Markov Chains for programmers

Markov Chains for programmers

First Edition – April/2022

Ricardo M. Czekster
Birmingham, United Kingdom

Self Publishers Worldwide

This book was typeset using LaTeX software (using OverLeaf – online).
It employs proprietary software (mostly MS-Excel) for demonstrating MC.
It uses the PRISM Statistical Model Checker, version 4.6 (2022).
For demonstrating direct solution methods, it worked with MATLAB®.

Preface

This book originated as I perceived a need for better (and simpler) explanations surrounding Markov Chains (MC). If you take the (formal) literature on this subject and if you do not have a (rather strong) mathematical background to understand the concepts, you are probably restricted to use MC, which I always thought that I could change that. In my view, this is a nice opportunity to sharp programming skills, as the numerical methods proposed here, for larger models, pose interesting challenges.

The purpose of this book is to present MC firstly to programmers (at any level), but it is not restricted though; I think that broader audiences might enjoy it as well. The idea is to grasp the basic notions and then implement solutions that are reproducible. Prove to that is that I have shared spreadsheets and code in GitHub, so there are multiple ways of verifying answers.

I hope you enjoy the next pages. Please, feel free to send me your remarks, suggestions, comments, and (eventual) critics.

I shall keep track of changes between editions, errata, etc., in a yearly rate.

- **01/04/2022**: first edition, MC, DTMC, CTMC, basic notions;

Table of Contents

List of Figures

List of Tables

Introduction

Have you ever opened a book promising to teach you something and by page ten you were (already) completely lost, only to put it away and never again considered looking again? Well, I have, and I have done so exactly on the topic of this book: Markov Chains[1].

I dare you to go to your public library (or the Internet) and open a book, even introductory on this subject and start reading. Now, please answer honestly: did you pass page ten? Well, many people have not, you are not alone. However, working with modelling throughout the past 20 years and having revisited a lot of things, I embark on the challenge of making you, the reader, pass page ten and grasp Markov Chains, examples, and side programs that I will propose. I am on this mission and I will respect you, especially if you have a poor mathematical background but still want to delve into this technique. In pursuing this mission, I shall:

- Use as little mathematical notation as I possibly can; if I use, I promise to explain in detail what it means
- Combine any model or spreadsheet with code (usually in C/C++)
- Try to convey notions as simple as possible, but I understand that sometimes some formulas might get in our way (mainly in MS-Excel)

Please allow me to be quite candid here in the introduction. This book is dedicated to programmers and people initiating studies in Markov Chains. If you are looking for more formal aspects, please look elsewhere in the vast literature on this subject. This book is about understanding the basic principles surrounding Markov Chains and how to implement basic numerical methods to derive actionable indices for analysis. It has been written for programmers or engineers with enthusiasm for computing (and coding), eager to get their hands dirty on Markov Chains and willing to grasp results in short time.

In here there will not be discussions about Chapman-Kolmogorov equations, Perron-Frobenius theory, Krylov sub-spaces, Chebyshev methods, Arnoldi or GMRES solution methods, and other similar concepts. I urge you to feel free

[1]However, in my defence, I *persevered* and only stopped after I gathered a reasonable amount of knowledge in the subject.

to submerse in this literature if it is relevant to you. I assure there are some interesting notions lurking in there and depending on your background, it might prove useful in your career. For example, you could start looking at this book by Häggström [2] or Trivedi & Bobbio work on Reliability and Availability engineering [3] which are both remarkably interesting.

Now, let's go for it and hopefully pass page ten! I'm with you in this journey! Before we embark any further, a few housekeeping and notes of caution:

- From time to time, I will propose programming 'challenges'; the idea is that you try to code and test and come up with your own solution. I shall add working code and comments to challenges in the last chapter – most of the code will be produced using the C/C++ Programming Language.
  - All code is covered by the GPLv3 License – free to redistribute citing the source. Feel free to use it as you wish.
  - I have used proprietary MS-Excel for the spreadsheets, so I recommend running in this application – you may want to test on other (free) platforms.
  - I strongly recommend running the challenges in a GNU/Linux machine (I tested on a Virtual Machine running KUbuntu 18.04 and GNU/Linux kernel 5.4.100(generic).
- I have set up this website to accommodate all models, auxiliary code, and spreadsheets discussed here at: `https://github.com/czekster/markov` — feel free to download everything as needed (the book is CC-BY-4.0 and the code is GPLv3).
- At all times, try to grasp (fully?) the code, *what* it does, *how* it does, and *why* it does. Understanding the sequences of commands is crucial to improve reasoning about the subject.

Enjoy.

Chapter 1

Markov Chains

This is a gentle introduction to Markov Chains (MC). Before delving into MC however, we need to comment on basic key notions and ideas.

1.1 Model and system

A system, broadly speaking, comprises a set of interacting components that collaboratively perform functions to address a problem. A model abstracts a system, i.e., it captures the most relevant operational semantics (i.e., the essence of what it effectively does) into constructs for example a table (in mathematical words, in a matrix) of relationships among pre-modelled components. Here, it is true that the adage *"modelling is more of an art than a science"* comes up. Good modellers learn how to capture the system's function in essence and convey it into a working model that not only represents its behaviour, but it is amenable for reasoning and decision making.

Norbert Wiener (1894-1964), a mathematician, once stated that *"The best material model for a cat is another, or preferably the same cat"*. If your model represents the totality of parts belonging to a system under study, then why modelling? Why not inspecting and observing the real-world system for extracting properties and inferences?

1.2 Emergence of Markov Chains

Markov Chains (MC) were envisioned by Andrey Andreyevich Markov (1856-1922). His doctoral advisor was Pafnuty Chebyshev[1]. He published the first paper on the topic in 1906 and then applied later to study the distribution of vowels in Alexander Pushkin's opus *Eugene Onegin*. During those days, it attracted a reasonable amount of attention. Markov himself thought of the

[1]This was the last instance where I mention "Chebyshev" in this book, so, don't worry (obs.: what a first name!).

limited applicability of his result, perhaps only to the analysis of texts and identification of author's styles.

It was picked up by Alan Scherr in the 1960's in his thesis on time shared systems (MIT), where he applied to study scalability issues [4]. Nowadays, analysts employ MC to study a variety of phenomena, from economic models to Internet searching (PageRank® algorithm [5] [6]). If you have time, I suggest reading this paper [7]. It details the top five most significant MC applications, with interesting discussions.

1.3 Basic modelling primitives

MC models employ states, transitions, and values decorating transitions (I will comment more about these 'values' in Section 1.5). And that's it. It cannot be any simpler than that. The problem in MC is not about the primitives but about the abstractions modellers use to convey behavioural ideas to their investigations. You must make decisions on what and about the Level-of-Detail (LoD) of components, in a way that is not too abstract nothing is learned, and not too detailed that hinders comprehension.

There is a wealth of literature on MC [8, 9, 10], numerical methods [11], and applications [12], relating models to systems, decomposition techniques, division of responsibilities, coupling/cohesion, and so on. The vastness of notions should not overwhelm you because after all, what is important here is to capture the operational essence of systems and key component interactions.

1.4 Input and output

MC will take a model of a system with enumerable sets of states and transitions decorated with values as a Directed Graph (DG), i.e., a construct that has vertices or nodes (states), connected by arcs (transitions). A DG connects states where every transition has arrows pointing to other states (also known as a *digraph*).

After one finishes working on the model, tools prepare the model for numerical solution by applying procedures that creates matrices with interesting properties (to be inspected and discussed later). The procedure also checks whether the MC respects fundamental properties (mostly ergodicity – explained later as well) or checking impossible MC models (models having invalid states or values – I will not delve into this now, but I shall comment about it later. For *impatient* readers, the discussion here is about absorbent states, where the solution mechanism will differ from 'normality').

Even if a model is well-formed, it may not yield a solution in a timely fashion (steady state analysis vs transient analysis that will be discussed later). What analysts would like to compute are the permanence probability of each state in the model, i.e., the percentage of time spent in that state in a very long simulation. The computation is done by different ways, for instance, one might

employ simulation, direct solution (using a set of linear equations), numerical methods (power matrix, Vector-Matrix Multiplication), and so on. One computes the *eigenvector* of the matrix, i.e., the unique vector that 'describes' the matrix (let's put this way to simplify everything – I urge you to deepen your knowledge a bit more on this concept, in specialised literature). Another way of thinking about eigenvectors is to think that one is performing a dimension reduction, from a matrix (a two-dimensional structure) to an one-dimensional vector. Again, we compute this vector only if the matrix respects a set of properties which we shall discuss in due time.

1.5 Types

The literature provides a distinction for Markov Processes and Markov Chains. It explains that modellers use the first for describing continuous processes whereas the second one for discrete time considerations. I prefer to call Continuous Time Markov Chains (CTMC) for working with durations and rates when modelling states and Discrete Time Markov Chains (DTMC) when analysts would decorate transitions with probabilities [3].

Important

- **CTMC:** when working with durations (that are converted to rates), residence time in states (or sojourn times).
- **DTMC:** when decorating transitions with probabilities.

That is what will happen to the model, intuitively. A bit more formally, we care about observations of the system at time T. If these are discrete, e.g., $T = \{0, 1, 2, \ldots\}$, then we have a discrete stochastic (random) process whereas if T is continuous, where $T = \{0 < t \leq \infty\}$ the process is continuous [11]. In CTMC, to proper work with the Markov property, the time spent in a state must be independent of the time already spent in that state. Thus, this time must be exponentially distributed whereas in DTMC, the residence time in a state that respects the Markov property must be ruled by a geometrically distributed value, which are the only distributions that exhibit the memoryless property [11].

1.6 Values decorating transitions

After discussing the types of MC, a lot of concepts *floated* around. One must decorate transitions with rates, frequencies, or durations for CTMC, and probabilities for DTMC. Let's start with the easiest way of decorating transitions, employing durations. Modellers should observe a system for a considerable amount of time and then inspect the amount of time the observed entity (it

could be anything, it will be dependent on the model) has remained in *that* state. We say that the modeller is observing the *residence time in* the state (also called the *sojourn* time). Then, to decorate the transition, it will require to compute the *rate out* of the state (leaving the state) by using the inverse of the duration (i.e., $\texttt{Rate} = \frac{1}{\texttt{Duration}}$), also called the *frequency out* of the state [3]. Observe Table 1.1 next:

Table 1.1: Duration and rate for decorating transitions in CTMC models.

Observed duration (time spent IN the state) in min	Rate or frequency (OUT of the state) over one hour	Remarks
60	$60/60 = 1$	If any given entity has stayed in the state for 60 minutes, the rate exiting this state in one hour will be one, i.e., it would leave the state once in one hour.
30	$60/30 = 2$	If in a state for 30 min, the rate out would be modelled as two.
Note that one could have used $\frac{1}{60}$ and $\frac{1}{30}$ to decorate transitions without any difference for the solution. That is because of, in terms of *proportions*, they are the same.		

For DTMC, it is simpler: one decorates transition with perceived probabilities out towards other states or within the same state (loops). Loops are disregarded in CTMC for reasons that will be explained later (note for *impatient* readers: the diagonal is obliterated and the negative line sum of the rest of transitions is used). Now, let's reason about a simple model with two states as depicted in Figure 1.1 (this shall be referred as "The lighting model"):

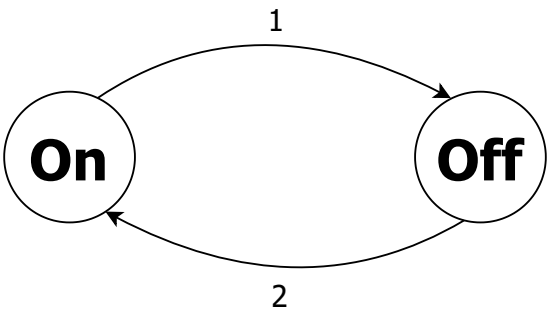


Figure 1.1: A simple CTMC model with two states and two transitions.

In this example, the modeller had observed that when a light switch was `On`, it remained lit for 60 minutes whereas when `Off`, someone would invariably turn back on every 30 minutes. As we shall discuss in detail next one could take this representation of the behaviour of a system and convert it to a table that will be subjected to a numerical method that will be able to answer the

following: *"At any given time of the day, what is the probability of the system in the `On` state or in the `Off` state"*.

In other words: *"You were somehow 'teleported' into this room: what is the probability that the light is either `On` or `Off`?"*. Having the ability to answer this one might take measures to decide on making changes to the system or promoting ways to help users turn the light off more frequently to save money, and so on. Of course, this is a *very simple* model, and it respects all requirements of MC in terms of well-formed models, and well-behaved durations, and so on. We shall discuss cases where these assumptions may not hold true.

1.7 Markov property or memoryless property

By now you can identify different types of MC (continuous time or discrete time), however, deep down, what does this *actually* mean? MC have a governing principle that states that the past is irrelevant to determine the future steps taken by the system. Let's imagine a frog in a pond with a set of lily pods[2]. This mindless frog cannot store its past movements in his little brain, so it jumps around lily pods never considering the visits taken beforehand. When jumping around, the frog has forgot about the past (perhaps because of his reduced brain capacity?), so it makes decisions on whether to stay or to move based on what it is perceiving in that moment in time. And then it keeps jumping between pads until it gets tired and go to sleep.

One observer (you?) might have been filming these for several hours and would like to know the set of lily pods that he most likely visits every time, by trying to extrapolate the frog's life until the infinite (looking for the steady state as one might say). So, you either take durations and mark the lily pods he visited (CTMC) or some probability given the observations (DTMC). And then you convert this system to a model and subject the model to a numerical method that will rank (by the highest probability), the set of most visited lily pads in the pond.

Now let's go back to those MC types, e.g., Continuous Time or Discrete Time once again, which is why we have started the discussion. Modelling with durations (and then with rates) and given the fact that you would like to simulate any given situation plus the fact that you are bounded by the memoryless property, you will have to model these durations also using a memoryless distribution. In this case, you would use the exponential distribution (mind that there are other distributions that are out of the scope of this work).

In this sense, you are modelling a continuous time 'evolution' (we might say) of this system. This distribution is better explained with an example. Let's suppose a person takes a bus to go home every day and after one year taking the bus and listing the time spent waiting for it in a bus stop, he (or she) ended up defining it as an exponential distribution of average parameter of 20 minutes.

[2]Attribution needed: the lily pads and frog example can also be found in [11], on page 4, detailing MC.

Table 1.2: Yearly observations of a person waiting for a bus.

	January	February	March	April	May	June	July	August	September	October	November	December
1	16.4	3.2	22.9	4.4	3.3	43.7	7.3	23.1	1.3	5.4	6.7	1.5
2	53.7	14.0	4.7	3.6	6.0	50.3	12.2	28.8	1.0	4.4	12.3	39.6
3	48.4	11.7	15.2	0.1	46.5	26.7	28.9	21.9	6.2	22.7	13.4	11.8
4	6.0	5.4	19.5	4.9	28.3	17.6	7.4	14.5	4.7	1.7	11.2	24.6
5	1.6	35.9	4.1	7.6	13.2	17.5	27.5	18.2	46.1	4.9	10.4	28.1
6	20.9	22.7	14.9	119.9	5.2	102.1	3.4	83.2	16.6	45.8	2.1	11.4
7	22.8	27.1	5.9	40.3	28.8	8.4	18.8	34.4	5.6	32.3	66.3	1.0
8	28.6	1.4	2.0	4.4	13.8	22.3	2.0	34.2	1.6	37.2	8.5	2.9
9	40.7	2.8	5.0	36.3	1.2	9.8	61.3	31.8	4.5	10.6	20.4	40.3
10	20.9	27.4	11.2	6.8	50.0	20.8	1.2	9.9	10.1	3.1	27.6	12.7
11	0.3	4.7	15.1	3.3	20.4	26.7	29.8	42.1	8.5	4.5	14.9	36.3
12	29.5	2.1	41.3	3.5	3.5	0.7	24.8	26.2	2.2	9.3	8.7	0.2
13	46.0	1.5	41.2	51.5	38.2	7.5	6.1	12.0	79.2	2.3	42.2	0.7
14	3.4	52.5	148.5	10.6	5.5	21.3	10.2	11.0	16.7	49.7	2.0	11.4
15	9.1	8.7	39.0	5.2	1.4	20.2	2.6	22.0	42.1	57.5	28.7	17.3
16	82.0	10.3	1.7	2.9	8.6	38.9	91.0	10.2	23.1	17.2	5.9	0.3
17	14.7	21.8	24.9	94.5	10.1	19.8	6.6	32.0	32.2	2.3	20.9	21.9
18	20.9	51.7	37.7	29.7	1.1	1.8	9.5	15.5	20.8	12.9	61.7	5.1
19	2.9	18.0	30.0	21.8	0.1	5.7	128.7	16.3	5.8	60.6	3.4	49.5
20	41.4	13.7	57.5	2.9	32.4	33.0	6.7	16.3	30.2	9.6	34.9	5.2
21	0.7	11.1	9.6	9.3	30.1	11.2	3.9	6.4	6.6	8.8	32.5	19.8
22	38.4	8.1	20.9	36.6	6.3	1.6	52.9	37.7	5.3	31.6	24.2	40.5
23	19.1	6.0	10.6	3.0	48.1	54.8	28.9	40.1	0.5	5.7	38.1	16.2
24	45.3	50.1	13.8	10.1	10.2	12.9	31.7	2.3	5.1	40.0	32.9	55.5
25	6.7	6.7	5.9	6.7	89.7	33.6	6.8	9.4	3.5	46.6	13.0	4.1
26	21.4	40.8	4.9	1.0	7.3	31.8	13.7	21.0	39.1	25.5	15.0	9.5
27	7.1	10.9	1.2	24.9	3.1	3.4	7.0	42.0	50.3	28.4	67.3	7.1
28	24.8	4.2	11.6	51.0	22.9	13.2	9.9	93.9	6.2	23.4	75.2	5.6
29	62.5		3.0	2.2	21.2	17.1	49.0	5.4	31.2	0.7	50.1	15.1
30	9.1		4.6	29.0	32.1	6.9	34.2	28.7	26.6	5.9	0.4	20.6
31	112.8		59.4		2.4		12.2	15.9		24.4		22.2
μ	27.7	17.0	22.2	20.9	19.1	22.7	23.8	26.0	17.8	20.5	25.0	17.3

This means that one could wait one minute, 10 minutes, 25 minutes, or even 78 minutes (or more!) for the bus to arrive, just because it is using an exponential distribution. On *average* (if one draws successive samples) it will correspond to 20 minutes. See Table 1.2 (`spreadhsheets/Chapter01-bus-times.xlsx`), with a list of some observations about the waiting time in the bus stop over a period of one year (365 observations).

To compute the time waiting in the bus stop for each day one uses the conversion from the uniform distribution (in MS-Excel it corresponds to the `RAND()` function, yielding uniformly distributed pseudo-random numbers between 0 and 1) to the exponential distribution: `Exp=(-1/PARAMETER) * LN(1-RAND())`.

The `PARAMETER` value for this case was set to 20 (if one computes the average of draws, it will be closer to this value).

Note that this is because we are modelling backwards here: we are generating numbers when we already know the distribution (we do not have the actual real-world observations here – we are trying to demonstrate one possible way of deriving these numbers). This is because one wants to work with the exponential distribution (at least for the time being).

What might happen in real life is that real measurements might not fit into the exponential distribution. In this case, we are in violation of employing MC to work with our problem and we shall resort to other technique, for instance, simulation. This shall be commented further in due time. For the time being, let's be happy with the fact that we can generate numbers from the exponential (memoryless) distribution. The last line (μ) in the table contains the monthly average.

If you take a closer look at this table, you will see that the average (for all table) is 21.7 minutes, the minimum value is 0.1 minutes (3.2 seconds!) and the maximum value is 148.5 minutes (2.5 hours!!). So, there are good days and bad days, but on average, the expected behaviour is to approximate to the rate I set forth: 20 minutes. It is true that it is memoryless, sometimes the bus took let's say 7 minutes (January 27^{th}) but on January 28^{th}, it took 24.8 minutes to arrive! As a matter of fact, it is capturing the memoryless property at its finest.

This is contrasted with Discrete Time. The best idea of understanding the concept is to consider a sort of *metronome*, with pulses ticking the same (without any losses). So, the system governed by a DTMC would change states following the rules set forth by the probabilities that decorated the transitions, without the notion of time passing (as it happened in CTMC).

To build a DTMC simulation, it suffices to model a situation where a visitor stands on a given state and draw a pseudo-random number from the uniform distribution and then use this value to consult and then jump to the next state. And then repeat this process many times and counts the visits to each state. Time is considered to jump on every metronome pulse, moment that the system changes state. We will stop now explaining about simulation, this topic is covered in detail in the next chapters.

1.8 Let's code!

For coding these challenges you will need to know how to process parameters from the command line and work with time constructs in C (for the pseudo-random[3] number generator).

[3]Most programming languages employ a pseudo-random generator that produces number from known formulas. See *Linear Congruential Generators* in the Rosetta code project: `https://rosettacode.org/wiki/Linear_congruential_generator#C`.

1.8.1 Challenge 01

File: `challenge01.c`

Do: write a program that given a parameter N (passed in the command line), it computes and shows N uniformly distributed numbers (each line shows 10 numbers, to ease output).

Notes:

- The parameter (N) is an integer, but you will have to convert from the command line using the `atoi` function (this function converts a string to an integer).
- Use `#include <stdlib.h>` and `#include <time.h>` and auxiliary function `random()`.
- Use a seed based on the current time (for increased randomness) with function `srand(time(NULL))`.

Good programming practice write a function to compute a uniform number and another function to show N uniform numbers, separating concerns in your code.

Suggested variations: (no code or auxiliary file will be provided)

- Do the same task in a spreadsheet (MS-Excel or other). Plot using a scatterplot. Plot the frequency on a bin size of 0.1. See that the numbers are *de facto* uniform.
- Count the number of values between 0.0 and 0.09, then from 0.1 to 0.19 and so on – they should have a uniform distribution, on average. See whether this is true.

1.8.2 Challenge 02

For the next challenge we shall use parts of the previous challenge (Section 1.8.1) to generate exponentially distributed numbers (adding the following 'cell' formula: `=(-1/A1)*LN(1-RAND()` – in MS-Excel, with parameter in cell `A1`).

File: `challenge02.c`

Do: write a program that given a parameter N (passed in the command line) corresponding to the PARAMETER of the desired exponential distribution and another parameter M (the number of samples to compute),

with functions to compute samples from the exponential distribution given the uniform distribution.

- The function prototype is `float next_exp(float rate);`

Notes:

- The parameter (N) could be a float value, so conversions will use the `atof` function
- You will use the `double log(double);` function in C
  - So, you must compile the file with `-lm` (this will link with the math API)

Good programming practice: write *functions* as the previous challenge recommended.

Suggested variations: (no code or auxiliary file will be provided)

- Write the output to a file (text).
- Do the same task in a spreadsheet (MS-Excel or other). Plot using a scatterplot. Then order numbers and plot using lines, adding a trendline based on the exponential function.

1.9 What is considered a 'proper' MC?

To be considered a MC, the model must adhere to a set of properties. We have discussed 'well-formed' or 'well-behaved' models in previous sections, now it is time to inspect what exactly this entails. MC considers so called ergodic chains, i.e., those that assume that all states belonging to models are visited in (eventually) in a random way.

In other ways, a MC is ergodic if a visitor can jump back and forth to and from every state (not necessarily in one step). In many definitions (in books), ergodic MC are also called irreducible (see Section 1.9.1). The MC should not be periodic, e.g., it returns to the original state in a pre-defined number of steps. For example, consider the MC in Figure 1.2:

Starting in state A one might return to state A after three hops (one to B, then to C, then back to A) every time. This characteristic defines a periodic MC, so we shall concern modelling chains that are aperiodic. Finally, we will address positive recurrent MC, i.e., those chains that are recurrent, they eventually return to the state after a number of steps and positive if the time it took to return is relatively fast (intuitively speaking).

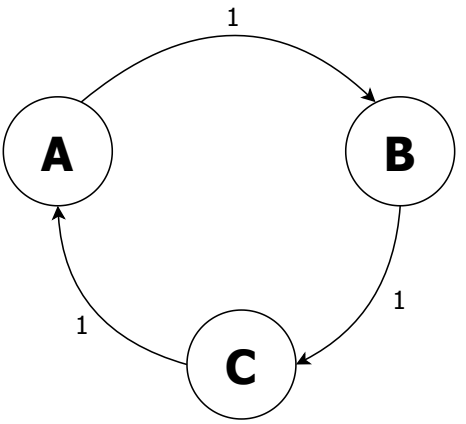


Figure 1.2: An example of periodic MC with three states.

Important In summary, we are dealing with MC that are ergodic (irreducible), aperiodic, and positive recurrent.

That's the mantra of MC in (almost every) book ever written about this subject. Start getting acquainted with the notion that not everything that you put down as a model will be considered a MC and have a steady-state solution. If your MC adheres to these properties, discovering the solution vector shall be possible within a decent amount of time, using appropriate numerical methods. We will not cover here MC with absorbent states, i.e., states with only incoming transitions and no outgoing transitions. It is worth noting that non-ergodic MC will not yield results if one employs a (let's say) classic numerical method. As we shall see these models might yield results if one resorts to simulation approaches.

1.9.1 Irreducible MC

Consider the next two MC, Q^0 and Q^1 depicted in Figure 1.3:

In model Q^0, whenever state E gets visited, it stays switching to F state indefinitely. In terms of result vector, we shall see probabilities only for those two states. In model Q^1, there are in fact two MC to consider, one with states A, B, C, and D, and another with E and F. Thus, one could solve each one separately, instead of it all. We are concerned here about chains that are irreducible, i.e., starting in any state, it is possible to reach any other state (not necessarily in one step). If Q^1 had a transition from F to A, B, C, or D, it would be sufficient to deem the matrix irreducible, because it would respect the property.

William Stewart's book [11], on page 38, discusses so called *Nearly Com-*

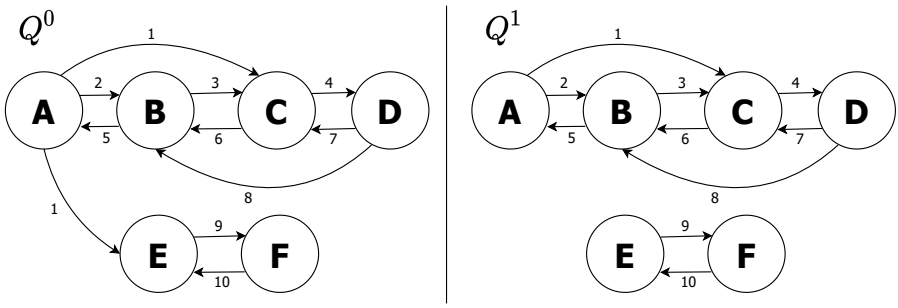


Figure 1.3: Two MC for demonstrating irreducibility (in this case, the lack thereof) properties (note that Q^0 and Q^1 are not irreducible).

pletely Decomposable (NCD) stochastic matrices, cases where transitions among states are weak (or low), where the matrix is irreducible however the model could have been broken in two (or more) to ease analysis. The literature refers to partitioning the state space in subsets with strong and weak interactions. Please, refer to the source for further information about NCD. Interested readers may also enjoy other properties such as *aggregation* or *lumpability* as well as other notions that surface as recurrent topics in MC.

1.10 Limitations of Markov Chains

MC are not free from drawbacks, unfortunately. For example, models should respect the memoryless property because *all* rates are drawn from the exponential distribution. Another limitation concerns the state space of problems: even for simple models, the number of states required to represent the MC could be enormously high (state space explosion problem), presenting difficulties in achieving solution.

To cope with some of the limitations of pure MC and its dimensionality problems, research has proposed higher-level constructs such as *structured MC*. Examples are Queueing Networks (QN), Stochastic Petri Nets (SPN), Performance Evaluation Process Algebra (PEPA), or Reactive Modules (formalism employed by the PRISM Statistical Model Checker to represent CTMC, DTMC, Markov Decision Processes – MDP and Probabilistic Timed Automata – PTA) [13].

1.11 Comments on the spreadsheets

For this chapter we have the following spreadsheet to inspect:

- `spreadsheets/Chapter01-bus-times.xlsx`: it shows the exponentially distributed bus times using the uniform distribution;

Play with the spreadsheet, change parameters, see what happens, build code that replicates (note: it will never yield the same results[4] due to the fact that you are employing different sets of pseudo-random numbers) the results of the spreadsheet.

[4]Unless you make the `seed` parameter to the `srand(int seed);` function constant.

Chapter 2

DTMC

Let's start discussing DTMC, where models have probabilities in the transitions, as depicted in Figure 2.1 (this model was covered and commented earlier, the only difference is that for the other one we used rates – CTMC):

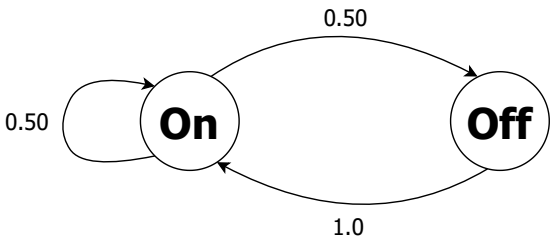


Figure 2.1: A simple DTMC with 2 states (the "lighting model").

For this MC, one may assume that since the On state is the most visited (because of the self-loop present in this model), the final probability vector π (our result) will be higher on this state (intuitively). The restriction of DTMC models is that they must have each line summing to 1.0, and for the case of numerical solution, that it also respects previously discussed MC properties. There are several ways of solving this DTMC. For instance, after representing the model in a matrix, one could multiply it by itself many times, until it reaches convergence (steady state). This is known as the Power Matrix method. But first, let's create the model appropriately in Table 2.1:

Table 2.1: A two state representation of the lighting problem.

	On	**Off**
On	0.50	0.50
Off	1.00	0.00

After solution using the Power Matrix method (please, consult file in repository called `spreadsheets/Chapter02-DTMC-2states.xlsx`), we compute that $\pi_{\texttt{on}} = 0.66667$ (around 67%) and $\pi_{\texttt{Off}} = 0.33334$ (around 33%). Note that in the MS-Excel file we have employed the function `MMULT` for the same matrix. To do this, one must first select an empty 2x2 result cell (where the results will be), pressing F2, then putting the formula `=MMULT(D3:E4,D3,E4)` and pressing `CTRL-SHIFT-ENTER`. This is required in MS-Excel when operating with matrix multiplication (where the results are matrices as well). It was necessary to multiply the matrix by itself seven times (M^7) before reaching convergence (steady state) for this model. You may try to replicate these results as well.

Another way of reaching the same conclusions is to employ a Vector-Matrix Multiplication (VMM) method that from an initial vector multiplies the matrix iteratively until it finds the results. This method is more lightweight than the Power Method, because instead of a Matrix-Matrix multiplication, now it is a Vector-Matrix product, which is less cumbersome. It suffices to say that this method produces the same results as before. It took 21 steps for reaching the steady state.

Another difference in the MS-Excel for this method is needed: we should fix the matrix position (using the symbol '$' before cell placements). The initial vector chosen was `[ 1 0 ]` however, any initial vector summing one would work – perhaps the iterative method will have the side effect of performing better, i.e., producing the results in fewer iterations. That is what is called *preconditioning*, and there is a wealth of research on methods to accelerate solution of MC. If the vector does not sum to one, you will have to adjust the results, because let's say, it sums 0.25, this 0.25 will be distributed 66% to the `On` state (≈ 0.16667) and 34% to the `Off` state (≈ 0.08333), which is equal to the answer.

2.1 Remembering matrix operations

For MC solution mechanisms we will require to review matrix operations, especially matrix-matrix and vector-matrix multiplication. For example, when multiplying (operator 'x' below) two matrices X and Y, both of order 3x3, then $X \times Y$ is shown in Equation 2.1:

$$\texttt{Let } X = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \texttt{ and } Y = \begin{bmatrix} J & K & L \\ M & N & O \\ P & Q & R \end{bmatrix}$$

Then,

$$X \times Y = \begin{bmatrix} AJ+BM+CP & AK+BN+CQ & AL+BO+CR \\ DJ+EM+FP & DK+EN+FQ & DL+EO+FR \\ GJ+HM+IP & GK+HN+IQ & GL+HO+IR \end{bmatrix} \quad (2.1)$$

There are several APIs for handling matrix multiplication efficiently. We are showing the basic process here because we are interested in coding this

down the line (for our next model). As we have shown, MS-Excel (and other spreadsheet-based applications) have built-in functions to help multiplying matrices altogether. For Vector-Matrix Multiplication, the process is easier, i.e., only the first line of X is used (Equation 2.2):

$$\texttt{Let } X = \begin{bmatrix} A & B & C \end{bmatrix} \texttt{ and } Y = \begin{bmatrix} J & K & L \\ M & N & O \\ P & Q & R \end{bmatrix}$$

Then,

$$X \times Y = \begin{bmatrix} AJ + BM + CP & AK + BN + CQ & AL + BO + CR \end{bmatrix} \quad (2.2)$$

This concludes the basic reviewing of matrix operations that we will require to code the solution.

2.2 The Belfast weather model

This model is depicted on William Stewart's work [11] (page 6), where it models the weather of Belfast, Northern Ireland[1]. Figure 2.2 shows a visual representation of the states and transitions of the DTMC model.

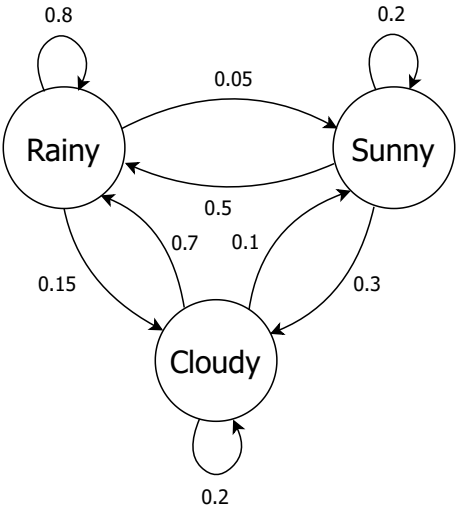


Figure 2.2: Model representing the weather of Belfast, in Northern Ireland.

Just by looking at transitions and their weight (the values associated to the modelled probabilities decorating transitions), one might assume that the

[1] Häggström [2] contrasts two other weather models, with comparisons, namely the *Gothenburg* model and the *Los Angeles* model.

expected probability result for both `Rainy` and `Cloudy` states will be high. Seldom it transitions to the `Sunny` state, and from this state, it returns with high probability towards `Rainy` (0.5) or `Cloudy` (0.3) states.

The solution of this model (`spreadsheets/Chapter02-DTMC-3states.xlsx`) shows that the final probability vector π is distributed as: $\pi_{\texttt{Rainy}} = 0.7625$, $\pi_{\texttt{Cloudy}} = 0.16875$, $\pi_{\texttt{Sunny}} = 0.06875$. So, `Rainy` or `Cloudy` days account for almost the totality of probabilities, `Rainy` + `Cloudy` $= 0.7625 + 0.16875 = 0.93125$, which is the same of thinking `Rainy` + `Cloudy` $= 1 -$ `Sunny` (for this model), yielding the same result. So, one could expect that any given day, in Belfast, to be either `Rainy` or `Cloudy`, and prepare for this.

2.3 Let's code!

This challenge will work with static structures and simple tests to see whether the model is well-written as expected for MC. You will need to review ergodicity, absorbent states, and the previously discussed properties of well-formed MC for this challenge.

2.3.1 Challenge 03

Before we start discussing the challenge, it is worth noticing that we are doing *validation* instead of *verification*[2].

File: `challenge03.c`

Do: write a program that declares a static matrix and runs some tests (e.g., sum of lines equals to one, etc.) and output features.

Notes:

- The matrix corresponding to the model could be defined hard coded with static declarations (`float m[3][3];`), however, you could implement a way of opening a (text) file with the model (where values are separated by spaces or commas). That would require dynamic allocation and pointers.

Good programming practice:

- Use `#define LIN 3` and `#define COL 3`.
- Employ error codes (as `#define`) for returning values in the function, for example, `ERR_NOT_SQUARE, ERR_ABSORBENT, ERR_LINE_SUM, SUCCESS`).

[2]The word *verification* conjures the notion of *formal verification* processes that models might undergo against specifications.

Implement:

- Function `void print_static_dtmc(float m[LIN][COL]);`
- Function `void print_static_array(float a[LIN]);`
- Function `int validate_static_dtmc(float m[LIN][COL]);` that checks whether every line of matrix m sums 1.0 and also if the matrix m has absorbent states
  - Return 1 if matrix checks out and it is well-formed, no absorbent, summing lines to 1.0;
  - Return -1 if the matrix is not square (`#lines` ! = `#columns`);
  - Return -2 if sum of each line (show line) is not 1.0;
  - Return -3 if matrix has absorbent states: check whether there is one state with no outgoing transitions to other states other than to itself;

Tips:

- Work on incoming and outgoing transitions: if there are no incoming or there is no incoming except to itself, then it is absorbent.
- Implement a way to learn about the state which is absorbent (pass a pointer to an integer by reference to the `validate_static_dtmc` function, where it saves it to this variable. Then show the state in the output.

Suggested variations: (no code or auxiliary file will be provided)

- 'Open' the model from a file, and then 'Save' the output to a file.
- Use a library (external API) to power a matrix or multiply a vector by a matrix.
- Run the model on MATLAB® or GNU/Octave and compute the same set of results.

2.4 Solution methods for DTMC

After the model definition and after passing basic tests to see whether it is well-formed and no absorbent states (which would require other solution methods), it is time to solve it and extract state (occurrence) probabilities.

We will work next with three methods: i) Power matrix; ii) VMM; and iii) forward simulation. For i) and ii) methods, the idea is to call the functions iteratively until one achieves convergence, i.e., the resulting matrix has

the same vectors in all positions, or the resulting vector does not change between the last two iterations. If the solution reaches the boundaries set by the numerical methods (e.g., number of iterations), it then diverged (the opposite of converged), and you should state this to whomever call the method and show the last computed vector.

2.4.1 Power matrix, or Matrix-Matrix Multiplication

This method consists on multiplying a matrix (in our case, one where each line sums to 1.0) by itself, iteratively, until one observes negligible differences (or no difference at all) between lines in the final matrix. Because this method employs ensembles of multiplication/addition pair each time, depending on the size of the model, it may behave computationally heavy in terms of required processing power.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Note that there is no convergence insurances that this method will produce valid π probabilities vector after execution. It is the job of the modeller to analyse (partial) results and investigate causes when diverging, refining models, reviewing transition probabilities, and so on.

2.4.2 Vector-Matrix Multiplication

The VMM method will operate in simpler terms, it will multiply an initial vector (where the sum is 1.0) by the matrix iteratively, until this vector has negligible differences between two iterations.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}$$

The power matrix method may converge faster than this method, however, VMM requires less mathematical operations to complete on each iteration. On each iteration of VMM it is possible to see the π vector being redistributed throughout the positions, always summing to 1.0 (as a matter of fact it will sum to the value set earlier).

Let's revisit the Belfast weather model and apply the VMM method:

$$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 0.80 & 0.15 & 0.05 \\ 0.70 & 0.20 & 0.10 \\ 0.50 & 0.30 & 0.20 \end{pmatrix}$$

Here is a step-by-step view of the VMM process.

Consult file `spreadsheets/Chapter02-DTMC-3states.xlsx`, on Tab 'Vector-Matrix Multiplication':

```
Initial Vector: 1 0 0
Iter.1: 0.8          0.15         0.05
Iter.2: 0.77         0.165        0.065
Iter.3: 0.764        0.168        0.068
Iter.4: 0.7628       0.1686       0.0686
Iter.5: 0.76256      0.16872      0.06872
Iter.6: 0.762512     0.168744     0.068744
Iter.7: 0.7625024    0.1687488    0.0687488
Iter.8: 0.76250048   0.16874976   0.06874976
Iter.9: 0.762500096 0.168749952 0.068749952
```

Note that by Iteration 9, the residue between Iteration 8 and this one is negligible. Depending on accuracy requirements, this could be handled by continuing the process until the residue is smaller than a pre-defined threshold.

2.4.3 Forward simulation

For the DTMC simulation (also called *forward* simulation), the process is different, and one must follow this process:

- Define the desired number of visits or samples (e.g., 1000);
- Define a start state to visit initially (any state belonging to the model);
- Modify the probability matrix of the DTMC to withhold the cumulative probabilities on each position, that will ease the selection of the next state;

For example, suppose a three state DTMC with the following probabilities:

$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{0} & \mathbf{1} & \mathbf{2} \end{array} \\ \begin{pmatrix} 0.2 & 0.4 & 0.4 \\ 0.1 & 0.2 & 0.7 \\ 0.3 & 0.6 & 0.1 \end{pmatrix} \end{array} \texttt{becomes} \begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{0} & \mathbf{1} & \mathbf{2} \end{array} \\ \begin{pmatrix} 0.2 & 0.6 & 1.0 \\ 0.1 & 0.3 & 1.0 \\ 0.3 & 0.9 & 1.0 \end{pmatrix} \end{array}$$

On the left-hand side, there is the original matrix, and on the right-hand side, an accumulated version, on each state, of the same matrix.

When making draws for each state, let's say in state 0, and a pseudo-random number equals to 0.345632, then it suffices to consult the accumulated table and see that if the number was between 0.0 and 0.2, then the next state would be 0, if the number was between 0.200001 and 0.6 the next state would 1, otherwise state 3.

Let's continue the process. From that chosen start state, one:

1. Check if the number of samples reached the desired value: if yes, stop the process (and go directly to step 5);

2. Add the visit to this state in a counting vector withholding all visits;
3. Draw a uniform random number;
4. Look at the transitions from that state;
5. If the probability is less than the position at the cumulative matrix, jumps to that position;
6. Repeat steps 1-4;
7. Compute visits statistics (e.g., divide each position of the counting visits vector by the total number of samples);
8. Present the vector to the user (it should approximate to the analytical counterpart);

To feel convinced that this actually works, nothing better than a demonstration. Consult `spreadsheets/Chapter02-DTMC-3states.xlsx`, Tab 'DTMC simulator'. It shows the accumulated matrix and $1,000$ samples from the Belfast weather model (note that the results approximate to the analytical answer).

2.4.4 Useful comments

Note that sometimes running a method and inspecting the resulting probability vector may help you investigate other problems with models. Depending on the problem, one could resort on modifying the original matrix to *force its way* into becoming a MC, i.e., adding transitions or even new states (that is the approach employed by Google's PageRank® when (attempting to) *fixing* the matrix).

One clear indication of problems is, for example, if just one state has all the probability – this means that your model has an absorbent state If only a small set of states have probabilities, then one should inspect whether the MC is irreducible or perhaps NCD (with weak links).

2.4.5 Direct solution method

It is worth discussing that these three solution methods are not the only way of solving MC. For instance, one might employ direct solution methods (linear equations) and specialised tools. We shall briefly comment on these two methods next.

The direct solution methods employ for example MATLAB® or GNU/Octave. The idea is to work with Equation 2.3 [11, 3]:

$$\pi \times P = \pi \tag{2.3}$$

Recall that π is a vector of unknowns, i.e., for the Belfast weather model (with three states, according to Section 2.2 with a probability vector having size three: $[\pi_{\texttt{Rainy}}, \pi_{\texttt{Cloudy}}, \pi_{\texttt{Sunny}}]$). This is what we would like to compute, i.e.,

the probability of each state in the model. To simplify the discussion, we shall refer this symbolic probability vector as $[\pi_1, \pi_2, \pi_3]$.

If you recall the Belfast weather model, here are the state transitions and the required operations for performing $\pi \times P = \pi$:

$$\begin{bmatrix}\pi_1 & \pi_2 & \pi_3\end{bmatrix} \times \begin{pmatrix}0.80 & 0.15 & 0.05\\ 0.70 & 0.20 & 0.10\\ 0.50 & 0.30 & 0.20\end{pmatrix} = \begin{bmatrix}\pi_1\\ \pi_2\\ \pi_3\end{bmatrix}$$

Refer to the review in matrix multiplication we conducted in Section 2.1.

Let's multiply this symbolically:

$$\begin{cases}0.80\pi_1 + 0.70\pi_2 + 0.50\pi_3 &= \pi_1\\ 0.15\pi_1 + 0.20\pi_2 + 0.30\pi_3 &= \pi_2\\ 0.05\pi_1 + 0.10\pi_2 + 0.20\pi_3 &= \pi_3\end{cases}$$

We should work with the variables and also add the following equation to the mix: $\pi_1 + \pi_2 + \pi_3 = 1$. So, after operating (after some algebra) on the equations, we come up with the following system to solve:

$$\begin{cases}(0.8-1)\pi_1 + 0.70\pi_2 + 0.50\pi_3 &= 0\\ 0.15\pi_1 + (0.2-1)\pi_2 + 0.30\pi_3 &= 0\\ 0.05\pi_1 + 0.10\pi_2 + (0.2-1)\pi_3 &= 0\\ 1.00\pi_1 + 1.00\pi_2 + 1.00\pi_3 &= 1\end{cases} \quad \begin{cases}-0.20\pi_1 + 0.70\pi_2 + 0.50\pi_3 &= 0\\ 0.15\pi_1 - 0.80\pi_2 + 0.30\pi_3 &= 0\\ 0.05\pi_1 + 0.10\pi_2 - 0.80\pi_3 &= 0\\ 1.00\pi_1 + 1.00\pi_2 + 1.00\pi_3 &= 1\end{cases}$$

This is a system of linear equations with three unknowns. The best (and fastest) way to solve this is using MATLAB® (or GNU/Octave) for discovering the π probability vector (refer to file `matlab/matlabsolution_dtmc.m` in the GitHub repository).

```
% MATLAB Model
M = [ -0.20,  0.70,  0.50;
       0.15, -0.80,  0.30;
       0.05,  0.10, -0.80;
       1.00,  1.00,  1.00 ];
b = [0; 0; 0; 1];
pi = linsolve(M,b);
```

The solution vector is: $[\pi_{\texttt{Rainy}}, \pi_{\texttt{Cloudy}}, \pi_{\texttt{Sunny}}] = [0.7625, 0.1688, 0.0688]$ which (not surprisingly) corresponds to the solution for this model (using the Power method or VMM as explained earlier).

2.4.6 PRISM DTMC model

The PRISM model is straightforward, being sufficient to describe the model using the appropriated syntax, i.e.:

```
dtmc

const int N=3; // N - number of states

const int sRainy  = 0;
const int sCloudy = 1;
const int sSunny  = 2;

module Module1
n : [0..N] init sRainy; // total states, initial state sRainy
// transitions
[] (n=sRainy)  -> 0.80 : (n'=sRainy)  +
                  0.15 : (n'=sCloudy) +
                  0.05 : (n'=sSunny);
[] (n=sCloudy) -> 0.20 : (n'=sCloudy) +
                  0.70 : (n'=sRainy)  +
                  0.10 : (n'=sSunny);
[] (n=sSunny)  -> 0.20 : (n'=sSunny)  +
                  0.50 : (n'=sRainy)  +
                  0.30 : (n'=sCloudy);
endmodule
```

The model is located in `prism/belfast-dtmc.pm` and it should be run in PRISM Statistical Model Checker tool (it was tested on version 4.6).

After running the tool, go to `Open Model` (choose the file), then click on `Model`>`Compute`>`Steady-state probabilities`. The following results are shown to modellers:

```
Printing steady-state probabilities in plain text format below:
0:(0)=0.7624998618647714
1:(1)=0.16875010721113942
2:(2)=0.06875003092408917
```

Observe that it yielded the same probability vector as output as all other methods explored here so far.

2.5 Let's code!

The next programming challenge will take into account those aforementioned matrix and vector operations.

2.5.1 Challenge 04

We will implement a solution to work with the power method and the VMM.

File: `challenge04.c`

Do: write a program to work with the following methods:

1. Power matrix
2. Vector-Matrix Multiplication

Notes:

- We will need to use dynamic allocation (employing pointers) in C for the next tasks.

Good programming practice:

- When programming with pointers you need to exert caution and patience.
- Your code should not run indefinitely. You will have to code auxiliary functions to help you, for example, one that checks residuals between two arrays of floats. If the residuals are less than a predefined threshold you may stop computing the process.

Implement a program accepting the following parameters:

- <FILENAME> <OP>
  - <FILENAME>: a text based file with a DTMC (a squared matrix, each line sums 1.0) describing the model
  - OP (will trigger one of the following methods):
    - [OP=0] `void multiply0(float** m, int size);`
      - Returns the solution vector for a model of a model represented by m of order size;
    - [OP=1] `void multiply1(float* v, float** m, int size);`
      - Employs VMM and returns the solution vector v for a model in m, of a matrix of order size;
  - The usual tests (see Section 2.3.1) apply and at the end, it prints the full resulting matrix (for $OP = 0$), the resulting vector (for $OP = 1$).

Suggested variations:

- Output the current result vector per iteration on the screen (according to an option passed in command-line);

2.5.2 Challenge 05

This challenge will implement a basic forward DTMC simulator.

File: `challenge05.c`

Do: write a program that runs a simple DTMC simulator, where given a model, a visitor jumps from state to state during a pre-defined number of steps.

Notes:

- Implement a function `void dtmc_simulator(float** m, int size, int samples, int* results);` that simulates a DTMC (the function returns the solution vector π in `results`);
- The results should approximate to the analytical solution as observed in previous challenge (Section 2.5.1).

Implement a program accepting the following parameters:

- `<FILENAME> <SAMPLES>`

Implementation tips:

- Not necessarily you will need to code the accumulating matrix. You could just use an accumulating variable each time you traverse the line of the matrix under consideration (this will be explored in the solution for this challenge).

Suggested variations:

- Output the top N states (passed in command line) with the highest visits (so far) as the method is running);

2.6 Comments on the spreadsheets

The spreadsheets for DTMC are:

- The Lighting model: `spreadsheets/Chapter02-DTMC-2states.xlsx`;
- The Belfast model: `spreadsheets/Chapter02-DTMC-3states.xlsx`;

Each file has three tabs: 1. Power-Matrix method; 2. Vector-Matrix Multiplication; and 3. DTMC simulator.

The first method multiplies the model's matrix to the n^{th} power (in this case, to M^5), whereas the VMM method iteratively multiplies a start vector

(summing to one) by the model's matrix until the last two resulting vectors are the same between two iterations (in this case, it ran for nine iterations).

Finally, the DTMC simulator will run a simulation using pseudo-random numbers (uniformly distributed) mimicking the idea of visiting states one after the other for a fixed number of times (in the spreadsheet, it is set to run until it produces 1,000 state samples).

After it runs, it computes the resulting probabilities. In terms of results, one notices that they are quite close to the analytical ones (computed previously), however, a larger number of samples would produce even closer results.

Chapter 3

CTMC

In terms of modelling, one could wonder about the expressiveness of CTMC over DTMC. The idea is that one could think about state permanence, or durations, instead of probabilities. CTMC are different than DTMC when modelling situations, because one thinks on the time spent *in* the state (residence or sojourn time) and how it transitions to other states, using `rate` $= \frac{1}{duration}$ to represent the rate *out* of the state.

Figure 3.1 shows how the balancing process works for each state: the negative sum of outgoing rates is used to even out the rates, so each line in the model will sum to zero.

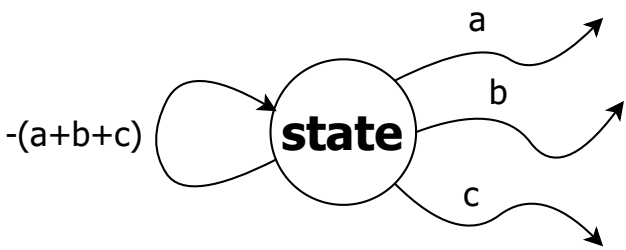


Figure 3.1: Representation of balancing rates in states in CTMC.

Here, this is not a MC *per se*, but only a representation of how and why one does not consider self-loop transitions. The side-effect of this is that, in CTMC, the diagonal is not used in the modelling process, because it is destroyed to accommodate the sum of exits and balance out the rates in and out the state.

3.1 Infinitesimal Generator

The process to model CTMC differs from DTMC in the sense that we must work with a special kind of matrix called the Infinitesimal Generator (IG). It

consists on destroying the diagonal (self-loops in the model) and substituting it with the negative sum of rates of this state, balancing *rates in* and *rates out*.

Let's revisit the Lighting model once again in Figure 3.2:

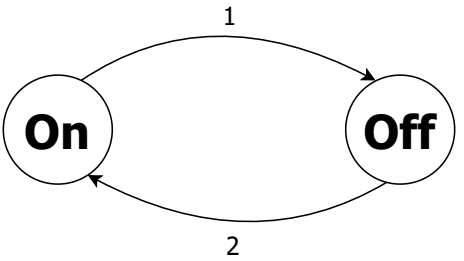


Figure 3.2: Lighting model with two states as a CTMC.

The IG (referred as matrix Q) for this model is[1]:

$$\mathtt{Q} = \begin{pmatrix} -1 & 1 \\ 2 & -2 \end{pmatrix}$$

The same process is repeated for higher order matrices, i.e., a Generic model with four states (Figure 3.3):

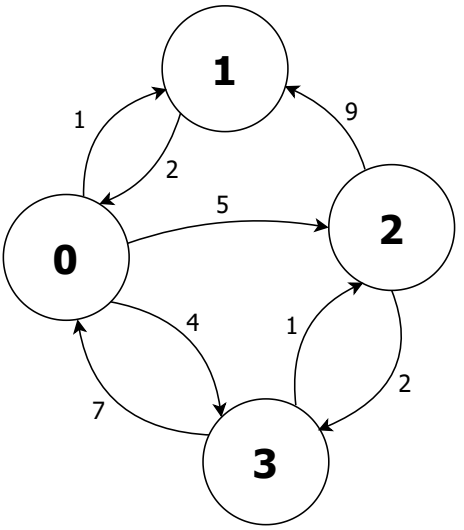


Figure 3.3: Generic CTMC model with four states.

[1]It contrasts with matrix M in DTMC, where each line sums 1.0, whereas Q sums 0.0.

The corresponding IG for this model is:

$$\mathtt{Q} = \begin{pmatrix} -10 & 1 & 5 & 4 \\ 2 & -2 & 0 & 0 \\ 0 & 9 & -11 & 2 \\ 7 & 0 & 1 & -8 \end{pmatrix}$$

Note that both models respect previously mentioned properties to be MC, i.e., they are ergodic (Section 1.9).

3.2 Computing the embedded DTMC

Given an IG, one could convert the CTMC into its embedded DTMC. The process is as follows.

1. Let Q be an Infinitesimal Generator of a MC model;
2. Let I^n be an Identity Matrix of order n;
3. Find $\mathtt{max}_Q$, i.e., the maximum value of Q_{ij}:
   - Because of the way the IG is built, this value will be *always* located in the diagonal.
   - Find the highest value in absolute terms and then use it with its signal – more about this in a moment.
4. For each cell of Q, i.e., Q_{ij}, compute $M'_{ij} = I_{ij} - \frac{Q_{ij}}{\mathtt{max}_Q}$;

Let's examine this example using the Generic model presented in previous section and use an Identity Matrix of order 4 (I^4) to derive the embedded DTMC. For this case, $\mathtt{max}_Q = -11$.

$$\overbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{I^4}, \overbrace{\begin{pmatrix} -10 & 1 & 5 & 4 \\ 2 & -2 & 0 & 0 \\ 0 & 9 & -11 & 2 \\ 7 & 0 & 1 & -8 \end{pmatrix}}^{Q}, \overbrace{\begin{pmatrix} 0.09 & 0.09 & 0.45 & 0.36 \\ 0.18 & 0.82 & 0.00 & 0.00 \\ 0.00 & 0.82 & 0.00 & 0.18 \\ 0.64 & 0.00 & 0.09 & 0.27 \end{pmatrix}}^{M'}$$

We created a new version of the model and called it M', which is the embedded DTMC of Q. Let's perform the required computation for the first cell. $M'_{00} = \frac{\mathtt{I}_{00}}{\mathtt{max}_Q} = 1 - (\frac{-10}{-11}) = 0.090909091$ (the matrix shows a less accurate number for presentation purposes). The process is subsequently repeated for *all* cells of Q.

For reference, these are all intermediary computations for reaching M':

$$M' = \begin{array}{|c|c|c|c|} \hline \mathbf{1} - (\frac{\mathbf{-10}}{\mathbf{-11}}) & 0 - (\frac{1}{-11}) & 0 - (\frac{5}{-11}) & 0 - (\frac{4}{-11}) \\ \hline 0 - (\frac{2}{-11}) & \mathbf{1} - (\frac{\mathbf{-2}}{\mathbf{-11}}) & 0 & 0 \\ \hline 0 & 0 - (\frac{9}{-11}) & \mathbf{1} - (\frac{\mathbf{-11}}{\mathbf{-11}}) & 0 - (\frac{2}{-11}) \\ \hline 0 - (\frac{7}{-11}) & 0 & 0 - (\frac{1}{-11}) & \mathbf{1} - (\frac{\mathbf{-8}}{\mathbf{-11}}) \\ \hline \end{array}$$

So, this is the embedded DTMC of the CTMC, the discrete time version of it (where each line sums to one). One may use any of those previously discussed solution methods available for DTMC to unveil the probability vector π.

Take some time to look at the resulting DTMC. Observe that now there are some self-loop transitions (from 0 to 0, from 1 to 1 and from 3 to 3). If one was modelling from the onset using DTMC, one should take those transitions into account. That is why one might think that employing CTMC for modelling is sometimes best to represent complex behaviour.

3.3 The Lily Pad model

Let's consider the "Lily Pad model" now: it consists of a frog hopping lily pads in a fictitious pond. The idea is that it jumps around mindlessly, staying some time on each lily pad and then moving on to the next.

Figure 3.4 shows a pond with four lily pads and transitions among them.

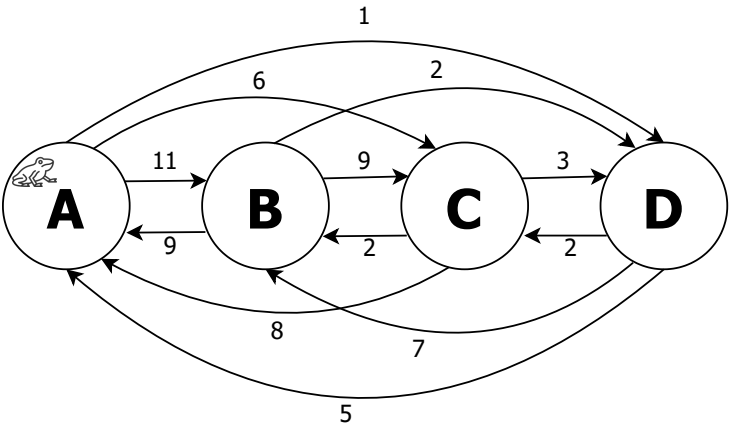


Figure 3.4: Lily Pad model showing possible pads and transitioning rates.

Consider Table 3.1 for thinking about the rates decorating transitions in the model. Note that for modelling time, the observer has attended the pond and remained collecting frog residence times over a period of time, to have a nice idea of jumps and durations.

Table 3.1: Residence time observed in lily pads and corresponding rates.

Time (min)	From	To	Rate
5.4	A	B	11
10		C	6
60		D	1
6.7	B	A	9
6.7		C	9
30		D	2
7.5	C	A	8
30		B	2
20		D	3
12	D	A	5
8.6		B	7
30		C	2

One might think that, when modelling, it is easier to consider states and 'entities' changing states where each has remained some time in the state before transitioning to other state. The reasoning is that sometimes, time is the only observable thing to model the system. Recall that for modelling DTMC one requires the probabilities for switching states. Because of the formula to convert durations to rates, lower time spent in the state corresponds to higher rates out of it, as seen in the table. For example, from pad A to pad B with a duration of 5.4 min, the converted rate was equal to 11, whereas from pad A to pad D, the frog stayed for 60 min and the rate out in one hour was one.

The IG (Q) for this model ($\texttt{max}_Q = -20$) and its embedded DTMC M' is:

$$\texttt{Q} = \begin{pmatrix} -18 & 11 & 6 & 1 \\ 9 & -20 & 9 & 2 \\ 8 & 2 & -13 & 3 \\ 5 & 7 & 2 & -14 \end{pmatrix} \qquad \texttt{M'} = \begin{pmatrix} 0.10 & 0.55 & 0.30 & 0.05 \\ 0.45 & 0.00 & 0.45 & 0.10 \\ 0.40 & 0.10 & 0.35 & 0.15 \\ 0.25 & 0.35 & 0.10 & 0.30 \end{pmatrix}$$

As previously mentioned, with the embedded DTMC M', one could employ Power matrix, VMM, simulation, or direct methods to discover π. Using the VMM, we reach the following results (starting with vector `[ 1 0 0 0 ]`):

$$\pi = \begin{pmatrix} 0.302449696 & 0.243404608 & 0.327575347 & 0.12657035 \end{pmatrix}$$

We come to the conclusion that the frog has a preference for lily pads A, B, and C ($1 - 0.12657035 = 0.87342965$, $\approx$87% of the time), given the probabilities that we have computed. It is worth mentioning that those are the same results if one employed the Power matrix method using the embedded DTMC.

3.4 Let's code!

Let's put those ideas into practice now.

3.4.1 Challenge 06

This challenge 'fix' the diagonal of a CTMC and then convert it to its embedded DTMC representation.

File: `challenge06.c`

Do: write a program that will take a CTMC as parameter and then compute the main diagonal accordingly and convert it to its embedded DTMC.

Notes:

- Implement a function `void fix_ctmc(float** q, int size);` that will destroy the current main diagonal and replace it with the sum of line accordingly
- Implement a function `float** convert2dtmc(float** q, int size);` that creates another matrix in memory and returns the DTMC of the CTMC matrix passed by parameter (variable q)
  - Remember to use previous function `fix_ctmc` because it has the 'corrected' value for the diagonal
- Remember that you will have to code auxiliary functions to help you, such as `float discover_max(float** q, int size);`
- You will also need to consider the use of *identity matrix* (without necessarily creating it in memory
  - The idea is that if you are traversing Q, then if you are on the diagonal position (when `i==j`), then you proceed with the proper computation (as discussed earlier), otherwise consider it as zero

Suggested variations:

- Remember to use previously implementations and solution mechanisms for DTMC (earlier challenges).
- Offer users a way of outputting the newly generated DTMC in a file (that you pass both the option and the name of this file in the command-line).

3.5 Solution methods for CTMC

3.5.1 Forward simulation

In the DTMC case, one would choose an initial state, and according to a model, visit states using the uniform distribution. The process ends when enough visits reached a pre-defined value, and then modellers would compute state statistics on visits.

For CTMC we would like to pursue more or less the same venue, however, now, we would like to generate pseudo-random numbers from the exponential distribution. The idea of visiting states remains, also for running for a large number of samples and then calculating state probabilities.

Before we start discussing the simulation per se, we would like to state the process and how it would work, for example:

1. Given a model written in CTMC, and assuming that the diagonal contains the sum of the rest of the elements (negative value);

2. The model fills a matrix Q, that is a valid (well-written) Infinitesimal Generator;

3. Now, we will compute a new matrix called Q': for each cell Q_{ij}, and employing an Identity Matrix of same order (I^4), and then we will compute $Q'_{ij} = Q_{ij} - \texttt{MAX}_Q \times I_{ij}$;

4. From an initial state S_0, draw a number from the exponential distribution (considering the rates in Q) and then visit the next state, accumulating all these visits in a counting vector (to present to modellers afterwards);

    - The formula to generate an exponential distributed variable using a pseudo-random number generated using the uniform distributions is $\texttt{Exp(rate)} = (-\frac{1}{\texttt{rate}}) \times log(1 - \texttt{U}(0;1), e)$, where $\texttt{U}(0,1)$ is a pseudo-random number between 0 and 1 and log is the natural logarithm of a number (base e);

Recall the Q matrix for the Generic model using CTMC (Section 3.2):

$$\texttt{Q} = \begin{pmatrix} -10 & 1 & 5 & 4 \\ 2 & -2 & 0 & 0 \\ 0 & 9 & -11 & 2 \\ 7 & 0 & 1 & -8 \end{pmatrix}$$

For reference, these are all intermediary computations for computing Q' ($\texttt{MAX}_Q = -11$):

$\mathbf{-10} - (\mathbf{-11} \times \mathbf{1})$	$1 - (-11 \times 0)$	$5 - (-11 \times 0)$	$4 - (-11 \times 0)$
2	$\mathbf{-2} - (\mathbf{-11} \times \mathbf{1})$	0	0
0	$9 - (-11 \times 0)$	$\mathbf{-11} - (\mathbf{-11} \times \mathbf{1})$	$2 - (-11 \times 0)$
$7 - (-11 \times 0)$	0	$1 - (-11 \times 0)$	$\mathbf{-8} - (\mathbf{-11} \times \mathbf{1})$

The new matrix Q' is equal to (observe that the only change is in the main diagonal):

$$\texttt{Q} = \begin{pmatrix} -10 & 1 & 5 & 4 \\ 2 & -2 & 0 & 0 \\ 0 & 9 & -11 & 2 \\ 7 & 0 & 1 & -8 \end{pmatrix} \qquad \texttt{Q'} = \begin{pmatrix} 1 & 1 & 5 & 4 \\ 2 & 9 & 0 & 0 \\ 0 & 9 & 0 & 2 \\ 7 & 0 & 1 & 3 \end{pmatrix}$$

If you simulate this matrix, you will reach results that are approximations of the other solution methods for this model.

3.6 Race condition

When modelling CTMC and looking at individual states, one might ponder: if one has to choose a state to visit, given the rates, which one to choose? This is important in simulation, because visiting a state one cannot look at the past (after all this is a Markovian process) to decide which state to visit next.

Look at Figure 3.5 next, given that you are in a state with those transitions, which one you choose to trigger next?

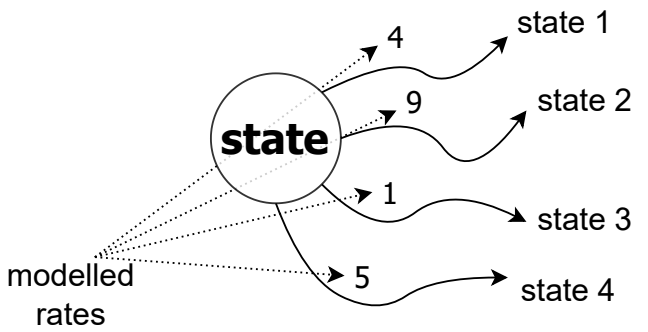
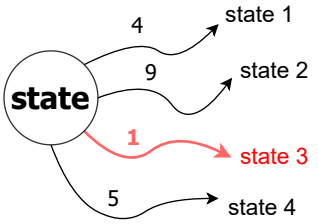


Figure 3.5: Hypothetical situation showing the problem of choosing the next state in CTMC simulation.

The answer is: the fastest. As we have discussed earlier, it does not matter the highest rate when you use the exponential distribution. Only after you draw a number from the distribution you know the parameter to use to decide *"which transition will win the 'race' and become the next state visited"*.

Important Remember that Figure 3.5 does not show previous transitions for reaching the state: recall that in MC the past is *irrelevant*, what only matters is where to 'jump' next.

In a CTMC simulation you are working with time; so this means that you will draw pseudo-random numbers using each state rates and then you will choose the one that happened first in time. Figure 3.6 details this process, and also shows the winning transition for this particular case.



Rate	Uniform	Exponential	Rank
4	0.3024550	0.0900470	2
9	0.8295417	0.1965850	4
1	0.0528990	0.0543496	**1**
5	0.6072475	0.1869151	3

$$\texttt{Exp(rate)} = \left(\frac{-1}{rate}\right) \times log(1 - \texttt{Unif(0;1)}, e)$$

Figure 3.6: Drawing pseudo-numbers from uniform and exponential distributions to help simulate time in CTMC (`log` is the natural logarithm, in base e).

So, given the pseudo-random numbers, and after establishing a rank (to simulate the lowest time), one decides that `state 3` will be the next state visited[2]. This is the core of the CTMC simulator, visiting states, and depending on the rates on each state, deciding where to go next, simulating time. Since you are simulating, you could choose how long would you like to simulate in terms of time, e.g., 10 hours.

3.7 Comments on the spreadsheets

This are the MS-Excel files detailing the solution possibilities for this model:

- Generic model: `spreadsheets/Chapter03-CTMC-4states-generic.xlsx`
- Lily Pad model: `spreadsheets/Chapter03-CTMC-4states-lily.xlsx`

[2]Pure chance has selected the minimum rate in this example – be mindful of this. I suggest you open a blank spreadsheet to test this.

3.8 Let's code!

For this challenge we will build a CTMC simulator.

3.8.1 Challenge 07

File: `challenge07.c`

Do: write a program that will take a CTMC as parameter and then simulates it for some time (passed as parameter in the command line).

Implement a program accepting the following parameters:

- <FILENAME> <TIME>

Notes:

- Implement a function `float** convert2ctmc_prime(float** q, int size);` that converts a matrix Q by another (Q') using the method discussed in Section 3.5.1;
- Use the function that generates a pseudo-number from the exponential distribution that you have coded in previous challenge (Challenge 02 in Section 1.8.2) – the function needs the `rate` parameter;
  - Function prototype: `float exponential(float n);`
- You will need to implement a function to find out the next state to visit. The idea is to draw N exponentially distributed numbers (a line of the current state) and then select the one that has the least value other than zero, so code the following function:
  - `int lowest_not_zero(float* qline, int size, float* sim_time);`
    - It returns the next state according to values drawn
    - You must pass the time in order to conveniently update it (since the function returns the next state);
- Implement a function `int ctmc_simulator(float** qprime, int size, float sim_time, int* results);` that simulates a CTMC (the function returns the solution vector π) for some time, i.e., `sim_time`;
  - The function returns the number of samples produced for simulated that time amount (units of time);

Suggested variations:

- Make the residue either small enough (configured in command line) or after sampling a number of samples (as before), whichever event comes early. Show each position's residue after every $1,000$ samples computation;
- Build a program that computes series of simulation 'batches' consisting of N samples each (for instance, 10 runs of 1000 samples), then present to the modeller basic statistics over the batches, e.g., averages, standard deviation, boxplot output (1^{st} quarter, 2^{nd}, 3^{rd}, 4^{th}). Look at Jain's book [14] for multiple batch runs;
- Implement transitions having other probability distributions (you are running a simulation after all);

There is a CTMC simulator written for MS-Excel in the following files (on Tab CTMC simulator):

- Lighting model: `spreadsheets/Chapter03-CTMC-2states.xlsx`
- Generic model: `spreadsheets/Chapter03-CTMC-4states-generic.xlsx`
- Lily Pad model: `spreadsheets/Chapter03-CTMC-4states-lily.xlsx`

All methods yield the same results, for each model.

3.9 Solution methods (cont.)

Now let's continue working with other solution methods for CTMC, more precisely, the direct solution method and then some PRISM models.

3.9.1 Direct solution method

For direct solution of CTMC one must satisfy $\pi Q = 0$. Let's recall the Generic model (CTMC) and state transitions and add the required operations accordingly:

$$\begin{bmatrix}\pi_1 & \pi_2 & \pi_3 & \pi_4\end{bmatrix} \times \begin{pmatrix} -10 & 1 & 5 & 4 \\ 2 & -2 & 0 & 0 \\ 0 & 9 & -11 & 2 \\ 7 & 0 & 1 & -8 \end{pmatrix} = \begin{bmatrix}0\\0\\0\\0\end{bmatrix}$$

After multiplication, one has the following system of equations (plus the one for ensuring that all π will sum to 1.0, i.e., $\pi_1 + \pi_2 + \pi_3 + \pi_4 = 1$):

$$\begin{cases} -10\pi_1 + 2\pi_2 + 7\pi_4 &= 0 \\ \pi_1 - 2\pi_2 + 9\pi_3 &= 0 \\ 5\pi_1 - 11\pi_3 + \pi_4 &= 0 \\ 4\pi_1 + 2\pi_3 - 8\pi_4 &= 0 \\ \pi_1 + \pi_2 + \pi_3 + \pi_4 &= 1 \end{cases}$$

For this model one could tackle by defining $\pi_1 = 1$ and then doing algebra to discover π_3 and π_4 (on the third and fourth equations), then circling back to π_2 (second equation). Then, the next step would be to uniformise the results, where all probabilities sum to 1.0.

We submitted this system of linear equations to MATLAB® (refer to file `matlabgeneric_model_dtmc.m` in the GitHub repository).

```
% MATLAB Model
Q = [-10, 2, 0, 7;
1, -2, 9, 0;
5, 0, -11, 1;
4, 0, 2, -8;
1, 1, 1, 1];

% auxiliary vector with one position set as 1
b = [0; 0; 0; 0; 1];

pi = linsolve(Q,b);
```

Again, the computed vector π is equivalent to the other solutions.

3.9.2 PRISM CTMC models

We present next the Lily Pad model and the Generic model written in CTMC for PRISM. We start with the Lily Pad model (file `prism/lily-ctmc.sm`):

```
// Lily Pad model (frog in the pond)
ctmc

const int N=4; // N - number of states

const int sA=0;
const int sB=1;
const int sC=2;
const int sD=3;

module Module1
n : [0..N] init sA; // total states, initial state sA
```

```
// transitions
[] (n=sA) -> 11 : (n'=sB) + 6 : (n'=sC) + 1 : (n'=sD);
[] (n=sB) ->  2 : (n'=sD) + 9 : (n'=sC) + 9 : (n'=sA);
[] (n=sC) ->  2 : (n'=sB) + 8 : (n'=sA) + 3 : (n'=sD);
[] (n=sD) ->  5 : (n'=sA) + 7 : (n'=sB) + 2 : (n'=sC);
endmodule

------------------------
Printing steady-state probabilities in plain text format below:
0:(0)=0.30244974694693344
1:(1)=0.24340456372498753
2:(2)=0.32757532070253004
3:(3)=0.12657036862554888
```

And here is the Generic model (file `prism/generic-ctmc.sm`):

```
// Generic model
ctmc

const int N=4; // N - number of states

const int s0=0;
const int s1=1;
const int s2=2;
const int s3=3;

module Module1
n : [0..N] init s0; // total states, initial state s0
// transitions
[] (n=s0) -> 1 : (n'=s1) + 5 : (n'=s2) + 4 : (n'=s3);
[] (n=s1) -> 2 : (n'=s0);
[] (n=s2) -> 9 : (n'=s1) + 2 : (n'=s3);
[] (n=s3) -> 7 : (n'=s0) + 1 : (n'=s2);
endmodule

------------------------
Printing steady-state probabilities in plain text format below:
0:(0)=0.20235285239955195
1:(1)=0.567058916201716
2:(2)=0.1035294076461007
3:(3)=0.1270588237526314
```

Chapter 4

More projects and models

In this chapter I will comment about other projects and some models based on MC that are worth looking at.

4.1 Projects

The code present here and in the repository is far from bug-free. As a starting project, you could improve the code, and also translating it to C++, adding more controls, working with increased number of states, and so on[1].

4.1.1 Combine it all!

The idea here is to combine all pieces of code into one C project, putting functions in a different file for proper organisation. In the repository you may find a folder called `project01`, with a full project with all challenges explored here (combined in one place).

The 'services' provided by the project are the following command-line options (the executable is `markov`):

```
printf("This is the Markov chain solver [DD/MM/YYYY].\n");
printf("To run DTMC: markov <FILE> <OP>\n");
printf(" where <OP> is one out of these options:\n");
printf("   0 Solve DTMC - Power Matrix\n");
printf("   1 Solve DTMC - Vector-Matrix Multiplication\n");
printf("   2 Simulate DTMC (pass <SAMPLES> parameter!)\n");
printf("   3 Convert CTMC to Embedded DTMC\n");
printf("   4 Simulate CTMC (pass <TIME> parameter!)\n");
```

For `OP=2`, the user must also inform the number of samples at some point and for `OP=4`, the simulation time (in time units).

[1] Please consider doing this and then a subsequent Pull Request on the GitHub repository.

Suggestion: before looking at the solution, take all challenges and try to come up with your own solution.

4.1.2 Visual MC

Find a way of representing MC visually, using an auxiliary library or API like `graphviz` or some other.

4.1.3 Parallelisation of simulation samples

Vendors ship modern CPUs with more than one core. The idea of this project is to explore parallelism in simulations by distributing the computation of samples among multiple processing core and then assembling results altogether, showing to the modeller. This approach would make a lot of sense in huge models.

4.2 Models

4.2.1 Birth and Death model

Figure 4.1 shows a 4 state Birth and Death process in CTMC with symbolic rates λ (for arrivals, i.e., births) and μ (for departures, or deaths).

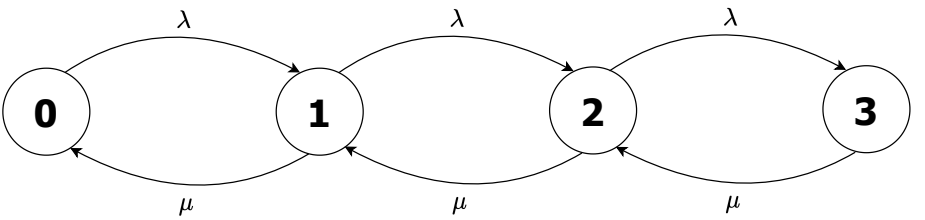


Figure 4.1: A four state Birth and Death CTMC model.

The Q matrix for this model follows a distinctive pattern:

$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} \end{array} \\ \begin{pmatrix} -\lambda & \lambda & 0 & 0 \\ \mu & -(\mu+\lambda) & \lambda & 0 \\ 0 & \mu & -(\mu+\lambda) & \lambda \\ 0 & 0 & \mu & -\mu \end{pmatrix} \end{array}$$

There are known equations that work with this type of models (closed form) [15, 14]. And also check out the model `S6-birth-and-death-ctmc.txt` in the GitHub repository for a 6 state Birth and Death process.

4.2.2 Mouse Maze model

This model was extracted from the book "Introduction to Probability" (page 440) [16]. The model employed DTMC and it has 9 states, mapping the situation explained in Figure 4.2.

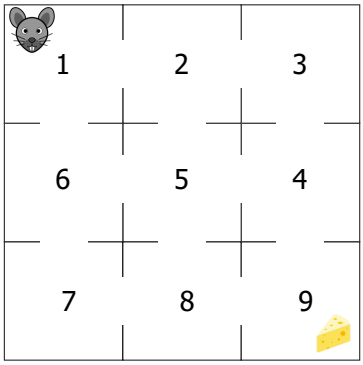


Figure 4.2: A mouse in a maze mindlessly visiting cells.

The model, as follows, shows the probabilities for reaching adjacent cells in the maze.

	1	2	3	4	5	6	7	8	9
1	0	0.50	0	0	0	0.50	0	0	0
2	0.33	0	0.33	0	0.33	0	0	0	0
3	0	0.50	0	0.50	0	0	0	0	0
4	0	0	0.33	0	0.33	0	0	0	0.33
5	0	0.25	0	0.25	0	0.25	0	0.25	0
6	0.33	0	0	0	0.33	0	0.33	0	0
7	0	0	0	0	0	0.50	0	0.50	0
8	0	0	0	0	0.33	0	0.33	0	0.33
9	0	0	0	0.50	0	0	0	0.50	0

The transition probabilities depend on the number of doors in the cell. For example, cells 1, 3, 7, and 9 have two doors, so the probability is 0.50 whereas cells 2, 4, 6, and 8 have three doors each (probability of 0.33) and finally, cell 5 has 4 doors (probability of 0.25). This model will not yield an analytical solution, however, via simulation, it is possible to reach satisfactory results.

The mouse in the maze model is in GitHub's repository:

- File: `S9-maze-model-dtmc.txt`.

4.2.3 Availability model

Next model was discussed by Trivedi and Bobbio [3] (Example 9.2, page 311), about a blade server in a data centre. They investigated the availability (and

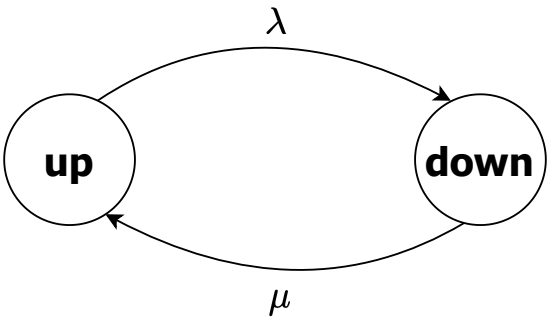


Figure 4.3: Availability modelling of a blade server in a data centre.

conversely, the unavailability) using CTMC. The model is depicted in Figure 4.3.

For this case, the IG of the CTMC is:

$$\begin{array}{c} \\ \textbf{up} \\ \textbf{down} \end{array} \begin{array}{c} \begin{array}{cc} \textbf{up} & \textbf{down} \end{array} \\ \begin{pmatrix} -\lambda & \lambda \\ \mu & -\mu \end{pmatrix} \end{array}$$

The authors use special numerical methods to solve this CTMC (e.g., Laplace transform method – they explain *why* they do it in the book) and then discuss the following equations for assessing availability:

$$\begin{cases} \pi_{up} &= \dfrac{\mu}{\lambda + \mu} \\ \pi_{down} &= \dfrac{\lambda}{\lambda + \mu} \end{cases}$$

The instantaneous system *Availability* is `A(t)` $= \pi_{up}$ and correspondingly, the instantaneous system *Unavailability* is `U(t)` $= 1 -$ `A(t)` $= \pi_{down}$.

4.2.4 Aging and Rejuvenation model

Next model was discussed by Huang et al. (1995) [1], about software aging and rejuvenation [17]. The basic idea behind these concepts is to employ preemptive mechanism to restart applications to healthy states to prevent failures in the future. The authors have come up with a CTMC model depicted in Figure 4.4.

In the model, S_0 is the robust state of an application where it stays for a period equivalent to its base longevity interval, then it transitions to a failure probable state S_P. From this state (S_P), it goes to a failure state S_F where it remains until it is repaired. If only these states (S_0, S_P, and S_F) were considered, the model would correspond to a normal process without any rejuvenation.

The model as described in [1], uses the following assumptions:

- $\lambda = \frac{1}{12*30*24} = 0.000115741 \rightarrow$, i.e., the Mean Time Between Failure (MTBF) is 12 months

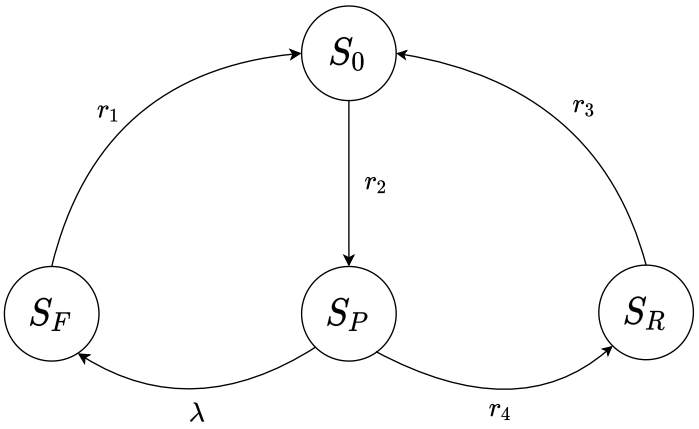


Figure 4.4: Software aging and rejuvenation model [1].

- $r_1 = 2 \rightarrow$ it takes 7 days to recover from an unexpected failure
- $r_2 = \frac{1}{7*24} = 0.005952381 \rightarrow$ base longevity interval (from S_0 to S_P), equal to 7 days
- $r_3 = 3 \rightarrow$ the mean repair time after rejuvenation is 20 min
- $r_4 = \frac{1}{(14-7)*24} = 0.005952381 \rightarrow$ rejuvenation frequency for an application

$$
\begin{array}{c} \\ S_0 \\ S_F \\ S_P \\ S_R \end{array}
\begin{array}{c}
\begin{array}{cccc} S_0 & S_F & S_P & S_R \end{array} \\
\left(\begin{array}{cccc}
0 & 0 & 0.005952381 & 0 \\
2 & 0 & 0 & 0 \\
0 & 0.000115741 & 0 & 0.005952381 \\
3 & 0 & 0 & 0
\end{array} \right)
\end{array}
$$

The insertion of an intermediary state S_R that models a preemptive action from the probable failure state to replenish the application is inserted, as shown in the model. After this process is concluded, the application returns to the stable (robust) state. The rates for each state are discussed in the paper, the point the authors make is that rejuvenating applications costs less and helps applications stay (or return) in stable states more frequently. The seminal paper discusses how they modelled each rate and the reasoning behind it and other interesting notions about aging and rejuvenation.

It is worth pointing out that the model has only four states, modelling a CTMC that helps analysts consider different possibilities when designing or handling systems. Solving the model and having numerical outputs enormously help analysts understand behaviours and effectively cope with trade-offs.

Chapter 5

Final considerations

I sincerely hope you have had a good journey reading this book, and your knowledge about MC has improved. Not only that but now you can address hard problems using MC as another tool in your toolbox. Don't think that because you have a good fancy hammer (MC), every problem is a nail, and you must torture it until it fits everything discussed here. No, no, and no. You should be able to apply MC depending on the problem at hand, the data you have, and the circumstances (the context) surrounding your analysis.

The code present here (and in the repository) is far from perfect. I bet if you look deeper you may find a lot of defects and problems. It handles models with few number of states, having line sizes no greater than 1k (it does not even say that this is problem while trying to open). So, you could improve the code, translate it to C++ perhaps, add more controls, work with increased number of states, and so on. The sky is the limit – go for it!

I guess you noticed that I have tackled models with extremely reduced number of states, for educational purposes only. It is worth remarking that useful[1] real world modelling tends to require much larger order matrix to represent somewhat complex behaviours and relationships. I like to think that your *Markovian Journey* has just began.

5.1 What's next?

The next step will require further understanding of MC. For instance, one might:

- Investigate structured MC: Queueing Networks (QN) [18, 12], Stochastic Petri Nets (SPN), PEPA [19], or PTA/MDP;
- Understand and implement "Perfect Simulation" (also called *backwards* simulation) notions;

[1] Remember that quote from George Box, *"All models are wrong, some are useful."*

- Explore other related formalisms, such as Generalized Stochastic Petri Nets [20], Well Formed Networks, Layered Queueing Networks [21], Superposed Generalized SPNs (SGSPN) [22], and so on;

I sincerely think you should give more time to PRISM and its modelling formalisms. It is a well-established and stable tool that researchers around the world use in a daily basis. You could for example explore other tool features such as investigate (in depth) other formalisms (suitable for CTMC/DTMC, PTA, and MDP), generation of automatic charts with variables, simulation, transient analysis, synchronising events. The tool comes with several examples and has a decent User Guide with lots of useful information to start modelling.

Using PRISM over the years I can attest its robustness and usefulness, however I urge you to look into other tools. Mind that there is a wealth of tools to consider, for instance, the Modest Toolset (`https://www.modestchecker.net/`) or the Möbius Tool (`https://www.mobius.illinois.edu/`). And also, you may explore other quantitative analysis tools such as SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) [23] – `https://sharpe.pratt.duke.edu/`.

You could start learning more about *dependability* reading the landmark paper by Avizienis et al. (2004) [24], and then about reliability and availability modelling [3]. Finally, look at Jain's book on performance [14], the "Queueing networks and Markov chains" book [15], and Trivedi's "Reliability and Availability engineering" [3].

Your 'real' journey have just started!

About the author

Ricardo Melo Czekster is a curious individual based in Birmingham, United Kingdom (at least in 2022, time of writing this book). He promises he has no relation to Andrey A. Markov as his ancestors were from Vitebsk (Viciebsk) in Belarus. From what he has learned over time, Markov was from St. Petersburg and the distance between these two cities is 328 miles.

In 1906, two major events happened in the world history: first, Markov published his *magna opus* and second, avoiding impending wars, Ricardo's ancestors eventually settled down in Brazil in the southernmost region of Rio Grande do Sul, precisely in Santo Ângelo (distant 272 miles from the state capital, Porto Alegre).

He is a travelled and seasoned world visitor, having spent substantial time in Porto Alegre/Brazil (his home town), Grenoble/France, Edinburgh/Scotland, Princeton/USA, St Andrews/Scotland, Newcastle upon Tyne/England and now Birmingham/England.

Ricardo has been studying Markov Chains since 2004, whilst pursuing his PhD. Since then, he has done a lot of thinking and messing around this formalism, devising methods and solutions for tackling large state space models.

When he is not teaching or researching, he is doing some light manual woodworking, to try to forget about things for a while and engage in creating things out of wood and keep his fast-paced thought-stream in check.

Bibliography

[1] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Symposium on Fault-Tolerant Computing*. IEEE, 1995, pp. 381–390.

[2] O. Häggström, *Finite Markov chains and algorithmic applications*. Cambridge University Press, 2002, vol. 52.

[3] K. S. Trivedi and A. Bobbio, *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.

[4] A. L. Scherr, *An analysis of time-shared computer systems*. Massachusetts Institute of Technology, 1965, vol. 535.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[6] A. N. Langville and C. D. Meyer, *Google's PageRank and beyond*. Princeton University Press, 2011.

[7] P. Von Hilgers and A. N. Langville, "The five greatest applications of Markov chains," in *Proceedings of the Markov Anniversary meeting*. Citeseer, 2006, pp. 155–158.

[8] K. L. Chung, "Markov chains," *Springer-Verlag, New York*, 1967.

[9] J. R. Norris and J. R. Norris, *Markov chains*. Cambridge university press, 1998, no. 2.

[10] S. P. Meyn and R. L. Tweedie, *Markov chains and stochastic stability*. Springer Science & Business Media, 2012.

[11] W. J. Stewart, *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[12] ——, *Probability, Markov chains, queues, and simulation*. Princeton University Press, 2009.

[13] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *International conference on computer aided verification*. Springer, 2011, pp. 585–591.

[14] R. Jain, *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

[15] G. Bolch, S. Greiner, H. De Meer, and K. S. Trivedi, *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[16] C. M. Grinstead and J. L. Snell, *Introduction to probability*. American Mathematical Soc., 1997.

[17] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A survey of software aging and rejuvenation studies," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, pp. 1–34, 2014.

[18] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 248–260, 1975.

[19] J. Hillston, *A Compositional Approach to Performance Modelling*, ser. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[20] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, "Modelling with generalized stochastic Petri nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 2, p. 2, 1998.

[21] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2008.

[22] S. Donatelli, "Superposed generalized stochastic Petri nets: definition and efficient solution," in *International Conference on Application and Theory of Petri Nets*. Springer, 1994, pp. 258–277.

[23] R. A. Sahner, K. Trivedi, and A. Puliafito, *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*. Springer Science & Business Media, 2012.

[24] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

Index

Yet another venture by *the lazy panda collection*
URL: `https://pixabay.com/photos/mammal-wildlife-animal-zoo-panda-3074618/`

Markov Chains for programmers

Why should you read this book?

- It is directed at programmers (at any level) interested in Markovian numerical methods.
- It addresses simple Markov Chains and explains the numerical methods behind the formalism.
- It explains simple models and discusses trade-offs about computation effort concerning solutions.