfaction of a path formula, $\overset{\text{min}}{\models}$ is given by:

$$\sigma \overset{\text{min}}{\models} \varphi \quad \text{iff} \quad \sigma \models \phi \text{ and } \forall \dot{\sigma} \in \mathit{Pref}(\sigma) \backslash \{\sigma\} : \dot{\sigma} \not\models \varphi.$$

We shall use the following expressions as a shorthand to denote a particular set of paths starting in a state s that satisfy a path formula φ:

- $\mathit{Paths}(s, \varphi) = \{\sigma \in \mathit{Paths}(s) \mid \sigma \models \varphi\}$
- $\mathit{Paths}_{\mathit{fin}}(s, \varphi) = \{\sigma \in \mathit{Paths}_{\mathit{fin}}(s) \mid \sigma \models \varphi\}$
- $\mathit{Paths}_{\mathit{min}}(s, \varphi) = \{\sigma \in \mathit{Paths}(s) \mid \sigma \overset{\text{min}}{\models} \varphi\}$

Given a $\sigma \in \mathit{Paths}(s, \varphi)$, we define $\mathit{Pref}_{\mathit{min}}(\sigma, \varphi)$ to be:

$$\mathit{Pref}_{\mathit{min}}(\sigma, \varphi) = \dot{\sigma} \in \mathit{Pref}(\sigma), \text{ s.t. } \dot{\sigma} \overset{\text{min}}{\models} \varphi$$

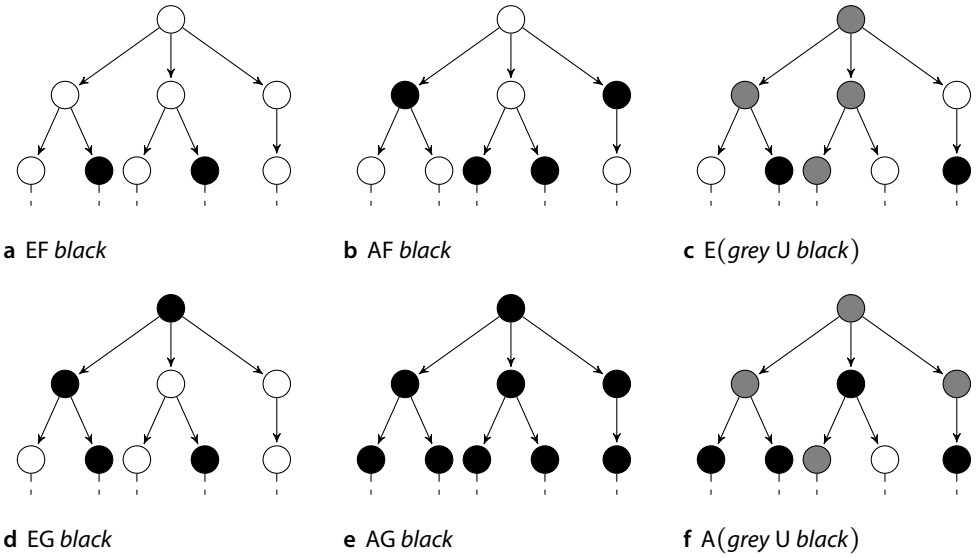
Remark 1.5.7. Note that, the $\mathsf{U}^{\leq h}$ (until) and $\mathsf{W}^{\leq h}$ (unless) are very closely related. The unless operator does not require its right hand side to hold at a future moment in time if the left hand side is always true. For this reason it is also called 'weak until'. For a single path σ we could even define the equivalence $\sigma \models \Phi \mathsf{W}^{\leq h} \Psi \equiv \sigma \models \Phi \mathsf{U}^{\leq h} \Psi \vee \mathsf{G}^{\leq h} \Phi$, where we could express $\mathsf{G}^{\leq h} \Phi$ as $\neg \mathsf{F}^{\leq h} \neg\Phi$; in this way the W operator would not need a separate definition. However, this does not work if the A and E operators come into play. Since $\Phi \mathsf{U}^{\leq h} \Psi$ is a path formula, we cannot say $\Phi \mathsf{U}^{\leq h} \Psi \vee \mathsf{G}^{\leq h} \Phi$, since $\vee$ can only connect state formulae. Furthermore $\mathsf{A}(\Phi \mathsf{U}^{\leq h} \Psi) \vee \mathsf{A}(\mathsf{G}^{\leq h} \Phi)$ means something different than $\mathsf{A}(\Phi \mathsf{W}^{\leq h} \Psi)$. The former requires either $\Phi \mathsf{U}^{\leq h} \Psi$ or $\mathsf{G}^{\leq h} \Phi$ or both to hold on all future paths, whereas the latter requires to that at least one of $\Phi \mathsf{U}^{\leq h} \Psi$ or $\mathsf{G}^{\leq h} \Phi$ holds on each single path. That is, there can be some paths on which $\Phi \mathsf{U}^{\leq h} \Psi$ holds, but not $\mathsf{G}^{\leq h} \Phi$ and vice versa, as long as one of them is valid on each path. The first expression is much stronger.

Again, we can derive the semantics of the other formulae using only these definitions. In understanding these semantics, it may be helpful to recall some of the plain English interpretations of the formulae in the introduction of this section.

Another way of illustrating the semantics of these operators, and illustrating in which states the state operators (i.e. $\mathsf{E}\,\varphi$ and $\mathsf{A}\,\varphi$) are valid is by taking an unwound automaton. This is illustrated in figure 1.6.

1.6 Probabilistic CTL

Probabilistic Computational Tree Logic (PCTL) is a variant of CTL. The main difference is that the universal A and existential E path qualifier are replaced by one probabilistic



a EF *black* **b** AF *black* **c** E(*grey* U *black*)

d EG *black* **e** AG *black* **f** A(*grey* U *black*)

Figure 1.6 Different examples of unwound transition structures and the validity of basic CTL formulae.

operator $\mathcal{P}_{\trianglelefteq p}(\varphi)$. In a sense, this is a generalization of A and E operators, but in fact, they are not comparable. If we recall the example of the mouse in the maze in section 1.4.4 on page 17, we see that there is a possibility that the mouse will never see freedom, because it will, for example, always stay in the starting cell. So, AFf (see figure 1.4b) does not hold in the starting state. However, the probability that the mouse will escape the maze is exactly equal to 1, since the set of paths in which the mouse stays in the maze is a null set, that is, its measure is zero. Or, more informally, the probability it will never escape is negligible. Indeed, this can be gauged by considering the fact that the mouse moves randomly and once it has escaped, it will never come back. One can see that is very unlikely the mouse will not escape into freedom sooner or later. One can also look at the coin flipping example in figure 1.5b. Here it is in theory possible one will never flip heads, so EG *tails* is valid, but the probability of never flipping tails, assuming we use a fair coin, if you keep trying is very small. In fact, it is equal to $\frac{1}{2^n}$ after n throws. Which is, for $n = 40$, less than a trillionth (10^{-12}).

As mentioned, Hansson & Jonsson (1994) introduced PCTL. As they explain in their introduction, it is a logic for reasoning about properties such as 'after a request for service, there is at least a 98 percent probability that the service will be carried out within 2 seconds.' For many real life applications one cannot guarantee that, for example, a packet transmitted over a network will always arrive. A packet can always get lost because another computer tried to send a packet over the physical link at the same moment. In a supermarket it is hard to guarantee that there will never be more than three people waiting in line; it might happen that a bus with tourists stops who all want to buy something. This is where probabilistic modelling comes into play.

1.6.1 Syntax and semantics

PCTL keeps most of the syntax of CTL, except for the A and E operator, and it adds the following state-formula (Cf. definition 1.5.1 on page 20).

Definition 1.6.1. The state-formulae of PCTL are those of CTL except for A and E, but including:

- $\mathcal{P}_{\trianglelefteq p}(\varphi)$ where $p \in [0,1]$ is a probability, $\trianglelefteq \in \{<, \leq, >, \geq\}$ and φ is a path operator.

The semantics for the probabilistic operator are as follows: syntax

$$s \vDash \mathcal{P}_{\trianglelefteq p}(\phi) \qquad \text{iff} \quad \Pr\{\sigma \mid \sigma \in \mathit{Paths}(s, \varphi)\} \trianglelefteq p.$$

We can again express other operators in terms of the $\mathsf{U}^{\leq h}$ and $\mathsf{W}^{\leq h}$ operator. For example, we still have the following identities:

$$\mathcal{P}_{\trianglelefteq p}(\mathsf{F}^{\leq h}\,\Phi) = \mathcal{P}_{\trianglelefteq p}(\mathsf{tt}\;\mathsf{U}^{\leq h}\,\Phi)$$
$$\mathcal{P}_{\trianglelefteq p}(\mathsf{G}^{\leq h}\,\Phi) = \mathcal{P}_{\trianglelefteq p}(\Phi\;\mathsf{W}^{\leq h}\,\mathsf{ff}).$$

And furthermore also the two following identities hold, which show the close connection between the until and unless operator:

$$\mathcal{P}_{\geq p}(\Phi\;\mathsf{W}^{\leq h}\,\Psi) \equiv \mathcal{P}_{\leq 1-p}((\Phi \wedge \neg\Psi)\;\mathsf{U}^{\leq h}\,(\neg\Phi \wedge \neg\Psi))$$
$$\mathcal{P}_{\geq p}(\Phi\;\mathsf{U}^{\leq h}\,\Psi) \equiv \mathcal{P}_{\leq 1-p}((\Phi \wedge \neg\Psi)\;\mathsf{W}^{\leq h}\,(\neg\Phi \wedge \neg\Psi)).$$

The main focus of this thesis, as mentioned previously, is the $\mathsf{U}^{\leq h}$ in connection with the $\mathcal{P}_{\leq p}$ operator. We see from this identity that this enables us also to handle cases in which a $\mathcal{P}_{\geq p}$ operator along with a $\mathsf{W}^{\leq h}$ operator features. We do not discuss the next operator X in a probabilistic setting, nor do we discuss the case of $\mathcal{P}_{\geq p}(\Phi\;\mathsf{U}^{\leq h}\,\Psi)$. A discussion of the latter can be found in the article by Han & Katoen (2007).

Finally we recall remark 1.5.4 and equation 1.1; because the validity of a until formula can be witnessed by a finite path, we use the cylinder set to define the set of infinite paths which all start with this finite path fragment.

Counterexamples for PCTL

2

One of the core applications of the logics introduced in the previous chapter lies within the field of *model checking*. The name model is slightly ambiguous. It has a precise mathematical meaning in the context of modal logics, namely as a triple consisting of a set of states, a binary relation over this set of states and a valuation function; we have already seen this structure thinly disguised as a DTMC. Another meaning of model is 'abstraction of an underlying system'. Both coincide.

In the field of computer science, model checking is used to check whether a piece of software or hardware works or whether a protocol works according to its (formal) specifications. But, in order to check this, we first have to derive a model of the software or the protocol. Usually this involves abstracting or generalising certain aspects which reduce the complexity and the size of the model. For example, when modelling the mouse in the maze (See section 1.4.4 on page 17), we assumed the mouse would move randomly in the maze, which might be an assumption that is appropriate enough and of course far easier to model than intelligent behaviour.

The result of these abstractions, assumptions and generalisations is called the model, which in turn is also a model in the sense that we can verify logical properties on it. In the mouse example, we could try whether $\mathcal{P}_{=1}(\mathit{free})$ holds, i.e., the mouse will surely[1] reach freedom. Verifying such a property is done using model checking.

Only reporting whether a property holds or not however, is not the main strength of model checking. Its strength is the possibility to illustrate by means of an example in which case a property does not hold. Using this information, the software engineer can check the software and hopefully correct the error (or conclude that its model was too coarse). As software and hardware become increasingly complex and increasingly ubiquitous, it becomes more and more important that this software is functioning correctly.

A word processor crash can result in a few hours of work having to be redone, which is annoying, but not life threatening. Bugs in software or hardware developed for medical use however, might be life-threatening. Also, situations in which it is very costly or impossible to send an engineer to repair the software, such as software on board of spaceships, might want to verify their software in advance by checking models of it.

[1] Actually, *almost* surely. The scenario's for which it does not reach freedom are negligible, or, mathematically speaking, form a *null set*.

Much effort has therefore been applied to developing techniques that can find counterexamples. In some a single path, suffices. Suppose one would state, in the maze example, that the mouse could never reach the cheese, or expressed in CTL $\mathsf{AG}(\neg \mathit{cheese})$, one could falsify this by giving the path $\mathit{start} \cdot c_1 \mathit{cheese}$ as a counterexample. Sometimes, we need an infinite path, for example, if we would state $\mathsf{AF}(\mathit{free})$. This is not true, because $(\mathit{start})^\omega$, that is, the mouse will only stay in the top left cell, is a possibility.

For our case of Markovian models, many tools, such as PRISM (Kwiatkowska *et al.*, 2002) and MRMC (Katoen *et al.*, 2005) have been developed. These tools can be used to verify whether a Markovian model satisfies a property or not, but they do not provide the user with a counterexample if the property does not hold. In the remainder of this chapter we shall expound how PCTL counterexamples look like and how they can be found. As it turns out, we will usually require a *set* of paths to refute a property, instead of only a single one.

We will start by introducing and defining some terminology in the first section. The second section will outline the procedure to actually find the counterexamples, and the transformation on the DTMC we need to apply. Finally it will discuss appropriate algorithms to solve the shortest and k-shortest path problems we encounter when looking for counterexamples, depending on whether the property we want to check involves a bounded or unbounded U operator. We start with the unbounded case, which is easier, and proceed to the unbounded case afterwards.

It turns out that problems involving a bounded operator can actually be solved by transforming the model so that the bound becomes irrelevant. This powerful transformation also serves to elucidate some of the choices we make in adapting the first versions of the algorithm to suit the bounded case.

2.1 Evidences and counterexamples

As has been mentioned in the first chapter, this thesis mainly focuses on (counterexamples for) the $\mathsf{U}^{\leq h}$ operator. Most material in this section has previously been published by Han & Katoen (2007).

Before progressing further we shall first formally define what actually constitutes a counterexample in the context of PCTL, and then give an example.

To define a counterexample we can simply follow the semantics. Let us consider $\mathcal{P}_{\leq p}(\varphi)$ where $0 < p < 1$ – we handle the case where $p \in \{0, 1\}$ separately, and $\varphi = \Phi \,\mathsf{U}^{\leq h}\, \Psi$. If $\mathcal{P}_{\leq p}(\varphi)$ is not valid in a state s of a DTMC then:

$$\begin{aligned} & s \not\models \mathcal{P}_{\leq p}(\varphi) \\ \text{iff} \quad & \text{not } \Pr\{\sigma \mid \sigma \in \mathit{Paths}(s, \varphi)\} \leq p \\ \text{iff} \quad & \Pr\{\sigma \mid \sigma \in \mathit{Paths}(s, \varphi)\} > p \\ \text{iff} \quad & \Pr\{\sigma \mid \sigma \in \mathit{Paths}_{\min}(s, \varphi)\} > p. \end{aligned}$$

Hence, $\mathcal{P}_{\leq p}(\varphi)$ is not valid on a state s if the total probability mass of all φ-paths starting in s exceeds p. Because we can witness the validity of an until formula by a

finite sequence of states, *finite* paths suffice in our counterexample. This also explains the last equivalence in the equation above, because any path on which φ is valid, will give a minimal path on which φ is valid by taking a suitable prefix. Different paths will share the same prefix, and every minimal path can be extended in any way possible, and still remain a path on which φ is valid.

Differently expressed, the equivalence kernel of $Pref_{min}$ induces an equivalence relation $\sim^{\min}_{\models}$ on $Paths(s, \varphi)$, where $\sigma_1 \sim^{\min}_{\models} \sigma_2 \equiv Pref_{min}(\sigma_1, \varphi) = Pref_{min}(\sigma_2, \varphi)$. This in turn partitions $Paths(s, \varphi)$ in equivalence classes, where the probability mass of each equivalence class equals the probability of the shared prefix.

We therefore only need to look for enough *finite* paths that satisfy φ in order to find a counterexample. We call every path contributing to such a violation an *evidence*

Definition 2.1.1. An evidence for violating $\mathcal{P}_{\leq p}(\varphi)$ in state s is a path $\sigma \in Paths_{min}(s, \varphi)$.

Obviously, some paths will be more probable than others, and if we are to look actively for a violation, paths with a larger probability mass are most helpful. This motivates the following definition:

Definition 2.1.2. An *evidence* σ for violating $\mathcal{P}_{\leq p}(\varphi)$ in a state s is a strongest evidence if for any other evidence σ' it holds that:

$$\Pr\{\sigma\} \geq \Pr\{\sigma'\}$$

An evidence for violating $\mathcal{P}_{\leq p}(\varphi)$ dually is a witness for satisfying $\mathcal{P}_{>p}(\varphi)$, and a strongest evidence thus also is a strongest witness.

However, a sole evidence usually does not form a counterexample, because its probability mass will lie far below the bound. A counterexample will be a *set* of evidences:

Definition 2.1.3. A *counterexample* for $\mathcal{P}_{\leq p}(\varphi)$ in state s is a set C of evidences, such that $C \subseteq Paths_{min}(s, \varphi)$ and $\Pr(C) > p$.

We will denote the number of evidences in a counterexample C by $|C|$. A counterexample C combined with another finite path $\sigma \in Paths_{min}(\sigma, \varphi)$, such that $\sigma \notin C$, is of course still a counterexample. Usually, we shall be more interested in counterexamples with fewer paths, because they are easier for the programmer to interpret and understand. The following definition formalises what we mean by a minimal counterexample:

Definition 2.1.4. A counterexample C for $\mathcal{P}_{\leq p}(\varphi)$ in s is a *minimal* counterexample if for any other counterexample C' for $\mathcal{P}_{\leq p}(\varphi)$ $|C| \leq |C'|$.

Note that the minimal counterexample need not be unique. There can be any number of counterexamples with the same number of evidences. Among these however, some will have a greater probability mass than others, the one with the largest probability mass will be called a *smallest counterexample*.

Definition 2.1.5. A counterexample C for $\mathcal{P}_{\leq p}(\varphi)$ in s is a *smallest counterexample* if it is both minimal and $\Pr(C) \geq \Pr(C')$ for any minimal counterexample C' for $\mathcal{P}_{\leq p}(\varphi)$.

Intuitively, the smallest counterexample is most indicative. It differs by a maximum amount from the desired probability bound, given that it has a minimal number of paths.

Remark 2.1.6. Any smallest counterexample contains a strongest evidence. Suppose C would be a smallest counterexample and not contain a strongest evidence, then it would could create a new counterexample with a higher probability mass by replacing any evidence in C by the strongest evidence. The number of paths stays the same, but its probability mass increases, so that would mean C was not a smallest counterexample after all.

Remark 2.1.7. For now it suffices to think of a smallest counterexample as a collection of paths. Whether these paths share some common prefix, or whether they are all disjunct does not matter; we solely use the number of paths to determine what constitutes a smallest counterexample. However, in practice, a counterexample with an infinite number of paths, which all obey a simple pattern, such as $s(s_1)^* t$, might well be easier to understand than a finite, but very long list, of disjunct paths. Chapters 4 and 5 will discuss this in-depth.

Lemma 2.1.8. *A smallest counterexample C for $s \vDash \mathcal{P}_{\leq p}(\varphi)$ is finite.*

Proof. A smallest counterexample is, by definition, a minimal counterexample, hence, it has a minimal number of paths. Let us assume that C is a counterexample for $s \vDash \mathcal{P}_{\leq p}(\varphi)$. If $|C| < \infty$, then there is nothing left to prove, since then for sure the smallest counterexample will be finite. So, suppose $|C|$ is infinite; we shall prove that this cannot be a minimal counterexample, and by implication, not a smallest.

Since C is a counterexample $\sum_{\sigma \in C} \Pr\{\sigma\} = \Pr(C) = q > p$. Furthermore, since $0 \leq \Pr(C) \leq 1$, by definition, and $\Pr(C) = \sum_{\sigma \in C} \sigma$, we have that this is a convergent series with non-negative terms so the order of summation is irrelevant. Therefore, we assign some arbitrary but fixed order to the elements of $C = \{\sigma_1, \sigma_2, \dots\}$, so the partial sum S_n of C is:

$$S_n = \sum_{i=1}^{n} \sigma_i$$

We now know that for every $\varepsilon > 0$ there is an integer N such that for all $n \geq N$: $q - S_n \leq \varepsilon$, by the definition of a convergent series. If we pick $\varepsilon = q - p > 0$, since $q > p$, this N is guaranteed to exist. Hence, the set $\{\sigma_1, \dots, \sigma_N\}$ also forms a counterexample, so C cannot be a minimal counterexample, let alone a smallest counterexample. □

Remark 2.1.9. The above proof also shows that the same reasoning cannot hold for $\mathcal{P}_{<p}(\varphi)$, in that case a counterexample may indeed contain infinitely many paths summing exactly to p, because the sum of the per-path-probabilities forms a convergent series. An example is shown in figure 2.1, where the counterexample for $\mathcal{P}_{<\frac{1}{2}}(\mathsf{F}\, a)$ consists of all paths that traverse the self-loop at s zero or more times, and end in t. There are infinitely many of those paths, and all are needed.

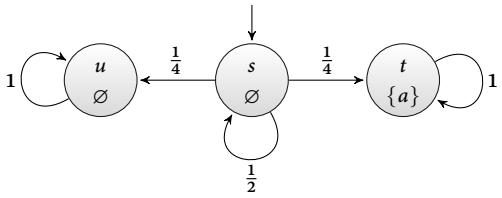


Figure 2.1 A DTMC with an infinite smallest counterexample for $\mathcal{P}_{<\frac{1}{2}}(\mathsf{F}\, a)$

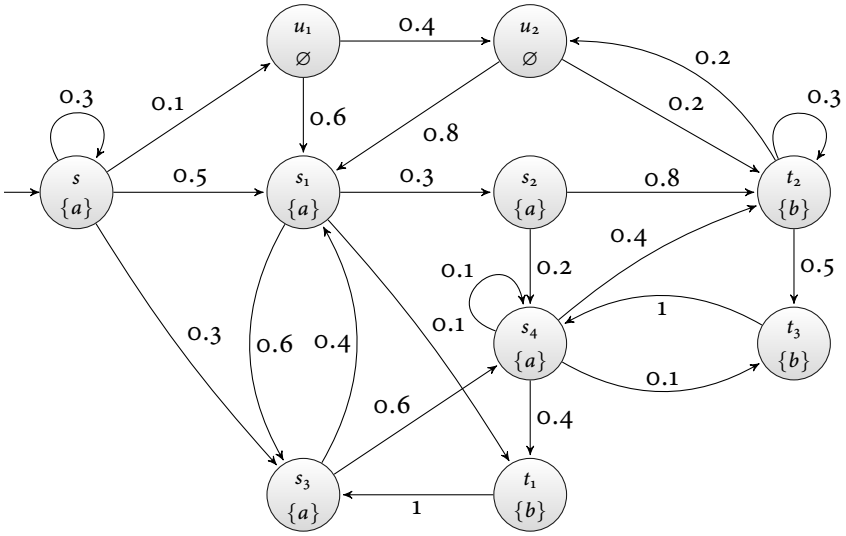


Figure 2.2 An example DTMC, for which $s \not\models \mathcal{P}_{\leq 0.27}(a \mathsf{U}\, b)$.

Example 2.1.10. Figure 2.2 shows a simple DTMC with ten states. The initial state is indicated by the arrow, and is s. The set of atomic propositions $AP = \{a, b\}$ and $L(s) = L(s_i) = \{a\}$, for $1 \leq i \leq 4$, $L(t_i) = \{b\}$ for $1 \leq i \leq 3$, and finally $L(u_1) = L(u_2) = \varnothing$. There are no absorbing states in this DTMC.

Let us define the following paths:

1. $\sigma_1 = s{\cdot}s_1{\cdot}s_2{\cdot}t_2$, with $\Pr\{\sigma_1\} = 0.12$;
2. $\sigma_2 = s{\cdot}s_1{\cdot}s_3{\cdot}s_4{\cdot}t_1$, with $\Pr\{\sigma_2\} = 0.072$;
3. $\sigma_3 = s{\cdot}s_1{\cdot}s_3{\cdot}s_4{\cdot}t_2$, with $\Pr\{\sigma_3\} = 0.072$;
4. $\sigma_4 = s{\cdot}s_3{\cdot}s_4{\cdot}t_1$, with $\Pr\{\sigma_4\} = 0.072$;
5. $\sigma_5 = s{\cdot}s_3{\cdot}s_4{\cdot}t_2$, with $\Pr\{\sigma_5\} = 0.072$;
6. $\sigma_6 = s{\cdot}s_3{\cdot}s_4{\cdot}t_3$, with $\Pr\{\sigma_6\} = 0.018$.

Suppose we are looking for a counterexample for $s \vDash \mathcal{P}_{\leq 0.27}(a \mathsf{U}\, b)$. We define the following sets:

1. $C_1 = \{\sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6\}$, with $\Pr\{C_1\} = 0.306$;

2. $C_2 = \{\sigma_2, \sigma_3, \sigma_4, \sigma_5\}$, with $\Pr\{C_2\} = 0.288$;
3. $C_3 = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, with $\Pr\{C_3\} = 0.336$.

Clearly, each of these sets is a counterexample. C_1 however, is clearly not a minimal counterexample, since it consists of five paths, and C_2 only of four. C_2 is minimal, since removing any paths from C_2 would not make it a counterexample; but since $|C_3| = |C_2|$ and $\Pr(C_3) > \Pr(C_2)$, C_2 is not a smallest counterexample. C_3 however is a smallest counterexample, and therefore contains the strongest evidence, σ_1.

Note however, that the smallest counterexample is not unique, we could, for example, remove σ_4 and replace it by σ_5, which has the same probability, and obtain another counterexample with probability 0.336. Indeed, we could take σ_1 and any three paths from C_2 and have a smallest counterexample. This shows that if different paths have the same probability, the counterexample need not be unique.

Hitherto we have defined what counterexamples are, and we have shown an example, but we shall now proceed to explain how one can systematically find a Strongest Evidence (SE) and a Smallest Counterexample (SC).

2.2 Conversion of the DTMC

The main idea is to transform the problem of the strongest evidence and the smallest counterexample to a k-shortest path problem, which can be solved by well-known algorithms. The first step involves the introduction of a target state and modifications to the structure of the DTMC based on the formula to be refuted. The second step involves the conversion of the edge weights. The procedure outlined here has previously been published by Han & Katoen (2007).

2.2.1 Adaptation of the DTMC

The way the DTMC is adapted, is intimately related to algorithms used for CTL model-checking, because these algorithms are also applicable to the equal part of PCTL such as the boolean operators. Appendix A contains a short outline of these procedures, and the curious reader may read it.

This leaves us with the particular case where the formula under scrutiny is of the form $\mathcal{P}_{\leq p}(\Phi \mathsf{U}^{\leq h} \Psi)$, we use the sub-formulae Φ and Ψ as our basis for the adaptation of the DTMC. We first determine the states in which Φ holds, i.e. $Sat(\Phi) = \{s \in S \mid s \vDash \Phi\}$. Analogously we define $Sat(\Psi)$ to be $\{s \in S \mid s \vDash \Psi\}$. Clearly states that are neither in $Sat(\Phi)$ nor in $Sat(\Psi)$ do not lie on a path that fulfils $\Phi \mathsf{U}^{\leq h} \Psi$. We furthermore notice, that if Ψ holds in a state, every path starting in that state will automatically fulfil $\Phi \mathsf{U}^{\leq h} \Psi$.

We therefore make every state which neither fulfils Φ nor Ψ absorbing, and every state which fulfils Ψ is connected to a new state t with probability 1 – other transitions are removed. Formally:

Let $\mathcal{D} = (S, \mathbf{P}, L)$ be our DTMC, we define a derived DTMC $\mathcal{D}' = (S', \mathbf{P}', L')$, where:

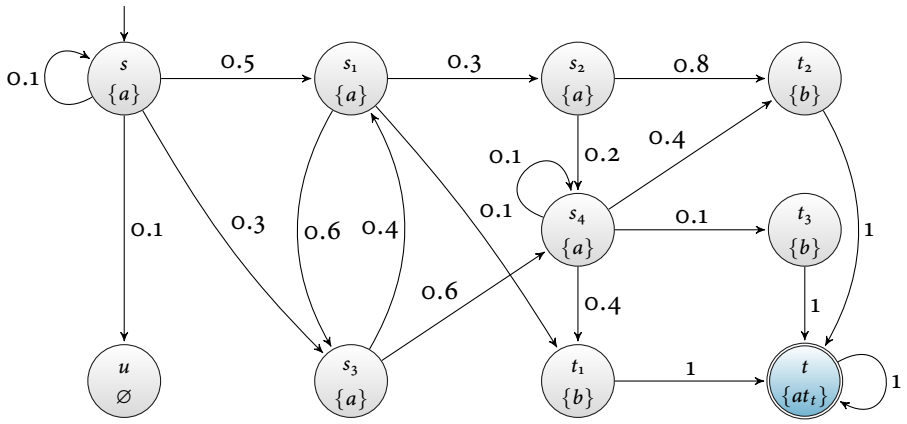


Figure 2.3 The example DTMC after the first transformation step. Note that in addition to this u_1 and u_2 have been collapsed into u.

- $S' = S \cup \{t\}$, where t is new, i.e. $t \notin S$.
- $\mathbf{P}'$ is defined as follows:
  - If $s \notin (Sat(\Phi) \cup Sat(\Psi))$ or $s = t$:

$$\mathbf{P}'(s,s) = 1 \text{ and } \mathbf{P}'(s,s') = 0 \text{ for } s \neq s'.$$

  - If $s \in Sat(\Psi)$:

$$\mathbf{P}'(s,t) = 1 \text{ and } \mathbf{P}'(s,s') = 0 \text{ for } s \neq t.$$

  - Otherwise, i.e. $s \in (Sat(\Phi) - Sat(\Psi))$:

$$\mathbf{P}'(s,s') = \mathbf{P}(s,s') \text{ for } s' \in S$$
$$\mathbf{P}'(s,t) = 0.$$

- $L' = L(s)$ for $s \in S$ and $L'(t) = \{at_t\}$, where at_t is a new label, i.e. $at_t \notin AP$.

If $Sat(\Phi)$ and $Sat(\Psi)$ are available, which is normally the case with a bottom-up model checking algorithm, the construction of $\mathcal{D}'$ requires $\mathcal{O}(|S|)$ time, if the structure is represented as an adjacency list.

This construction evidently preserves the validity of $\Phi \, U^{\leq h} \, \Psi$ on the underlying structure. We also see that $\Phi U^{\leq h} \Psi$ will hold on every path ending in t; more specifically, if $\sigma = s_1 \cdots s_n \cdot t$, $\Phi \, U^{\leq h} \, \Psi$ will also hold on $\hat{\sigma} = s_1 \cdots s_n$ where $\hat{\sigma}$ and σ are equally probable.

Example 2.2.1. The result of the first step of the transformation of the DTMC in figure 2.2 is shown in figure 2.3. Note that the two states u_1 and u_2, which both become absorbing, are replaced by a single state u. Our construction does not do this by default, although this could be done in general.

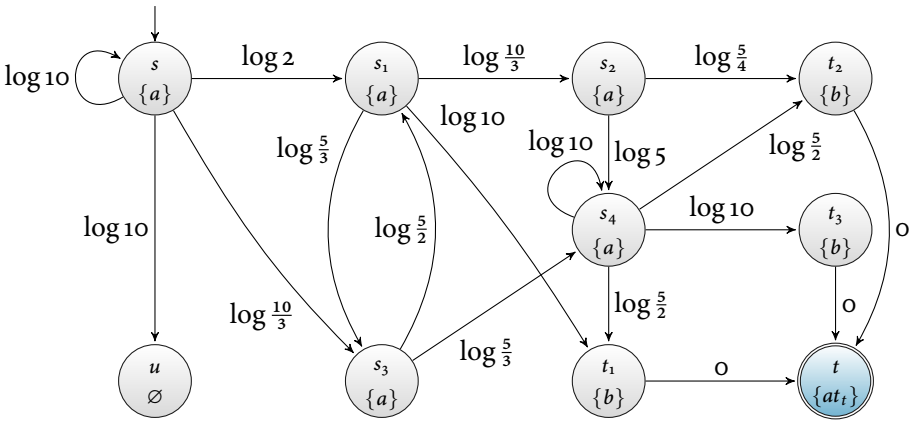


Figure 2.4 The example DTMC after the second transformation step. The probabilities have been replaced by weights and the self-loop on the final state has been removed.

2.2.2 Conversion to a weighted digraph

In the next step, we shall convert the DTMC to a weighted directed graph, *digraph* for short, such that the path with the lowest weight in the digraph will correspond to the path with the highest probability in the DTMC. Before proceeding, we first provide the definition of a digraph.

Definition 2.2.2. A weighted *digraph* $\mathcal{G}$ is an ordered triple (V, E, w) where:

- V is a set, whose elements are called *nodes*, or *vertices*.
- E is a set of ordered pairs of vertices. Its elements are called *arcs* or *edges*. If $(u, v) \in E$, we say the edge goes *from* u *to* v.
- w is a function $w : E \to \mathbb{R}$, mapping every edge on to a real number, its *weight*.

If $(u, v) \in E$, we call v the *successor* of u, and we call u the predecessor of v. Edge weights can be negative, but we shall assume they are positive unless stated otherwise.

A DTMC $\mathcal{D} = (S, \mathbf{P}, L)$is as follows transformed to a weighted digraph $\mathcal{G}_\mathcal{D} = (V, E, w)$:

- $V = S$; the vertex or node set.
- $(v, v') \in E$ iff $\mathbf{P}(v, v') > 0$; the edge set.
- $w(v, v') = \log\left(\frac{1}{\mathbf{P}(v,v')}\right)$; the weight function.

Finally, remove the self-loop on the target state, because it otherwise would cause an infinite number of shortest paths to t. This construction, which involves recalculating the weight of every edge, can be done in $\mathcal{O}(m)$ time, where $m = |\mathbf{P}|$, that is, the number of non-zero elements in $\mathbf{P}$ if the graph is represented as an adjacency list.

Example 2.2.3. The result of the second step of the transformation of the DTMC in figure 2.2 is shown in figure 2.4. Note that the self-loop on the final state is removed.

Note that, if $\mathbf{P}(s, s') > 0$, then $w(s, s') \in [0, \infty)$. The sum of the weights of the edges on a path is called its *weight* or *length*.

Definition 2.2.4. The *length* ℓ of a finite path $\sigma = v_0 v_1 \cdots v_n$ in a weighted graph $\mathcal{G}$:

$$\ell(\sigma) = \sum_{i=0}^{j-1} w(v_i, v_{i+1}).$$

Depending on the context we either use the term *weight*, which is the most general, or *length*, which sounds more appropriate in the context of a shortest path algorithm. As such, a shortest path is defined as a path of minimum length between two nodes, cf. definition 2.3.1.

The following states that the conversion indeed transforms the most probable paths to the shortest or lowest weight paths:

Lemma 2.2.5. *Let σ and σ' be finite paths in a DTMC $\mathcal{D}$ and its corresponding weighted digraph $\mathcal{G}_{\mathcal{D}}$, then:*

$$\Pr\{\sigma'\} \geq \Pr\{\sigma\} \text{ iff } \ell(\sigma') \geq \ell(\sigma)$$

.

The proof uses the well-known fact that $\log(a \cdot b) = \log(a) + \log(b)$ and the fact that $\log(\cdot)$ is monotonic.

Proof. Let $0 < p_1, p_2, q_1, q_2 \leq 1$ and suppose $p_1 p_2 \leq q_1 q_2$. Then: $\frac{1}{p_1 p_2} \geq \frac{1}{q_1 q_2}$ and by the monotonicity of log and because $\log(ab) = \log(a) + \log(b)$, we obtain:

$$\log\left(\frac{1}{p_1}\right) + \log\left(\frac{1}{p_2}\right) \geq \log\left(\frac{1}{q_1}\right) + \log\left(\frac{1}{q_2}\right).$$

Applying this (repeatedly) to a path σ trivially obtains the desired result. □

Remark 2.2.6. Note that the probability-weight mapping from $(0, 1] \to \mathbb{R}^+$ is a bijective mapping. So given the weight w of a path in the digraph, the probability mass of the corresponding path in the DTMC is e^{-w}. Calculating the weight of the shortest path thus gives us directly the probability mass in the DTMC.

To conclude this section, we give the following lemma which states the equivalence between shortest paths and most probable paths.

Lemma 2.2.7. *For any path σ from s to t in a DTMC $\mathcal{D}$, $k > 0$ and $h \in \mathbb{N} \cup \infty$: σ is a k-th most probable path of h hops in $\mathcal{D}$ iff σ is a k-th shortest path of h hops in $\mathcal{G}_{\mathcal{D}}$.*

Proof. The proof follows directly from lemma 2.2.5. □

Remark 2.2.8. In case a graph contains two paths with the same length, the k shortest path problem, or even the shortest path problem in the cases these are the shortest paths, does not have a unique solution.

2.3 Finding the strongest evidence

Now that the Strongest Evidence (SE) problem has been reduced to a Shortest Path (SP) problem, we can use well-known shortest path algorithms to find the SE. We shall first discuss the unbound case, after which we discuss a reduction from the bound to the unbound case and various algorithms for the bounded version of the until-operator, including lower and upper bounds.

2.3.1 Unbounded until

For the case of the unbounded until, we simply need to find the shortest path in $\mathcal{G}_{\mathcal{D}}$ from the initial state to t. In principle, any shortest path algorithm suffices, since the weights on the edges are non-negative.

Definition 2.3.1. Given a weighted graph $\mathcal{G} = (V, E, w)$ and two nodes $s, t \in V$, the *shortest path* SP problem is to find a path σ from s to t such that $\ell(\sigma) \leq \ell(\sigma')$ for any path σ' from s to t in $\mathcal{G}$.

We have seen that the transformation of a DTMC to a weighted digraph can be done in $\mathcal{O}(n + m)$ time, where n is the number of nodes and m the number of edges. If we use Dijkstra's algorithm (Dijkstra, 1959), of which it is well known it has a time complexity of $\mathcal{O}(m + n \log n)$, provided Fibonacci heaps (Fredman & Tarjan, 1987) are used in the implementation.[2]

We can therefore conclude:

Theorem 2.3.2. *The SE problem for unbounded until is in PTIME.*

2.3.2 Bounded until

First of all, we assume, in this section that the hop bound h is less than the number of nodes n in the graph. If it is at least n, we can treat the case as an unbounded case, because a shortest path can at most visit every node in the graph, and hence have length $n - 1$; if it visits a node twice, this indicates a cycle, which contradicts the fact it is a shortest path.

The problem for bounded until is harder, of course, since we have to take the hop limit into account too. The shortest path may well exceed this hop limit h. We shall first state the definition of this Hop constrained Shortest Path (HSP) problem:

Definition 2.3.3. Given a weighted digraph $\mathcal{G} = (V, E, w)$ with $s, t \in V$, and a hop limit $h \in \mathbb{N}$, the HSP problem is to find a path σ in $\mathcal{G}$ from s to t such that $\ell(\sigma) \leq \ell(\sigma')$ for any other path σ' from s to t with $|\sigma'| \leq h$.

[2] Not so well known is that in the article by Goldberg & Tarjan (1996) the inventor of Fibonacci heaps explains why in practice an implementation with binary heaps perform as well as Fibonacci heaps, due to the fact that the number of times a node is moved in the queue (which should give the Fibonacci heap its advantage) is relatively few in practice. Other operations in binary heaps are absolutely speaking, that is in number of clock cycles, faster than Fibonacci heaps.

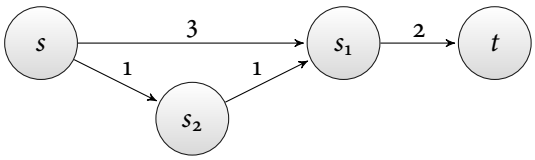


Figure 2.5 A graph in which the optimality principle does not hold for a hop bound of 2.

Remark 2.3.4. This problem can be seen as a special case of a more general problem, namely the resource constrained path problem. This problem is discussed in more detail in Ahuja *et al.* (1993). This is a shortest path problem where a shortest path has to be found fulfilling some other constraints, a far more difficult problem, and even NP-complete, as explained in Garey & Johnson (1979).

For this to work, we need to adapt existing shortest path algorithms to take the hop constraint into account. There are a few caveats, however, which every implementation should be aware of. For a shortest path $\sigma = s{\cdot}s_1 \cdots s_n{\cdot}t$ from s to t in a graph, we have that any prefix of σ is also a shortest path. That is, if $\sigma = s{\cdot}s_1 \cdots s_i$ would not be a shortest path from s to s_i, then we could replace that segment in σ by the shortest path from s to s_i and obtain an even shorter path, contradicting our assumption that σ was a shortest path.

This so called *optimality principle* need not hold in case of the hop constrained shortest path, as figure 2.5 shows. With a hop bound of 2, the shortest path from s to t is $s{\cdot}s_1{\cdot}t$, since going via s_2 would yield a path which exceeds the bound. The shortest path from s to s_1 however, is $s{\cdot}s_2{\cdot}s_1$; and this does not violate the hop bound.

To see why this is an issue, shortest path algorithms usually store the shortest path implicitly, by having each node point to its predecessor, so that afterwards the shortest path can be constructed by going backwards from the target. The above example above shows that this does not work in the hop bound case. So even if all nodes are reachable within two steps, implicitly, paths of more than two steps can be formed. If we simply record the predecessor, then after two steps s_1 will be recorded as being the predecessor of t, and s_2 as being the predecessor of s_1, hence implicitly yielding $s{\cdot}s_1{\cdot}s_2{\cdot}t$. This problem is easily addressed by storing different predecessors for different hop counts.

We therefore store a predecessor *for every hop bound* at every node. In the case of s_1 this would mean that the predecessor for $h = 1$ would be s, but for $h = 2$ it would be s_2. Even though $s{\cdot}s_1$ is not optimal in terms of length, it does allow us to reach node t in two steps, which is not possible through s_2. In general we note that if we have found two paths P_1 and P_2, with respective lengths l_1 and l_2, and respective hop-counts h_1 and h_2 that only if $l_1 \leq l_2$ and $h_1 \leq h_2$, with at least one inequality, P_1 is preferred over P_2. However, if $h_1 < h_2$, but $l_1 > l_2$, then P_1 might still be of interest, because, if extended further, it might reach nodes that are otherwise unreachable.

The introduction of a hop constraint has some important repercussions for the viability of the algorithms that can be successfully used, because Dijkstra's algorithm hinges on the optimality principle: if this does not hold, its strategy to mark nodes as

finalised as soon as they're on front of the queue does not hold up.

Before investigating specialised algorithms that can solve the hop constrained problem immediately, we shall discuss a reduction to the unconstrained problem. The results and analysis of this reduction will greatly help in the understanding of the constrained problem.

2.3.2.1 Reduction to an unconstrained problem

It is possible to reduce the hop constrained version of the problem to an unconstrained problem, by transforming the graph. I have not found a similar procedure in the literature, although I do not expect it is a novelty. This transformation is helpful because it makes explicit what is implicitly done in the adapted versions of the Recursive Enumeration Algorithm (REA) and Bellman-Ford (BF) algorithm. It furthermore helps to explain why, for example, Dijkstra's algorithm performs so badly when modified for a hop constraint.

Although we are mainly interested in graphs which are the results of a DTMC converted to a digraph by the procedure of section 2.2.2, there are no principle objections why this procedure would not work on digraphs in general. As an example of this, and for reasons of conciseness our running example of figure 2.5 is not the result of a converted DTMC.

The procedure is as follows: if we want to solve a shortest path problem between s and t with a hop constraint h on a graph $\mathcal{G}$, we take the original graph and replace it by $h+1$ copies. We number the copies from 1 up to $h+1$. The transitions within a copy are replaced by a transition between copies, more precisely, a transition to the next copy. If the original graph has a transition $s_1 \to s_2$, we replace it by a transition from state s_1 in the first copy to s_2 in the second copy, and also with a transition from s_1 in the second copy to s_2 in the third copy. Every transition thus points to the next copy, except for transitions in the final copy, these are completely removed.

The target state t of the shortest path problem in the original graph, has $h+1$ copies in the new graph. We can extend the graph with one extra node $\hat{t}$ and add edges from every copy of t to $\hat{t}$ with weight 0, thus introducing $\hat{t}$ as the new target state. This way the problem is reduced to a shortest path to a single node $\hat{t}$. As a starting node state s in the first copy is chosen for obvious reasons.

Using the graph of figure 2.5 as a basis, we give a visual depiction of the transformed graph in figure 2.6.

We formalise the outlined procedure using product graphs.

Definition 2.3.5. The tensor or direct product $\mathcal{P} = (V_\mathcal{P}, E_\mathcal{P})$ of two graphs $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ and $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H})$ is defined as:

- $V_\mathcal{P} = V_\mathcal{G} \times V_\mathcal{H}$, where $\times$ denotes the Cartesian product.
- Let $u, v \in V_\mathcal{G}$ and $\hat{u}, \hat{v} \in V_\mathcal{H}$, then $((u, \hat{u}), (v, \hat{v})) \in E_\mathcal{P}$ if and only if $(u, v) \in E_\mathcal{G}$ and $(\hat{u}, \hat{v}) \in E_\mathcal{H}$; in other words, $(u, \hat{u})$ and $(v, \hat{v})$ are adjacent in $\mathcal{P}$ if and only if u and v are adjacent in $\mathcal{G}$ and $\hat{u}$ and $\hat{v}$ are adjacent in $\mathcal{H}$.

We write $\mathcal{P} = \mathcal{G} \times \mathcal{H}$ to denote the product.

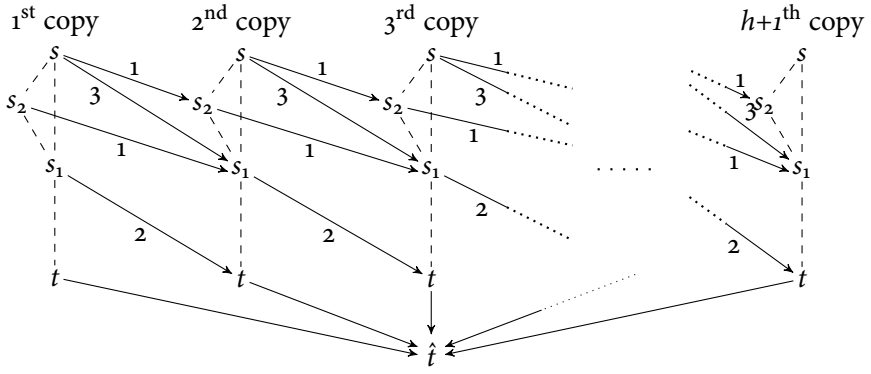


Figure 2.6 Schematic representation of the result of the graph transformation. The original graph is replaced by $h + 1$ copies. The original edges are removed (shown dashed) and instead a new edge is inserted, to the corresponding successor in the next copy, e.g. $s \to s_1$ in the first copy is replaced by a transition from s in the first copy to s_1 in the second copy, and so one. Finally all copies of the original target state t are connected to a new target state $\hat{t}$.

The tensor product almost gives us the complete transformation, with the exception of the addition of the extra target state and the definition of the weights:

Definition 2.3.6. The reduction of a shortest path problem with start node s and end node t on a weighted graph $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G}, w_\mathcal{G})$ with hop constrained h to a shortest path problem without a hop constraint on a weighed graph $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T}, w_\mathcal{T})$ with start node (s, q_0) and end node $\hat{t}$ is defined as follows. Assume $\mathcal{G}_q = (V_{\mathcal{G}_q}, E_{\mathcal{G}_q})$ represents the path graph with $h+1$ nodes, labelled $q_0, q_2, \ldots, q_h$, that is $\mathcal{G}_q$ looks likes $q_0 \to q_1 \to \cdots \to q_h$. The intuitive interpretation of the index could be 'the number of steps taken'.

- Let $\mathcal{G}' = (V_\mathcal{G}, E_\mathcal{G})$ then $(V'_\mathcal{T}, E'_\mathcal{T}) = \mathcal{G}' \times \mathcal{G}_p$. $V_\mathcal{T} = V'_\mathcal{T} \cup \hat{t}$ and $E_\mathcal{T} = E'_\mathcal{T} \cup \{((t, q_1), \hat{t}), \ldots, ((t, q_n), \hat{t})\}$.
- $w_\mathcal{T}((s_x, q_i), (s_y, q_j)) = w_\mathcal{G}(s_x, s_y)$, with $s_x, s_y \in V_\mathcal{G}$; $w_\mathcal{T}((t, q_i), \hat{t}) = 0$.

Theorem 2.3.7. *The shortest path problem from s to t with hop constraint h on a graph $\mathcal{G}$ can be reduced to a shortest path problem on a graph $\mathcal{T}$, obtained by applying the graph transformation of definition 2.3.6. The counterpart of a path from (s, q_0) to $\hat{t}$ in $\mathcal{T}$ is a hop constrained shortest in $\mathcal{G}$ path if and only if it is a shortest path in $\mathcal{T}$.*

Proof. Assume the nodes in $\mathcal{G}$ are labelled $s, s_1, s_2, \ldots s_n, t$, where s and t are the nodes between which we want to find the hop constrained shortest path. Also assume the nodes in path graph are labelled $q_0, q_1, \ldots, q_h$. The extra target node wherewith the product graph is extended is called $\hat{t}$.

The counterpart of a path $(s, q_0)\cdot(s_x, q_1)\cdots(t_y, q_n)\hat{t}$ in $\mathcal{T}$ in $\mathcal{G}$ is simply $s\cdot s_x \cdots t$. Vice versa, a path $s\cdot s_x \cdots s_y \cdot t$ has a counterpart in $(s, q_0)\cdot(s_x, q_1)\cdots(s_y, q_{i-1})\cdot(t, q_i)\cdot\hat{t}$. Remember that we are only interested in hop constrained (shortest) paths from a specific s to a specific t; in our transformed graph this translates from a path from

(s, q_0) to $\hat{t}$ via some (t, q_i), with $0 \leq i \leq h$ – it does not matter which copy of the target state.

That the counterparts exist follows directly from the construction. There is an edge $(s_x, q_i) \to (s_y, q_{i+1})$ if and only if s_x and s_y are adjacent in $\mathcal{G}$, q_i and q_{i+1} are adjacent because of the shape of the path graph.

Note that in particular the length of a path and its counterpart are equal, because of the definition of the weight function. The added edges in the transformed graph have weight zero.

If we have a shortest path, obeying the hop constraint, from s to t in the original graph, we can find the counterpart in the transformed graph. Suppose this is not the shortest path in the product graph from s to $\hat{t}$, yet in this case the shorter path would have a counterpart in the original graph, with the same length, this is absurd.

For the other direction: Suppose we have found a shortest path in the transformed graph from s to $\hat{t}$, we can find the counterpart in the original graph: this counterpart surely obeys the hop constraint, due to the construction of the transformed graph. Suppose however that the original graph has a shorter path: this is possible, but if this path in addition satisfies the hop constraint, then the transformed graph should have a shorter path, which is again absurd. □

The time complexity of the graph transformation is $\mathcal{O}(hn + hm)$ time, m being the number of edges in the original graph, and n being the number of nodes. We assume the data structure involves a list of nodes and an adjacency list per node. We have to make $h + 1$ copies of every node, and give these a unique name; then we have to place hm edges between them, finally we have to add another node, which can be done in $\mathcal{O}(1)$ time, and add $h + 1$ edges to this new target state. With the above transformation, we trivially obtain:

Corollary 2.3.8. *A hop constraint shortest path problem, and by implication the strongest evidence problem for bounded until is an unconstrained shortest path problem.*

With the graph transformation, which has polynomial time complexity, we can run a shortest path algorithm, which also has polynomial time complexity, and we can therefore conclude:

Theorem 2.3.9. *The SE problem for bounded until is in PTIME.*

The crux of the transformation is that every node has been duplicated h times. Remember that the optimality principle does not hold in this problem, and that the optimal route from a node to the target can be different depending on the number of hops used to arrive at the node; in the product graph the number of hops used to arrive at a node, and hence the number that can still be spent is encoded in the state, and the shortest path algorithm need not take care of it.

The most important aspect of this transformed graph however, has not been mentioned yet, viz. it is acyclic. Even if the original graph contains a cycle, the transformed graph does not, because it is constructed via a product with an acyclic graph. A cycle in a tensor product always indicates a cycle in both of the original graphs.

Algorithm 2.3.1 Typical version of Bellman-Ford's shortest path algorithm, unconstrained version.

Require: $\mathcal{G} = (V, E)$ is a graph with no negative weight cycles, $s \in V$
Ensure: $\forall v \in V$: distance$[v]$ is the length of the shortest path from s to v and the predecessor of v on this path is in previous$[v]$.

```
1:  for each v ∈ V:
2:      distance[v] := ∞
3:      previous[v] := ⊥
4:  distance[s] := 0
5:  previous[s] := ∅
6:  repeat |V| - 1 times:
7:      for each (u, v) ∈ E:
8:          d = distance[u] + weight(u, v)
9:          if d < distance[v]:
10:             distance[v] := d
11:             previous[v] := u
```

This means the graph we have constructed is a Directed Acyclic Graph (DAG). Because DAGs are known to have a topological ordering, and since it is well-known the shortest path of graphs with a topological ordering can be computed in linear time, we can use this on the product graph, first however, we shall discuss the modification of existing algorithms.

2.3.2.2 Hop constrained Bellman-Ford

The most obvious algorithm to adapt, in order to solve the HSP problem, is the Bellman-Ford (BF) algorithm (Bellman, 1958), since it proceeds by increasing hop count. For discussion purposes, a typical implementation of the BF algorithm has been provided in algorithm 2.3.1. This version uses an implicit path representation by means of array previous[].

Starting in s, after n iterations the algorithm will have found all nodes reachable within at most n steps. In the adapted algorithm we need to make sure that the implicit path representation functions correctly. Because even if all nodes are reachable within two steps, implicitly, paths of more than two steps can be formed, as we saw in the example of figure 2.5. We therefore duplicate the arrays distance[] and previous[] for every hop count, as shown in algorithm 2.3.2.

Looking closely at the constrained algorithm, we can see that for any specific number of hops h distance$[v][h]$ will in fact contain the length of a path from s to v with *exactly* h hops, or ∞ if no such path exists. It therefore calculates at most h shortest paths to a node. Usually we are only interested in the shortest only. This can be found by selecting $\min_h$ distance[v][h], and the corresponding path can be reconstructed by using previous$[v][h]$. Alternatively we can introduce a variable that

Algorithm 2.3.2 Bellman-Ford's shortest path algorithm, hop constrained version.

Require: $\mathcal{G} = (V, E)$ is a graph with no negative weight cycles, $s \in V$, H_c is a positive integer.

Ensure: $\forall v \in V, \forall h \leq H_c$: the length of shortest path from s to v with exactly h hops is in distance$[v][h]$ and the predecessor of v on this path is in previous$[v][h]$.

```
1:  for each v ∈ V:
2:      ∀0 ≤ h ≤ H_c : distance[v][h] := ∞
3:      ∀0 ≤ h ≤ H_c : previous[v][h] := ⊥
4:  distance[s][0] = 0
5:  previous[s][0] = ∅
6:  for h from 0 up to min{H_c, |V| − 1}:        // Not including H_c or |V| − 1
7:      for each (u, v) ∈ E:
8:          d = distance[u][h] + weight(u, v)
9:          if d < distance[v][h + 1]:
10:             distance[v][h + 1] := d
11:             previous[v][h + 1] := u
```

keeps track of the value of h for which distance[v][h] is minimal.

Referring to our example graph of figure 2.5, we see that this algorithm indeed functions correctly. Suppose we run the algorithm with a hop constraint of 3; when the algorithm ends, we obtain the following results. Here $\{h \to x\}$ means in case of the distance array that for a hop count of h the best distance is x, and similarly for the previous array that the predecessor on this path is x.

	Nodes			
Array	s	s_1	s_2	t
distance	$\{0 \to 0\}$	$\{1 \to 3, 2 \to 2\}$	$\{1 \to 1\}$	$\{2 \to 4, 3 \to 3\}$
previous	$\{0 \to \varnothing\}$	$\{1 \to s, 2 \to s_2\}$	$\{1 \to s\}$	$\{2 \to s_2, 3 \to s_1\}$

We notice that in case of t the smallest distance can be found for $h = 3$, and we can reconstruct this path by going through the previous array, which points to s_1, then to s_2 and finally to s, which does not have a predecessor.

The complexity of this algorithm is easily seen to be $\mathcal{O}(hm)$, since the outer loop is executed at most h times, whereas in each loop every edge will be investigated to see whether an improvement can be found. This confirms the result of theorem 2.3.2.1.

The BF algorithms can be adapted to perform in practice much better than the one presented here; one could for example, in iteration 0 only check the successors of the starting node, put those in a set, and use that set for the next iteration, so one really expands on the previous results. This does not improve the theoretical complexity, since in the worst case, the graph is fully connected. However, if this is not the case, one

can only, each iteration, use those nodes that were updated in the previous iteration as a basis.

Comparison with the product graph approach We have seen two approaches to solve the HSP problem. The first consisted of transforming the graph, after which we could run any (unconstrained) shortest path algorithm, and the second approach consisted of modifying Bellman-Ford.

It is interesting to compare these two approaches, and in particular the case in which we would run an unconstrained BF algorithm on a transformed graph, versus the specialised, constrained algorithm.

If we compare unconstrained version of algorithm 2.3.1, with the constrained version of algorithm 2.3.2 we see that they are very similar, the control structures are similar, the main difference is that the arrays distance and previous in the constrained algorithm have acquired a second index, namely h. If we run the unconstrained algorithm on the transformed graph, we see that this second index h is in fact there, but in that case it is encoded in the nodes of the graph. We explicitly made $h + 1$ copies of every node, so we could run the unconstrained algorithm, the constrained algorithm makes these copies too, but implicitly, by introducing an extra index. This underlines the similarity of both approaches.

We now take a look at the time complexity of both approaches. Viewed naïvely, we could state that running the unconstrained algorithm on our transformed graph gives a time complexity of $\mathcal{O}(hm{\cdot}hn)$, because the transformed graph has $\mathcal{O}(hm)$ edges and $\mathcal{O}(hn)$ nodes. This is a rather rough approximation, and we can see this if we take the acyclic structure of the transformed graph into account. Provided we run an optimised version of the unconstrained BF algorithm on the transformed graph, that is, the version that only checks the successors of the starting node in the first iteration, and in the second iteration only checks the successors' successors and so on, we see that we effectively get a layer by layer exploration in the transformed graph, and we thus get a breadth first exploration. In this case this surely is an improvement, because in this way we again arrive at the $\mathcal{O}(hm)$ complexity, instead of $\mathcal{O}(hm{\cdot}hn)$. Because of the structure we see that if we optimise BF (and hence turn it more or less into a breadth-first exploration) we need at most h iterations, because after the h^{th} iteration the algorithm surely has arrived at $\hat{t}$, which has no successors. Also, during each iteration it can only visit at most n nodes, which can have no more than m successors.

The main difference now is that the product graph is constructed up front, whereas the constrained algorithm can very easily explore the graph lazily, only defining a value previous$[v][h]$ if it is actually used. For large graphs and large hop count h this can be a huge improvement. A different point of view could therefore be that the constrained Bellman-Ford algorithm actually performs the graph-transformation in a lazy manner.

2.3.2.3 Hop constrained Dijkstra

Before I had found the reduction to the unconstrained problem, and the fact that the adapted BF algorithm implicitly explores the transformed graph too, some thinking

had been spent already on adapting Dijkstra's algorithm to handle hop constraints. Although this can be done using a simple transformation of the algorithm, the result will less optimal than Bellman-Ford, which is not surprising if we look at the performance of Dijkstra's algorithm on DAGs.

As we have seen, BF proceeds by increasing hop count, whereas Dijkstra selects the next closest node. In order to be able to select this node quickly, it uses some sort of queue. In case of Dijkstra's algorithm, we also have to take hop counts into account, which leads to an implicit encoding of the transformed graph. We now know that this implicit graph is in fact a DAG. Dijkstra's algorithm is not optimal for these kinds of graphs, because it spends 'too much' time ordering its nodes, whereas as simple Breadth First Search (BFS) suffices[3]. There is no need to spend so much (costly) time on keeping order.

In practice in some specific cases Dijkstra's algorithm might turn out to still be better, because it can reach $\hat{t}$ early, and be able to terminate the algorithm. For the general case however, the complexity is $\mathcal{O}(hm + hn\log(hn))$. We have not tried to find specific cases in which the graph structure favours Dijkstra's algorithm, partly because the DTMCs used do not exhibit a very specific structure in general.

2.4 Finding the smallest counterexample

The smallest counterexample is closely related to finding the strongest evidence; that is, instead of only needing the shortest path, we also need the next to shortest and the next to next to shortest, et cetera. Recall that by definition 2.1.5 of *smallest counterexample* (q.v.) it contains the first, say k, most probable paths. If not, we could exchange one of the paths with a more probable one and obtain an even smaller counterexample, hence refuting the claim that the original counterexample was smallest. In the weighted digraph derived from the DTMC this will mean finding the k shortest paths.

Since we do not know in advance how many paths we will need, we would prefer to be able to continue finding paths until we hit our target limit, or the paths are exhausted. Several algorithms for finding the k shortest paths require us, however, to specify k in advance. Since such a restriction is in general not feasible in our case, we shall not investigate these algorithms.

This left us with two main candidates, namely the algorithm by Eppstein (1998), and the algorithm by Jiménez & Marzal (1999). The first one performs better in theory, with an asymptotic time complexity of $\mathcal{O}(m + n\log n + k)$ whereas the latter has a worse time complexity of $\mathcal{O}(m + kn\log\frac{m}{n})$, but is much simpler. Practical experiments however, favour the algorithm by Jiménez and Marzal, because Eppstein's

[3] Actually, we can see Dijkstra's algorithm as a BFS variant. Normal BFS only works on a non-acyclic graph if every edge has the same length. A graph can be transformed into such a graph by dividing an edge of weight w into w edges of equal length, and running BFS on it. The order in which the original nodes are discovered is the same as for Dijkstra's algorithm. Dijkstra uses a queue in lieu of dividing the edges, but since the latter is not necessary for BFS to work correctly in case of a DAG, the former is also superfluous.

algorithm has an expensive first phase, which has to be performed always. After this phase any number of paths can be extracted quickly, but it takes a very long time for this first phase to pay off. An improved version of Eppstein's algorithm is described in Varó & Jiménez (2003), this version is in essence a lazy version of Eppstein's algorithm, which postpones much of the computing the original algorithm does in the first phase to a moment when it's necessary, but experiments in the same paper show that in practice the algorithm of Jiménez & Marzal (1999) is still superior in almost all cases.

Note that both algorithms depend on k, which is the number of paths we need, since we do not know this number in advance, the complexity is pseudo-polynomial because k may be exponential, and in fact is, as shall be discussed in chapter 4.

The implementation of Eppstein's algorithm in both cases requires rather specialised data structures and is not very simple. We therefore mainly discuss the Recursive Enumeration Algorithm (REA) for the unbounded case and the modified REA as described by Han & Katoen (2007) for the unbounded case and their relation with the shortest path algorithms described above.

After the experiments and implementation had been done, a new algorithm, named K*, based on the lazy variant of Eppstein's algorithm, appeared, which was developed by Aljazzar & Leue (2008b). This algorithm has the same time complexity as Eppstein's algorithm, and appears to be faster in practice than the other algorithms. Furthermore, it is able to work with only portions of the graph in memory (both other algorithms start with by computing the shortest path tree of the entire graph).

2.4.1 Unbounded until

The unbounded until case is, again, the most straightforward one. We shall first formalise the problem at hand by defining the k Shortest Paths (KSP) problem.

Definition 2.4.1. Given a weighted directed graph $\mathcal{G} = (V, E, w)$ with $s, t \in V$ and $k \in \mathbb{N}^*$, the KSP problem is to find k distinct shortest paths between s and t, if they exist.

From the correspondence between shortest paths and most probable paths, as defined in lemma 2.2.5, we can deduce that we can find a smallest counterexample by solving a KSP problem on a graph $\mathcal{G} = (V, E)$.

This problem, can be solved by the Recursive Enumeration Algorithm (REA), as mentioned before. We shall first give an overview of this algorithm. The full details and experimental results can be found in (Jiménez & Marzal, 1999). The problem is first presented as a system of equations. We denote the k-th shortest path from s to v, where $s, v \in V$ as $\pi^k(s, v)$. This yields the following equations:[4]

$$\pi^k(s,v) = \begin{cases} s & \text{if } k = 1 \text{ and } v = s \\ \sigma \text{ s.t. } \ell(\sigma) = \min\left\{\ell(\sigma) \mid \sigma \in \mathcal{C}^k(s,v)\right\} & \text{otherwise} \end{cases} \tag{2.1}$$

[4] For $k = 1$ these equations are the Bellman equations for the single shortest path problem, see Bellman (1958).

where $\mathcal{C}^k(s,v)$ is a set of candidates from which $\pi^k(s,v)$ is selected.

Before providing the formal definition of this candidate set, we describe the idea behind it. First of all, we note that, except in the case where $v = s$ and $k = 1$ a k shortest path from s to v always has to go via a predecessor node of v. Now we note that one candidate path for each predecessor node suffices, because we are looking for the shortest path. If we would have two or more candidates for each predecessor, we could discard all except the shortest candidate path, the others would never give an optimal result. So, our candidates are formed by the shortest paths leading to predecessors of v. For the case where $k > 1$ we have to make sure we do not reuse paths. For example, if we find that the third shortest path from s to v is formed by extending the shortest path from s to u with the edge uv, this path should be removed from the candidates for the fourth shortest path. The second shortest path from s to u extended with uv would however be a candidate for the fourth shortest path.

This gives us a definition of $\mathcal{C}^k$ in terms of $\mathcal{C}^{k-1}$, with the candidate removed that was used for the $k-1$-th shortest path, and replaced by the next-shortest path leading to the same node, if such a path exists.

In the next definition let u be the node and k' the index such that $\pi^{k-1}(s,v) = \pi^{k'}(s,u)\cdot v$:

$$\mathcal{C}^k(s,v) = \begin{cases} \{\pi^1(s,u)\cdot v \mid (u,v) \in E\} & \text{if } k = 1,\ v \neq s; \\ & \text{or } k = 2,\ v = s. \\ \left(\mathcal{C}^{k-1}(s,v) - \left\{\pi^{k'}(s,u)\cdot v\right\}\right) \cup \left\{\pi^{k'+1}(s,u)\cdot v\right\} & \text{otherwise} \end{cases} \tag{2.2}$$

Note however, that $\pi^{k'+1}(s,u)$ need not exist, which happens if $\mathcal{C}^{k'+1}(s,u)$ is empty; if this is the case we assume $\pi^{k'+1}(s,u)\cdot v = \perp$, where $\perp$ denotes the non-existence of a path.

Note that in particular, the values for $\pi^1(s,v)$ can be computed by a single run of any single source shortest path algorithm, such as Dijkstra's algorithm of Bellman-Ford, depending on whether the graph has negative edge weights: the REA works too for negative edge weights, but not negative weight cycles of course. For a proof of the equations the reader is referred to the original article by Jiménez & Marzal (1999) or to theorem 2.4.3 on page 54, which gives a proof for a generalised version of these equations.

2.4.1.1 Algorithmic description

A solution of the shortest path equations of the previous section can be computed by the Recursive Enumeration Algorithm (REA). It first starts out by computing the shortest path tree rooted in s and then continues to calculate the next shortest paths to v until there are no more paths or no more paths are needed, which in our case means that we have reached a certain probability threshold.

Looking at the algorithm, we see that there is a special case for $\pi^1(s,v)$ on line 2; which equation 2.1 does not have. However, as we see in equation 2.2, the candidate set equation has a special case for $k = 1$ (except for $v \neq s$, but this constraint is actually

Algorithm 2.4.1 The Recursive Enumeration Algorithm

Require: $\mathcal{G} = (V, E)$ is a graph with no negative weight cycles, $s, t \in V$
Ensure: The first k shortest paths from s to t are computed, where k is the total number of paths from s to t, or less if a threshold condition causes the algorithm to terminate early.

1: { *Dijkstra, Bellman-Ford or any adequate algorithm that can be used to compute the shortest path tree depending on the additional constraints (such as positive edge weights) on the graph is suitable.* }
2: $\pi^1(s, \cdot) :=$ ShortestPathTree$(\mathcal{G}, s)$
3: $k := 1$
4: **while** $\pi^k(s,t) \neq \bot \wedge \neg$ThresholdReached:
5: { *In our case,* ThresholdReached *could be defined as* $p \leq \sum_{i=1}^{k} \Pr(\pi^i(s,t))$ }
6: $\pi^k(s,t) =$NextPath(t, k)

7: **procedure** NextPath(t, k): // $\pi^1(s,t), \ldots, \pi^{k-1}(s,t)$ *should exist.*
8: **if** $k = 2$:
9: { *The set of candidates is initialised with all the shortest path to the predecessor nodes of* v*, except for the path used in the shortest path from* s *to* v. }
10: $\mathcal{C}[t] := \{\pi^1(s,u)\cdot v \mid (u,v) \in E \wedge \pi^1(s,v) \neq \pi^1(s,u)\cdot v\}$
11: **if** $\neg(t = s \wedge k = 2)$:
12: Assign u and k' s.t. $\pi^{k-1}(s,v) = \pi^{k'}(s,u)\cdot v$
13: **if** Undefined$(\pi^{k'+1}(s,u))$:
14: $\pi^{k'+1}(s,u) :=$ NextPath$(u, k'+1)$
15: **if** $\pi^{k'+1}(s,u) \neq \bot$:
16: $\mathcal{C}[t] := \mathcal{C}[t] \cup \pi^{k'+1}(s,u)$
17: **if** $\mathcal{C}[t] \neq \varnothing$:
18: select σ s.t. $\ell(\sigma) = \min\{\ell(\sigma) \mid \sigma \in \mathcal{C}[t][h]\}$
19: $\mathcal{C}[t][h] := \mathcal{C}[t][h] - \sigma$
20: **return** σ
21: **else**:
22: **return** $\bot$

superfluous, because for $v = s$ equation 2.1 already tells us that the answer is s and we need not evaluate the candidate set).

The reason for this is twofold, first of all, when being able to use Dijkstra, this will result in a much fast calculation of the shortest path tree (the tree formed by all the shortest paths from the root to every node in the graph) and the other reason is that it makes the algorithm easier to implement.

Since we know that if $k_2 > k_1$ we always first have to compute $\mathcal{C}^{k_1}$ before we compute $\mathcal{C}^{k_2}$, we can suffice by keeping one candidate set per node, which is iteratively updated as needed.

Also note that the calculation of π^1 starts at the source, and ends in the target, whereas the calculation of each next path starts at the target and looks backward instead. This does not matter for the unbounded until, but causes the bounded until to be trickier.

It can be proven that for $k > 1$ a call to NextPath(v, k) will only generate calls to NextPath(u, j) for nodes u in $\pi^{k-1}(s, v)$ and does not generate a call NextPath(v, j) for any j; the details of this proof can be obtained in the article of Jiménez & Marzal (1999).

2.4.2 Upper bounded until

First of all, the approach to reduce the problem to an unbounded problem by using the product graph transform also works. We can then run the unbounded version of the REA on a transformed graph. We can also adapt the algorithm, which is not very hard because the bounded until problem is very similar to the unbounded until, but has a slight catch. The translation of the algorithm is very straight forward, and we can almost succeed by simply replacing $\mathcal{C}[t]$ by $\mathcal{C}[t, h]$, thus implicitly encoding the transformed graph. Before we continue however, we state the formal definition of the Hop constrained k Shortest Path (HKSP) problem:

Definition 2.4.2. Given a weighted directed graph $\mathcal{G} = (V, E, w)$ with $s, t \in V$ and $k \in \mathbb{N}^*$, the Hop constrained k Shortest Path (HKSP) problem is to find k distinct shortest paths between s and t not exceeding the hop bound h, if they exist.

Again, we can deduce from the correspondence between shortest paths and most probable paths, as defined in lemma 2.2.5, that we can find a smallest counterexample for bounded until by solving a HKSP problem on a graph $\mathcal{G} = (V, E)$. We first show the new set of recursive equations, inspired by equations 2.3 and 2.4, which take the hop count into account. These equations were first presented by Han & Katoen (2007) (again, $\perp$ denotes non-existence of a path):

$$\pi_h^k(s, v) = \begin{cases} s & \text{if } k = 1, v = s, h \geq 0 \\ \perp & \text{if } k > 1, v = s, h = 0; \\ & \quad \text{or } v \neq s, h = 0. \\ \sigma \text{ s.t. } \ell(\sigma) = \min\left\{\ell(\sigma) \mid \sigma \in \mathcal{C}_h^k(s, v)\right\} & \text{otherwise} \end{cases} \tag{2.3}$$

The main difference is that we take the hop count into account too. Note that the equations work backwards again, and h should basically be interpreted as 'the number of hops left to reach the starting vertex s when coming from t'; this is why the $h \geq 0$ condition is present in the first case, because one can arrive at s with a few hops to spend.

We also have different candidate set for each value of h, as indicated by the extra index $\mathcal{C}_h^k(s, v)$. Again, in the next definition let u be the node and k' the index such

that $\pi_h^{k-1}(s,v) = \pi_{h-1}^{k'}(s,u)\cdot v$:

$$C_h^k(s,v) = \begin{cases} \{\pi_{h-1}^1(s,u)\cdot v \mid (u,v)\in E\} & \text{if } k=1,\ v\neq s; \\ & \text{or } k=2,\ v=s \\ \left(C_h^{k-1}(s,v) - \{\pi_{h-1}^{k'}(s,u)\cdot v\}\right) \cup \left\{\pi_{h-1}^{k'+1}(s,u)\right\} & \text{otherwise} \end{cases} \tag{2.4}$$

When adapting the REA, we should take special care of the first stage of the algorithm, in which the values for $\pi_h^1(s,v)$ should be computed. We need this stage to compute the distances, which should be computed with respect to s, but also to compute the hop counts, which should be computed with respect to t, since the second stage works backwards.

For the time being, we ignore the actual implementation of the algorithm we need in the first phase, and instead focus on an issue to be taken into account. We take a new look at the computation by the hop constrained BF algorithm on the graph of figure 2.5, as shown in the table on page 40.

We assume the adapted algorithm does a forward pass to find the distances, much like BF does, and then and a backward pass to find the hop counts. The backwards pass fills a 'hopsleft' array which records the number of hops that we have left to reach the source when coming from the target.

	Nodes			
Array	s	s_1	s_2	t
distance	$\{0\to 0\}$	$\{1\to 3,\ 2\to 2\}$	$\{1\to 1\}$	$\{2\to 4,\ 3\to 3\}$
previous	$\{0\to\varnothing\}$	$\{1\to s,\ 2\to s_2\}$	$\{1\to s\}$	$\{2\to s_2,\ 3\to s_1\}$
hopsleft	$\{0,\ 1\}$	$\{2\}$	$\{1\}$	$\{3\}$

The difficulty is how to reconcile the first two arrays with the third one. We see that s_1 and t have multiple entries in the previous and distance array, but only one in hopsleft. The multiple entries in the previous and distance arrays indicate that there is more than one path from s to s_2 and t, but the single entry in hopsleft that there is only one path from s_2 and t to t. The situation for s however, is reversed: it has two entries in hopsleft, and only one in the other arrays, because when coming from t there are two different routes to get to s.

The solution to this is not so hard, for every entry h in hopsleft of a node v, we need to find the entry distance$[v]$, with the shortest distance, not exceeding h.

In this case the hop limit does not really pose a constraint, but if we would do the same with a constraint of 2 hops, we would find:

	Nodes			
Array	s	s_1	s_2	t
distance	$\{0\to 0\}$	$\{1\to 3,\ 2\to 2\}$	$\{1\to 1\}$	$\{2\to 4,\ 3\to 3\}$
previous	$\{0\to\varnothing\}$	$\{1\to s,\ 2\to s_2\}$	$\{1\to s\}$	$\{2\to s_2,\ 3\to s_1\}$
hopsleft	$\{0\}$	$\{1\}$	$\{0\}$	$\{2\}$

We see that although there is an option to get within two hops to s_1 when starting in s, the value in hopsleft tells us it does not have more than one hop to spend there to get to s, so its optimum will be a distance of 3. The hopsleft of 0 for s_2 indicates there is no path through s_2 that fulfils the hop constraint.

If an algorithm would do these two passes, the procedure for this reconciliation would have to be done for every node, and for every value h_l in hopsleft. When trying to adapt BF, a procedure like this invariably pops up. If we start at the target, we can get the hop counts directly correct, but not the distances, and vice versa. The solution lies with the set of equations that describe the problem, and we shall delve into this in the next section.

We first give the adapted version of the Recursive Enumeration Algorithm (REA) in algorithm 2.4.2. This algorithm was first published in a slightly different form by Han & Katoen (2007).

It can be proven that for $k > 1$ a call to NextPath(v, k, h) will only generate calls to NextPath$(u, j, h-1)$ for nodes u in $\pi_h^{k-1}(s, v)$; the details of this proof can be obtained in the article of Han & Katoen (2007). The other aspect proven for the original algorithm, namely that it won't run in a cycle is trivial, because of the added parameter h. A call to NextPath with parameter h will never result in a recursive call with the same (or higher) h.

Again, the similarity between both algorithm 2.4.1 and algorithm 2.4.2 is striking, the main difference involves the indexing with an extra h where otherwise only t was used.

We conclude this section stating that the time complexity of the adapted REA is $\mathcal{O}(hm + hk\log(\frac{m}{n}))$, the proof of which can also be found in Han & Katoen (2007). Due to the fact that k might be exponential, this is problem cannot be reduced to a polynomial time complexity.

2.4.2.1 Using DFS in the first phase

We first take a look at the recursive equations for the hop constrained shortest path problem, but we now fix the value of k to 1 in equation 2.3, this gives us (the second case cannot happen):

$$\pi_h^1(s,v) = \begin{cases} s & \text{if } v = s \text{ and } h \geq 0 \\ \sigma \text{ s.t. } \ell(\sigma) = \min\left\{\ell(\sigma) \mid \sigma \in \mathcal{C}_h^1(s,v)\right\} & \text{otherwise} \end{cases}$$

We still have a different candidate set for each value of h, as indicated by the extra index $\mathcal{C}_h^1(s,v)$. But the second case is gone: it is irrelevant, since it only applies for values of $k > 1$. The instantiated version of equation 2.4 therefore is:

$$\mathcal{C}_h^1(s,v) = \{\pi_{h-1}^1(s,u) \cdot v \mid (u,v) \in E\}$$

We now look at a small example graph and a computation run of $\pi_1^3(s,t)$, as shown in figure 2.7. We see that although nodes might have two predecessors, the computation graph itself is acyclic – note that this would *not* be the case if we didn't use hop counts,

Algorithm 2.4.2 The adapted Recursive Enumeration Algorithm

Require: $\mathcal{G} = (V, E)$ is a graph with no negative weight cycles, $s, t \in V$, $h \geq 0$ is an hop bound.

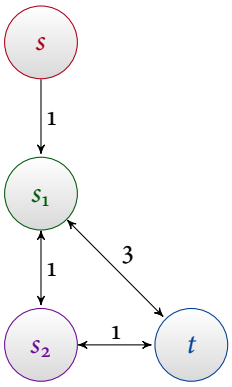
Ensure: The first k shortest paths from s to t are computed, where k is the total number of paths from s to t, or less if a threshold condition causes the algorithm to terminate early. The number of hops in each path does not exceed h

$\pi^1(s, \cdot) :=$ DFS$(\mathcal{G}, t)$
$k := 1$
while $\pi^k(s,t) \neq \bot \wedge \neg$ThresholdReached:
{ *In our case,* ThresholdReached *could be defined as* $p \leq \sum_{i=1}^{k} \Pr(\pi^i(s,t))$ }
$\pi^k(s,t) =$NextPath(t, k)

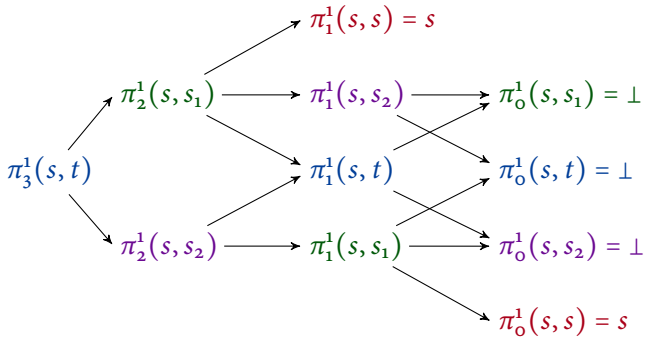
procedure NextPath(t, k): // $\pi_h^1(s,t), \ldots, \pi_h^{k-1}(s,t)$ *should exist.*
if $k = 2 \wedge h > 0$:
{ *The set of candidates is initialised with all the hop constrained shortest path to the predecessor nodes of v, except for the path used in the shortest path from s to v.* }
$\mathcal{C}[t][h] := \left\{\pi_{h-1}^1(s,u)\cdot v \mid (u,v) \in E \wedge \pi_h^1(s,v) \neq \pi_{h-1}^1(s,u)\cdot v\right\}$
if $\neg(t = s \wedge k = 2) \wedge h > 0$:
Assign u and k' s.t. $\pi_h^{k-1}(s,v) = \pi_{h-1}^{k'}(s,u)\cdot v$
if Undefined$(\pi_{h-1}^{k'+1}(s,u))$:
$\pi_{h-1}^{k'+1}(s,u) :=$ NextPath$(u, k'+1, h-1)$
if $\pi_{h-1}^{k'+1}(s,u) \neq \bot$:
$\mathcal{C}[t][h] := \mathcal{C}[t][h-1] \cup \pi_{h-1}^{k'+1}(s,u)$
if $\mathcal{C}[t][h] \neq \varnothing$:
select σ s.t. $\ell(\sigma) = \min\{\ell(\sigma) \mid \sigma \in \mathcal{C}[t][h]\}$
$\mathcal{C}[t][h] := \mathcal{C}[t][h] - \sigma$
return σ
else:
return $\bot$

in that case computation graph would effectively yield a copy of the graph, but with the arrow directions reversed, as shown in figure 2.8. So only in the hop constrained case, we see we end up with a DAG if we unroll the equations. This is not surprising, after the discussion how a hop constrained problem can be reduced by transforming the graph to a DAG; we see this is done implicitly in these equations. When evaluated lazy the graph is only constructed as it is explored, which gives an advantage in memory usage in practice.

We need to point out it is not fortuitous that our example results in a DAG: that a cycle could never occur is easily seen if we realise that for a cycle exist somewhere on

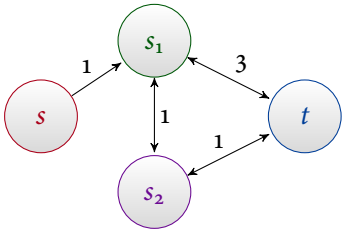


a A cyclic graph.

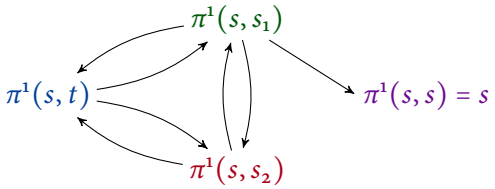


b A hop constrained shortest path computation in the opposite graph.

Figure 2.7 An example of a cyclic graph and the computation of a hop constrained shortest path from s to t with a hop limit of 3. The upper branch yields $s{\cdot}s_1{\cdot}t$ and the lower branch yields $s{\cdot}s_1{\cdot}s_2{\cdot}t$. Note that the computation starts at the target.



a A cyclic graph.



b An unconstrained shortest path computation tree in the opposite graph. Note that the result is *not* a DAG.

Figure 2.8 An example of a cyclic graph and the computation of a un constrained shortest path from s to t. We see that the computation results in a copy of the original graph, with cyclic dependencies. The solution to these unconstrained equations cannot thus be given by DFS.

the cycle we need to have an arc from $\pi^1_{h_l}(s,s_x)$ to $\pi^1_{h_u}(s,s_y)$ where $h_l < h_u$; this is impossible.[5]

Hence, seeing the computation results in a DAG, we know this structure should have a topological ordering, and that this ordering can be found using DFS, as first described by Tarjan (1976). Because we are working with a unique target, the computation tree always starts at a single node, and we can simply run DFS from this target node

[5]It is obvious that a DFS computation in the non-hop constrained case would very often miss the shortest path, because of these cycles. For example, looking at the computation graph in figure 2.8b if we would start the search in t, it might pick s_1 first to extend the search to, then it might pick s_2, as a predecessor of s_1 and conclude that t and s_1 have already been visited, thus leaving nothing to explore at s_2, and it needs to mark the node as done. Backtracking to s_1 it finds s as its predecessor, and can find the distance, which is propagated to t, and the result is that s_2 has no known distance, and t has distance 4.

Algorithm 2.4.3 DFS(v, h) calculates $\pi_h^1(s, v)$ on an acyclic graph.

```
1:  procedure DFS(v, h):
2:      if h < 0: return ⊥
3:      if DFS(v, h) has been computed before:
4:          return π¹_h(s, v)
5:      if s = v ∧ 0 ≤ h:
6:          π¹_h(s, s) = s                                   // By definition
7:          for each u ∈ {u | (u, s) ∈ E}:
8:              DFS(u, h − 1)
9:      else:
10:         C¹_h(s, v) = {DFS(u, h − 1) · v | (u, v) ∈ E}
11:         π¹_h(s, v) = σ s.t. ℓ(σ) = min{ℓ(σ) | σ ∈ C¹_h(s, v)}
12:     return π¹_h(s, v)
```

to compute the shortest paths. While DFS does the forward pass, it registers the hop counts, as soon as it hits s or the hop count is zero, it knows the distance from s to that node. In the first case it is zero, in the latter case we register it as ∞. This might not be correct when viewing from s, because the node might well be connected to s, however, the path we have found violates the hop constraint, and hence we need not save it, it will never be of use. Then, when back tracking after having exhaustively explored a branch, it can propagate the distance to s. If more paths lead to s, it of course needs to choose the one with minimum distance.

The pseudo code for the DFS is shown in algorithm 2.4.3. Although the case where $s = v$ is a terminal case in a certain sense, the DFS cannot halt here. Suppose, for example, the graph would look like  and we would run a DFS, with a hop constraint of 3. If we halt immediately when reaching s, after starting in t, we would only register $\pi_1^2(s, s)$, and cause trouble when the next phase of the algorithm would try to find the second shortest path to s, because for this it would look for the shortest path to the predecessor of s, but for this value of h we would not have explored s yet.

Theorem 2.4.3. *The Depth First Search (DFS) presented in 2.4.3 can be used to calculate $\pi_h^1(s, v)$ on an acyclic graph.*

Proof. We proceed by induction on h. For $h = 0$, the claim is correct, since DFS$(s, 0)$ yields s, which is obviously correct, and for a node $v \neq s$ we see that for each predecessor on line 10 a recursive call is made, which yields an immediate return on line 2. Hence, the candidate set will be empty, and it will return $\perp$ in these cases.

Now that we have proven the base case, we continue with the inductive step. Suppose a call DFS(s, h) is made. In this case the result will be s, which is correct (note that the recursive call that is made at line 8 does not influence the computation directly and only serves to pre-compute values needed in the next phase of the REA.). If

DFS(v, h) is called, with $v \neq s$, the shortest paths to its neighbours are first computed, from which the shortest path is selected, and this path is extended with edge (u, v). □

Note that the introduction of the hop bound makes the proof easier, if there would have been no hop bound, we could not use induction so easily. Also, this gives us the guarantee that the recursion will stop:

Theorem 2.4.4. *The maximum depth of recursion of a call to* DFS(v, h) *presented in 2.4.3 is* $h + 1$.

Proof. We use a similar induction reasoning. For $h = 0$ we see, that if DFS$(v, -1)$ is called for the predecessors of v. This call returns immediately, thus limiting the recursion depth to 1.

For the inductive step, consider some call to DFS(v, h) with $h > 0$. This might result in calls at line 8 or line 10, but not both. Either way, the recursive call has the form DFS$(v, h - 1)$; we know this has a maximum depth of h, hence the total recursion depth is $h + 1$ at most. □

2.4.3 Double and lower bounded until

The previous section explored the translation of the simple bounded until operator $\Phi \, \mathsf{U}^{\leq h} \, \Psi$, there are no reasons however why the bound should only be an upper bound, as mentioned in remark 1.5.5 on page 21. Although not the main focus of this thesis, we shall discuss the adaptations need for the more general operator $\Phi \, \mathsf{U}^{[h_l, h_u]} \, \Psi$ that can take both a lower and upper bound.

The observations in this section are mainly intended as a starting point for further investigations: not all aspects have been examined thoroughly.

The semantics for the dual bounded operator are (Cf. definition 1.5.3):

$$\sigma \vDash \Phi \, \mathsf{U}^{[h_l, h_u]} \, \Psi \qquad \text{iff} \quad \exists h_l \leq i \leq h_u : \big(\sigma[i] \vDash \Psi \wedge \forall 0 \leq j < i : (\sigma[j] \vDash \Phi)\big)$$

We furthermore require $h_l \leq h_u$. The original bounded operator $\mathsf{U}^{\leq h_u}$ can be expressed as $\mathsf{U}^{[0, h_u]}$, and dually a lower bounded operator $\mathsf{U}^{\geq h_l}$ can be defined as $\mathsf{U}^{[h_l, \infty]}$, finally an operator $\mathsf{U}^{=h}$ could be defined as $\mathsf{U}^{[h,h]}$. Because the case where $h_u = \infty$ is a bit more difficult, for now we assume that $h_u < \infty$. Note that the interpretation of a lower bound only affects Ψ in $\Phi \, \mathsf{U}^{[h_l, h_u]} \, \Psi$, Ψ, it has to hold after h_l steps, if it holds before, Φ has to hold too, because otherwise the formula is not valid, so basically these early Ψ's are ignored.

We start by giving the equations describing the dual bounded problem, these are in fact generalisations of the previous equations (2.1–2.4):

$$
\begin{aligned}
&\text{if } k = 1,\ v = s \text{ and } h_l \le 0 \le h_u\text{:}\\
&\qquad \pi^k_{[h_l,h_u]}(s,v) \quad = s;\\
&\text{if } k > 1,\ v = s, \text{ and } h_u = 0\text{; or } v \neq s \text{ and } h_u = 0\text{:}\\
&\qquad \pi^k_{[h_l,h_u]}(s,v) \quad = \perp;\\
&\text{otherwise:}\\
&\qquad \pi^k_{[h_l,h_u]}(s,v) \quad = \sigma \text{ s.t. } \ell(\sigma) = \min\left\{\ell(\sigma) \mid \sigma \in \mathcal{C}^k_{[h_l,h_u]}(s,v)\right\}.
\end{aligned}
\tag{2.5}
$$

The main alteration involves the addition of a lower bound. It is easily seen that if we provide $h_l = 0$, we get the previous hop constrained equations of 2.3-2.4), and that for $h_l = 0$ and $h_u = \infty$ we get the unconstrained equations of (2.1-2.2).

The equations for the candidate set are altered likewise. Again, let u be the node and k' the index such that $\pi^{k-1}_{[h_l,h_u]}(s,v) = \pi^{k'}_{[h_l-1,h_u-1]}(s,u)\cdot v$:

$$
\begin{aligned}
&\text{if } k = 1 \text{ and } (h_l > 0 \text{ or } v \neq s)\text{; or } k = 2 \text{ and } v = s\text{:}\\
&\qquad \mathcal{C}^k_{[h_l,h_u]}(s,v) = \{\pi^1_{[h_l-1,h_u-1]}(s,u)\cdot v \mid (u,v) \in E\};\\
&\text{otherwise:}\\
&\qquad \mathcal{C}^k_{[h_l,h_u]}(s,v) = \left(\mathcal{C}^{k-1}_{[h_l,h_u]}(s,v) - \{\pi^{k'}_{[h_l-1,h_u-1]}(s,u)\cdot v\}\right) \cup \left\{\pi^{k'+1}_{[h_l-1,h_u-1]}(s,u)\right\}.
\end{aligned}
\tag{2.6}
$$

The addition of the h_l lower bound might be a bit misleading, since it suggests the number of entries in the candidate set and the number of entries (implicitly) encoded in the graph explodes, but we will always have that $h_u - h_l = d$, where d is some fixed number, so memory-wise this is no worse than the single bounded operator.

The addition of the indices is self explanatory, the $h_l \le 0 \le h_u$ may be less so. The condition $h \ge 0$ in the original hop constrained equations basically states 'it does not matter if we end up early in s'. As soon as we hit s, we have found a path. If the condition would have been $h = 0$, it would only find paths of *exactly* h hops: this would plainly lead to incorrect results. In the case of a lower bound, we can end up too soon in s, say, after only 2, with a lower hop bound of 3. This is avoided by the condition $h_l \le 0$.

Before delving into the formal correctness proof of the algorithm, an example run is shown in figure 2.9. It is similar to the run of figure 2.7b, and also uses the same graph, but we see that the first time a path to s is found, it is deemed invalid because it is only two hops away from t, whereas a path needs to have at least 3 hops. We also see that the lower hop bound can become negative, this is intended. One could also, instead of saying $h_l - 1$ in the equations do something like $\max(0, h_l - 1)$, so the lower bound could not drop below zero, which might be preferable due to implementation concerns.

It should be noted that the lower hop bound can cause even the shortest path to have a cycle. The possible introduction of a cycle, even in the shortest path explains the softening of the first condition of the candidate set. Previously, if we would arrive

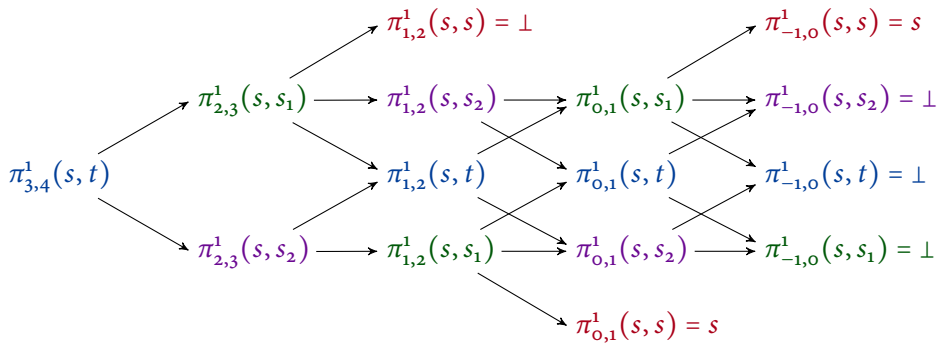


Figure 2.9 Example of computation run in the graph of figure 2.7a for the shortest path computation with a lower bound of 3 and an upper bound of 4. We start from t and are looking for paths from s. Note that first s is found, the lower limit has not been reached and thus no path is found.

at s, we would always have found a shortest path in case of $k = 1$, hence, the condition if $k = 1$ and $v \neq s$, in fact, this condition was superfluous, because it implies 'otherwise' should have been the case in the case of π^1, which cannot be if $k = 1$ and $v = s$ are true. In this case however we can arrive at s too soon, and need to extend the path, hoping we'll find a cycle and end up at s again. For completeness' sake, we give a proof of the theorem, since the adaptation is so small, the proof is almost a verbatim copy from the original by Jiménez & Marzal (1999).

Theorem 2.4.5. *The equations 2.5 and 2.6 characterise the double hop constrained k shortest paths from s to v with at least h_l and at most h_u hops.*

Proof. Let $\mathcal{P}^k_{[h_l,h_u]}(s,v)$ denote the set of the k shortest paths from s to v with at least h_l and at most h_u hops. Every path from s to v arrives at v through some predecessor node u, i.e. $u \in Pred(v) = \{\hat{u} \in V \mid (\hat{u}, v) \in E\}$.

In order to compute $\pi^k_{[h_l,h_u]}(s,v)$ we could consider every path from s to some predecessor u, which, extended with (u,v) would not yield a path already in $\mathcal{P}^{k-1}_{[h_l,h_u]}(s,v)$, that is, we cannot use the second shortest path again as third shortest, so these have to be left out. This would be unnecessary though. For each predecessor u, of course $w(u,v)$ is always the same. Given two paths through u, which abide the hop bounds $[h_l - 1, h_u - 1]$ we can always opt for the shortest[6], since the total length, when extended with $w(u,v)$ will be our best choice. Hence, we only need to consider the shortest path to a predecessor u which, if extended, would not yield a path in $\mathcal{P}^{k-1}_{[h_l,h_u]}(s,v)$.

The set containing one candidate for each predecessor node is characterised recursively by equation 2.6. □

If both an upper and lower bound are specified, we can use only the upper bound as an index variable, since the difference between h_u and h_l is fixed. This means that when it comes to the number of invocations on NextPath, this is no different from the

[6] If they are of equal length, another criterion can be introduced to break the tie, or we can pick one non-deterministically, if necessary. This does not affect the argument.

number when only specifying an upper bound. Therefore, we can conclude that for $k > 1$ a call to NextPath(v, k, h_u, h_l) will only generate calls to NextPath$(u, j, h_u - 1, h_l - 1)$ for nodes u in $\pi^{k-1}_{[h_l, h_u]}(s, v)$. This follows immediately from the result in Han & Katoen (2007).

We can interpret the recursive equation for the k-th shortest path as saying, create the new candidate set by removing the path through predecessor u that was used for the $k-1$-th shortest path, and add a new candidate for u in the form of the next shortest path to u (if it exists).

2.4.3.1 Algorithmic Description

Introducing a lower bound is not particularly tricky if we use the graph transformation to reduce the problem to a non-constrained problem. If there is both an upper and lower bound, the extra transitions to the new target state of the first l_h copies can be removed. If the upper bound is however set at ∞, one cannot unroll the graph indefinitely, but instead we do not remove the internal edges of the last copy, this idea is shown in figure 2.10. Equivalently, we can choose not to remove the internal edges of the first copy.

If we want to look at direct algorithmic solution, that not explicitly transforms the graph, we can still use the DFS approach, provided an upper bound is present. Some obvious constraints on the lower bound can be imposed, i.e. checking whether $h_l \leq 0$ when arriving at s, and if not so, not registering the path. That is, the following code fragment in algorithm 2.4.3:

5: **if** $s = v \wedge 0 \leq h$:
6: $\quad \pi^1_h(s, s) = s$ *//By definition*
7: $\quad \ldots$

should be replaced by:

5: **if** $s = v \wedge h_l \leq 0 \leq h_u$:
6: $\quad \pi^1_{h_u}(s, s) = s$ *//By definition*
7: $\quad \ldots$

also the parameter h_l should be passed, but there is no reason to use it for indexing.

If this upper bound is infinite, the DFS will not terminate by itself, since h_u will never drop below zero, which is what effectively stops the recursion. An approach could be to split the shortest path computation phase in two, first running an unconstrained shortest path algorithm such as Dijkstra on the original graph, and then running BF for h_l iterations, the paths of exactly h_l hops can then be connected to the shortest path starting in their ending vertices. For the bounded operator $\Phi \, U^{[h_l, \infty]} \, \Psi$ this would imply that Φ has to hold in the first h_l states and from then one Ψ can hold.

Another approach could be to impose an upper bound that is not an actual upper bound, namely $h_l + |V| - 1$, V being the vertex set. In the worst case a node lies on a single cycle forming a Hamiltonian path trough the graph and the first arrival is below the hop bound: this does not mean there is no shortest path with the hop bound, the cycle can be extended as often as needed. We just need to make sure we look deep

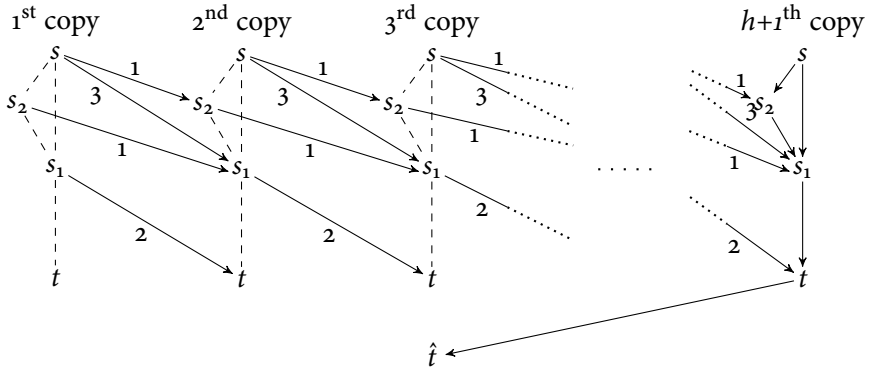


Figure 2.10 Schematic representation of the result of the graph transformation with a lower bound of h. The original graph is replaced by $h + 1$ copies. The original edges are removed (shown dashed) and instead a new edge is inserted, to the corresponding successor in the next copy, e.g. $s \to s_1$ in the first copy is replaced by a transition from s in the first copy to s_1 in the second copy, and so one. The edges in the final copy are not removed however. This way it takes at least h transitions to get to the final copy, and this one is the only to be connected with the extra target state. In the context of the $U^{[h,\infty]}$ operator, this would mean that in the states in the first h copies Φ has to hold, and that Ψ holds in $\hat{t}$.

enough.

If this first phase is done correctly, the second of phase of the algorithm works without problems. Some more aspects will be discussed in the implementation part, but otherwise we shall not look into this in further detail.

2.4.4 Arbitrary bounded operators

This has not been pursuit in any way, but the idea of the graph transformation lends itself, in principle, to more arbitrary bounds on the operator. An upper bound, lower bound and upper and lower bound have been discussed in the previous section, but similarly a bound of 'an even number of steps' could be evaluated by making two copies of the original graph, as we would do with a upper bound of two, but then connecting the second copy again with the first. Only the first copy in turn would be connected with the added target state $\hat{t}$. So, a first hop would go the second copy, the second hop would go back to the first one (possibly making $\hat{t}$ reachable), etc.

Intervals like $[3 \ldots 8]$ hops or $[12 \ldots 20]$ hops would also be easily implementable by removing the transitions to $\hat{t}$ from the first, second, ninth, et cetera, copy.

2.4.5 Lazy algorithms

The algorithms described in this section, REA and the modified REA share the disadvantage that they start by computing the shortest path tree; which has to be kept in memory in its entirety. The original Eppstein (and lazy Eppstein) algorithms also have this problem. Especially in the area of model checking, where one often would like to explore only parts of the state space, this can be considered very undesirable.

The K* algorithm by Aljazzar & Leue (2008b), which was mentioned at the start of this section, does not suffer from this problem because it is able to work without traversing the entire graph beforehand. Nodes are are only expanded as needed. Of course this requires that one can implicitly specify the successor of a node, but this is very often the case in model checking where a state's successor is computed from the information in the current state.

If such a function is available, let us assume it is called *Succ*, then this could also be used to specify a new function $Succ'$ which implicitly creates the product graph construction. For the outline, say that we have $Succ(s) = s'$, then we introduce an extra parameter x for $Succ'$, so that it will look like $Succ'(s, x)$. Assume that in addition we have an upper hop-bound of h. We then can define $Succ'$ as follows:

If $x > h$ or $x = h \wedge s \neq t$ then:

$$Succ'(s, x) = \bot$$

If $s = t$ then:

$$Succ'(t, x) = \hat{t}$$

Otherwise (that is, $x < h \wedge s \neq t$):

$$Succ'(s, x) = (Succ(s), x + 1)$$

Implementation

3

Due to the fact that no program was available that implemented the algorithms explained in the previous chapter, we decided that it would be best to start out developing such an implementation. Using this implementation we would be able to obtain hands-on results with respect to several case-studies.

This implementation, or prototype, is described in this section, it consists of a parser for formulae and transition systems, modelled after MRMC; and it can output a strongest evidence or smallest counterexample. The next sections will describe the functionality in more detail. A small manual for the program, which gives a taste of its basic functionality, is found in appendix B. This chapter only provides a high-level overview, for the nitty-gritty details one is referred to the actual implementation and its documentation.

3.1 Requirements and design goals

The application is deliberately designed as a prototype, which mean we do not aim to deliver a highly tuned and optimised product, which can handle very large models. In principle it can be considered a throw-away prototype which explores the practical viability of the theory in the previous chapter. The following sums up the requirements and the expectations:

- Implement a (basic) parser for PCTL-formulae and DTMC specifications.
  - Functionality can first be restricted to basic formulae without nested probability operators, i.e. $\mathcal{P}_{\leq p_1}(a \,\mathsf{U}\, \mathcal{P}_{\leq p_2} \,\mathsf{F}\, b)$ need not be supported.
  - Also, only probability operators of the form $\mathcal{P}_{\leq p}$ need to be supported, i.e. support for $\mathcal{P}_{<p}$, $\mathcal{P}_{>p}$, etc. is unnecessary.
- Implement the translation of the Markov-chain to a weighted direct graph, as described in section 2.2.2, this means a small model checker for a subset of CTL must be implemented.
  - At first the W operator need not be supported (and by the previous point, nor need nested probability operators).
- Implement at least one shortest path algorithm for finding the Strongest Evidence (SE), including hop-constrained versions.

- Implement at least one k-shortest path algorithm, and a hop-constrained version for finding k-shortest paths with bounded until-operators.
- Present the user with a message whether the counterexample has been found, and if so a list of traces in the counterexample.
- The program itself can be implemented as a command-line program which reads its input from files and outputs to standard out.
- The program need not be very fast or very scalable, but it should at least be able to handle models with a couple of hundred of few thousand states such that the behaviour of small-scale case studies can be studied.

Together, these requirements specify a very basic model checker for PCTL, yet good enough to check a few well-known cases, such as synchronised leader election and the crowds protocol.

In addition to the above requirements, which have been implemented, during program development a few extra features have found their way into the program:

- Hop-constrained problems can be reduced to a non-constrained problem using the product graph method of section 2.3.2.1.
- Lower-bounds in addition to upper-bounds are supported.
  - But: if there is no upper-bound , the problem has to be reduced to an unconstrained problem.

And finally I have tried to remain as close to the representation of the algorithms in this thesis, or the original papers, and to choose obvious data structures, and not to optimise the understandability of the code away.

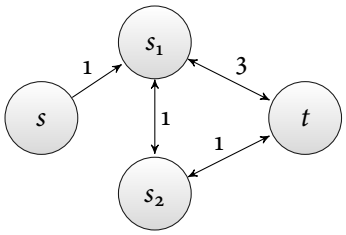
3.2 Program design

The above requirements description allows the program to be naturally divided into the following parts:

- A parser and formula rewriter for PCTL-formulae and Markov chains.
- A (basic) CTL model-checker which can calculate $Sat(\Phi)$ and $Sat(\Psi)$ for the inner formula of $\mathcal{P}_{\leq p}(\Phi \cup \Psi)$.
- A DTMC to weighted graph converter which uses the information of the previous step.
- Strongest Evidence (SE) computation using a shortest path algorithm.
- Smallest Counterexample (SC) computation using a k shortest path algorithm.

The first two parts are rather mundane, and are more like a warm-up exercise than anything else. These parts will not be discussed in detail, appendix A contains a small exposition on the algorithm used for model checking the CTL part, whereas appendix B contains a short description on the file formats used for the PCTL-formulae and DTMC.

The other parts will be described in more detail in the following sections, with particular focus on the algorithms and used data structures.




```
1  G = {'s' : {'s1' : 1 },
2       's1': {'s2' : 1, 't': 3},
3       's2': {'s1' : 1, 't': 1},
4       't' : {'s1' : 3, 's2': 1},
5  }
```

Figure 3.1 An example of a cyclic graph and a possible representation in Python using nested dictionaries, where simple strings are used as states.

3.2.1 Language choice

First of all, a language had to be chosen for the prototype. Since the emphasis of a prototype lies at rapid application development, and not so much speed, I have decided to use Python, an interpreted, interactive, multi-paradigm programming language, supporting object oriented programming throughout, but also allowing typical functional programming constructs such as list comprehensions and first-class functions. It is also quite portable, running on anything from Nokia Phones to Windows, OS X and UNIX. Furthermore interpreters for the Java Virtual Machine (Jython) and Microsoft's Common Language Runtime (IronPython) exist.[1]

Furthermore, the language has built-in support for important data structures, such as sets, dictionaries (hash maps), linked lists, et cetera. This makes it quite convenient to implement algorithms. Because these data structures have a heavily optimised C implementation, they are quite fast in practice, and it allows for rather straightforward translation from pseudo-code to real code. Resulting in faster development than programming in Java or C (PRECHELT, 2000)

3.2.2 DTMC and graph representation

Python's internal hash-table, called a *dictionary* in Python terms, lends itself very well to representing a weighted graph, and for that matter, a DTMC. The idea is a variant on an adjacency list. Using Python's notation, we represent a dictionary as follows: `{'a':1, 'b':2, 'c':3}`, meaning that in this hash table the *key* `a` is mapped onto the integer `1`, which is the *value*, `b` is mapped onto `2`, et cetera. The keys need to be immutable, but the values need not so. Exploiting this, we can map each node onto a new dictionary, which holds the weight (in case of a weighted graph) or the probability of the transition (in case of a DTMC), we see an example of this in figure 3.1.

The advantage of using Python's own hash table is that it's very optimised.[2] With

[1] The de-facto standard implentation can be found at `http://www.python.org/`; Jython is developed under auspices of Sun and can be found at `http://www.jython.org`, IronPython is supported by Microsoft and can be downloaded from their open source project hosting website: `http://www.codeplex.com/IronPython`.

[2] Exact time bounds are hard to give. If your input data is well behaved, and you have specified a suitable hash function for your own objects, insertion and deletion should be $\mathcal{O}(1)$. Practical considerations

this structure it is very easy to check whether a node is a successor of another node, simply by stating `if v in G[u]`, this requires two lookups, first for `u`, and then in the dictionary returned by this lookup, both are very fast. The weight of an arc uv can be obtained by `G[u][v]`, equally fast.

Further advantages are that one can get a list of keys in the dictionary with a simple call, thus obtaining all neighbours, and insertion and deletion of neighbours is equally fast. The construction of a predecessor list however, does require either a pass over the data structure (to find the predecessors of v we have to check for every u whether `v in G[u]` is true), or another adjacency list which is organised the other way around.[3]

Perhaps most importantly, using these data structures, the algorithm of section 2.2 can be implemented faithful to its description, without a loss in performance.

3.2.3 Strongest Evidence algorithms

The algorithm for the strongest evidence in the unconstrained case is rather straightforward. It should be Dijkstra's algorithm; for the hop-constrained case Bellman-Ford is preferable, or, if, more paths have to be found, DFS as explained in section 2.4.2.1.

Therefore, our main point of concern is the choice of an appropriate heap structure to be used for Dijkstra's shortest path algorithm. The better its performs, the better Dijkstra performs. The best known theoretical bounds are given by Fibonacci heaps (Fredman & Tarjan, 1987), but in practice binary heaps or pairing heaps (Fredman *et al.*, 1986) are as fast as Fibonacci heaps (Goldberg & Tarjan, 1996), due to the fact that not enough updates occur in a typical run of Dijkstra's algorithm to warrant the overhead of Fibonacci heaps for the other operations.

A final concern in the case of Python also is the quality of the underlying implementation. Fibonacci heaps implemented in pure Python will probably fare worse than binary heaps that can use its underlying array representation, because of the overhead involved with object creation and destruction. Of course this can be circumvented by implementing an extension module in C, which does not suffer as much of these speed problems, but this would mean implementing an optimised Fibonacci heap in plain C, which in turn, is not a trivial task. Luckily, the latter has been done by Andrew Snare,[4] sadly this implementation is incompatible with the latest versions of Python.

Experiments using different heaps however, showed that the difference in speed for for the algorithms was, although significant, only of slight importance, since the sizes for which the speed really became intolerably slow where not of any practical limitations for the prototype. As a non-comprehensive test, an implementation of Dijkstra's algorithm was run with different heaps on a graph created by a Delaunay triangulation in a plane (see figure 3.2). This is not meant to be a representative model

however govern a design which makes the hash table to perform really well in frequent cases, on the expense of uncommon cases (and exact theoretical bounds). A discussion on the aspects and different concerns of this implementation can be found in *Beautiful Code* (Oram & Wilson, 2007, chapter 18).

[3] In this case, a combination of a dictionary mapping to a set – another built-in data structure – is most appropriate.

[4] The PQueue extension module: which could, at the time of writing be found at: `http://www.csse.monash.edu.au/hons/projects/2001/Andrew.Snare/#pqueue`

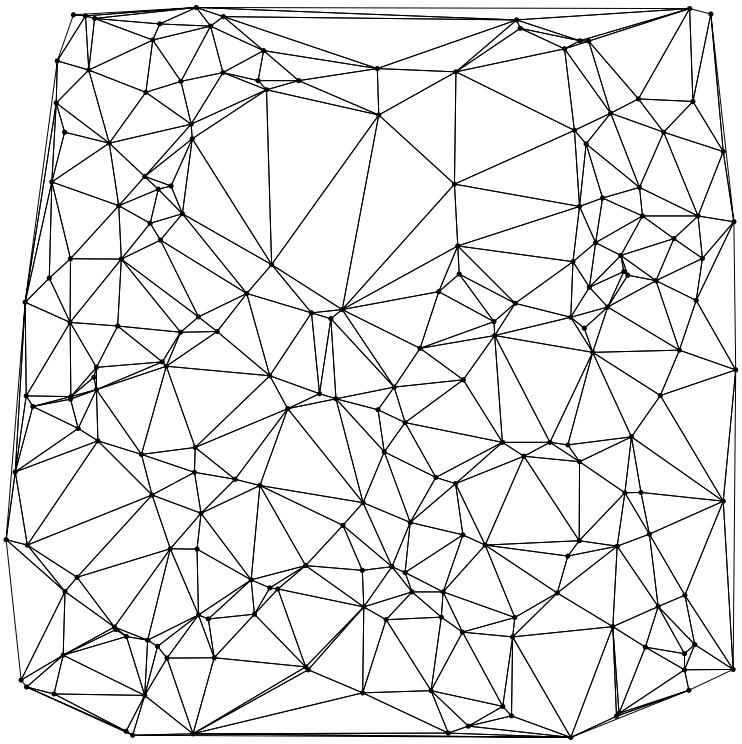


Figure 3.2 An example Delaunay triangulation of a set of 200 random points in the plane.

for structures typically encountered when probabilistic model checking, rather, it serves as a test to see how fast Dijkstra's algorithm can run on a reasonably large graph where every node has mainly local connections.

From the results in figure 3.3, it can be seen that the implementations that are written in pure Python, which are the Fibonacci heap and both pairing heaps, are clearly the slowest, although even six seconds is not unbearably slow for a graph with 50 000 points. This indeed illustrates that the time the shortest path computation will take is not a bottleneck in the execution of the algorithms. The pure C implementation of the Fibonacci heaps is the fastest, whereas the other two implementations that use Python's provided data structures, which also have an underlying C implementation, lie in between.

3.2.4 Smallest Counterexample algorithms

The implementation of the REA follows rather straightforwardly from the pseudo code presented in algorithm 2.4.1. There is only one important optimisation to be made, and that involves storing every path implicitly in order to save massive amounts of space. The pseudo code doesn't show how this is done, but the idea is similar to the way one stores the path implicitly in Dijkstra's algorithm, although in this case we have to store several paths per node, the shortest path, the second shortest path, and so on, and keep a candidate set for every node.

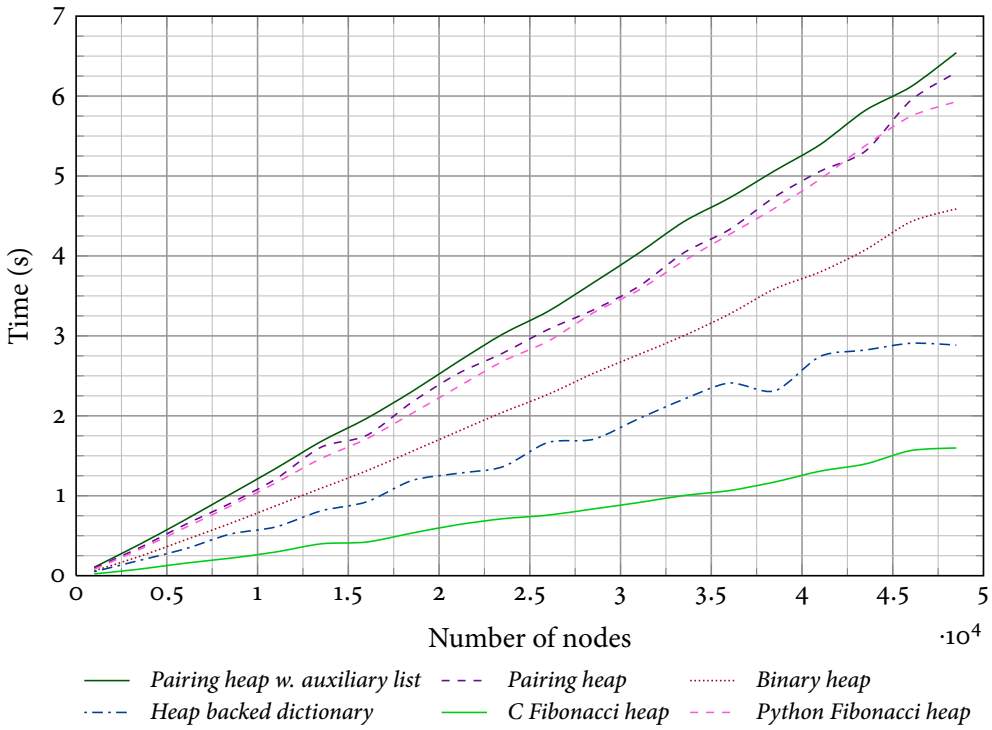


Figure 3.3 Execution times of a run of Dijkstra's single source all shortest path algorithm on differently sized Delaunay triangulation graphs.

The way this is implemented is shown in figure 3.4; this picture is redrawn from the original in the article by Jiménez & Marzal (1999). The candidate set is also represented in this article, as a simple list, but in practice this set is implemented as a heap, which allows easier selection of the shortest candidate.

After a shortest path has been found in this way, it can be reconstructed in time linear to the length of the path by following the back pointers.

The situation for the hop-constrained shortest path is rather similar, except that another index is needed to keep track of the hop-constraint.

Because the algorithm for the smallest counterexample starts out with a shortest path tree as a basis, we also take care that the shortest path tree delivered by the Dijkstra, Bellman Ford or DFS run in the first phase is built so that it can be used in the second phase.

In case of the k hop-constrained shortest path algorithm the data structure of figure 3.4 becomes even more elaborate; the fact that we need to record the hop-count adds another dimension to the data structure. For every node there are several candidate lists associated, not only for each shortest path, but also for each hop-count.

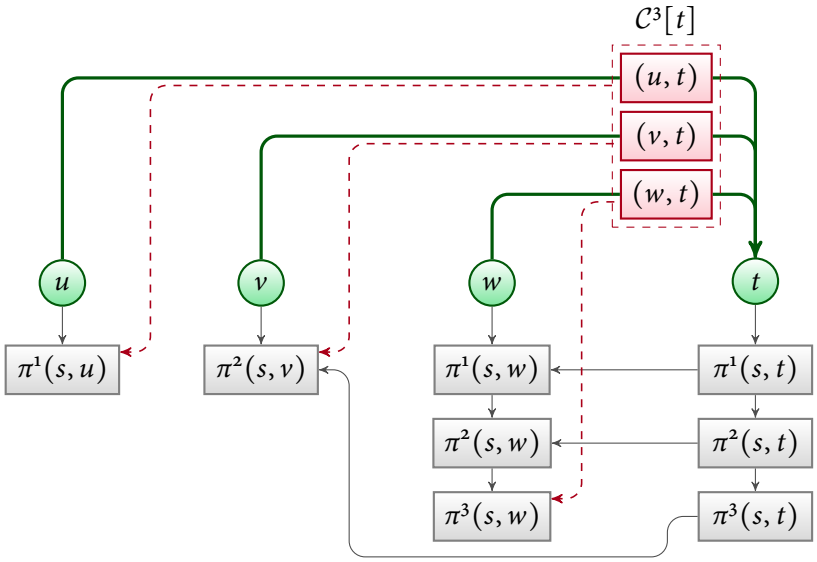


Figure 3.4 The implicit representation of paths and candidate sets in the REA. The thick (green) lines represent the structure of the graph $\mathcal{G}$. For u and v one shortest path has been computed, for w and t three have. The paths are represented by back-pointers to the predecessor node on the path. For example, the third shortest path, $\pi^3(s,t) = \pi^2(s,v)\cdot t$. Also represented is the candidate set $\mathcal{C}$ for t, which holds a candidate path for every predecessor node of t. The pointers indicates the path last considered for every node. Of these $\pi^1(s,v)$ was the shortest. If $\pi^4(s,t)$ would be computed, $\pi^1(s,v)$ would be removed from the candidate set, and $\pi^2(s,v)$ would be added, if it exists.

3.2.4.1 Alternative algorithms

An implementation of Eppstein (1998) was attempted, but abandoned. The main problem with this algorithm is that its implementation requires quite specific data structures, which, if implemented in Python, would result in quite a discrepancy in speed, simply because the underlying algorithms are not available in C. Therefore, these two implementations would not be comparable. The same problem applied to the lazy variant by Varó & Jiménez (2003).

The K* algorithm by (Aljazzar & Leue, 2008b) was only developed after the implementation had been completed, due to time constraints it was not included in the comparison, although it is in principle quite suited since it supports on the fly exploration of the graph.

3.2.5 Product graph construction

The graph product has been implemented too, but without any finesse. It simply calculates the graph product with an added final state, as described in definition 2.3.6. It is mainly meant to be able to check the outcomes by the hop-constrained k shortest path algorithm against the outcomes that are obtained by running the unconstrained algorithm on the original graph.

3.2.6 Regular expression

Implementation details about algorithms to obtain regular expressions are discussed in chapter 5.

Experimental results

4

With only the theoretical algorithms, the question remained open: how well will these algorithms perform on actual problems. This chapter presents several case studies and shows how the number of paths per counterexample grows. We first discuss the synchronous leader election, which also includes a mathematical analysis. This analysis confirms that in practice the number of paths per counterexample can grow exponentially.

The selection of case studies was based, by recommendation of MRMC author Ivan Zapreev, on those that were also covered by KATOEN *et al.* (2007). An advantage of these case studies is that these are available as PRISM files, which can be readily converted to MRMC format, and hence our software. This chapter presents a selection of three of these case studies.

A test framework was therefore put into place which is able to generate a PRISM file[1]; convert it to MRMC format and run the counterexample finding algorithm.

For the results that include timing, a problem was run several times. The average, standard deviation and fastest time was recorded. Because the problems are all deterministic in nature, that is, the same computations and outcome are expected for every run, we take the *fastest* result. The reasoning for this is that a computer even if it does 'nothing' might still have to task switch occasionally because of network traffic, a scheduled job or something like this. The fastest time is the time closest to an undisturbed run. The average and standard deviation were merely used to see whether the computer could reasonably assumed to be idle: a high standard deviation means the result is probably unreliable.

We of course tried to let the program run as undisturbed as possible by disabling the garbage checking in Python and making sure no other CPU-intensive applications were running. Nevertheless, on a multi-tasking operating system connected to the network one always has some disturbance.

We first present the synchronous leader election protocol, followed by the Crowds protocol; both case studies show that the number of paths per counterexample is enormous.

[1] Although PRISM is able to obtain some sort of parametrisation by using process renaming, in general one needs to manually add or remove some lines if one, for example, adds an extra node or computer to the example. Therefore shell-scripts were written that would generate an appropriate PRISM file, given a few parameters relevant to the problem.

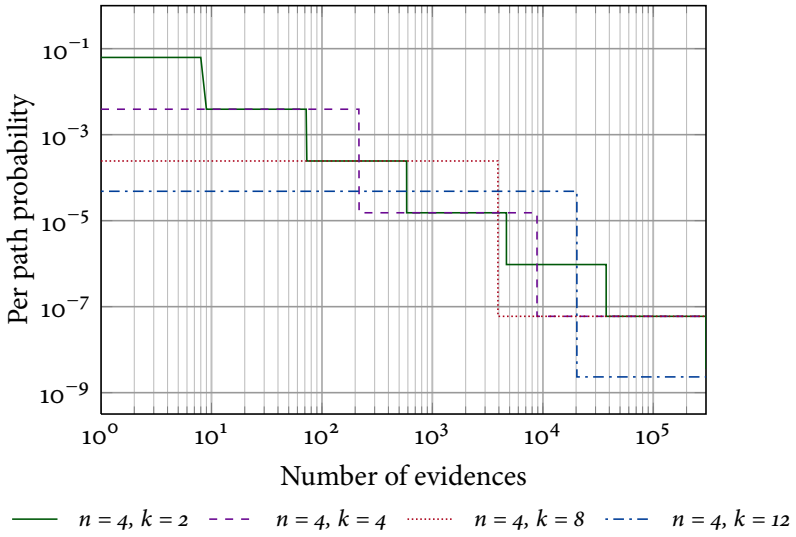


Figure 4.1 Per path probability for different instances of the synchronous leader election. n is the number of nodes in the ring, k the number of values they choose from.

4.1 Synchronous leader election

4.1.1 The protocol

The synchronous leader election protocol is rather easy. There are n processes, connected in a ring, and one of these has to become a leader. In order to establish who the leader is, they all pick a random number from some predefined range $1 \ldots r$, and then pass these numbers around on the ring. After passing $n - 1$ times – so everybody has seen every token – they decide whether a unique ID has been seen, and if so, the one with the highest ID wins and becomes a leader (or, if there is only one unique ID, the only one). If the vote fails, they try again. This gives the formula we want to check: $\mathcal{P}_{\leq 1}(\mathsf{F}\, \textit{leader elected})$[2].

The model can be analysed by hand for the greater part. It's clear that after $n + 1$ steps a decision has been reached. There is one step needed to choose a random value, then $n - 1$ to pass the values around, and then there is a final step to a 'finished'-state, if there was a unique ID, or to the start, if there was none. Each step corresponds to a transition in the DTMC. After $n + 1$ steps the DTMC is either in the starting state again, or in the accepting state, where a leader has been elected.

If we feed our formula, with the high upper bound to the software, it will need to exhaustively search every evidence it can find, because it will never be able to find a counterexample which violates this bound. Therefore, one could say this is an exercise in futility, but it very nicely shows how the contribution per path becomes less and less and how the total weight only very slowly attains the expected limit of 1.

[2] Strictly speaking, we should say $\leq 0.999\ldots$, with some finite number of nines for the upper bound, but we simply write 1.

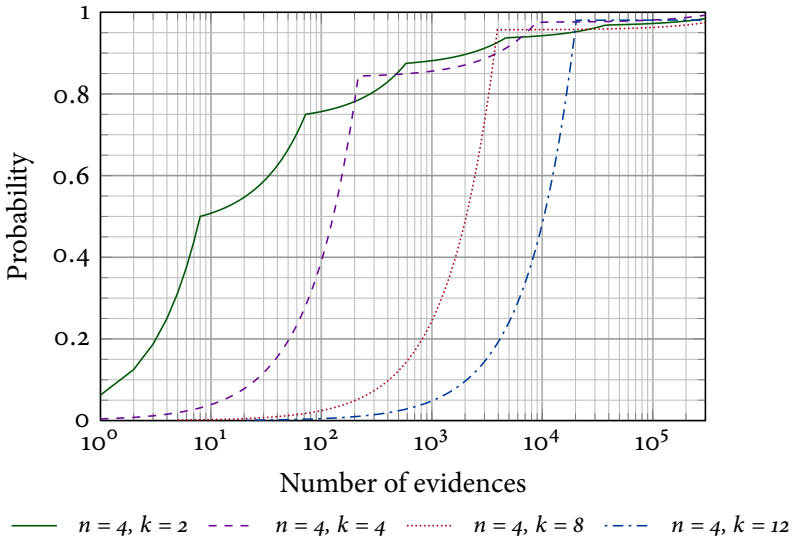


Figure 4.2 Cumulative probability for different instances of the synchronous leader election. n is the number of nodes in the ring, k the number of values they choose from.

If we look at figure 4.1, we see a very distinct staircase pattern. In every case the graph first goes horizontally, steeply drops – note that the y axis is logarithmic – and proceeds horizontally. If we look at the case $n = 4$, $k = 12$, for example, we see that this line remains horizontal after about the twenty-thousandth evidence. This means that all of these paths are equally likely. We shall explain this shape shortly, because it has a very obvious reason.

First however, we shall look at the graph of the cumulative probability, which is shown in figure 4.2. Here we have made only the x axis logarithmic; in fact this graph is piecewise linear, but without the logarithmic x axis the structure would not be so clear. What is very notable is that although all the graphs come close to the limit, in an absolute sense, there convergence is very slow.

4.1.2 Mathematical analysis

The first step of each round is probabilistic, namely, all processes pick a random value. Because the processes are ordered, and there are n of them, choosing from k values, this gives k^n possible allocations after starting, each of them is as probable as the other. After they have chosen, the process is deterministic. They pass $n - 1$ times, and then they go to the start or the end, depending on whether the allocation contained a unique value.

For example, in case of 3 nodes and 2 values to choose from, there are evidently $2^3 = 8$ possible outcomes after choosing. Of these 8 outcomes, 6 have a unique value (if we choose from $\{1, 2\}$ only 111 and 222 are not unique). So, there is a probability of $3/4$ that a leader is elected after the first round. If not, a second round is started. The probability for a second round is $1/4$, and the probability that a leader is elected in the

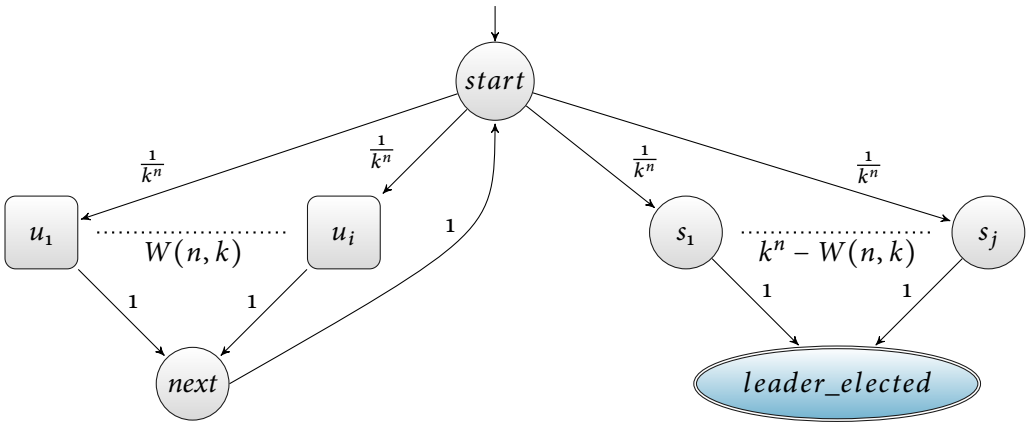


Figure 4.3 The example DTMC after the second transformation step. The probabilities have been replaced by weights and the self-loop on the final state has been removed.

second round is $\frac{1}{4}\frac{3}{4}$, or $\frac{3}{16}$. The probability a leader is elected in the third round is $\frac{1}{4}\frac{1}{4}\frac{3}{4}$, et cetera. In general we find, in this case, that a leader is elected in the n-th round is:

$$\frac{3}{4} \cdot \left(\frac{1}{4}\right)^{n-1}$$

In the same way we find that there are 6 paths to success in the first round, and $2 \cdot 6 = 12$ paths to success in the second round, and 24 in the third round, and so one. So the number of paths in each round increases, whereas the probability to be 'spread over' these paths is decreasing. Every path will have a weight that is less and less probable.

4.1.2.1 The general case

It is not trivial to find an explicit formula that gives the number of distributions given n processes and k values to in which at least one process has a unique value. The way the PRISM model is designed, each of these assignments is a different path. So the starting node has k^n successor nodes, since each of the n nodes can choose from k values.

This structure is schematically shown in figure 4.3. This structure also explains the staircase pattern. The 'highest step' of the staircase consists of a path in which a leader is immediately elected, then it drops, which consists of paths in which there is one 'next', that is, there are two election rounds; the next drop indicates third election round.

We shall now describe an expression for $W(n,k)$. This is given by the r-associated Stirling numbers of the second kind (Comtet, 1974, p.221). This give the number of ways to partition a set of n values into k (non-empty) subsets with in each subset at least r elements. The 2-associated Stirling numbers of the second kind give the number of partitions in which each subset contains at least two elements. This relation has the following recursive definition (Comtet, 1974, p.222):[3]

[3] See also: http://www.research.att.com/~njas/sequences/A008299, Sloane's Encyclopedia of

$$S_2(n,k) = kS_2(n-1,k) + (n-1)S_2(n-2,k-1). \tag{4.1}$$

One can read this as: 'The number of ways to distribute n objects over k boxes such that each box at least contains two objects is given by the way to distribute $n-1$ objects over k boxes and putting the last element in any box (with k boxes) or by keeping another element apart (for which there are $n-1$ candidates), and putting the remaining $n-2$ elements in $k-1$ boxes. The boxes are unlabelled, the elements are labelled. So, given k boxes it only matters which elements are put together in some box.

There also exists a non-recursive equation for equation 4.1, given by:

$$S_2(n,k) = \sum_{i=0}^{k}(-1)^i\binom{n}{i}\left(\sum_{j=0}^{k-i}(-1)^j\frac{(k-i-j)^{n-i}}{j!(k-i-j)!}\right) \tag{4.2}$$

Of course, this formula only makes sense for $k \leq \lfloor n/2 \rfloor$.

Given this formula, we can obtain a direct expression for the number of ways to put n labelled objects into k labelled boxes, such that no box contains exactly one object:

$$W(n,k) = \sum_{j=1}^{\min(\lfloor n/2 \rfloor, k)} S_2(n,j)\frac{k!}{(k-j)!}. \tag{4.3}$$

It does not make sense to put the objects in more than $\lfloor n/2 \rfloor$ boxes of course[4], because one has to allocate two elements to each box at least. This explains the upper bound in the summation. Then we take the number of ways to put n objects into j unlabelled boxes. Including the empty boxes, there are of course $k!$ possibilities to arrange these boxes, but we should leave out the permutations in which only the empty boxes (of which there are $k-j$) are different. This yields formula 4.3.

With equation 4.3 it is possible, given the number of processes n and the k values to find the number of allocations in which no process has a unique value. We could view the phase where each process chooses a random number from $1 \ldots k$ as assigning each process to a labelled box. If at least one of these is alone in its box, there is a unique highest value.

We thus obtain the following to express the probability that with n processes and k values the leader is elected in round m:

$$P(n,k,m) = \frac{k^n - W(n,k)}{k^n} \cdot \frac{W(n,k)}{k^n}^{m-1} \tag{4.4}$$

4.1.3 More or less experimental results

Although we have an explicit formula that gives the number of distributions in which no element obtains a unique value, we can also use the model checker to obtain this number.

integer sequences.

[4] it does make sense to have more than $\lfloor n/2 \rfloor$ boxes, because even though not more than half of the boxes will be allocated, it does matter which boxes are allocated.

The aspect about the structure of this model is that the hop bound of $n + 1$, will only give paths for which a leader is chosen in the first round. In general the formula cannot be falsified, because a leader will always be elected, but it's interesting to try, because this forces the program to find all the paths it can actually find.

This number coincides with the number of allocations in which some node has a unique value, or, $k^n - W(n, k)$. If we put the upper and lower hop-limit at $2(n + 1)$, we will find all paths in which the leader is chosen in the second round, this however can be calculated too. In general, the number of paths added each round is:

$$T(n, k, m) = (k^n - W(n, k)) \cdot W(n, k)^{m-1} \tag{4.5}$$

We can see here that this is exponential in the number of rounds. On the other hand the probability mass each round gets less and less – also exponentially. The mass per path gets hence less and less very rapidly! In table 4.1, the calculations have been performed for $k = 2$ and $n = 3 \ldots 6$. It can be clearly seen how in case of $n = 6$, the number of paths becomes terribly large after round three.

Of course, if n is large with respect to k the situation becomes worse and worse as n gets larger, but also with a large k with respect to n, the situation becomes really bad after the first round. A calculation of this can be found in table 4.2 and 4.2. In the case of $n = 6$ and $k = 2$, the number of states (329, with 392 transitions) is not excessively high at all, but the number of traces becomes enormous, really soon.

That a large n with respect to a small k is bad, is obvious since there are much more nodes than values, especially if $n > 2k$ it is expected that on average every value will be chosen at least twice. In case of a high k however, the chance at success is much larger, but just because of that the number of ways to achieve success also increases a lot.

Note that for every case presented in these tables, the final probability will be 1. In some cases, such as the first in table 4.1, it is rather clear from the progression that 1 will be reached, but for the third entry it is not very obvious. Even after more than 20 trillion (10^{12}), the cumulative probability is not more than 0.1; if one would look at the progression as new paths are found, one could readily, and wrongly, conclude that the limit would be around 0.1.

4.1.4 Tables

4.2 Crowds protocol

The previous example is quite useful because it is so easily analysed mathematically. This example doesn't allow such an easy mathematical analysis, however, it does exhibit similar behaviour when it comes to convergence and many paths with the same mass.

The protocol of this example is motivated by the lack of privacy for transactions on the www or the internet in general. This is a well-documented fact. New kinds of privacy-sensitive information, such as location information etc. about the context can possibly be traced and collected by traditional means, such as cookies, to create extensive user profiles and be disseminated (Andersson *et al.*, 2004).

	$n = 2, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	2	0.5	0.25	0.5
Round 2	4	0.25	0.0625	0.75
Round 3	8	0.125	0.015625	0.875
Round 4	16	0.0625	0.00390625	0.9375
Round 5	32	0.03125	0.000976562	0.96875
	$n = 6, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	12	0.1875	0.015625	0.1875
Round 2	624	0.152344	0.000244141	0.339844
Round 3	32 448	0.123779	$3.8147 \cdot 10^{-06}$	0.463623
Round 4	1 687 296	0.100571	$5.96046 \cdot 10^{-08}$	0.564194
Round 5	87 739 392	0.0817137	$9.31323 \cdot 10^{-10}$	0.645907
	$n = 10, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	20	0.0195312	0.000976562	0.0195312
Round 2	20 080	0.0191498	$9.53674 \cdot 10^{-07}$	0.038681
Round 3	20 160 320	0.0187758	$9.31323 \cdot 10^{-10}$	0.0574568
Round 4	20 240 961 280	0.018409	$9.09495 \cdot 10^{-13}$	0.0758658
Round 5	20 321 925 125 120	0.0180495	$8.88178 \cdot 10^{-16}$	0.0939153
	$n = 16, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	32	0.000488281	$1.52588 \cdot 10^{-05}$	0.000488281
Round 2	2 096 128	0.000488043	$2.32831 \cdot 10^{-10}$	0.000976324
Round 3	137 304 768 512	0.000487805	$3.55271 \cdot 10^{-15}$	0.00146413
Round 4	8 994 011 556 610 048	0.000487566	$5.42101 \cdot 10^{-20}$	0.00195169
Round 5	589 143 733 004 184 584 192	0.000487328	$8.27181 \cdot 10^{-25}$	0.00243902

Table 4.1 Overview of number of paths and probability mass, per path mass and the cumulative probability per round. For fixed k and varying n

	$n = 4, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	8	0.5	0.0625	0.5
Round 2	64	0.25	0.00390625	0.75
Round 3	512	0.125	0.000244141	0.875
Round 4	4 096	0.0625	$1.52588 \cdot 10^{-05}$	0.9375
Round 5	32 768	0.03125	$9.53674 \cdot 10^{-07}$	0.96875

	$n = 4, k = 4$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	216	0.84375	0.00390625	0.84375
Round 2	8 640	0.131836	$1.52588 \cdot 10^{-05}$	0.975586
Round 3	345 600	0.0205994	$5.96046 \cdot 10^{-08}$	0.996185
Round 4	13 824 000	0.00321865	$2.32831 \cdot 10^{-10}$	0.999404
Round 5	552 960 000	0.000502914	$9.09495 \cdot 10^{-13}$	0.999907

	$n = 4, k = 8$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	3 920	0.957031	0.000244141	0.957031
Round 2	689 920	0.0411224	$5.96046 \cdot 10^{-08}$	0.998154
Round 3	121 425 920	0.00176698	$1.45519 \cdot 10^{-11}$	0.999921
Round 4	21 370 961 920	$7.59249 \cdot 10^{-05}$	$3.55271 \cdot 10^{-15}$	0.999997
Round 5	3 761 289 297 920	$3.2624 \cdot 10^{-06}$	$8.67362 \cdot 10^{-19}$	1

	$n = 4, k = 12$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	20 328	0.980324	$4.82253 \cdot 10^{-05}$	0.980324
Round 2	8 293 824	0.0192888	$2.32568 \cdot 10^{-09}$	0.999613
Round 3	3 383 880 192	0.000379525	$1.12157 \cdot 10^{-13}$	0.999992
Round 4	1 380 623 118 336	$7.4675 \cdot 10^{-06}$	$5.40879 \cdot 10^{-18}$	1
Round 5	563 294 232 281 088	$1.4693 \cdot 10^{-07}$	$2.60841 \cdot 10^{-22}$	1

Table 4.2 Overview of number of paths and probability mass, per path mass and the cumulative probability per round. For fixed n and varying k

	$n = 5, k = 2$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	10	0.3125	0.03125	0.3125
Round 2	220	0.214844	0.000976562	0.527344
Round 3	4 840	0.147705	$3.05176 \cdot 10^{-05}$	0.675049
Round 4	106 480	0.101547	$9.53674 \cdot 10^{-07}$	0.776596
Round 5	2 342 560	0.0698137	$2.98023 \cdot 10^{-08}$	0.84641

	$n = 5, k = 4$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	900	0.878906	0.000976562	0.878906
Round 2	111 600	0.10643	$9.53674 \cdot 10^{-07}$	0.985336
Round 3	13 838 400	0.012888	$9.31323 \cdot 10^{-10}$	0.998224
Round 4	1 715 961 600	0.00156066	$9.09495 \cdot 10^{-13}$	0.999785
Round 5	212 779 238 400	0.000188986	$8.88178 \cdot 10^{-16}$	0.999974

	$n = 5, k = 8$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	32 200	0.982666	$3.05176 \cdot 10^{-05}$	0.982666
Round 2	18 289 600	0.0170335	$9.31323 \cdot 10^{-10}$	0.9997
Round 3	10 388 492 800	0.000295259	$2.84217 \cdot 10^{-14}$	0.999995
Round 4	5 900 663 910 400	$5.11801 \cdot 10^{-06}$	$8.67362 \cdot 10^{-19}$	1
Round 5	3 351 577 101 107 200	$8.87155 \cdot 10^{-08}$	$2.64698 \cdot 10^{-23}$	1

	$n = 5, k = 12$			
	Paths	Prob. mass	Per-path mass	Cum. Prob
Round 1	247 500	0.994647	$4.01878 \cdot 10^{-06}$	0.994647
Round 2	329 670 000	0.00532435	$1.61506 \cdot 10^{-11}$	0.999971
Round 3	439 120 440 000	$2.85013 \cdot 10^{-05}$	$6.49055 \cdot 10^{-17}$	1
Round 4	584 908 426 080 000	$1.52568 \cdot 10^{-07}$	$2.60841 \cdot 10^{-22}$	1
Round 5	779 098 023 538 560 000	$8.16697 \cdot 10^{-10}$	$1.04826 \cdot 10^{-27}$	1

Table 4.3 Overview of number of paths and probability mass, per path mass and the cumulative probability per round. For fixed n and varying k

Crowds (Reiter & Rubin, 1998) is a protocol for anonymous communication to enhance privacy. It uses random routing within a group of nodes (a crowd) to establish a connection path between a sender and a receiver. Routing paths are reconstructed once the crowd changes; the number of such new route establishments is R, and this is an important parameter that influences the state space. Random routing serves to hide the secret identity of a sender.

The protocol works in the following way: Firstly: the sender selects a crowd member at random (possibly itself), and forwards the message to it, encrypted by the corresponding pairwise key. Secondly, the selected router flips a biased coin. With probability $1 - p_f$, where p_f (forwarding probability) is a parameter of the system, it delivers the message directly to the destination. With probability p_f, it selects a crowd member at random (possibly itself) as the next router in the path, and forwards the message to it, re-encrypted with the appropriate pairwise key. The next router repeats this step.

In our experiments, we assume that if a sender has been observed by the bad member twice, then it has been *positively identified* (*Pos* for short), thus the anonymity is not preserved; secondly the bad member will deliver the message with probability 1. This protocol is executed every time one crowd member wants to establish an anonymous connection to a Web server. We call one run of the protocol a *session* and denote the number of sessions by R. Other parameters are the number of good members N and the number of bad members B.

The rationale that a bad crowd member is actually able to identify the sender lies with the fact that these bad members can work together and can piece some of the information in the message together. Cookies can be inspected, and timing information (the loading of a web page probably initiates requests for style sheets and images) can be used to build a profile. This way, a profile can be build and routers can be more confident that the node they receive the message from is indeed the originator.

This protocol is able to guarantee 'probable innocence', that is, the probability that the real sender of a message sends it message to a bad crowd member is less than 0.5, if the following condition holds:

$$N + B \geq \frac{p_f}{p_f - \frac{1}{2}}(B + 1)$$

For our experiment we take the Crowds protocol modelled by prism and the property is $\mathcal{P}_{\leq p}(\mathsf{F}\, Pos)$ which characterizes the probability threshold that the original sender's id 0 is positively identified by the corrupt members. The relation between the number of evidences and the probability threshold for different number of sessions R is shown in figure 4.4 ($N = 5$, $B = 1$, $p_f = 0.8$). It is clear that a staircase pattern also occurs in this case, every horizontal line indicates paths of the same probability.

Finally, we take a look at the cumulative probability, this varies for each value of R. The graphs are shown in figure 4.5. From this graph it also becomes clear that, especially for the cases of $R = 4$ and $R = 5$, the graph does not even come close to the value as computed by prism (Kwiatkowska *et al.*, 2002). It is only for $R = 3$ that after 300,000 paths we are approaching the limit, although in all three cases the per

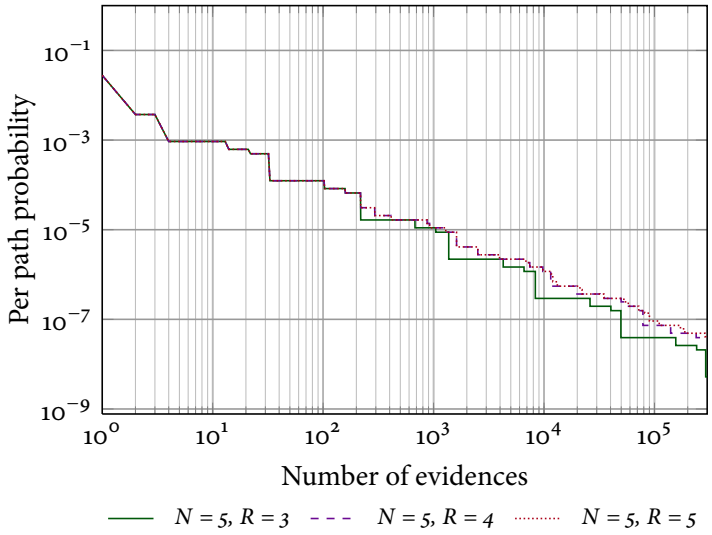


Figure 4.4 Per path probability for different instances of the crowds protocol. The number of path reformulations R, varies. There is one bad crowd member.

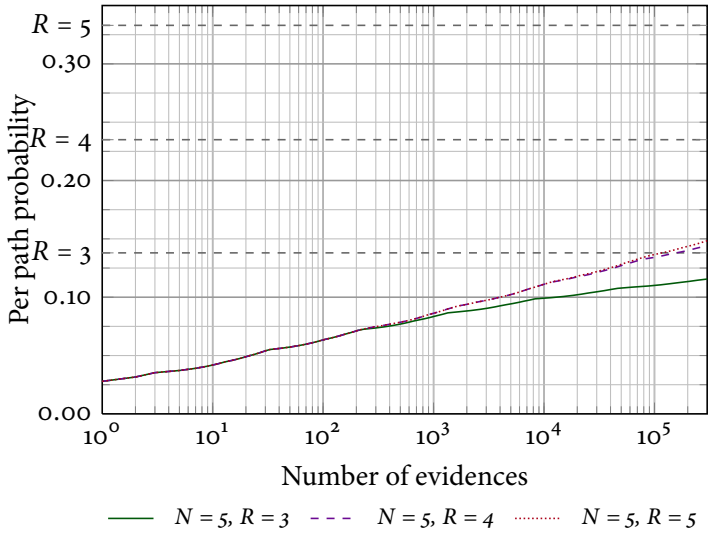


Figure 4.5 Cumulative probability for different instances of the Crowds protocol. N is the number of crowd members in the ring, R the number of path reformulations. The forwarding probability p_f is 0.8 and there is one bad member. The theoretical limits are also indicated by dashed lines.

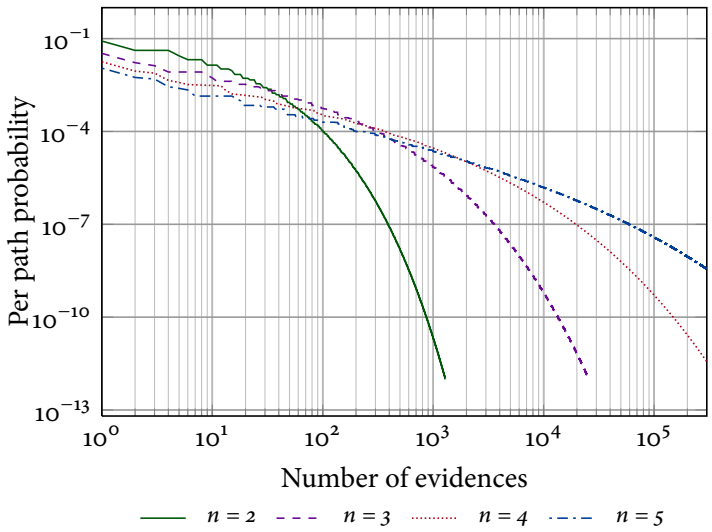


Figure 4.6 Per path probability for different instances of the randomised mutual exclusion algorithms, shown for different number of processes.

path probability mass has already dropped to less than 10^{-7}, as can be seen in figure 4.4, it will therefore require many more paths to reach the limit.

4.3 Randomised mutual exclusion

The randomised mutual exclusion protocol by Pnueli & Zuck (1984) gives a probabilistic solution to the n process mutual exclusion problem. There are n processes that try to enter a critical section, while basing their choices on coin tosses. The protocol guarantees that eventually every process can enter the critical section, but not at the same time. We verify the property $\mathcal{P}_{\leq 1}\left(\bigwedge_{j\neq 1}^{N} \neg enter_j \,\mathsf{U}\, enter_1\right)$.

The interesting aspect about this example is that it, in a way, behaves differently from the previous two. The typical 'staircase pattern', this can be seen in figure 4.6. There are not very many horizontal parts in the graph – there are some, but they are not very clearly notable. The per path probability drops so quickly that it falls beyond the precision of cumulative result. After about 1200 paths, the cumulative probability for example does not increase any further. Therefore the computation is halted. This can also be seen in figure 4.7, where the theoretical limits – which are simply $1/n$ are also drawn. The fact that it is possible to exhaustively search the model indicates that there are no cycles, also the lack of horizontal lines indicates that there are probably not very many 'similar' scenario's. We shall see the importance of these observations when we try to compact the counterexample.

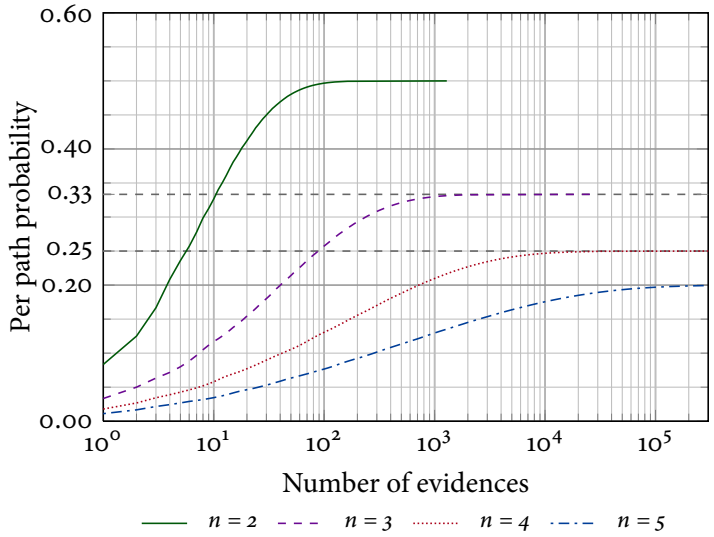


Figure 4.7 Cumulative probability for different instances of the mutual exclusion algorithm. n is the number of processes. The theoretical limits $\left(\frac{1}{n}\right)$ are also indicated by dashed lines.

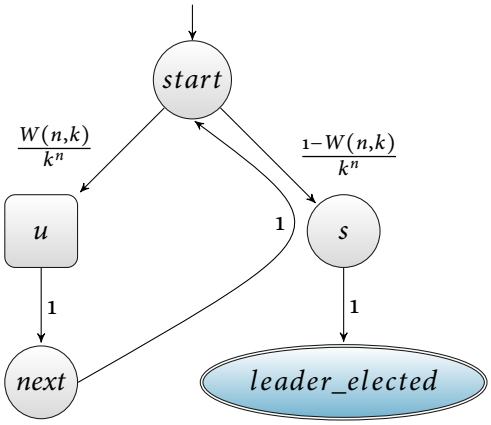


Figure 4.8 Schematic structure of the synchronous leader election DTMC after bisimulation minimization.

4.4 Bisimulation minimisation

The previous experiments show that counterexamples can become very large. Investigation of the counterexamples however shows, that in the cases of synchronous leader election and the Crowds protocol, many paths with the same value occur. This leads to the suspicion that successes could be achieved by trying to lump states together, at least in these two cases. From the results in (KATOEN *et al.*, 2007) we can deduce that large reductions in state space are possible, and hence, for the number of paths. The bisimulation minimised version of figure 4.3 is shown in figure 4.8.

In order to obtain these models however, we needed the help of MRMC; but by default it only reads the model, reduces it and runs the requested verification. There was no possibility to tell MRMC to export the reduced model, we therefore wrote a small patch that allowed exporting the model after minimisation.

We first provide an overview of the achieved reductions for the cases presented above in table 4.4.

Table 4.4 Showing the state space reductions bisimulation minimisation can achieve. As expected, the reductions are very large in case of the leader election and Crowds protocol, but less so in case of randomised mutual exclusion.

	Synchronous leader election ($n = 4$)			
	Original		Minimised	
	States	Transitions	States	Transitions
$k = 2$	55	70	10	11
$k = 4$	782	1 037	10	11
$k = 8$	12 302	16 397	10	11
$k = 12$	62 222	82 957	10	11
	Crowds Protocol ($N = 4$)			
	Original		Minimised	
	States	Transitions	States	Transitions
$R = 3$	1 198	2 038	93	130
$R = 4$	3 515	6 038	170	245
$R = 5$	8 653	14 935	269	395
	Randomised mutual exclusion			
	Original		Minimised	
	States	Transitions	States	Transitions
$n = 2$	188	455	151	376
$n = 3$	2 368	8 272	2 109	7 461
$n = 4$	27 600	123 883	25 385	114 440
$n = 5$	308 800	1 680 086	282 709	1 538 585

We now reproduce each of the graphs of the previous section. It is clear that the bisimulation minimisation removes the staircase pattern, as can be seen in the graphs of figure 4.10 and figure 4.11. We can even see that the program halts before 300 000 paths have been found in the case of leader election. This is a bit misleading, because in principle there are infinitely many paths, but the value drops so quickly that it drops below the accuracy of the floating point value. Therefore, as can now be seen in figure 4.10, the counterexample becomes very small.

In the case of the Crowds protocol, convergence has also improved. For the cases

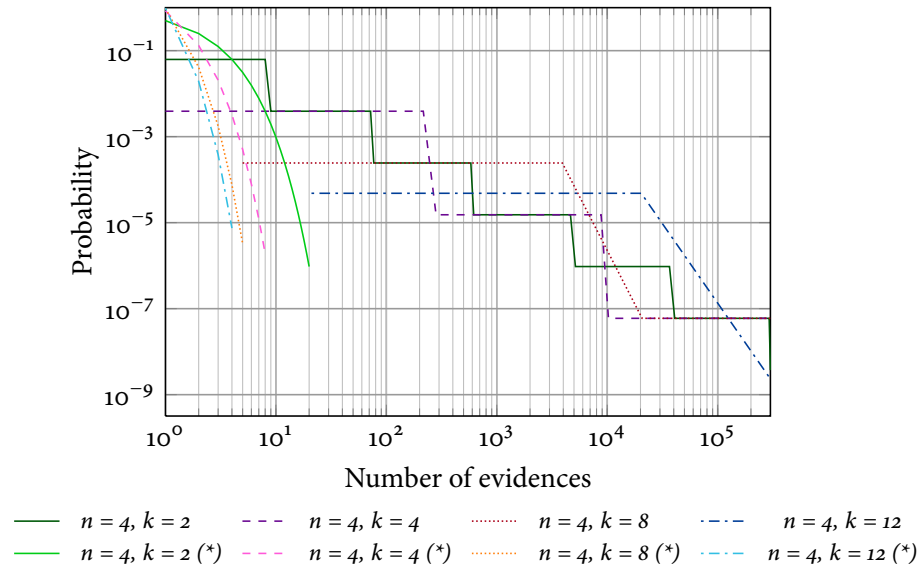


Figure 4.9 Per path probability for different instances of the synchronous leader election. n is the number of nodes in the ring, k the number of values they choose from. The starred (*) data indicates minimised versions.

$R = 4$ and $R = 5$, it is still a long way to the theoretical limit, as can be seen in figure 4.12, but the difference with the original versions is obvious.

However, in the case of the randomised mutual exclusion algorithm, we can see that bisimulation minimisation show no discernible improvement; we had expected this partly because of the fact that the reduction of states was very slim. Measured bisimulation reduction would yield better results, but as explained, we were not able to use this.

So, bisimulation minimisation is sometimes really able to reduce the number of paths in the counterexample. We should note however that if one checks formulae with a hop-bound, the minimisation may very well yield results that violate the hop-bound in the original model. Also, if the original model has clear underlying semantics, the lumped model might be harder to interpret.

Still, however, counterexample sizes of this size are very large. Bisimulation especially does not help if a cycle appears in the model: new solutions can be formed by just repeating this cycle.

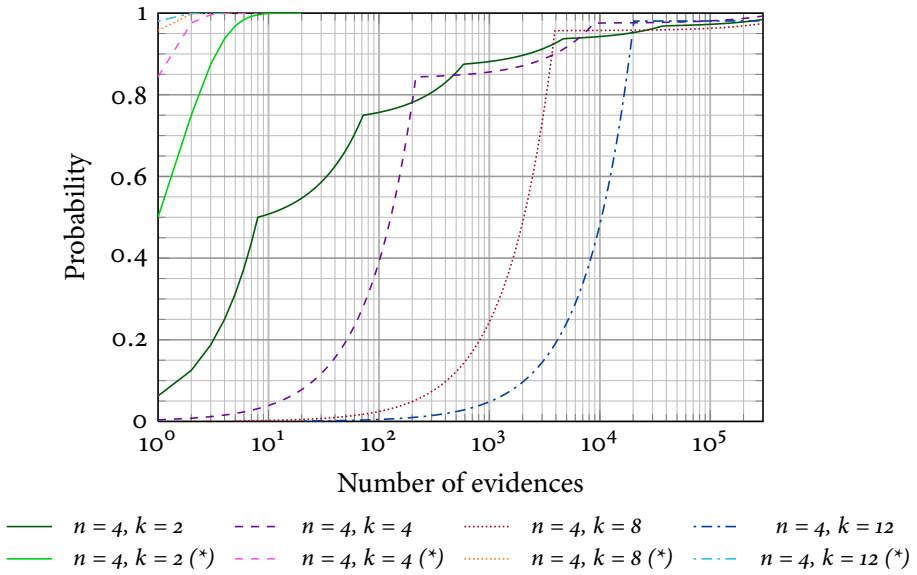


Figure 4.10 Cumulative probability for different instances of the synchronous leader election. n is the number of nodes in the ring, k the number of values they choose from. The starred (*) data indicates minimised versions.

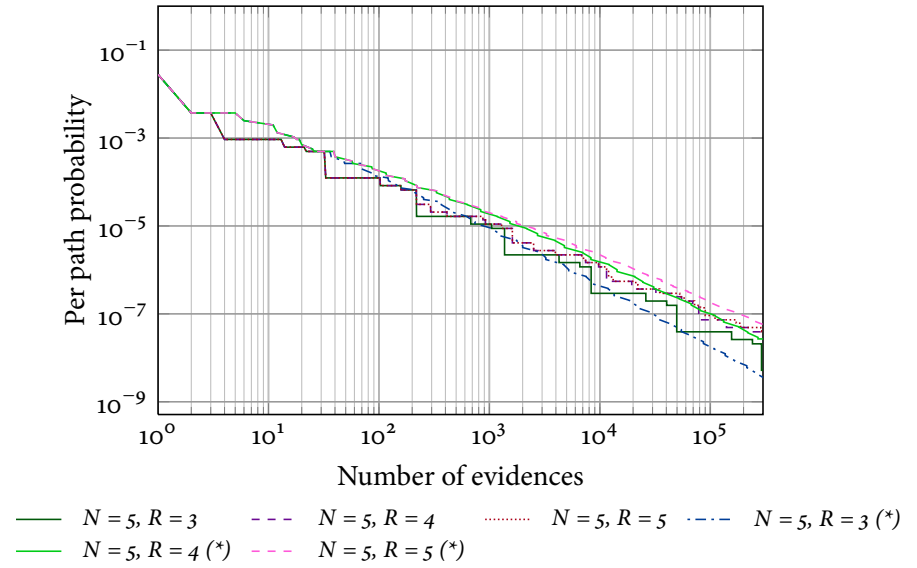


Figure 4.11 Per path probability for different instances of the crowds protocol. The number of path reformulations R, varies. There is one bad crowd member. The starred (*) data indicates minimised versions.

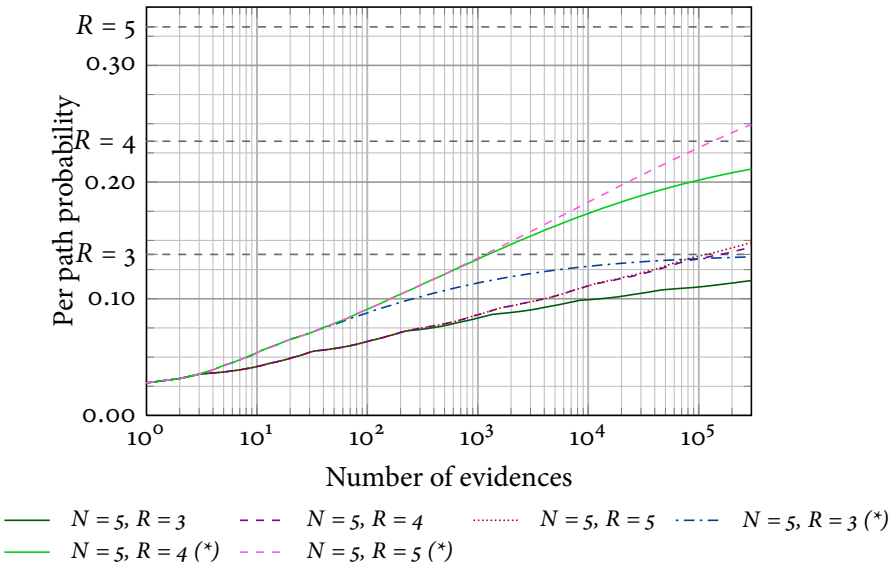


Figure 4.12 Cumulative probability for different instances of the Crowds protocol. *N* is the number of crowd members in the ring, *R* the number of path reformulations. The forwarding probability p_f is 0.8 and there is one bad member. The theoretical limits are also indicated by dashed lines. The starred (*) data indicates minimised versions.

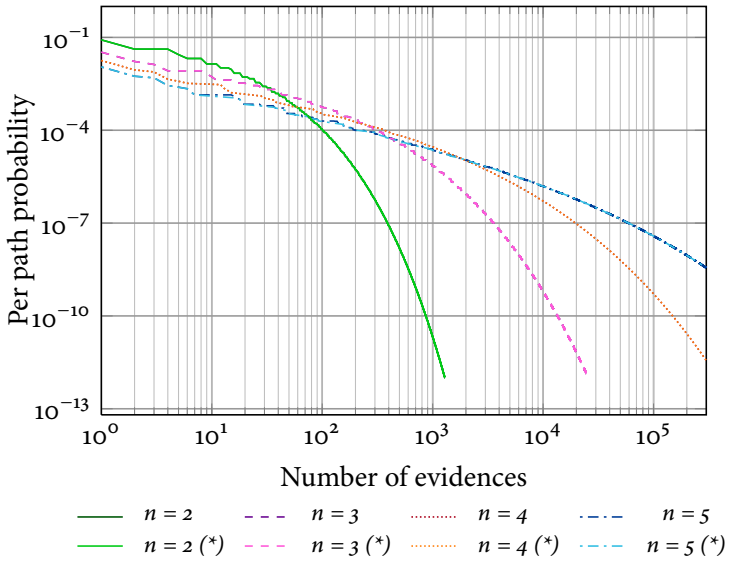


Figure 4.13 Per path probability for different instances of the randomised mutual exclusion algorithms, shown for different number of processes. The starred (*) data indicates minimised versions.

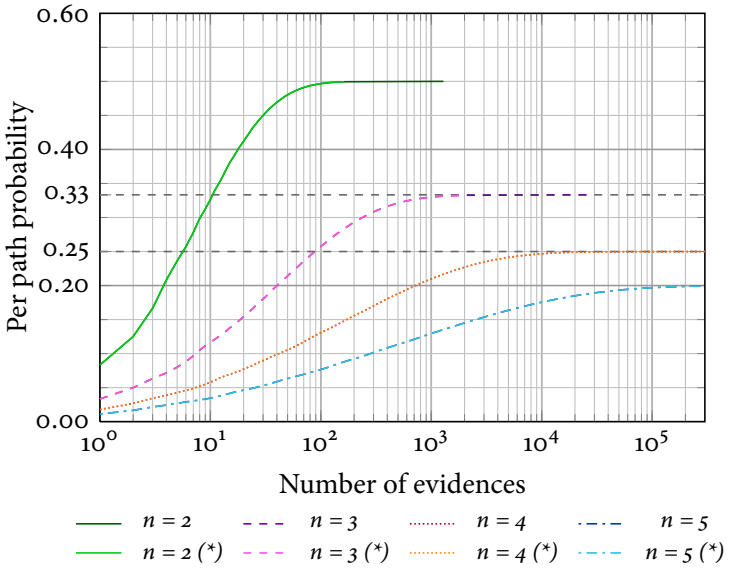


Figure 4.14 Cumulative probability for different instances of the mutual exclusion algorithm. n is the number of processes. The theoretical limits $\left(\frac{1}{n}\right)$ are also indicated by dashed lines. The starred (*) data indicates minimised versions.

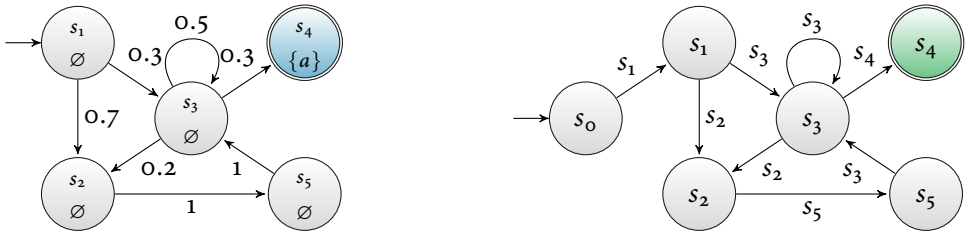
Regular representations

5

The previous chapter shows that very often counterexamples comprise of thousands, if not millions of paths. We have observed that there are two reasons for this: on the one hand cycles in the automaton, which introduce paths in which a pattern is repeated, on the other hand branching such as in the leader election example.

In this chapter we propose a way to compactly represent counterexamples by means of regular expressions, this can be used in addition to bisimulation minimisation. Our main aim is to be able to capture cycles compactly. Both methods can be used orthogonally, it will be shown that the regular expressions indeed capture the cyclic behaviour compactly, whereas the bisimulation minimisation is excellent for similar behaviour.

This chapter is an extended coverage of the material in Damman *et al.* (2008). We show how this, combined with the bisimulation minimisation, can yield very compact representations of counterexamples. We conclude this chapter with some remarks on the practical implementation and a few results obtained from the case studies presented in the previous chapter.



a Example DTMC

b A DFA derived from the DTMC.

Figure 5.1 An example of a DTMC with target state s_4 and a DFA with final state s_4. Note that the string $s_1 \cdot s_3 \cdot s_4$ is accepted by the DFA, and that this is also a path in the DTMC and an evidence for $\mathcal{P}_{\leq p}\, \mathsf{F}\, a$.

5.1 From DTMCs to regular expressions.

5.1.1 Introduction

We first show the idea by means of an example. A DTMC is given in figure 5.1a. We could list evidences of the formulae $\mathcal{P}_{\leq p}\,\mathsf{F}\,a$ as $s_1{\cdot}s_3{\cdot}s_4$, $s_1{\cdot}s_3{\cdot}s_3{\cdot}s_4$, $s_1{\cdot}s_3{\cdot}s_3{\cdot}s_3{\cdot}s_4$, et cetera. We see from this pattern that in general, we could say that the set described by the regular expression $s_1 s_3 (s_3)^* s_4$ consists of evidences. Also, we can easily calculate the probability of all these evidences. We obtain the following expression for the total probability of these expressions:

$$
\begin{aligned}
&\sum_{i=0}^{\infty} 0.3 \cdot 0.5^i \cdot 0.3 \\
&= 0.3 \left(\sum_{i=0}^{\infty} 0.5^i \right) 0.3 \\
&\quad \textit{Geometric series} \\
&= 0.3 \cdot 2 \cdot 0.3 \\
&= 0.18.
\end{aligned}
$$

We see that we can easily calculate the total probability of such a regular expression. It is clear that the whole set of evidences for a formula can be seen as a language over the alphabet of state names. Every evidence is a word in this language. If we would be able to find a regular expression for this language, we could easily calculate the probability of this expression, and thus of this set of evidences, and hence conclude whether this set would form a counterexample.

Noting that regular expressions and DFA can be used interchangeably, because they are equivalent, it would also suffice to find a DFA that describes the language. An example of such a DFA is shown in figure 5.1b. It is created by adding an extra state as a predecessor of the original starting state, and by copying the state names onto the incoming transitions as a label. Since state names are unique, and there is only at most one transition between every pair of states this will yield a DFA. Every path through this DFA obviously has a corresponding path in the DTMC.

The previous represents the main idea to go from DTMCs to finite automata and this is described more formally in the next section.

5.1.2 Formal definition

A major difference with usual regular expressions and usual edge labels is that we need to keep track of the transition probabilities too; we did not do this explicitly in our example, but shall do this in the formal definition of this section. To tackle this, we adopt the approach proposed by Daws (2004). He uses regular expressions to represent sets of paths and calculates the exact rational value of the probability measure in DTMC model checking (provided all transition probabilities are rational). His main

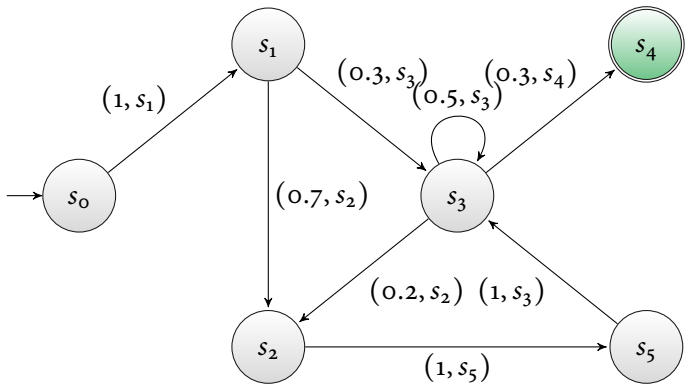


Figure 5.2 The DFA derived from the DTMC of figure 5.1a. Note that the symbols on the transitions are tuples consisting of a probability and a label.

goal is to have expressions with certain parameters, which allow for parametric model checking.

We do not consider parametric model checking, but we do adapt his approach to obtain compact representations of counterexamples. The main idea is to consider a counterexample as a set of probable branches (sub-expressions) that go from the initial state to the goal state. In example, besides the paths starting with $s_1 \cdot s_3$, there is another set starting with $s_1 \cdot s_2$. This set would form a second branch in the regular expression.

We also provide a function to evaluate the probability measure of those expressions. To simplify the presentation we will assume that the DTMC at hand has been subject to the first step of the transformation described in section 2.2.1. This is not a limitation, since $s \vDash \mathcal{P}_{\trianglelefteq p}(\Phi \, \mathsf{U}^{\leqslant h} \, \Psi)$ in a DTMC if and only if $s \vDash \mathcal{P}_{\trianglelefteq p}(\mathsf{F}^{\leq h+1} \; at_t)$ in the transformed DTMC where at_t uniquely identifies t.

Definition 5.1.1. For DTMC $\mathcal{D} = (S, \mathbf{P}, L)$ with initial state $\hat{s} \in S$ and goal state t, let the deterministic finite automaton DFA $\mathcal{A}_{\mathcal{D}} = (S', \Sigma, \tilde{s}, \delta, \{t\})$, where:

- $S' = S \cup \{\tilde{s}\}$ is the state space with start state $\tilde{s} \notin S$;
- $\Sigma \subseteq (0, 1] \times S$ is the (finite) alphabet;
- $\delta \subseteq S' \times \Sigma \times S'$ is the transition relation such that $\delta\,(s, (p, s')) = s'$ iff $\mathbf{P}(s, s') = p$, and $\delta(\tilde{s}, (1, \hat{s})) = \hat{s}$;
- $t \in S$ is the accepting state.

The automaton is equipped with a start state $\tilde{s}$ and a transition of probability one to the initial state of $\mathcal{D}$. The symbols in the alphabet are pairs (p, s) where p is a probability and s a state. A transition $s \xrightarrow{p} s'$ in $\mathcal{D}$ is converted into a transition from s to s' labelled with (p, s'). (Again, obviously, this yields a deterministic automaton.) This is a slight, though important deviation from (Daws, 2004), where labels are just probabilities. The probabilities are needed to determine the path probabilities (see definition 5.1.3), while the target states are used for recovering the evidences. For simplicity, probability labels are omitted if they are clear from the context.

Example 5.1.2. Figure 5.1a depicts an abstract example of a DTMC $\mathcal{D}$ with initial state $\hat{s} = s_1$ and goal state $t = s_4$, and its DFA $\mathcal{A}_\mathcal{D}$ given by definition 5.1.1 is shown in figure 5.2. The new start state is $\tilde{s} = s_0$, which has a transition equipped with symbol $(1, s_1)$ to s_1.

5.1.3 Evaluation of regular expressions

The previous makes clear how regular expressions can be used to represent a counterexample C. To determine the probability of C, $\Pr(C)$, from this regular expression we use an evaluation function. Let $\mathcal{R}(\Sigma)$ be the set of regular expressions over the finite alphabet Σ. It contains the elements of Σ, the empty word ε, and is closed under union ($|$), concatenation ($\cdot$) and Kleene star ($*$). Let $\mathcal{L}(r)$ denote the regular language (a set of words) described by the regular expression $r \in \mathcal{R}(\Sigma)$ and $\mathcal{L}(\Sigma)$ denote the regular language that can be generated by any regular expression over Σ. The length $|w|$ and $|r|$ denote the number of symbols in the word w and regular expression r, respectively. We sometimes omit $\cdot$ and write $r{\cdot}r'$ as rr' for short. Note that in our setting, $\Sigma \subseteq (0, 1] \times S$.

Definition 5.1.3 (Daws (2004), Evaluating regular expressions)**.** Let $\mathrm{val} : \mathcal{R}(\Sigma) \to \mathbb{R}$ be defined as:

$$\mathrm{val}(\varepsilon) = 1 \qquad \mathrm{val}(r \mid r') = \mathrm{val}(r) + \mathrm{val}(r')$$
$$\mathrm{val}((p, s)) = p \qquad \mathrm{val}(r{\cdot}r') = \mathrm{val}(r) \times \mathrm{val}(r')$$

$$\mathrm{val}(r^*) = \begin{cases} 1 & \text{if } \mathrm{val}(r) = 1 \\ \frac{1}{1-\mathrm{val}(r)} & \text{otherwise} \end{cases}$$

If we limit the transition probabilities to be rational values, exact values are obtained. It can be proven that $\mathrm{val}(r) = \Pr\left(Paths_{fin}\min(\hat{s}, \mathsf{F}^{\leq h}\, at_t)\right)$, for $h = \infty$. Daws (2004)

Definition 5.1.4. r_1 is a *maximal union subexpression* (MUS) of a regular expression r if $r = r_1 | r_2$ modulo $(\mathbf{R_1})$-$(\mathbf{R_3})$, for some $r_2 \in \mathcal{R}(\Sigma)$, where:

$$(\mathbf{R_1}) \quad r \equiv r \mid \varepsilon$$
$$(\mathbf{R_2}) \quad r_1 \mid r_2 \equiv r_2 \mid r_1$$
$$(\mathbf{R_3}) \quad r_1 \mid (r_2 \mid r_3) \equiv (r_1 \mid r_2) \mid r_3$$

r_1 is maximal because it is at the topmost level of a union operator. If the topmost level operator is not union, then $r_1 = r$ (cf. $\mathbf{R_1}$). A regular expression represents a set of paths and each MUS can be regarded as a main branch from the start state to the accepting state.

Example 5.1.5. A regular expression for the automaton $\mathcal{A}_\mathcal{D}$ in figure 5.2 is:

$$r_0 = \underbrace{s_1 s_3 s_3^* s_4}_{r_1} \mid \underbrace{s_1(s_2 \mid s_3 s_3^* s_2)(s_5 s_3 s_3^* s_2)^* s_5 s_3 s_3^* s_4}_{r_2}.$$

r_1 and r_2 are the MUS s of r_0 with $\mathrm{val}(r_1) = 1 \times 0.3 \times \frac{1}{1-0.5} \times 0.3 = 0.18$ and $\mathrm{val}(r_2) = 0.82$. Note that $|r_1| = 4$ and $|r_2| = 13$; $w = s_1 s_3 s_3 s_3 s_4$ is a word generated by r_1 and

$|w| = 5$. We can distribute $|$ over . in r_2 and obtain two more MUS s instead: $r_3 = s_1 s_2 (s_5 s_3 s_3^* s_2)^* s_5 s_3 s_3^* s_4$ and $r_4 = s_1 s_3 s_3^* s_2 (s_5 s_3 s_3^* s_2)^* s_5 s_3 s_3^* s_4$. r_1, r_3 and r_4 characterize all paths from s_1 to s_4, which fall into the above three branches. Note that r_1 *cannot* be written as $s_1 s_3^+ s_4$, since from the full form of $r_1 = (1, s_1)(0.3, s_3)(0.5, s_3)^*(0.3, s_4)$, the probability of the first s_3 is different from that of s_3^*.

Remark 5.1.6. The previous example already shows that there is more than one way to write a regular expression. We could also have presented the expression: $s_1(s_2 s_5 s_3 \mid s_3)\cdot(s_2 s_5 s_3 \mid s_3)^* s_4$ which also describes the language. This one is arguably a bit more intuitive, and at least shorter. Finding such an 'optimal' regular expression is hard however, we shall discuss this after the next paragraph.

5.1.3.1 Interpretation of valuations

This paragraph tries to elucidate some aspects with respect to the interpretation of the valuation function. For this, we first go back to the theory of non-regular Markov chains (i.e. those without a limiting distribution. These are the ones we have, because the extra state (t)) is always absorbing. We can write the Markov Chain in matrix form as:

$$\begin{bmatrix} Q & R \\ 0 & I \end{bmatrix}$$

Where I is an identity matrix for the absorbing states (besides our target state t, for example states in which the formula Φ we want to check is not satisfied), Q is the transient part of the matrix, and R encodes the edges from transient to absorbing states; finally 0 consists of only zeroes.

Using this, the matrix (the I here is a different identity matrix of course, with the dimensions appropriate for Q) we define:

$$W = (I - Q)^{(-1)}$$

as the fundamental matrix of the DTMC; the entries in it are the sojourn times of every state. It is furthermore known that the hitting probabilities U can be defined in terms of this matrix:

$$U = WR$$

Where R is the matrix fragment of the transition matrix. For example, for the DTMC of figure 5.1a we have (the order is s_1, s_2, s_3, s_5)

$$Q = \begin{bmatrix} 0 & \frac{7}{10} & \frac{3}{10} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{1}{5} & \frac{1}{2} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The corresponding W is:

$$W = \begin{bmatrix} 1 & \frac{41}{3} & \frac{41}{30} & \frac{10}{3} \\ 0 & \frac{5}{3} & \frac{10}{3} & \frac{5}{3} \\ 0 & \frac{2}{3} & \frac{10}{3} & \frac{2}{3} \\ 0 & \frac{2}{3} & \frac{10}{3} & \frac{5}{3} \end{bmatrix}$$

The entry $W_{x,y}$ now gives the expected sojourn time in state y, if we start x. So, we only spend 1 'unit' in state 1 (which is s_1) if we start there; and we expect to spend 10/3 units in s_3 if we start there – or alternatively: after we have arrived there. If we take a look at remark 5.1.6, and look at the part of the regular expression that expresses the looping in s_3, viz. $(s_2 s_5 s_3 \mid s_3)^*$, and calculate the valuation we find that this is $\frac{1}{1-(0.2+0.5)} = \frac{10}{3}$.

It might look odd at first sight that this valuation function can yield values which are larger than one, since we expect to obtain a *probability*. But this is not always the case: in a sense, this is not unexpected either, since the probability of leaving s_3 is $\frac{3}{10}$, and we end up with probability 1 in s_4, and of course $\frac{10}{3} \cdot \frac{3}{10} = 1$, and indeed this is what the following mathematical analysis tells us

We can define R simply as a single column:

$$R = \begin{bmatrix} 0 & 0 & 3/10 & 0 \end{bmatrix}$$

Which allows us to compute $U = WR = [1\,1\,1\,1]$, which means: No matter where we start, the probability of hitting s_4 is 1; this is obvious, since it is the only absorbing state.

The proper interpretation of val for the $*$ appears closely related to this. For the other parts it works to just think of probabilities being multiplied, although, in a sense these can be interpreted as expected sojourn times too. What is clear however that the valuation of the whole expression, of a MUS can always be interpreted as a probability, since this involves multiplying with a transition leading to an absorbing state.

When dealing with larger models and larger regular expressions the interpretation is a bit more cumbersome, since subexpressions can appear several times in the formula.

5.1.4 Regular expressions as counterexamples

The equivalence of DFAs and regular expressions, as well as converting DFAs to regular expressions has been widely studied. Several techniques are known, e.g., the transitive closure method first described by KLEENE (1956), or the algebraic method by BRZOZOWSKI (1964) and elaborated by BERRY & SETHI (1986), and different variants of state elimination by DU & KO (2001) and LINZ (2001). State elimination is based on removing states one by one, while labelling transitions by regular expressions. It terminates after only the start and accepting state remain; the transition connecting these states is labelled with the resulting regular expression. This technique is suitable for manual inspection but is less straightforward to implement efficiently. Humans can easily pick a sensible state to remove, recognizing the patterns that occur: picking the 'wrong' state yields a very long expression.

The transitive closure method gives a clear and simple implementation but tends to create rather long regular expressions. The algebraic method is elegant and generates reasonably compact regular expressions. In order to obtain a minimal counterexample in an *on-the-fly* manner, we take the state elimination method. This allows us to stop once the value of the obtained regular expression exceeds the probability threshold. The algebraic method does not support this.

By using regular expressions for representing counterexamples, we will, instead of obtaining evidences one by one, derive a larger number of evidences at a time, which hopefully yields a quick convergence to the required probability threshold and a clear explanation of the violation. As a result, we will not insist on obtaining the smallest counterexample – in fact, every expression with a Kleene star describes an infinite set in fact – but instead prefer to find the branches (MUS s) with large probabilities and short length. Thus, a (good) regular expression should be:

1. shorter (wrt. its length), to improve comprehensibility;
2. more probable, such that it is more informative and the algorithm will terminate with less MUS s;
3. minimal, where a compact counterexample is *minimal* if the omission of any of its MUS s would no longer result in a counterexample.

However, it has been recently proven by Gramlich & Schnitger (2007) that the size of a shortest regular expression of a given DFA cannot be efficiently approximated. Therefore, it is not easy to, e.g., by state elimination, compute an optimal removal sequence for state elimination in polynomial time. We could adapt the heuristics proposed by Han & Wood (2007) or by Delgado & Morais (2004) to get a better order to eliminate states. For our second point, we could take the advantage of the KSP or HKSP algorithms as well as the model-checking results. The states on the more probable evidences should be eliminated first.

This observations yield the following iterative strategy: In each iteration, we take the strongest evidence $\sigma = \tilde{s}\cdot\hat{s}\cdot s_1\cdots s_j\cdot t$ in the remaining automaton – recall that this amounts to an SP problem – and eliminate all the intermediate states on σ (i.e., $s_1, \ldots, s_j$) one by one according to a recently proposed heuristic order Han & Wood (2007). After eliminating a state, possibly a new MUS r_k, say, is created where k MUS s have been created so far, and $\mathrm{val}(r_k)$ can be determined. If $\sum_{i=1}^{k} \mathrm{val}(r_i) > p$, then the algorithm terminates. Otherwise, the transition labelled with r_k is removed from the automaton and either a next state is selected for elimination or a new evidence is to be found, cf. algorithm 5.1.1.

Priority queue q keeps the states to be eliminated in the current iteration. The order in which states are dequeued from q is given by the heuristics in Han & Wood (2007). The function "eliminate(·)" can both eliminate states and regular expressions, where the latter simply means the deletion of the transitions labelled with the regular expression.

Example 5.1.7. Let us apply the algorithm on $\mathcal{A}_D$ of figure 5.2 and $\mathcal{P}_{\leq 0.7}(\mathsf{F}\, s_4)$. In the first iteration, $s_0\cdot s_1\cdot s_2\cdot s_5\cdot s_3\cdot s_4$ is found as the strongest evidence. Assuming the order

Algorithm 5.1.1 Regular expression counterexamples

Require: DFA $\mathcal{A}_{\mathcal{D}} = (S, \Sigma, \tilde{s}, \delta, \{t\})$, and $p \in [0, 1]$
Ensure: regular expression $r \in \mathcal{R}(\Sigma)$ with $\mathrm{val}(r) > p$

```
1:  A := A_D; pr := 0; Priority queue q := ∅; k := 1;
2:  while pr ⩽ p:
3:      σ := the strongest evidence in A;
4:      for each s' ∈ σ ∖ {s̃, ŝ, t}:
5:          q.enqueue(s')
6:      while q ≠ ∅:
7:          A :=eliminate(q.dequeue())
8:          r_k := the created MUS
9:          pr := pr + val(r_k)
10:         A :=eliminate(r_k)
11:         if pr > p:
12:             break
13:         else:
14:             k := k + 1
15: return  r_1 | ... | r_k
```

to eliminate the states by Han & Wood (2007) is s_5, s_2, s_3, we obtain the regular expression $r_5{=}s_1(s_3|s_2s_5s_3)(s_3|s_2s_5s_3)^*s_4$ with $\mathrm{val}(r_5){=}1$. Since all states are eliminated and the threshold 0.7 is exceeded, the algorithm terminates. This expression gives a clear reason that traversing the cycle s_3 or $s_2s_5s_3$ infinitely many times causes the probability exceeding 0.7.

Let us *change the elimination order* to s_5, s_3, s_2. Then the regular expression is $r_0 = s_1s_3s_3^*s_4 \mid s_1(s_2|s_3s_3^*s_2)(s_5s_3s_3^*s_2)^*s_5s_3s_3^*s_4$. After eliminating s_3, the first MUS $r_1 = s_1s_3s_3^*s_4$ is generated and the probability is $0.18 < 0.7$. The algorithm continues (i.e., eliminates s_2) to find more MUS s, till r_0 is found. Note that r_0 is longer than r_5, and thus less intuitive to comprehend. The cycles s_3 and $s_3s_2s_5$ are however indicated.

Let us pick a *less probable evidence* $s_0{\cdot}s_1{\cdot}s_3{\cdot}s_4$ to be eliminated in the first iteration. After eliminating s_3, the resulting expression is $r_1{=}s_1s_3s_3^*s_4$. Then r_1 is removed from the automaton and the strongest evidence in the remaining automaton is $s_0{\cdot}s_1{\cdot}s_2{\cdot}s_5{\cdot}s_4$. After eliminating s_2, s_5, we obtain the regular expression: r_2, as in Example 5.1.5. The final regular expression is again r_0 and the analysis in the last case applies.

Corollary 5.1.8. *The regular expression counterexample generated by Alg. 5.1.1 is minimal.*

This property immediately follows from the fact that Alg. 5.1.1 terminates immediately once the cumulative probability exceeds the threshold. We like to emphasize that the regular expression representation is not applicable for formulae with nested probabilistic operators, e.g., $\mathcal{P}_{\leq p_1}\left(\mathsf{F}\,\mathcal{P}_{\leq p_2}(\mathsf{F}\,a t_t)\right)$. However, this is not a real constraint in practice, since those formulae are rarely used. In addition, it is important to mention

that the algorithm in this section not only applies to non-strict probability bounds, but also to strict bounds as, e.g., $\mathcal{P}_{<p}(\mathsf{F}^{\leq h}\, at_t)$.

5.1.5 Bounded expressions

For bounded reachability formula $\mathsf{F}^{\leq h}\, at_t$, a regular expression, e.g. $r = r_1|r_2^*$, may not be valid because it is possible that the length of the words generated by r_1 or the expansion of r_2 exceeds h. Thus, $\mathrm{val}(r)$ might be larger than the actual probability. In the case of starred expressions this is even clearer. As it turns out, this is quite challenging to work with.

We shall nonetheless provide a valuation function for bounded expressions, which we name *constrained regular expressions*, but this is mainly theoretical.

Definition 5.1.9 (Constrained regular expressions)**.** For $r \in \mathcal{R}(\Sigma)$ and $h \in \mathbb{N}$, $\mathcal{L}(r[h]) = \{w \in \mathcal{L}(r) \mid |w| \leqslant h\}$.

In fact, $\mathcal{L}(r[h]) \subseteq \mathcal{L}(r)$ and $r[h]$ can be expressed equivalently by a union of possible enumerations, namely $r[h] = r\langle 0\rangle|r\langle 1\rangle|\cdots|r\langle h\rangle$, where $r\langle i\rangle$ denotes the set of words generated by r and having exactly i symbols. Constrained regular expressions can be obtained in the same way as presented just before, only their valuation is different:

Definition 5.1.10. For $r \in \mathcal{R}(\Sigma)$ and $h \in \mathbb{N}_{\geqslant 0}$, the function val for $r[h]$ and $r\langle h\rangle$ is defined by:

$$\mathrm{val}(r[h]) = \sum_{i=0}^{h} \mathrm{val}(r\langle h\rangle)$$

$$\mathrm{val}(\varepsilon\langle h\rangle) = \begin{cases} 1 & \text{if } h = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{val}((p,s)\langle h\rangle) = \begin{cases} p & \text{if } h = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{val}(r_1 \mid r_2\langle h\rangle) = \mathrm{val}(r_1\langle h\rangle) + \mathrm{val}(r_2\langle h\rangle)$$

$$\mathrm{val}(r_1{\cdot}r_2\langle h\rangle) = \sum_{i=0}^{h} \mathrm{val}(r_1\langle i\rangle){\cdot}\,\mathrm{val}(r_2\langle h-i\rangle)$$

$$\mathrm{val}(r^*\langle h\rangle) = \mathrm{val}(\varepsilon\langle h\rangle) + \sum_{i=1}^{h} \mathrm{val}(r\langle i\rangle) \times \mathrm{val}(r^*\langle h-i\rangle)$$

Note that the complexity of the above evaluation function is, however, very high. It remains to establish that constrained regular expressions are counterexamples for bounded until-formulae.

Theorem 5.1.11. *Let r be the regular expression for* DFA $\mathcal{A}_{\mathcal{D}} = (S', \Sigma, \tilde{s}, \delta, \{t\})$ *where $\mathcal{D} = (S, \mathbf{P}, L)$ with initial state $\hat{s}$ and $h \in \mathbb{N}$*

$$\mathrm{val}(r[h]) = \Pr\left(Paths_{fin}\min(\hat{s}, \mathsf{F}^{\leq h}\, at_t)\right).$$

An alternative possibility would be to use the graph product construction of section 2.3.2.1. However, this won't yield very good results, because resulting product is acyclic, which means that the resulting regular expression will simply consist of many branches, without being able to capture repeating behaviour.

5.2 Case studies

The idea of capturing the cyclic behaviour using regular expressions only came to us after studying the typical patterns shown in counterexamples that were generated by the original algorithm. Since the program was already there, with its data structures, it was a bit hard to faithfully implement algorithm 5.1.1. However, it was feasible to simply convert the whole automaton to a DFA and convert this in its entirety to a regular expression. This conversion is guided by the heuristic presented by Delgado & Morais (2004); this heuristic was easy to implement and gave very reasonable results.

A first try without any heuristic and fully random behaviour resulted in cases where the resulting expression was impossibly large, so in any case a heuristic is vital to obtain anything useful.

Incremental generation, possibly combined with the visualisation techniques presented by Aljazzar & Leue (2008a), could result in very attractive exploration of counterexamples. However, this is clearly beyond the scope of this thesis.

5.2.1 Leader election example

We conclude this section by reconsidering the leader election protocol. For the original DTMC, the regular expression, denoted $r(n,k)$, is:

$$start.\big((u_1|\cdots|u_i)\,.next.start\big)^*.\,\big(s_1|\cdots|s_j\big)\,.leader,$$

where *start*, *next* and *leader* are the obvious short forms. The regular expression lists all the unsuccessful configurations, as well as the successful ones. As a result, $|r(n,k)| = k^n+4$. Compared to the number of evidences computed directly, $|r(N,K)|$ is much shorter, but it is still exponentially long. On the other hand, however, the structure of $r(N,K)$ clearly indicates the reason of violation, i.e., the repeated unsuccessful configurations followed by a successful one.

And indeed, our implementation yields an expression similar to the one described here – similar because in fact what we indicate as s_1 is a sequence of states, but each transition has probability 1. By first using bisimulation minimization all parallel tracks are lumped together, and the resulting regular expression after computing the quotient DTMC indeed is:

$$\hat{r}(n,k) = start(u{\cdot}next{\cdot}start)^*{\cdot}s{\cdot}leader\_elected$$

The following table provides an overview how the two techniques together yield very compact regular expressions, in terms of the number of symbols in the expression.

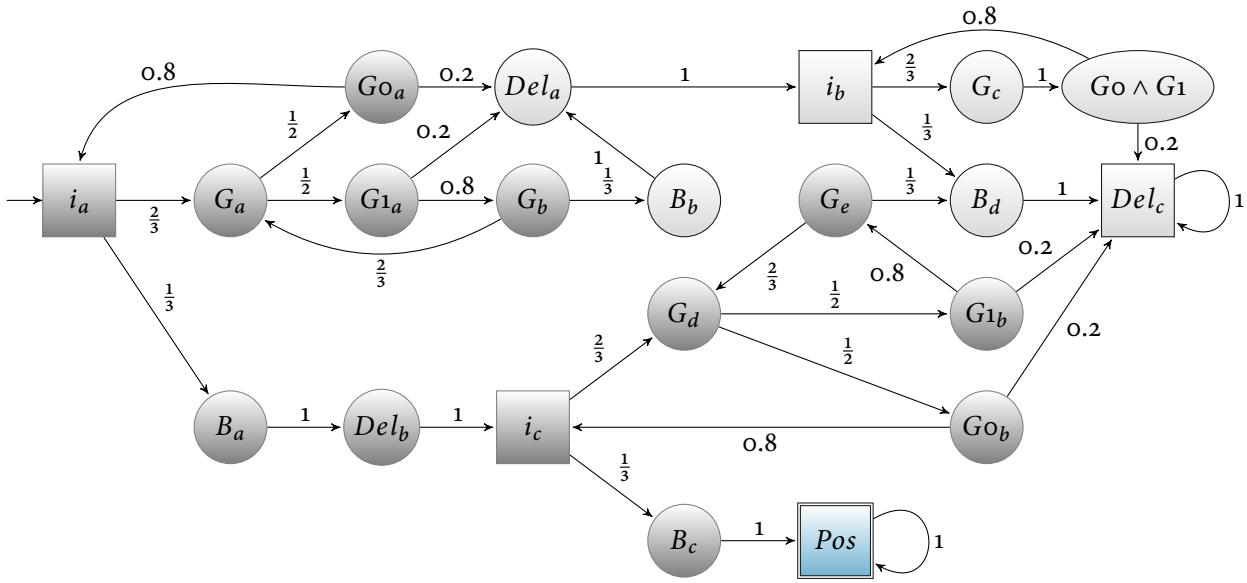


Figure 5.3 The state space of the quotient DTMC with two good crowd members $N = 2$, two path reformulations $R = 2$ and one bad crowdmember, and a forwarding probability of 0.8.

	Original	Minimised
$k = 2$	73	11
$k = 4$	1 061	11
$k = 8$	20 481	11
$k = 12$	87 151	11

As we can see, the branching in the original model is not handled very well by the regular expression, it is still quite large.

5.2.2 Crowds protocol

Although bisimulation minimisation improves the convergence, as we have seen in the previous example. As can be seen in the next table the combination of bisimulation minimisation and regular expressions yields very short expressions, which are highly informative. Because it is a bit hard too appreciate how the model is reduced, and what the structure of the regular expression is, we show an example in figure 5.3.

To fit the figure on the page, we group a path of states with probability 1 by a square state. States *i*, *G*, *B*, *Del*, *Pos* represent *i*nitiating a new session, sending a message to a *G*ood member, to a *B*ad member, a message being *Del*ivered, a *Pos*itive result obtained, respectively. *G*o and *G*1 are the two good members, where *G*o is assumed always to be the original sender when a new session starts. $Go \vee G1$ is a lumped state where either *G*o or *G*1 is reached. The subscripts $a, b, \ldots$ are to distinguish the states in similar situations. Since the goal state *Pos* can be reached by only the grey states, the regular

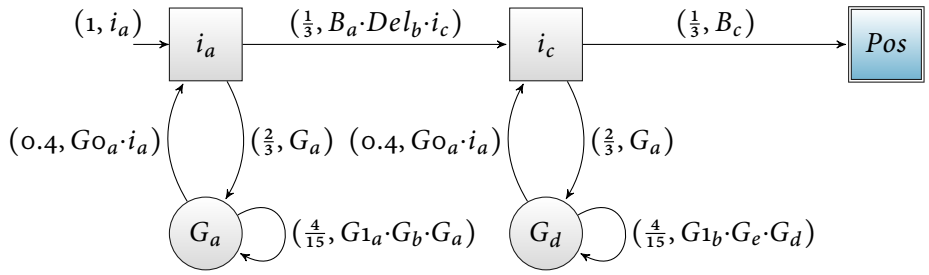


Figure 5.4 The compacitified state space of the quotient DTMC with two good crowd members $N = 2$, two path reformulations $R = 2$ and one bad crowdmember, and a forwarding probability of 0.8.

expression (thus the automaton) only depends on those states. Note that Del_a and Del_b denote the end of the first session, while Del_c and Del_b denote the end of the second. Only the case that two messages are both delivered by the bad member indicates a positive identification of the sender.

An intermediate automaton (see figure 5.4) can be derived after eliminating some states. This shows the basic structure of the model: i_a and i_c are the starting points of two sessions. The horizontal transitions indicate the observation of Go by the bad member, which lead to *Pos*. In each session, a message can be forwarded to Go or $G1$ many times (captured by the self loops). Once a message is delivered, a new session is assumed to be started (the transitions back to i_a and i_c). Thus, a regular expression that can be generated from the automaton is $r = r_0 r_1^* r_2 r_3^* r_4$, where:

$$
\begin{aligned}
r_0 &= (1, i_a),\\
r_1 &= (\tfrac{2}{3}, G_a)(0.267, G1_a.G_b.G_a)^*(0.4, Go_a.i_a),\\
r_2 &= (\tfrac{1}{3}0.333, B_a.Del_b.i_c),\\
r_3 &= (\tfrac{2}{3}0.667, G_d)(0.267, G1_b.G_e.G_d)^*(0.4, Go_b.i_c),\\
r_4 &= (\tfrac{1}{3}, B_c.Del_d.Pos).
\end{aligned}
$$

If we omit the probabilities and the subscripts and merge the stuttering steps G, then we obtain:

$$
r' = i\,\underbrace{(G.(G1.G)^*Go.i)^*}_{good}.\underbrace{(B.Del.i)}_{bad}.\underbrace{(G.(G1.G)^*Go.i)^*}_{good}.\underbrace{B}_{bad},
$$

which is highly compact and informative in the sense that it indicates the observation of the bad members twice with arbitrary number of observing the good members.

The probability of r is $\mathrm{val}(r) = 0.274$, which coincides with the model checking result. These probabilities depend, among others, on the parameters of the protocol (N, C, R, p_f, etc.). For instance, the probability of the strongest evidence is $(\frac{C}{N+C})^R = (\frac{1}{3})^2 = \frac{1}{9}$, which loops 0 times at r_1 and r_3. The probability of r_2 and r_4 is $\frac{a}{1-a} = \frac{4}{11}$, where a is the probability of the inner loop: $\frac{1}{N+C}{\cdot}PF{\cdot}(1 - \frac{C}{N+C}) = 0.267$, as is shown in the intermediate automaton. Note that this closed-form expression can now be used for arbitrary parameter values.

The table below shows some numbers for the actual reductions in regular expression length that are obtained after lumping, and the expressiveness of these expressions.

	Original	Minimised
$R = 3$	56 888	98
$R = 4$	40 1883	200
$R = 5$	6 384 840	346

5.2.3 Randomised mutual exclusion

As we saw in the previous chapter, bisimulation minimisation did not result in very short expressions. However, as it turns out, the regular expression is already very short for this automaton, the computation of this expression however is very memory intensive, because the automaton is very large.

We refrain from a detailed analysis.

	Original	Minimised
$n = 2$	11	11
$n = 3$	37	37
$n = 4$	155	155

5.3 Concluding remarks

These examples show that the combination of bisimulation minimisation and regular expression representations looks very promising. Very compact expressions can be obtained, which contain a lot of information. The computation of the regular expressions is usually quite fast when compared to the computation of the counterexamples, even if one converts the whole automaton instead of the evidence guided approach which is theoretically much more promising.

Conclusion and future work

6

6.1 Conclusion

We have seen that the approach of presenting a PCTL counterexample on a DTMC, simply by a long list of paths leads to very large lists in practice. Mathematical evaluation of the particular case of leader election shows that this in fact is doubly exponential.

In order to obtain a more manageable representation we propose the use of regular expressions along side bisimulation minimisation to represent a (possibly infinite) set of counterexamples. Using these we can provide a much more compact representation which is more useful for debugging. This approach looks very promising.

The reduction of the constrained problem to the unconstrained problem gives a general way to develop specialised algorithms for the hop-constrained case if needed, or to transform the input so one can use existing algorithms.

Besides the theoretical results, describing the necessary transformations to obtain a regular expression from an input DTMC and formula, we have also made a prototypical implementation, which combines the algorithms presented by Han & Katoen (2007) and the new applications of the algorithms described in this thesis. This implementation could be used as a starting point to implement the discussed algorithms in, for example MRMC.

6.2 Future work

Although one can never by exhaustive, I have tried to make a substantial effort towards a full exploration of these subjects. The following points are worth exploring:

- The whole implementation was meant as a prototype, and even though it is capable of handling models with more than hundred thousand states, it is not as efficient as could be. Since the approach with regular expressions seems promising, a tight integration, perhaps combined with an existing tool, such as MRMC would be promising.
- The recently developed K* algorithmAljazzar & Leue (2008b) would be a good algorithm to use as another k shortest path algorithm. Especially if combined with state elimination techniques, it would be possible to combine

the approach of Husain Aljazzar with our approach to find counterexamples. Discussions in Aachen about this combination seemed very promising.

- I would like a theoretically more fundamental reconciliation and treatment of the theory of finite automata and Markov Chains. The automata-theoretic approach of persons like Arto Salomaa, which rigorously encapsulates finite automata in the existing mathematical theory of groups and algebra would, in my opinion, be a powerful instrument to analyse these problems. After all, the state elimination procedure is a bit of an ad-hoc approach.
- The hop-constraint problem should be investigated more thoroughly, in our experience it was sometimes quite hard to give a solid interpretation to the meaning of a hop-constrained. In the case of the leader example a hop-constrained can be understood as the number of election rounds, but sometimes it seems that a hop-constraint cannot quite capture a limit on the number of rounds, because, for example, each round consists of a different number of steps.

CTL model checking

A

This chapter gives a short overview of CTL model checking. It is not intended to be complete, for a more thorough overview we refer to the work of Baier & Katoen (2008).

A.1 Outline

Given a formula Φ to check, and a structure to check it on, for example, a Kripke structure with a validation function, or as we use in this thesis, a DTMC with a labelling function, the algorithm works 'bottom-up'. In case of a simple state-formula $a \wedge (b \vee \neg c)$, it will first find all states on which a is valid; all states on which b is valid and all states on which c is valid. $\neg c$ is then found by taking the complement. $b \vee \neg c$ is subsequently found by taking the union of the result of b and $\neg c$, and finally this is intersected with the results for a.

The procedure to compute $Sat(\Phi)$ – that is, those states for which Φ is satisfied, thus functions according to the following steps:

- Construct the parse tree of Φ.
- First compute $Sat(p)$, where p is some atomic proposition, for the leaves of the tree (the leaves will of course only contain atomic propositions).
- Repeatedly go up one level and use the formula on the node to compute the result for that node, while reusing the result of the leaves, until you reach the root, which is Φ.

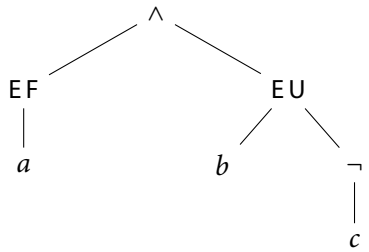


Figure A.1 The parse-tree of $\Phi = \mathsf{EF}\, a \wedge \mathsf{E}(b\ \mathsf{U}\ \neg c)$. A computation of $Sat(\Phi)$ will recursively descend the tree, computing the first result at leaves and then work bottom up.

An example of a formula and its parse tree is given in figure A.1. A computation of $Sat(\Phi)$ will need to compute $Sat(\mathsf{EF}\, a)$ and $Sat(b \,\mathsf{U}\, \neg c)$ first, which in turn will generate further recursive calls until the leaves are reached, after which the procedure returns and computes the results.

Since in practice, the results involving a single U operator, without nesting, are complicated enough, we do present the outline for computing $\mathsf{E}\,\mathsf{U}$ since its ideas are implicitly contained in the way the DTMC is transformed in section 2.2.1 on page 30.

We first give a characterization of $Sat(\Phi)$ which suffices to compute every possible propositional connective, like $\wedge$ or $\Rightarrow$, because we can rewrite these forms to forms without these connectives. It is needless to say that in practice something can be gained by stating directly that $Sat(\mathtt{ff}) = \varnothing$ instead of through computation of $Sat(\mathtt{tt})$ and negation.

Given a $\mathcal{D} = (S, \mathbf{P}, L)$, we characterize $Sat(\cdot)$ as follows:

$$\begin{aligned} Sat(\mathtt{tt}) &= S \\ Sat(a) &= \{s \in S \mid a \in L(s)\}, \text{ for any } a \in AP \\ Sat(\Phi \vee \Psi) &= Sat(\Phi) \cup Sat(\Psi) \\ Sat(\neg\Phi) &= S - Sat(\Phi) \end{aligned}$$

Secondly, $Sat(\mathsf{E}(\Phi \,\mathsf{U}\, \Psi)$ is characterized by the smallest subset T of S, such that:

1. $Sat(\Psi) \subseteq T$ and:
2. If $s \in Sat(\Phi)$, and s has a successor in T, then also $s \in T$.

Note that this definition is forward, although in practice a backward algorithm is easier. We can start with a set S of states that satisfy Ψ (these will surely satisfy $\mathsf{E}(\Phi \,\mathsf{U}\, \Psi)$) and repeatedly enlarge this set by looking at the states outside of S that are predecessors of the states in S and check whether they satisfy Φ, and if so, add them to our set, there are no predecessors left.

In our algorithm, by making states that satisfy neither Φ nor Ψ we make sure, that if we start at a state that satisfies Ψ, and we look at a predecessor, it will always satisfy Φ, and thus be a valid. Hence, we can be sure we will only find paths on which $\Phi \,\mathsf{U}\, \Psi$ is valid.

A.2 Time complexity

We state, without proof, that the time-complexity for deciding whether $T \vDash \Phi$ holds, where T is a transition system with n states and m transitions, and a Φ is a CTL formula, is $\mathcal{O}(|\Phi|(n+m))$.

Counterexample explorer manual

B

Note: *The full documentation can be found on the accompanying CD. This appendix is an (automatically generated) extract from this complete documentation. It sole purpose is to serve as a getting started guide, it describes the command-line options of the program, the requirements and the input file formats.*

Counterexample explorer is the main program, it can be used to find a strongest evidence or smallest counterexample given a DTMC and a formula.

B.1 Usage and requirements

The program is invoked as **ceexpl**, and requires at least Python 2.5 to run. Implementations other than CPython have not been tested, and may or may not work. At the moment of writing, Jython 2.5 is still in its alpha phase, and IronPython 2.0 has not been tested. There are no principle objections however why these should not work, because no external libraries are needed, although some extension modules can be used if they are available, see *Optional modules*.

The standard usage is:

```
$ ceexpl.py basename
```

where `basename` is the *base name* of the of the files containing the model, labelling and formula. The name of the model should be `<basename>.tra`, the name of the labelling should be `<basename>.lab` and finally the name of the file containing the formula should be `<basename>.pctl`. The section *File formats* describes the format of these files.

In addition the following options are supported:

`-h, --help`
: show a help message summarizing the options and exits

`-f FORMULA, --formula=FORMULA`
: use the specified formula, overriding the one in the .pctl file

`-H UHOPS, --maxhoplimit=UHOPS`
: specify a maximum hop limit, overriding the one in the PCTL-formula, see also *`-i`*

-L LHOPS, --minhoplimit=LHOPS
specify a minimum hop limit, overriding the one in the PCTL-formula.

Note: if you do not specify an upperbound with *-H*, or if you override one with *-i*, you also have to use *-p*, for this to be taken into account

-i, --nohoplimit
specify an infinite hop-limit, overriding the one in the PCTL-formula

-P PROBABILITY, --probability=PROBABILITY
specify a probability-bound, overriding the one in the PCTL- formula, the bound has to be in the range [0..1]

-t, --traces
output all traces in a smallest counterexample (default). Traces are written to stdout.

-T, --notraces
don't output all traces in a smallest counterexample

-d, --dot
writes .dot files of the DTMCs, these can be converted to pictures using GraphViz

-s, --se-only
only output the strongest evidence

--labfile=LABFILE
read the labelling from LABFILE, not from `<basename>.lab`

--pctlfile=PCTLFILE
read the PCTL-formula from PCTLFILE, not from `<basename>.pctl`

--outputdata
output a Gnuplot-compatible datafile with the number of paths versus the cumulative probability mass

-p, --product-graph
solve the hop-constraint k sp problem by building a product graph

-B BOUND, --trace-bound=BOUND
halt after having found `BOUND` traces

-v, --verbosity
run verbose (repeat for a higher level)

B.1.1 Example session

Assume the labelling and transitions inputs specify the *Example DTMC*, and suppose the formula to be checked is $\mathcal{P}_{\leq 0.8}(a \,\mathsf{U}^{\leq 3}\, b)$, the sample session might look like:

```
$ ./ceexpl.py example
* Formula succesfully parsed: P[≤ 0.8](a U[≤ 3] b).
* Now parsing the DTMC... Parse succesful!
    DTMC contains 10 states and 24 transitions.
    Starting states: {s}.
* Transforming the DTMC to a Directed Graph.... Done!
* Calculating smallest counter example.
```

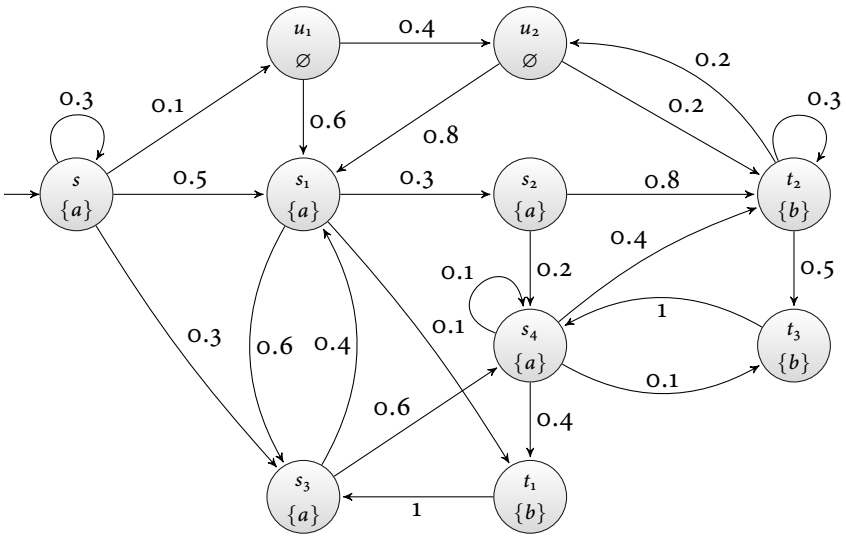


Figure B.1 Example DTMC

```
* No smallest counter example could be found.
   Total probability mass accumulated: 0.349.
   Total number of traces: 7.
   Time required: 0.0008 s.

List of traces:

Trace 1 (P:  0.12):    s-s_1-s_2-t_2
Trace 2 (P: 0.072):    s-s_3-s_4-t_1
Trace 3 (P: 0.072):    s-s_3-s_4-t_2
Trace 4 (P:  0.05):    s-s_1-t_1
Trace 5 (P: 0.018):    s-s_3-s_4-t_3
Trace 6 (P: 0.012):    s-s_3-s_1-t_1
Trace 7 (P: 0.005):    s-s-s_1-t_1
```

We see the formula is parsed, the DTMC is then parsed, it is transformed, and finally it tries to find a counterexample. Since we put the threshold on 0.8, and provided a hop-limit of 3, it cannot find such a counterexample. If we override the constraint, we get indeed get a counterexample:

```
$ ./ceexpl.py -i example
* Formula succesfully parsed: P[≤ 0.8](a U[≤ 3] b).
* Now parsing the DTMC... Parse succesful!
    DTMC contains 10 states and 24 transitions.
    Starting states: {s}.
* Transforming the DTMC to a Directed Graph.... Done!
* Nota Bene: Upper hop-limit override in effect!
* Calculating smallest counter example.
* A smallest counter example has been found!
```

```
   Total probability mass accumulated: 0.803.
   Total number of traces: 43.
   Time required: 0.0030 s.

List of traces:

Trace  1 (P:  0.12):    s-s_1-s_2-t_2
Trace  2 (P: 0.072):    s-s_1-s_3-s_4-t_1
....
Trace 42 (P: 0.00288):    s-s-s_3-s_1-s_2-t_2
Trace 43 (P: 0.00288):    s-s-s_1-s_3-s_1-s_2-t_2
```

We see that the second trace, with a probability 0.072 did not appear in the previous output, because the hop count was too large.

We removed some of the output-traces for clarity. In the following we override the probability bound with `-P`, lowering it to .25, we also provide a lower and upperbound:

```
$ ./ceexpl.py -P 0.25 -L 4 -H 4 example
* Formula succesfully parsed: P[≤ 0.8](a U[≤ 3] b).
* Now parsing the DTMC... Parse succesful!
    DTMC contains 10 states and 24 transitions.
    Starting states: {s}.
* Transforming the DTMC to a Directed Graph.... Done!
* Nota Bene: Lower hop-limit override in effect!
* Nota Bene: Upper hop-limit override in effect!
* Nota Bene: Probability override in effect!
* Calculating smallest counter example.
* A smallest counter example has been found!
   Total probability mass accumulated: 0.253.
   Total number of traces: 10.
   Time required: 0.0010 s.

List of traces:

Trace  1 (P: 0.072):    s-s_1-s_3-s_4-t_2
Trace  2 (P: 0.072):    s-s_1-s_3-s_4-t_1
Trace  3 (P: 0.0288):    s-s_3-s_1-s_2-t_2
Trace  4 (P: 0.018):    s-s_1-s_3-s_4-t_3
Trace  5 (P: 0.012):    s-s_1-s_3-s_1-t_1
Trace  6 (P: 0.012):    s-s_1-s_2-s_4-t_1
Trace  7 (P: 0.012):    s-s-s_1-s_2-t_2
Trace  8 (P: 0.012):    s-s_1-s_2-s_4-t_2
Trace  9 (P: 0.0072):    s-s-s_3-s_4-t_1
Trace 10 (P: 0.0072):    s-s-s_3-s_4-t_2
```

If we do not specify the upperbound, we need to provide `-p` (note that we explicitly override the upper hop-limit with `-i`, because the formula provides one):

```
$ ./ceexpl.py -P 0.25 -L 4 -i -p example
* Formula succesfully parsed: P[≤ 0.8](a U[≤ 3] b).
* Now parsing the DTMC... Parse succesful!
    DTMC contains 10 states and 24 transitions.
    Starting states: {s}.
* Transforming the DTMC to a Directed Graph.... Done!
* Nota Bene: Lower hop-limit override in effect!
* Nota Bene: Upper hop-limit override in effect!
* Removing hop-constraint, building product-graph instead.
* Time needed to compute the product graph: 0.218 ms.
* Product contains 62 nodes and 88 states.
    Starting state: s.
* Nota Bene: Probability override in effect!
* Calculating smallest counter example.
* A smallest counter example has been found!
   Total probability mass accumulated: 0.254.
   Total number of traces: 7.
   Time required: 0.0028 s.

List of traces:

Trace 1 (P: 0.072):    ('s', 0)-('s_1', 1)-('s_3', 2)-('s_4', 3)-t_1
Trace 2 (P: 0.072):    ('s', 0)-('s_1', 1)-('s_3', 2)-('s_4', 3)-t_2
Trace 3 (P: 0.0288):    ('s', 0)-('s_1', 1)-('s_3', 2)-('s_1', 3)-s_2-t_2
Trace 4 (P: 0.0288):    ('s', 0)-('s_3', 1)-('s_1', 2)-('s_2', 3)-t_2
Trace 5 (P: 0.018):    ('s', 0)-('s_1', 1)-('s_3', 2)-('s_4', 3)-t_3
Trace 6 (P: 0.0173):    ('s', 0)-('s_1', 1)-('s_3', 2)-('s_1', 3)-s_3-s_4-t_1
Trace 7 (P: 0.0173):    ('s', 0)-('s_3', 1)-('s_1', 2)-('s_3', 3)-s_4-t_1
```

We can see in the output that the algorihm is run on a transformed graph, because the statenames have been changed.

B.2 Optional modules

Counterexample explorer can make use of the PQueue extension modue, which provides an implementation of Fibonacci heaps in C for Python. If this modules is not installed, it relies on other heap structures for Dijkstra's algorithm, such as a pairing heap or binary heap. The program comes with a functioning pure Python pairing heap implementation. Before installing it, there is an important issue:

> **Warning:** PQueue does not function under Python 2.5 without patching it. The Python FAQ gives a description of the problem.

To fix it, in `pqueue_dealloc` the call `PyMem_DEL(pqp)` should be replaced by `PyObject_DEL(pqp)` in `pqueuemodule.c`. Otherwise using the module will most likely result in a segfault.

B.3 File formats

The tool uses a file format very similar to that of MRMC. The input should be split across three files, a file containing the states and transitions, a file containing the labbelling and finally a file with the PCTL-formula for which a counterexample is required.

B.3.1 Transition file

The transition file should end with the extension `.tra`. The file format is, as said, similar to MRMC, but with a few differences. The file is organised as follows:

- First, a line starting with `STATES` followed by whitespace and a positive integer `N` indicating the number of states.
- Optionally a line with exactly `N` whitespace separated words (no quoting possible) of names for the states. This can be used to provide names such as *init*, *final*, *deadlock*, et cetera. If this line is not available the states are represented by the integers [1..N]
- Then a line starting with `INITIAL` followed by whitespace and the name (if the states have names) or the number of the initial state.
- Then a line starting with `TRANSITIONS` followed by whitespace and positive integer `T` indicating the total number of transitions in the model.
- Finally `T` lines each specifying a transition. Each line should have the form `from to rate`. If the states have been given names, one can also use the names here, otherwise one has to use the numerical scheme.

Note: One can still use numbers even though the states are given names, internally they are numbered in the order of the names, starting at 1. This might be useful if you just want to annotate an MRMC model with state names.

Every state should have at least one successor and the total rate of per state should equal 1. Furthermore each transition should have a rate in the range (0..1] Exponentional notation is supported, for example a transition rate of '4.9e-06' is valid.

Note: MRMC does not support the optional state names, nor the INITIAL line. Using the input for MRMC then requires to remove this lines, this can easily be performed by doing something like:

```
$ sed '/^INITIAL/d' <file.tra> > <file-mrmc.tra>
```

Note: MRMC is much more restrictive when it comes to the transitions. First of all, names are not supported, so all states have to be numbered, and most importantly should be ordered by 'from' and then by 'to'. So first all transitions from state 1, then from state 2, et cetera. Within such a block the to-states should be order too, so 1 2 .5 and then 1 3 .5, not the other way around.

B.3.1.1 Example file

The transition file corresponding to the automaton of the *example* could be specified thusly:

```
STATES 10
s u_1 u_2 s_1 s_2 t_2 s_4 t_3 s_3 t_1
INITIAL s
TRANSITIONS 24
s s 0.1
s u_1 0.1
s s_1 0.5
s s_3 0.3
u_1 u_2 0.4
u_1 s_1 0.6
u_2 s_1 0.8
u_2 t_2 0.2
s_1 s_2 0.3
s_1 s_3 0.6
s_1 t_1 0.1
s_2 t_2 0.8
s_2 s_4 0.2
t_2 u_2 0.2
t_2 t_2 0.3
t_2 t_3 0.5
s_4 t_2 0.4
s_4 s_4 0.1
s_4 t_3 0.1
s_4 t_1 0.4
t_3 s_4 1
s_3 s_1 0.4
s_3 s_4 0.6
t_1 s_3 1
```

Alternatively, the input could specified without specifying the names in the transition list, as said, this is mostly useful when adapting MRCM models:

```
STATES 10
s u_1 u_2 s_1 s_2 t_2 s_4 t_3 s_3 t_1
INITIAL s
TRANSITIONS 24
1 1 0.1
1 2 0.1
1 4 0.5
1 9 0.3
2 3 0.4
2 4 0.6
3 4 0.8
3 6 0.2
4 5 0.3
```

```
4 9 0.6
4 10 0.1
5 6 0.8
5 7 0.2
6 3 0.2
6 6 0.3
6 8 0.5
7 6 0.4
7 7 0.1
7 8 0.1
7 10 0.4
8 7 1
9 4 0.4
9 7 0.6
10 9 1
```

And a third option would be to also remove the line of state names: this will of course cause the traces of the counterexample to be less intuitive too.

B.3.2 Labelling file

The labelling file follows exactly the same format as MRMC. The labelling file must end in `.lab`. This format is as follows:

- The first line should be `#DECLARATION`.
- The next line should contain a whitespace separated list of *atomic propositions*, these can be simply 'a' and 'b' or more complex like 'p1+p0<=p2+p3', as long as they do not contain spaces.
- Subsequently a line starting with `#END`, signalling the end of the label declarations.
- For every state a line starting with the state name or number, followed by whitespace and a whitespace separated list of atomic propositions valid in the state. This list can be empty. One has to specify each state exactly one time however, including the empty ones.

  Note: MRMC wants the states in numerical order, counterexample explorer need not so.

B.3.2.1 Example file

We continue our example, and show to possible ways to specify the labelling, first using the state names:

```
STATES 10
s u_1 u_2 s_1 s_2 t_2 s_4 t_3 s_3 t_1
INITIAL s
TRANSITIONS 24
s s 0.1
```

```
s u_1 0.1
s s_1 0.5
s s_3 0.3
u_1 u_2 0.4
u_1 s_1 0.6
u_2 s_1 0.8
u_2 t_2 0.2
s_1 s_2 0.3
s_1 s_3 0.6
s_1 t_1 0.1
s_2 t_2 0.8
s_2 s_4 0.2
t_2 u_2 0.2
t_2 t_2 0.3
t_2 t_3 0.5
s_4 t_2 0.4
s_4 s_4 0.1
s_4 t_3 0.1
s_4 t_1 0.4
t_3 s_4 1
s_3 s_1 0.4
s_3 s_4 0.6
t_1 s_3 1
```

Or secondly, only using the state numbers. The numbers of course correspond to the numbers in the transition-file:

```
STATES 10
s u_1 u_2 s_1 s_2 t_2 s_4 t_3 s_3 t_1
INITIAL s
TRANSITIONS 24
1 1 0.1
1 2 0.1
1 4 0.5
1 9 0.3
2 3 0.4
2 4 0.6
3 4 0.8
3 6 0.2
4 5 0.3
4 9 0.6
4 10 0.1
5 6 0.8
5 7 0.2
6 3 0.2
6 6 0.3
6 8 0.5
7 6 0.4
7 7 0.1
```

```
7 8 0.1
7 10 0.4
8 7 1
9 4 0.4
9 7 0.6
10 9 1
```

B.3.3 Formula file

The formula should contain exactly one PCTL formula, in the form of:

```
P[<operator>bound]( Path-Formula )
```

The parser is intended to be a quite flexible with the input it accepts. The best way to show the syntax is to provide a few examples, along with a LaTeX-version, showing how the formula is parsed:

- `P[<=.8](a U[3] b)`
  - $\mathcal{P}_{\leq 0.8}(a \,\mathsf{U}^{\leq 3}\, b)$
- `P[<=.8](a U b)`
  - $\mathcal{P}_{\leq 0.8}(a \,\mathsf{U}\, b)$
- `P[<=0.5](F[10,infinite] a => b => c & d | e)`
  - $\mathcal{P}_{\leq 0.5}(\mathsf{F}^{[10,\infty]}\,(a \Rightarrow (b \Rightarrow ((c \wedge d) \vee e))))$
- `P[>.3](a U[inf] P[<.5] (b W c /\\ d))`
  - $\mathcal{P}_{>0.3}(a \,\mathsf{U}\, \mathcal{P}_{<0.5}(b \,\mathsf{W}\, (c \wedge d)))$
- `P[>.1](tt U b)`
  - $\mathcal{P}_{>0.1}(\mathtt{tt} \,\mathsf{U}\, b)$

From these examples we can infer that the formula parser accepts different syntaxes, for example one can use '&' or '&&' or '/\' or even 'And' or '∧' to signify the boolean conjunctive. (To use the latter the file *must* be in UTF-8 encoding.) Furthermore, it knows the associativity and precedence rules for the operators, and it will fill in an infinite upper bound on a path-formula if none is provided. The last example shows that 'tt' (and 'true' and 'True') are interpreted as presenting a special value *true* which holds in every state.

> **Warning:** Counterexample does not support finding counterexamples for every feature the parser accepts, notably, nested probability operators, such as in the fourth example, are *not* supported. Also, probability bounds other than ≤ are not supported.

Finally, the parser does any necessary rewriting to obtain a formula in normal form, for example, an expression like $\mathsf{F}\, a$ will be rewritten to $(\mathtt{tt} \,\mathsf{U}\, a)$.

Note: Specifying incorrect input might lead to some unexpected results, because of the way the parsing engine works. It tries to parse as much as it can, and ignores the remainder. So, if you would accidentally specify `P(a U b)`, it would only parse `P`. The reason for this is that it expects a state formula, which can be only `P`, in which case `P` is interpreted as an atomic proposition. In order to recognize it as a probability operator one has to supply a probability-bound. One should therefore look closely whether the complete formula is parsed.

This also means that one can provide comments about the formula, in the same file.

Acronyms

C

BF	Bellman-Ford
BFS	Breadth First Search
CTL	Computational Tree Logic
DAG	Directed Acyclic Graph
DFA	Deterministic Finite State Automaton
DFS	Depth First Search
DTMC	Discrete Time Markov Chain
FSA	Finite State Automaton
HSP	Hop constrained Shortest Path
HKSP	Hop constrained k Shortest Path
KSP	k Shortest Paths
MRMC	Markov Reward Model Checker
NFA	Non-Deterministic Finite State Automaton
PCTL	Probabilistic Computational Tree Logic
PRISM	ProbabilistIc Symbolic Model checker
REA	Recursive Enumeration Algorithm
SC	Smallest Counterexample
SE	Strongest Evidence
SP	Shortest Path

Index

Bibliography

Ahuja, Ravindra K.; Magnanti, Thomas L. and Orlin, James B. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall. ISBN 0-136-17549-x.

Aljazzar, Husain and Leue, Stefan. 2008a. Debugging of dependability models using interactive visualization of counterexamples. *Pages 189–198 of: Qest '08: Proceedings of the 2008 fifth international conference on quantitative evaluation of systems*. Washington, DC, USA: IEEE Computer Society. ISBN 978-0-7695-3360-5. DOI:10.1109/QEST.2008.40.

Aljazzar, Hussain and Leue, Stephan. 2008b (July). *A directed on-the-fly algorithm for finding the k shortest paths*. Tech. rept. University of Konstanz.

Andersson, Christer; Fischer-Hübner, Simone and Lundin, Reine. 2004. Enabling anonymity for the mobile Internet using the mCrowds system. *Page 35 of: Ifip wg 9.2, 9.6/11.7 summer school on risks and challenges of the network society*.

Baier, Christel and Katoen, Joost-Pieter. 2008. *Principles of model checking*. Cambridge, MA: MIT press. ISBN 978-0-262-02649-9.

Bellman, Richard. 1958. On a routing problem. *Quarterly of applied mathematics*, **16**, 87–90.

Berry, Gérard and Sethi, Ravi. 1986. From regular expressions to deterministic automata. *Theor. comput. sci.*, **48**(3), 117–126.

Brzozowski, Janusz A. 1964. Derivatives of regular expressions. *J. ACM*, **11**(4), 481–494. DOI:10.1145/321239.321249.

Capinski, Marek and Kopp, Peter E. 2004. *Measure, Integral and Probability*. Springer-Verlag. ISBN 978-1-85233-781-0.

Clarke, Edmund M. and Emerson, E. Allen. 1981. Design and synthesis of synchronization skeletons using Branching-Time Temporal Logic. *Pages 52–71 of: Logic of programs, workshop*. Berlin: Springer-Verlag. DOI:10.1007/BFb0025774.

Comtet, Louis. 1974. *Advanced combinatorics: The art of finite and infinite expansion*. Boston: D.Reidel Publishing Company.

Damman, Berteun; Han, Tingting and Katoen, Joost-Pieter. 2008. Regular expressions for pctl counterexamples. *Pages 179–188 of: qest*. doi:10.1109/qest.2008.11.

Daws, Conrado. 2004. Symbolic and parametric model checking of discrete-time Markov chains. *Pages 280–294 of: Ictac, lncs 3407*. doi:10.1007/b107116.

Delgado, Manuel and Morais, José. 2004. Approximation to the smallest regular expression for a given regular language. *Pages 312–314 of: Ciaa, lncs 3317*. doi:10.1007/b105090.

Dijkstra, Edsger W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**, 269–271.

Du, Ding-Shu and Ko, Ker-I. 2001. *Problem solving in automata, languages, and complexity*. John Wiley and Sons, New York, NY.

Emerson, E. Allen. 1990. Temporal and modal logic. *Pages 997–1072 of:* Van Leeuwen, Jan (ed), *Handbook of theoretical computer science, volume B: Formal models and semantics*. Elsevier and mit Press. isbn 0-262-22039-3.

Emerson, E. Allen; Mok, Aloysius K.; Sistla, A. Prasad and Srinivasan, Jai. 1991. Quantitative temporal reasoning. *Pages 136–145 of: cav '90: Proceedings of the 2nd international workshop on computer aided verification*. London, UK: Springer-Verlag. isbn 3-540-54477-1. doi:10.1007/bf00355298.

Eppstein, David. 1998. Finding the k shortest paths. *siam J. computing*, **28**(2), 652–673. doi:10.1137/s0097539795290477.

Fredman, Michael L. and Tarjan, Robert E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. acm*, **34**(3), 596–615. doi:10.1145/28869.28874.

Fredman, Michael L.; Sedgewick, Robert; Sleator, Daniel D. and Tarjan, Robert E. 1986. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, **1**(1), 111–129. doi:10.1007/bf01840439.

Garey, Michael R. and Johnson, David S. 1979. *Computers and intractability: A guide to the theory of np-completeness*. W. H. Freeman. isbn 0-716-71045-5.

Goldberg, Andrew V. and Tarjan, Robert E. 1996. *Expected performance of Dijkstra's shortest path algorithm*. Tech. rept. nec Research Institute Report.

Gramlich, Gregor and Schnitger, Georg. 2007. Minimizing NFA's and regular expressions. *J. comput. syst. sci.*, **73**(6), 908–923.

Han, Tingting and Katoen, Joost-Pieter. 2007. Counterexamples in probabilistic model checking. *Pages 72–86 of: Tools and algorithms for the construction and analysis of systems*. Lecture Notes in Computer Science, vol. 4424. Springer Verlag. doi:10.1007/978-3-540-71209-1_8.

Han, Yo-Sub and Wood, Derick. 2007. Obtaining shorter regular expressions from finite-state automata. *Theor. comput. sci.*, **370**(1-3), 110–120.

Hansson, Hans and Jonsson, Bengt. 1994. A logic for reasoning about time and reliability. *Formal aspects of computing*, **6**(5), 512–535. doi:10.1007/bf01211866.

Jiménez, Víctor M. and Marzal, Andrés. 1999. Computing the *k* shortest paths: A new algorithm and an experimental comparison. *Pages 15–29 of: Wae '99: Proceedings of the 3rd international workshop on algorithm engineering*. London, UK: Springer-Verlag. doi:10.1007/3-540-48318-7_4.

Katoen, Joost-Pieter; Khattri, M. and Zapreev, Ivan S. 2005. A Markov reward model checker. *Pages 243–244 of: qest*. doi:10.1109/qest.2005.2.

Katoen, Joost-Pieter; Kemna, Tim; Zapreev, Ivan S. and Jansen, David N. 2007. Bisimulation minimisation mostly speeds up probabilistic model checking. *Pages 87–101 of: tacas, lncs 4424*. doi:10.1007/978-3-540-71209-1_9.

Kleene, Stephen C. 1956. Representation of events in nerve nets and finite automata. *Pages 3–40 of:* Shannon, Claude E. and McCarthy, John (eds), *Automata studies*. Annals of mathematics studies, vol. 34. Princeton, nj: Princeton University Press.

Kwiatkowska, Marta Z.; Norman, Gethin; Segala, Roberto and Sproston, Jeremy. 2002. Automatic verification of real-time systems with discrete probability distributions. *Theoretical compututer science*, **282**(1), 101–150. doi:10.1016/s0304-3975(01)00046-9.

Linz, Peter. 2001. *An introduction to formal languages and automata*. Jones and Bartless Publishers, Sudbury, MA.

McCulloch, Warren S. and Pitts, Walter. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematical biology*, **5**(4), 115–133. doi:10.1007/bf02478259.

Oram, Andy and Wilson, Greg. 2007. *Beautiful code: Leading programmers explain how they think*. O'Reilly. isbn 0-596-510004-7.

Panangaden, Praskash. 2001. Measure and probability for concurrency theorists. *Theoretical computer science*, **253**(2), 287–309. doi:10.1016/s0304-3975(00)00096-7.

Perrin, Dominique. 1990. Finite automata. *Pages 1–57 of:* Van Leeuwen, Jan (ed), *Handbook of theoretical computer science, volume B: Formal models and semantics*. Elsevier and mit Press. isbn 0-262-22039-3.

———. 1995. Les débuts de la théorie des automates. *Technique et science informatiques*, **14**(4), 409–433.

Pnueli, Amir and Zuck, Lenore. 1984. Verification of multiprocess probabilistic protocols. *Pages 12–27 of: Podc '84: Proceedings of the third annual acm symposium on principles of distributed computing*. New York, NY, USA: ACM. isbn 0-89791-143-1. doi:10.1145/800222.806732.

Prechelt, Lutz. 2000. An empirical comparison of seven programming languages. *Computer*, **33**(10), 23–29. doi:10.1109/2.876288.

Reiter, Michael K. and Rubin, Aviel D. 1998. Crowds: Anonymity for web transactions. *Acm trans. inf. syst. secur.*, **1**(1), 66–92. doi:10.1145/290163.290168.

Sudkamp, Thomas A. 1998. *Languages and machines*. 2nd edn. Reading, ma: Addison-Wesley. isbn 0-201-82136-2.

Tarjan, Robert E. 1976. Edge-disjoint spanning trees and depth-first search. *Acta informatica*, **6**(2), 171–185. doi:10.1007/bf00268499.

Varó, Andrés Marzal and Jiménez, Víctor M. 2003. A lazy version of Eppstein's *k* shortest paths algorithm. *Pages 179–190 of:* Jansen, Klaus; Margraf, M.; Mastrolli, M. and Rolim, José D. P. (eds), *Proceedings of 2nd international workshop on experimental and efficient algorithms (wea 2003)*. Lecture Notes in Computer Science, no. 2647. Springer-Verlag.

Yu, Sheng. 1997. Regular languages. *Pages 71–77 of:* Rozenberg, Grzegorz and Salomaa, Arto (eds), *Handbook of formal languages, volume 1: Word, language, grammar*. New York, ny: Springer-Verlag New York, Inc. isbn 3-540-60420-0.