Analysis Algorithms

for

Stochastic Models

Computer Science 512 Lecture Notes

Spring 2017

Andrew Miner

Last updated: May 9, 2017

Table of Contents

Part I

Core topics

Chapter 1

Introduction

1.1 Model taxonomy

We assume a "model" is a collection of "state variables" which are modified by "actions". We can characterize the following types of models:

Deterministic:

Actions are deterministic; no choices; nothing "random".

Examples:

- Execution of (single–threaded) code, with no inputs.
- Turing machines with fixed input.

Typical questions:

- Can a certain (type of) state be reached?
- Can a certain action or actions occur?
- E.g., "Will the Turing machine halt?"

Non-deterministic:

There may be arbitrary choices between actions. Can be used to model an unknown "environment" or user interactions.

Examples:

- Execution of (multi–threaded) code, with no inputs. (arbitrary choice of thread interleavings)
- Execution of code with user interaction. (arbitrary choice of user keystrokes)
- Games of strategy, e.g., chess or checkers. (arbitrary choice for each player's move)
- Puzzles, e.g., Rubik's cube (arbitrary choice for which face to turn)

Typical questions:

- Is there a sequence of choices that will cause a certain (type of) state to be reached?
- Will a certain (type of) state be reached, or certain actions occur, *regardless of choices*?

- E.g., “will the program halt regardless of user input”
- E.g., “will 2 threads modify a file simultaneously (regardless of thread interleaving)”
- E.g., “if a thread wants to enter critical section, will it eventually be able to do so?”
- Can a certain state be reached, regardless of choices *for some subset of actions*?
- E.g., “can I win regardless of the other player’s choice of moves?”

Stochastic (and deterministic):

Choices between actions are resolved by “random choice” or “random time” to execute actions.

Examples:

- Random inputs to deterministic algorithms
- Randomized algorithms
- Games of chance with no strategy. (Roulette, craps)
- Failure / repair models: failure times are “random”, repair times are “random”

Typical questions:

- Probability of reaching a certain state “eventually”
- E.g., “what is the probability to win craps?”
- Probability of reaching a certain state by a given time
- E.g., “what is the probability of a deadlock before 100 hours?”
- Expected time to reach a state, execute an action
- E.g., “what is the expected time of system failure?”
- Other performance queries
- E.g., “what is the average number of requests processed per second?”

Stochastic and nondeterministic:

The model contains both “random choices” and “nondeterministic choices”.

Examples:

- Games of chance with strategy. (Backgammon, Parcheesi, Monopoly)

Typical questions:

- Probability or performance, based on a given strategy.
- E.g., “If both players make optimal choices, what is the probability to win?”
- Find a strategy to optimize a performance measure.
- E.g., “what strategy will give me the best chance to win?”

We will spend most of our time discussing “nondeterministic” and “stochastic” models, but not mixed. Note that “deterministic” is a special case of both: deterministic is equivalent to “nondeterministic where there is always at most one choice” and to “stochastic where random choices are not really random”.

Also, we will focus on models with

- Discrete states (e.g., all state variables are integers)
- Actions occur at discrete time instances (rather than continuously). Continuous actions (e.g., rocket flight, aerodynamics, weather) typically require differential equations.

1.2 Model construction and analysis

One way to study a system is to build a model and analyze it "from scratch" each time, e.g., following the algorithm given by Leemis and Park from Simulation class:

1. **Determine the goals and objectives of analysis**. Can be queries about a specific system (e.g., want to know mean time until failure, is a deadlock possible). Can be design questions (e.g., how many redundant components are needed to keep the expected time until failure above 1000 hours).

2. **Build a conceptual model**. Typically an informal diagram. Model must be detailed enough to meet objectives in first step. But, too much detail leads to unnecessary complexity. Need to determine the important state variables.

3. **Build a specification model**. Fill in all the details of the conceptual model (usually requires much more information). E.g., machine failure times are "random", what is the distribution? E.g., threads use a locking protocol to prevent simultaneous writes; what are the details of the protocol?

4. **Build a computational model**. I.e., write a computer program to analyze the specification model. In ComS 555, this was a simulation, based on empirical probability measurements and generating random variates. In this class: numerical analysis of the underlying stochastic process, and/or model checking.

5. **Verify**. Make sure the computational model matches the specification model. In other words, debug your program.

6. **Validate**. Is the computational / specification model an accurate representation of reality? (Can compare results against the real system if it exists.)

The above approach may require several iterations.

The most expensive steps are implementation and debugging. For this class, we will instead develop a general–purpose implementation that must be debugged only once. Thus, the program must be able to read a specification model. As such, we need:

A specification "language" for models. These may differ slightly based on the type of model (i.e., stochastic or nondeterministic). These are typically called "modeling formalisms".

A specification "language" for queries. We need to express the quantities of interest (e.g., "is a deadlock possible", "what is the probability of a deadlock before time 1000") in a formal way.

So, we want a tool that takes a model and queries as input, and gives the answers as output.

Chapter 2

Review of Propositional Logic

2.1 Syntax

Propositional logic consists of

- propositional constants

`tt`	(true)
`ff`	(false)

- propositional variables
- operators

$\neg$	:	negation
$\wedge$	:	conjunction (and)
$\vee$	:	disjunction (or)
$\rightarrow$	:	conditional (implies)
$\leftrightarrow$	:	biconditional (if and only if)

- complex formulas of the form

$$\texttt{f} ::= \text{constant} \mid \text{variable} \mid \neg\texttt{f} \mid \texttt{f} \wedge \texttt{f} \mid \texttt{f} \vee \texttt{f} \mid \texttt{f} \rightarrow \texttt{f} \mid \texttt{f} \leftrightarrow \texttt{f}$$

2.2 Semantics

Operators can be defined by specifying, for all possible operands, the result of the operator on those operands. *Truth tables* are a natural way to do this. Truth tables may also be used to prove properties about complex formulas (this is a form of "proof by exhaustive enumeration").

2.2.1 Negation

The negation operator inverts the truth value of its operand:

a	$\neg a$
`ff`	`tt`
`tt`	`ff`

The negation operator has highest precedence.

2.2.2 Conjunction

The conjunction of two formulas is true if and only if both of the formulas evaluate to true:

a	b	$a \wedge b$
ff	ff	ff
ff	tt	ff
tt	ff	ff
tt	tt	tt

Note that the conjunction operator commutes.

2.2.3 Disjunction

The disjunction of two formulas is true if and only if at least one of the formulas evaluates to true:

a	b	$a \vee b$
ff	ff	ff
ff	tt	tt
tt	ff	tt
tt	tt	tt

Note that the disjunction operator commutes.

2.2.4 Conditional

The conditional, or implication, operator is used to express statements of the form, "if A then B". The operator is defined as:

a	b	$a \rightarrow b$
ff	ff	tt
ff	tt	tt
tt	ff	ff
tt	tt	tt

Note that $a \rightarrow b$ evaluates to *true* when a evaluates to *false*. Also, note that this operator **does not** commute.

2.2.5 Biconditional

The biconditional operator is used to express equivalence, or "if and only if". The operator is defined as:

a	b	$a \leftrightarrow b$
ff	ff	tt
ff	tt	ff
tt	ff	ff
tt	tt	tt

Note that the biconditional operator commutes.

2.3 Manipulating formulas

It is possible to express a formula in several different ways. To check if two formulas are equivalent, one sure method is to generate the truth tables for the two formulas: the formulas are equivalent if and only if the truth tables are identical. In practice, this can be done only for formulas with a few variables.

Property 2.1

$$a \to b \quad \equiv \quad (a \wedge b) \vee \neg a \quad \equiv \quad \neg a \vee b$$

Proof: construct the truth table

a	b	$a \to b$	$(a \wedge b) \vee \neg a$	$\neg a \vee b$
ff	ff	tt	tt	tt
ff	tt	tt	tt	tt
tt	ff	ff	ff	ff
tt	tt	tt	tt	tt

Property 2.2

$$\neg(a \to b) \quad \equiv \quad a \wedge \neg b$$

Property 2.3

$$a \leftrightarrow b \quad \equiv \quad (a \wedge b) \vee (\neg a \wedge \neg b)$$

Proof: construct the truth table

a	b	$a \leftrightarrow b$	$(a \wedge b) \vee (\neg a \wedge \neg b)$
ff	ff	tt	tt
ff	tt	ff	ff
tt	ff	ff	ff
tt	tt	tt	tt

Formulas can be manipulated just like algebraic expressions, where operator $\wedge$ acts like multiplication, and operator $\vee$ acts like addition. However, there are several extra rules that may be used to simplify or manipulate logic formulas; these are listed below.

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\neg\neg x & \equiv & x & & & \\[1ex]
x \wedge \texttt{ff} & \equiv & \texttt{ff} & x \vee \texttt{ff} & \equiv & x \\
x \wedge \texttt{tt} & \equiv & x & x \vee \texttt{tt} & \equiv & \texttt{tt} \\[1ex]
x \wedge x & \equiv & x & x \vee x & \equiv & x \\
x \wedge \neg x & \equiv & \texttt{ff} & x \vee \neg x & \equiv & \texttt{tt} \\[1ex]
\neg(a \wedge b) & \equiv & \neg a \vee \neg b & a \wedge b & \equiv & \neg(\neg a \vee \neg b) \\
\neg(a \vee b) & \equiv & \neg a \wedge \neg b & a \vee b & \equiv & \neg(\neg a \wedge \neg b) \qquad \text{(De Morgan's Laws)}
\end{array}$$

All of these are simple to prove using truth tables.

Example 2.1

Prove $a \leftrightarrow b \;\equiv\; (a \rightarrow b) \wedge (b \rightarrow a)$.

Solution:

$$\begin{aligned}
(a \rightarrow b) \wedge (b \rightarrow a) \;&\equiv\; \Big((a \wedge b) \vee \neg a\Big) \wedge \Big((b \wedge a) \vee \neg b\Big) \\
&\equiv\; (a \wedge b) \wedge (b \wedge a) \;\;\vee\;\; (a \wedge b) \wedge \neg b \;\;\vee\;\; \neg a \wedge (b \wedge a) \;\;\vee\;\; \neg a \wedge \neg b \\
&\equiv\; (a \wedge b) \;\;\vee\;\; \texttt{ff} \;\;\vee\;\; \texttt{ff} \;\;\vee\;\; \neg a \wedge \neg b \\
&\equiv\; (a \wedge b) \vee (\neg a \wedge \neg b) \\
&\equiv\; a \leftrightarrow b \qquad \text{(from Property 2.3)}
\end{aligned}$$

2.4 Adequate sets of operators

Note that some operators may be expressed in terms of other operators. For example, the biconditional operator can be expressed in terms of conjunction and conditional operators. Thus, the biconditional operator does not add any expressive power to propositional logic. An *adequate set* of operators is a (minimal) set that is powerful enough to express any formula. Often, logics are *defined* in terms of an adequate set.

Example 2.2

Propositional logic may be defined as

$\texttt{f} ::= \texttt{tt} \mid \text{variable} \mid \neg\texttt{f} \mid \texttt{f} \wedge \texttt{f} \mid$

because $\{\neg, \wedge\}$ is an adequate set for propositional logic:

$$\begin{aligned}
\texttt{ff} \;&\equiv\; \neg\texttt{tt} \\
a \vee b \;&\equiv\; \neg(\neg a \wedge \neg b) \\
a \rightarrow b \;&\equiv\; (a \wedge b) \vee \neg a \;\equiv\; \neg\Big(\neg(a \wedge b) \wedge a\Big) \\
a \leftrightarrow b \;&\equiv\; (a \wedge b) \vee (\neg a \wedge \neg b) \;\equiv\; \neg\Big(\neg(a \wedge b) \wedge \neg(\neg a \wedge \neg b)\Big)
\end{aligned}$$

2.5 Satisfiability

The *Satisfiability problem*, or SAT, may be expressed as follows.

> Given a formula over propositional variables, determine if it is possible to assign values to the variables so that the entire formula evaluates to true.

This problem is well–known to be NP–complete. All known algorithms that solve this problem have worst–case running times that are at least exponential in the number of propositional variables. However, many SAT solver tools work quite well in practice.

Chapter 3

Introduction to model checking

The idea behind model checking is as follows. Given

- a model of the system, specified via some formalism; and
- properties of the system, specified via some logic;

determine whether the model satisfies the properties. We begin with a basic, "low–level" formalism for describing systems. We will discuss logics starting in the next chapter.

3.1 Kripke structures

A Kripke structure is a formalism that consists of a directed graph, where graph vertices represent states of the system and graph edges represent transitions, along with propositions that may or may not hold depending on the current state of the system. Formally, we have the following.

Definition 3.1 *A* Kripke structure *is a tuple $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ where*

- *$\mathcal{S}$ is a finite set of states;*
- *$\emptyset \subset \mathcal{S}_0 \subseteq \mathcal{S}$ is the set of* initial *states;*
- *$\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, such that each state has at least one outgoing edge: $\forall s \in \mathcal{S}, \exists (s, s') \in \mathcal{R}$ for some $s' \in \mathcal{S}$;*
- *$L : \mathcal{S} \to 2^{\mathcal{P}}$ is a labeling function, where $L(s)$ gives the subset of propositions $\mathcal{P}$ that hold in state s.*

Example 3.1

As an example, consider the following simple CD player. There are three buttons:

1. open / close
2. stop
3. play

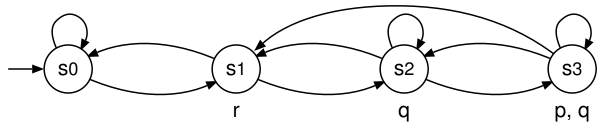
We can model this using a Kripke structure with:

- $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$ with
  - s_0: the tray is closed, without a CD
  - s_1: the tray is open
  - s_2: the tray is closed, with a CD, stopped
  - s_3: the tray is closed, with a CD, playing
- $\mathcal{S}_0 = \{s_0\}$
- $\mathcal{R} = \{(s_0, s_0), (s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_2, s_2), (s_2, s_3), (s_3, s_1), (s_3, s_2), (s_3, s_3)\}$
- Propositions $\mathcal{P} = \{p, q, r\}$ with
  - p : playing
  - q : tray is closed with a CD
  - r : tray is open
- Labeling:
  - $L(s_0) = \{\}$
  - $L(s_1) = \{r\}$
  - $L(s_2) = \{q\}$
  - $L(s_3) = \{p, q\}$

Normally, Kripke structures are *drawn*. We can do this by drawing the graph, indicating the start states, and indicating $L()$ beneath each state. Alternatively, when state names are unimportant, $L(s)$ can be drawn inside the graph vertex for state s. We will usually specify Kripke structures by drawing them in either of these ways.

Example 3.2

We can draw the Kripke structure for the CD player from Example 3.1:



Definition 3.2 *A* path *in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ is an infinite sequence of states $\pi = (p_0, p_1, p_2, \ldots) \in \mathcal{S}^\omega$ such that*

$$\forall i \geq 0, \qquad (p_i, p_{i+1}) \in \mathcal{R}.$$

Example 3.3

For the Kripke structure for the CD player from Example 3.1, $\pi = (s_0, s_1, s_1, s_2, \ldots)$ is *not* a path in the Kripke structure, because $(s_1, s_1) \notin \mathcal{R}$. $\pi = (s_3, s_3, s_2, s_1, s_2, s_2, s_1, s_0, s_1, s_0, s_0, \ldots)$ is a path in the Kripke structure[1].

[1]Technically it is the prefix of a path, because I did not write out infinitely many states.

3.2 Image operations

Two fundamental computations that we will need for Kripke structures are *pre-image* and *post-image*.

Pre-image: Given a set of states $\mathcal{X} \subseteq \mathcal{S}$, representing possible states of the system, "now". The Pre-image of $\mathcal{X}$ is the set of possible states for the system, "previously" (one step prior to the current state). Formally, we have

$$\mathit{PreImage}(\mathcal{X}, \mathcal{R}) = \{s : \exists s' \in \mathcal{X}, (s, s') \in \mathcal{R}\}$$

Post-image: Given a set of states $\mathcal{X} \subseteq \mathcal{S}$, representing possible states of the system, "now". The Post-image of $\mathcal{X}$ is the set of possible states for the system, after the next transition. Formally, we have

$$\mathit{PostImage}(\mathcal{X}, \mathcal{R}) = \{s' : \exists s \in \mathcal{X}, (s, s') \in \mathcal{R}\}$$

Whatever data structures we use to represent the Kripke structure and sets of states, will need to support these computations efficiently.

Example 3.4

For the CD player from Example 3.1, consider $\mathcal{X} = \{s_0, s_1\}$. The pre-image of $\mathcal{X}$ is $\{s_0, s_1, s_2, s_3\}$, and the post-image of $\mathcal{X}$ is $\{s_0, s_1, s_2\}$.

3.3 Internal representation

For practical models, the Kripke structure can be very large (millions of states is not uncommon). A typical representation uses

- an efficient graph data structure for representing $\mathcal{R}$, and
- an efficient data structure for representing subsets of $\mathcal{S}$,

where subsets of $\mathcal{S}$ are used to encode the labeling function L.

3.3.1 Graph representation

For now we will consider only the "classical" graph data structures.

Adjacency matrix:

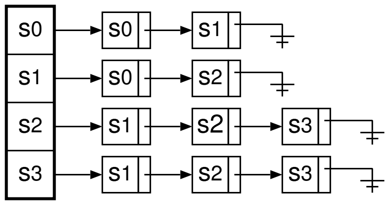
- Large Kripke structures are usually quite sparse, so a matrix representation is normally not used.
- The adjacency matrix *is* useful for describing operations. More on this, later.

Adjacency list:

- We can use a straightforward linked list for the outgoing edges from each state, while building the Kripke structure.
- Once the Kripke structure is complete, it often makes sense to convert to a more compact (less dynamic) representation.

Example 3.5

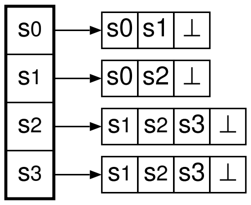
For the CD player from Example 3.1, using a linked list to store the outgoing edges for each state gives the following data structure.



The storage requirement is $|\mathcal{S}| + |\mathcal{R}|$ pointers, and $|\mathcal{R}|$ integers (state indexes).

Example 3.6

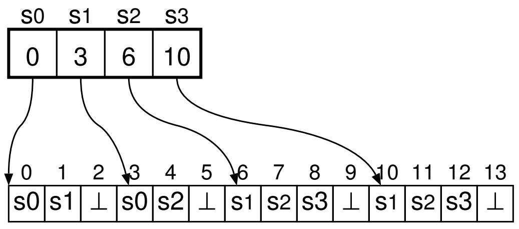
The data structure in Example 3.5 can be compacted into a less dynamic structure by replacing each linked list with a null–terminated array, as follows.



The storage requirement is $|\mathcal{S}|$ pointers, and $|\mathcal{S}| + |\mathcal{R}|$ integers (state indexes). With this data structure, it is more difficult to add new edges.

Example 3.7

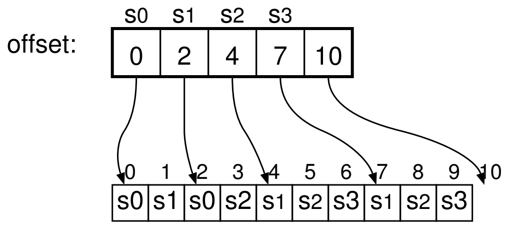
The separate arrays from Example 3.6 can be merged into a single array, and the pointers for each state can be replaced by array offsets, giving us the following data structure.



The storage requirement is $|\mathcal{S}|$ array offsets, and $|\mathcal{S}|+|\mathcal{R}|$ integers (state indexes). With this data structure, it is even more difficult to add new edges.

Example 3.8

Looking at the data structure from Example 3.7, we notice that the "$\perp$" sentinels serve only to indicate the end of a list. However, since the next list starts immediately after, it is possible to know the end of a list without using sentinels, as follows.



Since list i ends just before list $i+1$ begins, we have that list i runs from *offset*$[i]$ to *offset*$[i+1]-1$. The storage requirement here is $|\mathcal{S}|+1$ array offsets, and $|\mathcal{R}|$ integers (state indexes). This is known as *compressed sparse row format.*

3.3.2 Subset representation

To store a subset of $\mathcal{S}$, in practice[2] we need a way to store subsets of the integers $\{0, 1, \ldots, |\mathcal{S}|-1\}$. There are two straightforward ways to store $\mathcal{X} \subseteq \mathcal{S}$ (we will look at more advanced structures later).

- Use some dictionary structure or list of elements; this requires $\mathcal{O}(|\mathcal{X}|)$ storage. If we simply use a null–terminated array, then $|\mathcal{X}|+1$ integers are required.

- Use an array $\mathbf{x}$ where

$$\mathbf{x}[i] = \begin{cases} 1 & \text{iff } i \in \mathcal{X} \\ 0 & \text{otherwise} \end{cases}$$

  Using a straightforward implementation (for example, an array of type `bool` or `char`), this requires $|\mathcal{S}|$ bytes. However, with a more sophisticated implementation (using bitwise operators) we can "pack" the array and get a requirement of $|\mathcal{S}|$ bits.

For model checking, the sets $\mathcal{X}$ can be quite large (close or equal to $\mathcal{S}$ in size), in part because set complementation is a common operation. A vector of bits is more efficient when

$$|\mathcal{S}| < |\mathcal{X}| \cdot (\text{number of bits for an integer})$$

which happens frequently.

[2]If states have significant names, then we should use a data structure that gives an efficient mapping from $\mathcal{S}$ to $\{0, 1, \ldots, |\mathcal{S}|-1\}$.

3.4 Matrix descriptions

Let $\mathbf{E}$ be the adjacency matrix for a Kripke structure, where $\mathbf{E}[i,j]$ is one if and only if $(i,j) \in \mathcal{R}$. Let $\mathbf{x}, \mathbf{y}$ be vectors of bits representing $\mathcal{X}$ and $\mathcal{Y}$, subsets of $\mathcal{S}$, where $\mathbf{x}[i]$ is one iff $i \in \mathcal{X}$, and $\mathbf{y}[j]$ is one iff $j \in \mathcal{Y}$. Then we have

$$\begin{aligned}
\mathcal{Y} = \textit{PreImage}(\mathcal{X},\mathcal{R}) &= \{s : \exists s' \in \mathcal{X}, (s,s') \in \mathcal{R}\} \\
\mathcal{Y} &= \{i : \exists j, \mathbf{E}[i,j]\mathbf{x}[j] \neq 0\} \\
\mathbf{y}[i] &= \begin{cases} 1 & \text{if } \sum_j \mathbf{E}[i,j]\mathbf{x}[j] > 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

If we treat multiplication as conjunction and addition as disjunction, and use 0 for `ff` and 1 for `tt`, then we obtain

$$\begin{aligned}
\mathbf{y}[i] &= \begin{cases} 1 & \text{if } \sum_j \mathbf{E}[i,j]\mathbf{x}[j] > 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathbf{y}[i] &= \mathbf{E}[i,\bullet] \cdot \mathbf{x}
\end{aligned}$$

where $\mathbf{E}[i,\bullet]$ is the vector corresponding to row i of $\mathbf{E}$, and "$\cdot$" denotes vector dot product (using conjunction and disjunction, instead of multiplication and addition). But this gives us

$$\mathbf{y} = \mathbf{E}\mathbf{x} \tag{3.1}$$

as a concise way to describe the pre-image operation.

Similarly, for post-image, we have

$$\begin{aligned}
\mathcal{Y} = \textit{PostImage}(\mathcal{X},\mathcal{R}) &= \{s' : \exists s \in \mathcal{X}, (s,s') \in \mathcal{R}\} \\
\mathcal{Y} &= \{j : \exists i, \mathbf{x}[i]\mathbf{E}[i,j] \neq 0\} \\
\mathbf{y}[j] &= \begin{cases} 1 & \text{if } \sum_i \mathbf{x}[i]\mathbf{E}[i,j] > 0 \\ 0 & \text{otherwise} \end{cases} \\
\mathbf{y}[j] &= \mathbf{x} \cdot \mathbf{E}[\bullet,j] \\
\mathbf{y} &= \mathbf{x}\mathbf{E} \qquad (3.2)
\end{aligned}$$

where $\mathbf{E}[\bullet,j]$ is the vector corresponding to column j of $\mathbf{E}$.

Example 3.9

For the CD player from Example 3.1, compute the pre-image of $\mathcal{X} = \{s_0, s_1\}$.

Set $\mathcal{X}$ corresponds to vector $\mathbf{x} = [1,1,0,0]$. Using the matrix–vector multiplication operation, we have

$$\begin{aligned}
\mathbf{y} &= \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\
&= 1 \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + 1 \cdot \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} + 0 \cdot \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} + 0 \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
\end{aligned}$$

Therefore, the pre-image of $\mathcal{X}$ is $\{s_0, s_1, s_2, s_3\}$.

Example 3.10

For the CD player from Example 3.1, compute the post-image of $\mathcal{X} = \{s_0, s_1\}$.

Using the vector–matrix multiplication operation, we have

$$\begin{aligned}
\mathbf{y} &= [1,1,0,0] \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \\
&= 1 \cdot [1,1,0,0] + 1 \cdot [1,0,1,0] + 0 \cdot [0,1,1,1] + 0 \cdot [0,1,1,1] \\
&= [1,1,1,0]
\end{aligned}$$

Therefore, the post-image of $\mathcal{X}$ is $\{s_0, s_1, s_2\}$.

Chapter 4

Computation Tree Logic

Computation Tree (Temporal) Logic (CTL) is also known as Branching Time Temporal Logic. It is a popular logic for expressing properties, partly because it is powerful enough to be useful, but weak enough that its model checking algorithms are simple to implement. We will discuss CTL as applied to Kripke structures; later we will use different types of models.

4.1 CTL syntax

CTL formulas are *state formulas*, meaning we can determine if they hold or not for each state. A state formula ϕ in CTL has the form:

$$\begin{aligned} \phi \quad ::= \quad & \texttt{tt} \mid \texttt{ff} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{AX}\,\phi \mid \mathsf{EX}\,\phi \\ & \mid \mathsf{AF}\,\phi \mid \mathsf{EF}\,\phi \mid \mathsf{AG}\,\phi \mid \mathsf{EG}\,\phi \mid \mathsf{A}\ (\phi\ \mathsf{U}\ \phi) \mid \mathsf{E}\ (\phi\ \mathsf{U}\ \phi) \end{aligned}$$

where $p \in \mathcal{P}$ is an *atomic proposition*. Operators A and E are *path quantifiers*:

- A: "for all paths",
- E: "for at least one path (there exists a path)".

Operators X, F, G, and U are *temporal operators*:

- X: "in the ne**X**t state",
- F: "in a **F**uture state" (existential),
- G: "in all states (**G**lobally)" (universal),
- U: "**U**ntil"

Note that according to CTL syntax, the path quantifiers and temporal operators must be paired together. For example, $\mathsf{AX}\,p$ is a valid CTL formula, but $\mathsf{AGX}\,p$ and $\mathsf{AFGX}\,p$ are not valid CTL formulas. $\mathsf{AX}\,p$ means "for all paths, the next state satisfies p".

4.2 CTL semantics

We must give the rules for which states in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ satisfy a CTL state formula ϕ. If a state $s \in \mathcal{S}$ satisfies ϕ, we write

$$M, s \models \phi$$

although often the model M is omitted (if there is only one model, for instance). We write

$$M, s \not\models \phi$$

if state s does not satisfy ϕ. We give the rules for each type of formula in the syntax:

1. $M, s \models \texttt{tt}$, for all $s \in \mathcal{S}$
2. $M, s \not\models \texttt{ff}$, for all $s \in \mathcal{S}$
3. $M, s \models p$, if and only if $p \in L(s)$
4. $M, s \models \neg\phi$, if and only if $M, s \not\models \phi$
5. $M, s \models \phi_1 \wedge \phi_2$, if and only if $M, s \models \phi_1$ and $M, s \models \phi_2$
6. $M, s \models \phi_1 \vee \phi_2$, if and only if $M, s \models \phi_1$ or $M, s \models \phi_2$
7. $M, s \models \mathsf{AX}\,\phi$, if and only if, for all paths $(s, p_1, p_2, \ldots)$, $M, p_1 \models \phi$.
8. $M, s \models \mathsf{EX}\,\phi$, if and only if, there exists a path $(s, p_1, p_2, \ldots)$ such that $M, p_1 \models \phi$.
9. $M, s \models \mathsf{AF}\,\phi$, if and only if, for all paths $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, there exists an $i \geq 0$ such that $M, p_i \models \phi$.
10. $M, s \models \mathsf{EF}\,\phi$, if and only if, there exists a path $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, and an $i \geq 0$ such that $M, p_i \models \phi$.
11. $M, s \models \mathsf{AG}\,\phi$, if and only if, for all paths $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, we have $M, p_i \models \phi$ for all $i \geq 0$.
12. $M, s \models \mathsf{EG}\,\phi$, if and only if, there exists a path $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, such that $M, p_i \models \phi$ for all $i \geq 0$.
13. $M, s \models \mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2$, if and only if, for all paths $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, there exists an $i \geq 0$ such that
    (a) $M, p_i \models \phi_2$, and
    (b) $M, p_j \models \phi_1$, for all $0 \leq j < i$
14. $M, s \models \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$, if and only if, there exists a path $(p_0, p_1, p_2, \ldots)$ with $p_0 = s$, and an $i \geq 0$ such that
    (a) $M, p_i \models \phi_2$, and
    (b) $M, p_j \models \phi_1$, for all $0 \leq j < i$

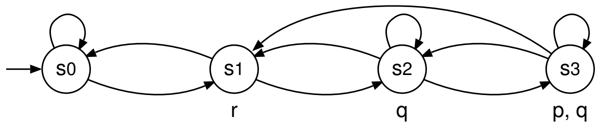
Finally, we say that the *model* satisfies a formula if *all* (or *some*) of its starting states satisfy the formula:

$$M \models \phi \quad \Leftrightarrow \quad \forall s \in \mathcal{S}_0,\ M, s \models \phi$$

where $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ is the Kripke structure.

Example 4.1

Recall the CD player Kripke structure from Example 3.1:



For this model, does state s_2 satisfy:

1. $\mathsf{EX}\, q$?
2. $\mathsf{AX}\, q$?
3. $\mathsf{EF}\, (\neg r \wedge \neg q)$?
4. $\mathsf{AF}\, r$?
5. $\mathsf{E}\, p\, \mathsf{U}\, q$?
6. $\mathsf{E}\, q\, \mathsf{U}\, r$?
7. $\mathsf{A}\, q\, \mathsf{U}\, r$?

Solutions

1. $\mathsf{EX}\, q$: Yes, because there exists a path of the form $(s_2, s_3, \ldots)$ and $s_3 \models q$.
2. $\mathsf{AX}\, q$: No, because there is some path that starts with s_2, where q does not hold in the next state. Specifically, a path of the form $(s_2, s_1, \ldots)$ has $s_1 \not\models q$.
3. $\mathsf{EF}\, (\neg r \wedge \neg q)$: Yes, because using path $(p_0 = s_2, p_1 = s_1, p_2 = s_0, \ldots)$ and $i = 2$, we have $p_i \models (\neg r \wedge \neg q)$.
4. $\mathsf{AF}\, r$: No. Consider the path $(p_0 = s_2, p_1 = s_2, p_2 = s_2, \ldots)$. For this path, there does not exist a j such that $p_j \models r$.
5. $\mathsf{E}\, p\, \mathsf{U}\, q$: Yes. Using path $(p_0 = s_2, \ldots)$ and $i = 0$, we have
   - $p_0 \models q$, and
   - $p_j \models p$, for all $0 \leq j < 0$.
6. $\mathsf{E}\, q\, \mathsf{U}\, r$: Yes. Using path $(p_0 = s_2, p_1 = s_1, \ldots)$ and $i = 1$, we have
   - $p_1 \models r$, and
   - $p_j \models q$, for all $0 \leq j < 1$.
7. $\mathsf{A}\, q\, \mathsf{U}\, r$: this looks promising, because any path that starts in s_2 and ends up in s_1 will satisfy q until s_1 is reached. However, consider the path $(p_0 = s_2, p_1 = s_2, p_2 = s_2, \ldots)$. There is no such i where
   - $p_i \models r$, and
   - $p_j \models q$, for all $0 \leq j < i$.

   The second condition holds for any i, but not the first.

4.3 Translating English to CTL

English is not a formal language, so it does not make sense to write an algorithm to convert from English to CTL. However, there are common English patterns that can be mapped to CTL. Here are several useful examples.

1. "The system never reaches a deadlocked state"

   $$\mathsf{AG}\ \neg \mathit{deadlocked}$$

   Technically, the above property is "the system always remains in a non-deadlocked state".

2. "Is it possible to reach a state where condition C holds?"

   $$\mathsf{EF}\ C$$

   Putting the first two together, we might have...

3. "Is it possible to reach a state from which the system never deadlocks?"

   $$\mathsf{EF}\ \mathsf{AG}\ \neg \mathit{deadlocked}$$

   Putting the first two together in the opposite order, we might have...

4. "It is always possible to reach a deadlocked state."

   $$\mathsf{AG}\ \mathsf{EF}\ \mathit{deadlocked}$$

5. "The system will eventually reach a deadlocked state, and remain deadlocked."

   $$\mathsf{AF}\ \mathsf{AG}\ \mathit{deadlocked}$$

6. "Condition C holds infinitely often."

   $$\mathsf{AG}\ \mathsf{AF}\ C$$

7. "Whenever we reach a state where condition B holds, we eventually reach a state where condition C holds."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{AF}\ C)$$

   Note that we effectively rewrote the original statement as "For every state, if condition B holds, then we eventually reach a state where condition C holds."

8. "Once condition B holds, it holds until condition C holds (and C must eventually hold)."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{A}\, B\, \mathsf{U}\, C)$$

9. "Whenever condition B holds, condition C holds after 2 or more steps."

   $$\mathsf{AG}\ (B \rightarrow \mathsf{AX}\ \mathsf{AX}\ \mathsf{AF}\ C)$$

4.3.1 Safety properties

A *safety* property is one that specifies that some undesired behavior never happens. For example, in a system of traffic lights, we would like to be sure that lights are never simultaneously green for northbound and westbound traffic. Safety properties can be expressed easily in CTL.

Example 4.2

The property

$$\neg \mathsf{EF}(north\_green \ \wedge \ west\_green)$$

says that, it is not possible to eventually reach a state where the northbound and westbound lights are both green.

4.3.2 Liveness properties

A *liveness* property is one that specifies that some desired behavior eventually happens. For example, in a system of traffic lights, we would like to be sure that the northbound traffic will eventually get a green light. Furthermore, we might like this to always be the case. Liveness properties of various strengths can be expressed in CTL.

Example 4.3

The property

$$\mathsf{AG}\ \mathsf{AF}\ north\_green$$

says that, from every state, we will always eventually reach a state where the northbound light is green.

Example 4.4

The property

$$\mathsf{AG}\ \mathsf{EF}\ north\_green$$

says that, from every state, it is always *possible* to reach a state where the northbound light is green.

4.4 Equivalences

As with propositional logic, some properties may be expressed in several different ways in CTL. Given a Kripke structure M and a property to express, sometimes the structure of M itself introduces equivalences. In this section, we discuss formulas that are equivalent in CTL, *for any Kripke structure.* We can manipulate CTL formulas in the same way that we did propositional logic formulas. To prove that formula ϕ_1 is equivalent to formula ϕ_2, we must show that, for any Kripke structure M and any state s, $M, s \models \phi_1$ if and only if $M, s \models \phi_2$. Conversely, to show that formulas ϕ_1 and ϕ_2 are *not* equivalent, it is sufficient to find a Kripke structure M and a state s such that $M, s \models \phi_1$ and $M, s \not\models \phi_2$.

Property 4.1

$$\neg\mathsf{EX}\,\phi \quad\equiv\quad \mathsf{AX}\,\neg\phi$$

Proof:

$$\begin{aligned}
M,s \models \neg\mathsf{EX}\,\phi &\Leftrightarrow M,s \not\models \mathsf{EX}\,\phi\\
&\Leftrightarrow \nexists\,\pi = (s,p_1,\ldots),\ M,p_1 \models \phi\\
&\Leftrightarrow \forall\pi = (s,p_1,\ldots),\ M,p_1 \not\models \phi\\
&\Leftrightarrow \forall\pi = (s,p_1,\ldots),\ M,p_1 \models \neg\phi\\
&\Leftrightarrow M,s \models \mathsf{AX}\,\neg\phi
\end{aligned}$$

Property 4.2

$$\neg\mathsf{EF}\,\phi \quad\equiv\quad \mathsf{AG}\,\neg\phi$$

Proof:

$$\begin{aligned}
M,s \models \neg\mathsf{EF}\,\phi &\Leftrightarrow M,s \not\models \mathsf{EF}\,\phi\\
&\Leftrightarrow \nexists\,i, \pi = (p_0 = s,p_1,p_2,\ldots),\ M,p_i \models \phi\\
&\Leftrightarrow \forall\pi = (p_0 = s,p_1,p_2,\ldots),\ M,p_i \not\models \phi, \forall i\\
&\Leftrightarrow \forall\pi = (p_0 = s,p_1,p_2,\ldots),\ M,p_i \models \neg\phi, \forall i\\
&\Leftrightarrow M,s \models \mathsf{AG}\,\neg\phi
\end{aligned}$$

Property 4.3

$$\neg\mathsf{EG}\,\phi \quad\equiv\quad \mathsf{AF}\,\neg\phi$$

Proof:

$$\begin{aligned}
M,s \models \neg\mathsf{EG}\,\phi &\Leftrightarrow M,s \not\models \mathsf{EG}\,\phi\\
&\Leftrightarrow \nexists\,\pi = (p_0 = s,p_1,p_2,\ldots),\ M,p_i \models \phi \quad \forall i\\
&\Leftrightarrow \forall\pi = (p_0 = s,p_1,p_2,\ldots),\ \exists i,\ M,p_i \not\models \phi\\
&\Leftrightarrow \forall\pi = (p_0 = s,p_1,p_2,\ldots),\ \exists i,\ M,p_i \models \neg\phi\\
&\Leftrightarrow \mathsf{AF}\,\neg\phi
\end{aligned}$$

Property 4.4

$$\mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,\phi \quad\equiv\quad \mathsf{EF}\,\phi$$

Property 4.5

$$\mathsf{A}\,\mathtt{tt}\,\mathsf{U}\,\phi \quad\equiv\quad \mathsf{AF}\,\phi$$

Property 4.6

$$\neg\mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2 \quad\equiv\quad \mathsf{E}(\ (\phi_1 \wedge \neg\phi_2)\,\mathsf{U}\,(\neg\phi_1 \wedge \neg\phi_2)\ ) \quad\vee\quad \mathsf{EG}\,\neg\phi_2$$

Property 4.7

$$\phi \vee \mathsf{EX}\,\mathsf{EF}\,\phi \quad \equiv \quad \mathsf{EF}\,\phi$$

Proof:

$$\begin{aligned}
M,s \models \phi \vee \mathsf{EX}\,\mathsf{EF}\,\phi \;&\Leftrightarrow\; M,s \models \phi \quad \vee \quad M,s \models \mathsf{EX}\,\mathsf{EF}\,\phi \\
&\Leftrightarrow\; M,s \models \phi \vee \exists \pi = (s, p_1, \ldots),\ M, p_1 \models \mathsf{EF}\,\phi \\
&\Leftrightarrow\; M,s \models \phi \vee \exists \pi = (s, p_1, \ldots), \exists i, \pi' = (p'_0 = p_1, p'_1, p'_2, \ldots),\ M, p'_i \models \phi \\
&\Leftrightarrow\; M,s \models \phi \ \vee\ \exists i, \pi = (s, p'_0, p'_1, \ldots),\ M, p'_i \models \phi \\
&\Leftrightarrow\; \exists i, \pi = (p_0 = s, p_1, \ldots),\ M, p_i \models \phi \\
&\Leftrightarrow\; M,s \models \mathsf{EF}\,\phi
\end{aligned}$$

Property 4.8

$$\phi \vee \mathsf{AX}\,\mathsf{AF}\,\phi \quad \equiv \quad \mathsf{AF}\,\phi$$

Property 4.9

$$\phi \wedge \mathsf{EX}\,\mathsf{EG}\,\phi \quad \equiv \quad \mathsf{EG}\,\phi$$

Property 4.10

$$\phi \wedge \mathsf{AX}\,\mathsf{AG}\,\phi \quad \equiv \quad \mathsf{AG}\,\phi$$

Property 4.11

$$\phi_2 \vee (\phi_1 \wedge \mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2) \quad \equiv \quad \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$$

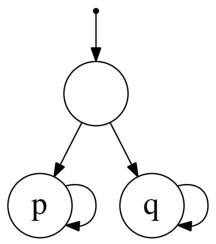
Property 4.12

$$\phi_2 \vee (\phi_1 \wedge \mathsf{AX}\,\mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2) \quad \equiv \quad \mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2$$

Example 4.5

Is the formula $\mathsf{EX}(p \wedge q)$ equivalent to $(\mathsf{EX}\,p) \wedge (\mathsf{EX}\,q)$?

Solution: It is easy to show that $\mathsf{EX}(p \wedge q) \;\rightarrow\; (\mathsf{EX}\,p) \wedge (\mathsf{EX}\,q)$. However, the reverse is not true. Consider the following Kripke structure:



Clearly, both $\mathsf{EX}\,p$ and $\mathsf{EX}\,q$ hold in the initial state, but $\mathsf{EX}(p \wedge q)$ does not.

4.4.1 Adequate sets of operators for CTL

From the above properties, and from the adequate sets for propositional logic, we see that the following operations are adequate to express all CTL formulas:

1. $\neg$
2. $\wedge$
3. AX or EX
4. EG or AF
5. EU (meaning, $\mathsf{E}\phi_1\mathsf{U}\phi_2$)

4.5 Algorithms for CTL operators

To automatically determine the set of states that satisfy an arbitrary CTL state formula ϕ, we need algorithms for all of the CTL operators. These are sometimes called *labeling* algorithms. For example, if we know which states are labeled with ϕ_1, and which states are labeled with ϕ_2, then we need an algorithm to label states satisfying $\phi = \phi_1 \wedge \phi_2$. We will discuss algorithms only for some operators (namely, the adequate ones: $\neg, \wedge, \mathsf{AX}, \mathsf{EX}, \mathsf{AF}, \mathsf{EG}, \mathsf{EU}$).

4.5.1 Labeling for $\neg$

There is a trivial algorithm for $\neg$: given the labeling for ϕ_1, we can determine the labeling for $\phi = \neg\phi_1$ as follows:

> For each state s, label s with ϕ if and only if s is not labeled with ϕ_1.

Complexity is $\mathcal{O}(|\mathcal{S}|)$.

4.5.2 Labeling for $\wedge$

There is also a simple algorithm for labeling $\phi = \phi_1 \wedge \phi_2$:

> For each state s, label s with ϕ if and only if s is labeled with both ϕ_1 and ϕ_2.

Complexity is $\mathcal{O}(|\mathcal{S}|)$.

4.5.3 Labeling for AX

We can label $\phi = \mathsf{AX}\,\phi_1$ using the following algorithm:

> For each state s, label s with ϕ if and only if every successor of s (i.e., all states s' with $(s, s') \in \mathcal{R}$) is labeled with ϕ_1.

Complexity is $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

4.5.4 Labeling for EX

We can label $\phi = \mathsf{EX}\,\phi_1$ using an algorithm similar to the one for AX:

For each state s, label s with ϕ if and only if some successor of s is labeled with ϕ_1.

Using preimage

Note that this algorithm is exactly the pre-image operation. Thus, if bit vector $\mathbf{x}$ has entries where $\mathbf{x}[i]$ is one if and only if state i is labeled with ϕ_1, then the bit vector $\mathbf{y}$ given by

$$\mathbf{y} = \mathbf{E}\mathbf{x}$$

where $\mathbf{E}$ is the adjacency matrix corresponding to $\mathcal{R}$, has entries one for states labeled with ϕ. Complexity is $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$; to obtain this complexity using $\mathbf{Ex}$, a suitable matrix–vector multiplication algorithm must be used (one that exploits the fact that $\mathbf{E}$ is sparse).

4.5.5 Labeling for AF

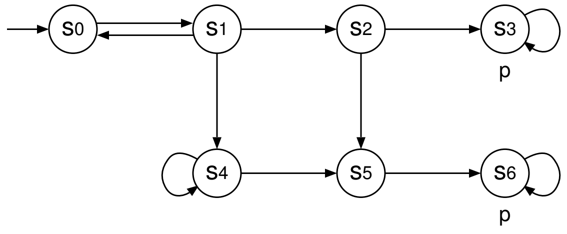
We can label $\phi = \mathsf{AF}\,\phi_1$ using an algorithm based on Property 4.8:

1. Any state labeled with ϕ_1 is also labeled with ϕ.
2. If all successors of state s are labeled with ϕ, then label s with ϕ
3. Repeat step (2) until no more changes are possible.

Note that the algorithm is guaranteed to eventually terminate, since at most $|\mathcal{S}|$ states can be labeled. This algorithm, if implemented cleverly, has a complexity of $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

Example 4.6

Compute the labeling for $\phi = \mathsf{AF}\,p$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we label s_3 and s_6 with ϕ. In step (2), we can label s_5 with ϕ. There was a change, so we repeat. In step (2), we can label s_2 with ϕ. There was a change, so we repeat. But no more states have *all* their successors labeled with ϕ, so the algorithm terminates. We therefore have that the states $\{s_2, s_3, s_5, s_6\}$ satisfy $\mathsf{AF}\,p$.

Verify: States s_3 and s_6 satisfy p, so they trivially satisfy $\mathsf{AF}\,p$. State s_5 has only one possible path, and state s_2 has only two possible paths, both of which eventually reach a state satisfying p. But what about the other states? State s_4 has a self loop, and therefore there exists a path, namely $(s_4, s_4, s_4, \ldots)$ that never reaches a state satisfying p. Similarly, states s_0 and s_1 can loop forever and never reach a state satisfying p.

4.5.6 Labeling for EG — iterative algorithm

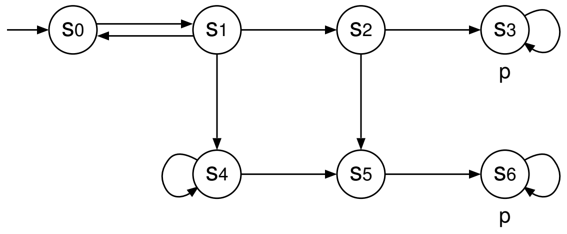
We can label $\phi = \mathsf{EG}\,\phi_1$ using an algorithm based on Property 4.9:

1. Any state labeled with ϕ_1 is labeled with ϕ.
2. If no successors of state s are labeled with ϕ, then remove the ϕ label from state s.
3. Repeat step (2) until no more changes are possible.

Note that the algorithm will eventually terminate, since in step 2 we are only removing labels from states, and there are only finitely many states. Also, note that this is essentially the opposite of the algorithm for AF.

Example 4.7

Compute the labeling for $\phi = \mathsf{EG}\,\neg p$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we label s_0, s_1, s_2, s_4, and s_5 with ϕ. In step (2), we remove label ϕ from s_5, since none of its successors are labeled with ϕ. There was a change, so we repeat. In step (2), we remove label ϕ from s_2. There was a change, so we repeat. But the remaining states all have at least one successor labeled with ϕ, so the algorithm terminates. We therefore have that the states $\{s_0, s_1, s_4\}$ satisfy $\mathsf{EG}\,\neg p$.

Verify: It is easy to see that, starting from states s_0, s_1, and s_4, it is possible to remain in states where p is not satisfied. This is not possible from states s_2, s_3, s_5, and s_6.

Using preimage

We can rewrite the above algorithm in terms of the pre-image operation, as follows. Assume we have a bitvector $\mathbf{x}$ that encodes the states satisfying ϕ_1. Then we can build the bitvector $\mathbf{y}$ that encodes the states satisfying $\phi = \mathsf{EG}\,\phi_1$ using the iteration

$$\begin{aligned}\mathbf{y}_0 &= \mathbf{x}\\ \mathbf{y}_{n+1} &= \mathbf{y}_n \wedge (\mathbf{E}\mathbf{y}_n)\end{aligned}$$

and stopping when $\mathbf{y}_{n+1} = \mathbf{y}_n$, and using $\mathbf{y} = \mathbf{y}_n$. It can be shown that the iteration

$$\begin{aligned}\mathbf{y}_0 &= \mathbf{x}\\ \mathbf{y}_{n+1} &= \mathbf{x} \wedge (\mathbf{E}\mathbf{y}_n)\end{aligned}$$

produces exactly the same sequence of bitvectors. (Hint: $\mathbf{y}_n$ is the set of all starting states from which it is possible to find a path where the first n states on the path satisfy ϕ_1.)

4.5.7 Labeling for EG — strongly connected components

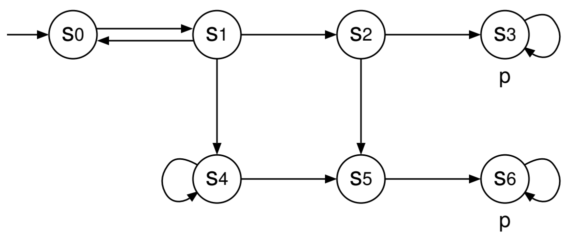
Another algorithm for labeling $\phi = \mathsf{EG}\,\phi_1$ is based on the observation that, the only way an infinitely–long path satisfying ϕ_1 can occur is if there is a cycle of states satisfying ϕ_1. The cycles of states are determined using strongly–connected components (SCCs). Recall that a SCC is a group of states such that, for any pair of states i and j in the SCC, there is a path from state i to state j.

1. Create a graph M' by removing all states that do not satisfy ϕ_1, and any edges associated with those states. (Note: this might not be a Kripke structure, because it is possible for a state to have no outgoing edges.)
2. Determine the SCCs for M'.
3. For each SCC,
   - if the SCC contains more than one state, then label all states in the SCC with ϕ
   - if the SCC contains only one state, and the state has an edge to itself[1], then label it with ϕ.
4. Any state that can reach (in M') a state labeled with ϕ should also be labeled with ϕ.

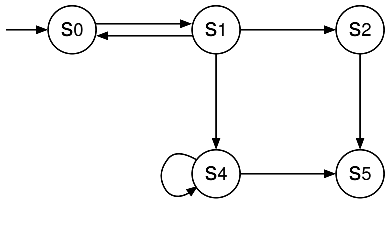
There is an algorithm to determine all SCCs for a graph, with complexity $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$. Thus, this algorithm has complexity $\mathcal{O}(|\mathcal{S}| + |\mathcal{R}|)$.

Example 4.8

Compute the labeling for $\phi = \mathsf{EG}\,\neg p$ for the following Kripke structure:



Solution: Using the algorithm, in step (1) we build a new graph containing only the states satisfying $\neg p$; this gives us:



This graph has the following SCCs: $\{s_0, s_1\}, \{s_2\}, \{s_4\}, \{s_5\}$. Following step (3) of the algorithm, we label $\{s_0, s_1\}$ and $\{s_4\}$ with ϕ. No new states are labeled in step (4); therefore, the states $\{s_0, s_1, s_4\}$ satisfy $\mathsf{EG}\,\neg p$.

Verify: This is the same set as the previous example.

[1]This check is necessary because, the SCC algorithm will put every state into some SCC.

4.5.8 Labeling for EU

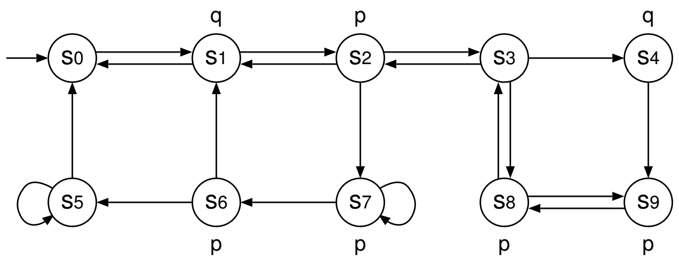
We can label $\phi = \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$ based on Property 4.11:

1. Any state labeled with ϕ_2 is labeled with ϕ.
2. If state s is labeled with ϕ_1, and some successor of s is labeled with ϕ, then label s with ϕ.
3. Repeat step (2) until no more changes are possible.

Since we never remove labels from states, the algorithm will eventually terminate, since $\mathcal{S}$ is finite. This algorithm, if implemented cleverly (using a single graph traversal), has complexity $\mathcal{O}(|\mathcal{S}|+|\mathcal{R}|)$.

Example 4.9

Compute the labeling for $\phi = \mathsf{E}\,p\,\mathsf{U}\,q$ for the following Kripke structure:



Solution: In step (1) of the algorithm, we label states s_1 and s_4 with ϕ, because they satisfy q. Then, during the first iteration of step (2), we can label states s_2 and s_6 with ϕ, because these states satisfy p and have a successor labeled with ϕ. In the next iteration, we can label state s_7 with ϕ. No more changes are possible after that; therefore, the states $\{s_1, s_2, s_4, s_6, s_7\}$ satisfy $\mathsf{E}\,p\,\mathsf{U}\,q$.

Verify: From each of (and only) $\{s_1, s_2, s_4, s_6, s_7\}$, there is a path that

- leads to[2] a state satisfying q, and
- for every state on the path before q is satisfied, p is satisfied.

Using preimage

We can rewrite the above algorithm in terms of the pre-image operation. Suppose the bitvector $\mathbf{p}$ encodes states satisfying p, and $\mathbf{q}$ encodes the states satisfying q. Then, we can build the bitvector $\mathbf{y}$ satisfying $\phi = \mathsf{E}\,p\,\mathsf{U}\,q$ using the iteration

$$\begin{aligned}\mathbf{y}_0 &= \mathbf{q}\\ \mathbf{y}_{n+1} &= \mathbf{y}_n \vee ((\mathbf{E}\cdot\mathbf{y}_n) \wedge \mathbf{p})\end{aligned}$$

and stopping when $\mathbf{y}_{n+1} = \mathbf{y}_n$, and using $\mathbf{y} = \mathbf{y}_n$.

[2]Starting with a state satisfying q is also allowed.

4.6 Counter examples and witnesses

Suppose we have a Kripke structure, and we want to verify the property $\mathsf{AG}(\neg \textit{deadlocked})$, but it turns out the answer is "no". Now what?

For debugging of the model or the underlying system, it would be nice to know "why not". In the case of this property, because we have

$$\neg\mathsf{AG}(\neg \textit{deadlocked}) \;\equiv\; \mathsf{EF}\ \textit{deadlocked}$$

we can see that the property does not hold because there exists a path that eventually reaches a *deadlocked* state. Such a path is a called a *counter example* to the property $\mathsf{AG}(\neg \textit{deadlocked})$. In general, whenever an "A" property does not hold, then the corresponding "E" property holds and can be used to give a counter example. Summarizing earlier properties, we have:

$$\begin{array}{lcl}
s \not\models \mathsf{AX}\,\phi & \to & s \models \mathsf{EX}\,\neg\phi \\
s \not\models \mathsf{AF}\,\phi & \to & s \models \mathsf{EG}\,\neg\phi \\
s \not\models \mathsf{AG}\,\phi & \to & s \models \mathsf{EF}\,\neg\phi \\
s \not\models \mathsf{A}\,\phi_1\,\mathsf{U}\,\phi_2 & \to & s \models \mathsf{EG}\,\neg\phi_2 \quad \vee \quad s \models \mathsf{E}\,(\phi_1 \wedge \neg\phi_2)\,\mathsf{U}\,(\neg\phi_1 \wedge \neg\phi_2)
\end{array}$$

What about "E" properties? If $\mathsf{EG}\,\phi$ does not hold, can we give a counter example? This would require us to show that $\mathsf{AF}\,\neg\phi$ holds, and in general we cannot give a single path to show this. For "E" properties, an example path that satisfies the formula is called a *witness*. Thus, a witness for $\mathsf{EG}\,\phi$ is an example path where ϕ holds in every state, and is also a counter example for $\mathsf{AF}\,\neg\phi$. We therefore need witness generation algorithms for EX, EG, and EU.

4.6.1 Witnesses for EX

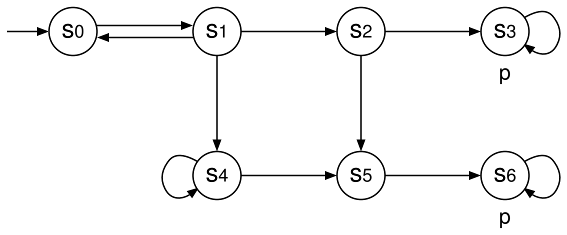
Given a state $s \models \mathsf{EX}\,\phi$, how do we generate a witness for $\mathsf{EX}\,\phi$ that starts in state s? This is trivial: check all successors of s for an $s' \models \phi$; the witness is a path $(s, s', \ldots)$, where of course we only display the "interesting" portion of the path.

4.6.2 Witnesses for EG

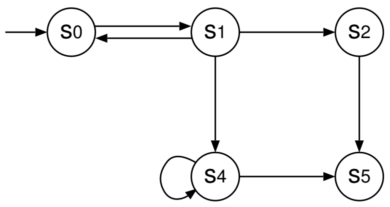
Given a state $s \models \mathsf{EG}\,\phi$, how do we generate a witness for $\mathsf{EG}\,\phi$ that starts in state s? Note that a witness will contain a cycle of states that satisfy ϕ; it suffices to terminate the displayed path with such a cycle (preferably a minimal cycle), rather than displaying an infinitely–long path. The SCC–based algorithm for labeling EG can inspire a witness generation algorithm. Consider the graph M' obtained by removing all states that do not satisfy ϕ, and their incoming and outgoing edges. If s has a self loop in this graph, then trivially $(s, s, \ldots)$ is a witness. If s belongs to a SCC with at least 2 states, then from any successor s' of s, there must exist a path in the graph from s' to s. Find and display such a path (ideally, find a *shortest* path). Otherwise, find a path from s to some state s' that either contains a self loop or belongs to a SCC with at least 2 states. Note that $s' \models \mathsf{EG}\,\phi$. Display the path from s to s', and then display a witness for $\mathsf{EG}\,\phi$ that starts in state s'.

Example 4.10

For the following Kripke structure, give a witness for $\mathsf{EG}\,\neg p$ starting in state s_0.



Solution: From an earlier example, removing the states that do not satisfy $\neg p$ gives us the graph:



Note that s_0 belongs to the SCC $\{s_0, s_1\}$. Therefore, a witness is s_0, followed by a path in the graph from s_1 (a successor of s_0) to s_0, which turns out to be trivial because there is an edge from s_1 to s_0. This gives us the witness

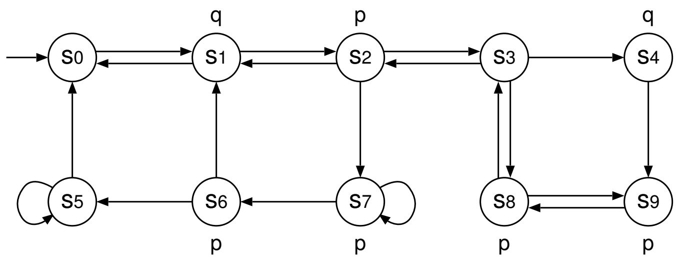
$$(s_0, s_1, s_0, \ldots)$$

4.6.3 Witnesses for EU

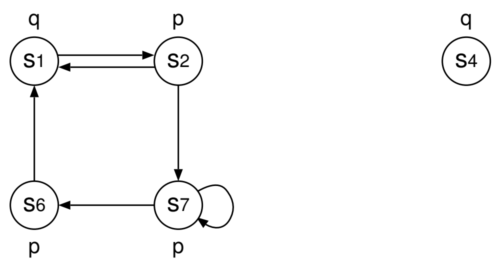
Given a state $s \models \mathsf{E}\,\phi_1 \,\mathsf{U}\, \phi_2$, a witness for $\mathsf{E}\,\phi_1 \,\mathsf{U}\, \phi_2$ starting from s can be generated by finding a path from s to a state satisfying ϕ_2, along only states satisfying ϕ_1. One way to do this is to find a shortest path from s to a state satisfying ϕ_2 in the graph obtained from the Kripke structure by removing all states (and their incoming and outgoing edges) that do not satisfy $\mathsf{E}\,\phi_1 \,\mathsf{U}\, \phi_2$.

Example 4.11

For the following Kripke structure, give a witness for $\mathsf{E}\,p \,\mathsf{U}\, q$ starting in state s_2.



Solution: From an earlier example, we know the set of states satisfying $\mathsf{E}\,p \,\mathsf{U}\, q$ is exactly $\{s_1, s_2, s_4, s_6, s_7\}$; restricting the Kripke structure to these states gives us the graph:



Finding a shortest path from s_2 to a state satisfying q (states s_1 and s_4) will generate the path s_2, s_1; that gives us the witness

$$(s_2, s_1, \ldots)$$

4.6.4 Nested formulas

Automatically generating a "complete" witness or counter example for a nested formula turns out to be impossible in general.

For instance, suppose we generate a witness for a formula $\mathsf{EG}\,\phi$ using the method discussed above. That gives us a path where each state satisfies ϕ. But if ϕ is a complex formula, it might not be obvious for each state that it satisfies ϕ. We should then generate witnesses for ϕ, starting from each state used in the witness for $\mathsf{EG}\,\phi$. Unfortunately, this is not always possible: if ϕ contains an "A" formula, then we cannot generate a witness for it.

However, in the case that a formula ϕ contains only E path quantifiers and holds for some state, we can recursively generate witnesses. Similarly, if the formula contains only A path quantifiers and does not hold, we can recursively generate counter examples.

4.7 Fixed points

In this section, we will discuss some of the theory behind the algorithms for the CTL operators. We consider functions f of the form $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, i.e., functions that take a set of states and return a set of states.

Definition 4.13 *A function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ is called* monotonic *if*

$$\mathcal{X} \subseteq \mathcal{Y} \quad \to \quad f(\mathcal{X}) \subseteq f(\mathcal{Y}), \qquad \forall \mathcal{X}, \mathcal{Y} \subseteq \mathcal{S}$$

Example 4.12

For $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$, is the function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$ monotonic?

Solution: This example is small enough to enumerate, but let's try a proper proof instead. For any sets $\mathcal{X}, \mathcal{Y}$ with $\mathcal{X} \subseteq \mathcal{Y}$, we must show that $f(\mathcal{X}) \subseteq f(\mathcal{Y})$:

$$f(\mathcal{X}) \;=\; \mathcal{X} \cup \{s_0\} \;\subseteq\; \mathcal{Y} \cup \{s_0\} \;=\; f(\mathcal{Y})$$

Example 4.13

For $\mathcal{S} = \{s_0, s_1, s_2, s_3\}$, the function

$$f(\mathcal{X}) \;=\; \begin{cases} \{s_0\} & \text{If } s_1 \in \mathcal{X} \\ \{s_0, s_1\} & \text{Otherwise} \end{cases}$$

is *not* monotonic, because $\emptyset \subseteq \{s_1\}$ but $f(\emptyset) = \{s_0, s_1\}$ is not a subset of $f(\{s_1\}) = \{s_0\}$.

Property 4.14 *The function $f_{\mathcal{R}}(\mathcal{X}) = \mathit{PreImage}(\mathcal{X}, \mathcal{R})$ is monotonic.*

Proof: *Suppose $\mathcal{X} \subseteq \mathcal{Y}$, and consider $s \in f_{\mathcal{R}}(\mathcal{X})$. Then there must exist some $s' \in \mathcal{X}$ such that $(s, s') \in \mathcal{R}$. But $\mathcal{X} \subseteq \mathcal{Y}$, so we must have $s' \in \mathcal{Y}$ also. This implies $s \in f_{\mathcal{R}}(\mathcal{Y})$.*

Definition 4.15 *For any function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, $\mathcal{X}$ is a* fixed point *of f if $f(\mathcal{X}) = \mathcal{X}$.*

Example 4.14

For the monotonic function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$, there are several fixed points, including

$$\begin{aligned} f(\{s_0, s_1\}) &= \{s_0, s_1\} \\ f(\mathcal{S}) &= \mathcal{S} \end{aligned}$$

Definition 4.16 *For any function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$, define f^n, for $n \geq 0$, inductively as*

$$\begin{aligned} f^0(\mathcal{X}) &= \mathcal{X} \\ f^{n+1}(\mathcal{X}) &= f(f^n(\mathcal{X})) \end{aligned}$$

Property 4.17 (Tarski–Knaster Theorem) *If f is monotonic over $2^{\mathcal{S}}$, where $|\mathcal{S}| = n$, then*

1. *$f^n(\emptyset)$ is the least fixed point of f*
2. *$f^n(\mathcal{S})$ is the greatest fixed point of f*

Proof: *Since $\emptyset \subseteq \mathcal{X}$ for any $\mathcal{X}$, we have*

$$\begin{array}{rcll} \emptyset & \subseteq & f(\emptyset) & \\ f(\emptyset) & \subseteq & f(f(\emptyset)) & \quad \textit{because } f \textit{ is monotone} \\ & \vdots & & \\ f^i(\emptyset) & \subseteq & f^{i+1}(\emptyset) & \quad \forall i \end{array}$$

Because $\mathcal{S}$ is finite, there exists a $j \geq 0$ such that $f^j(\emptyset) = f^{j+1}(\emptyset)$, and we have

$$f^0(\emptyset) \subset f^1(\emptyset) \subset f^2(\emptyset) \subset \cdots \subset f^j(\emptyset) = f^{j+1}(\emptyset) = f^{j+2}(\emptyset) = \cdots$$

Furthermore, we know $j \leq n$. Therefore, $f(f^n(\emptyset)) = f^n(\emptyset)$, and $f^n(\emptyset)$ is a fixed point. Now we must show that it is the least *fixed point. Consider any fixed point $\mathcal{X}$. Then we have*

$$\begin{array}{rcl} \emptyset & \subseteq & \mathcal{X} \\ f(\emptyset) & \subseteq & f(\mathcal{X}) \;=\; \mathcal{X} \\ & \vdots & \\ f^n(\emptyset) & \subseteq & \mathcal{X} \end{array}$$

and the proof of part (1) is complete. The proof for part (2) is similar.

Example 4.15

For the monotonic function $f(\mathcal{X}) = \mathcal{X} \cup \{s_0\}$, we have

$$\begin{aligned} f(\emptyset) &= \{s_0\} \\ f(\{s_0\}) &= \{s_0\} \end{aligned}$$

and therefore the least fixed point is $\{s_0\}$. Similarly, since $f(\mathcal{S}) = \mathcal{S}$, the greatest fixed point is $\mathcal{S}$.

4.7.1 Sets satisfying CTL state formulas

Definition 4.18 *For any CTL state formula ϕ, let*

$$[\![\phi]\!]_M \qquad \textit{denote the set of states in } M \textit{ satisfying } \phi$$

where the Kripke stucture M may be dropped if it is clear from context.

Using this notation, we can express the labeling algorithms in terms of the sets of states satisfying the formulas. For example, for Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$, we have:

- $[\![\texttt{tt}]\!]_M \;=\; \mathcal{S}$
- $[\![\texttt{ff}]\!]_M \;=\; \emptyset$
- $[\![\neg\phi]\!]_M \;=\; \mathcal{S} \setminus [\![\phi]\!]_M$
- $[\![\phi_1 \wedge \phi_2]\!]_M \;=\; [\![\phi_1]\!]_M \cap [\![\phi_2]\!]_M$
- $[\![\phi_1 \vee \phi_2]\!]_M \;=\; [\![\phi_1]\!]_M \cup [\![\phi_2]\!]_M$
- $[\![\mathsf{EX}\,\phi]\!]_M \;=\; \mathit{PreImage}([\![\phi]\!]_M, \mathcal{R})$

Now, define function $a_{\mathcal{R}} : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ as

$$a_{\mathcal{R}}(\mathcal{X}) \;=\; \{s : \forall (s, s') \in \mathcal{R}, s' \in \mathcal{X}\}$$

and note that

- $[\![\mathsf{AX}\,\phi]\!]_M \;=\; a_{\mathcal{R}}([\![\phi]\!]_M)$.

Property 4.19 *For any $\mathcal{R}$, function $a_{\mathcal{R}}$ is monotonic.*

Proof: *Suppose $\mathcal{X} \subseteq \mathcal{Y}$, and consider $s \in a_{\mathcal{R}}(\mathcal{X})$. Then we must have $\forall (s, s') \in \mathcal{R}$, $s' \in \mathcal{X}$. Since $\mathcal{X} \subseteq \mathcal{Y}$, we also have $\forall (s, s') \in \mathcal{R}$, $s' \in \mathcal{Y}$. Therefore, we must have $s \in a_{\mathcal{R}}(\mathcal{Y})$.*

4.7.2 Fixed points and AF

Property 4.20 *For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ and any state formula ϕ, define $F_\phi(\mathcal{X}) = [\![\phi]\!] \cup a_{\mathcal{R}}(\mathcal{X})$. Then $[\![\mathsf{AF}\,\phi]\!]$ is the least fixed point of F_ϕ.*

Proof: *First, note that when $\mathcal{X} \subseteq \mathcal{Y}$, we have $[\![\phi]\!] \cup a_{\mathcal{R}}(\mathcal{X}) \subseteq [\![\phi]\!] \cup a_{\mathcal{R}}(\mathcal{Y})$ because $a_{\mathcal{R}}$ is monotonic. Therefore, F_ϕ is monotinic.*

Now, using Property 4.8, we have

$$\begin{aligned}
[\![\mathsf{AF}\,\phi]\!] &= [\![\phi \vee \mathsf{AX}\,\mathsf{AF}\,\phi]\!] \\
&= [\![\phi]\!] \cup [\![\mathsf{AX}\,\mathsf{AF}\,\phi]\!] \\
&= [\![\phi]\!] \cup a_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \\
&= F_\phi([\![\mathsf{AF}\,\phi]\!])
\end{aligned}$$

and therefore $[\![\mathsf{AF}\,\phi]\!]$ is a fixed point of F_ϕ.

Finally, we show that it is the least *fixed point. Let $\mathcal{X}$ be some fixed point, and we will prove by contradiction that $[\![\mathsf{AF}\,\phi]\!] \subseteq \mathcal{X}$. Consider some $s \in [\![\mathsf{AF}\,\phi]\!] = [\![\phi]\!] \cup a_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!])$, and suppose $s \notin \mathcal{X}$. Because $\mathcal{X} = F_\phi(\mathcal{X}) = [\![\phi]\!] \cup a_{\mathcal{R}}(\mathcal{X})$, we have*

$$s \in [\![\phi]\!] \cup a_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \qquad \wedge \qquad s \notin [\![\phi]\!] \cup a_{\mathcal{R}}(\mathcal{X})$$

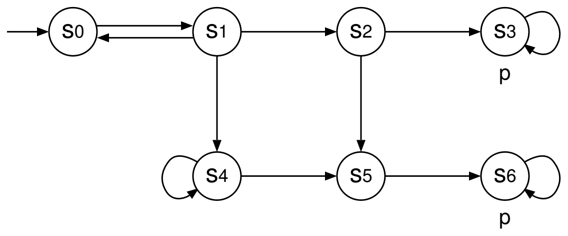
which implies

$$s \in a_{\mathcal{R}}([\![\mathsf{AF}\,\phi]\!]) \qquad \wedge \qquad s \notin [\![\phi]\!] \qquad \wedge \qquad s \notin a_{\mathcal{R}}(\mathcal{X}).$$

Now, $s \notin a_{\mathcal{R}}(\mathcal{X})$ implies that, for some edge $(s, s') \in \mathcal{R}$, $s' \notin \mathcal{X}$. But we again obtain that $s' \notin [\![\phi]\!]$ and there is some edge $(s', s'') \in \mathcal{R}$ with $s'' \notin \mathcal{X}$. Repeating this argument forever, we can obtain an infinite path $(s, s', s'', \ldots)$ where none of the states satisfy ϕ. But this is impossible if $s \models \mathsf{AF}\,\phi$. We have a contradiction, and our assumption $s \notin \mathcal{X}$ must be false, and therefore $s \in \mathcal{X}$.

Example 4.16

Show that $\mathcal{X} = \{s_2, s_3, s_4, s_5, s_6\}$ is a fixed point for F_p for the following Kripke structure:



(recall that $[\![\mathsf{AF}\,p]\!] = \{s_2, s_3, s_5, s_6\}$).

Solution:

$$\begin{aligned}
F_p(\{s_2, s_3, s_4, s_5, s_6\}) &= [\![p]\!] \cup a_{\mathcal{R}}(\{s_2, s_3, s_4, s_5, s_6\}) \\
&= \{s_3, s_6\} \cup \{s_2, s_3, s_4, s_5, s_6\} \\
&= \{s_2, s_3, s_4, s_5, s_6\}
\end{aligned}$$

Example 4.17

What is the greatest fixed point of F_ϕ?

Solution: $F_\phi(\mathcal{S}) \;=\; [\![\phi]\!] \cup a_\mathcal{R}(\mathcal{S}) \;=\; [\![\phi]\!] \cup \mathcal{S} \;=\; \mathcal{S}$. Therefore, $\mathcal{S}$ is the greatest fixed point of F_ϕ.

Property 4.21

$$\mathsf{AF}\,\mathsf{AF}\,\phi \quad \equiv \quad \mathsf{AF}\,\phi$$

Proof: *From Property 4.20, $[\![\mathsf{AF}\,\phi]\!]$ is the least fixed point of $F_\phi(\mathcal{X}) = [\![\phi]\!] \cup a_\mathcal{R}(\mathcal{X})$, which implies $[\![\phi]\!] \cup a_\mathcal{R}([\![\mathsf{AF}\,\phi]\!]) = [\![\mathsf{AF}\,\phi]\!]$ and therefore $a_\mathcal{R}([\![\mathsf{AF}\,\phi]\!]) \subseteq [\![\mathsf{AF}\,\phi]\!]$. Now, consider the least fixed point of $F_{\mathsf{AF}\,\phi}(\mathcal{X}) = [\![\mathsf{AF}\,\phi]\!] \cup a_\mathcal{R}(\mathcal{X})$:*

$$\begin{aligned} F_{\mathsf{AF}\,\phi}(\emptyset) &= [\![\mathsf{AF}\,\phi]\!] \cup a_\mathcal{R}(\emptyset) &&= [\![\mathsf{AF}\,\phi]\!] \\ F^2_{\mathsf{AF}\,\phi}(\emptyset) &= [\![\mathsf{AF}\,\phi]\!] \cup a_\mathcal{R}([\![\mathsf{AF}\,\phi]\!]) &&= [\![\mathsf{AF}\,\phi]\!] \end{aligned}$$

It follows that the least fixed point of $F_{\mathsf{AF}\,\phi}$ is $[\![\mathsf{AF}\,\phi]\!]$. But Property 4.20 says $[\![\mathsf{AF}\,\mathsf{AF}\,\phi]\!]$ is the least fixed point of $F_{\mathsf{AF}\,\phi}$. Therefore, $[\![\mathsf{AF}\,\mathsf{AF}\,\phi]\!] = [\![\mathsf{AF}\,\phi]\!]$.

4.7.3 Fixed points and EG

Property 4.22 *For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ and any state formula ϕ, define $G_\phi(\mathcal{X}) = [\![\phi]\!] \cap PreImage(\mathcal{X}, \mathcal{R})$. Then $[\![\mathsf{EG}\,\phi]\!]$ is the greatest fixed point of G_ϕ.*

Proof: *First, note that G_ϕ is monotonic since PreImage is monotonic. Using Property 4.9, we have*

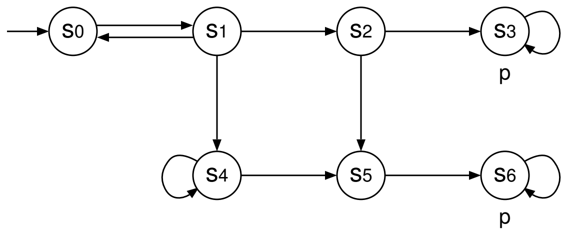
$$\begin{aligned} [\![\mathsf{EG}\,\phi]\!] &= [\![\phi \wedge \mathsf{EX}\,\mathsf{EG}\,\phi]\!] \\ &= [\![\phi]\!] \cap [\![\mathsf{EX}\,\mathsf{EG}\,\phi]\!] \\ &= [\![\phi]\!] \cap PreImage([\![\mathsf{EG}\,\phi]\!], \mathcal{R}) \\ &= G_\phi([\![\mathsf{EG}\,\phi]\!]) \end{aligned}$$

and therefore $[\![\mathsf{EG}\,\phi]\!]$ is a fixed point of G_ϕ.

Finally, we must show that it is the greatest *fixed point. Let $\mathcal{X}$ be some fixed point of G_ϕ. Since $\mathcal{X} = G_\phi(\mathcal{X})$, it follows that $\mathcal{X} \subseteq [\![\phi]\!]$. Furthermore, we have $\mathcal{X} \subseteq PreImage(\mathcal{X}, \mathcal{R})$, which says that, from any state $s \in \mathcal{X}$, there is an edge $(s, s') \in \mathcal{R}$ with $s' \in \mathcal{X}$. Therefore, for each state $s \in \mathcal{X}$, there exists a path which remains forever in the states in $\mathcal{X}$, which all satisfy ϕ. Thus, all states in $\mathcal{X}$ satisfy $\mathsf{EG}\,\phi$, and therefore $\mathcal{X} \subseteq [\![\mathsf{EG}\,\phi]\!]$.*

Example 4.18

Show that $\mathcal{X} = \{s_0, s_1\}$ is a fixed point for $G_{\neg p}$ for the following Kripke structure:



(recall that $[\![\mathsf{EG}\,\neg p]\!] = \{s_0, s_1, s_4\}$).

Solution:

$$\begin{aligned}
G_{\neg p}(\{s_0, s_1\}) &= [\![\neg p]\!] \cap \mathit{PreImage}(\{s_0, s_1\}) \\
&= \{s_0, s_1, s_2, s_4, s_5\} \cap \{s_0, s_1\} \\
&= \{s_0, s_1\}
\end{aligned}$$

Example 4.19

What is the least fixed point of G_ϕ?

Solution: $G_\phi(\emptyset) = [\![\phi]\!] \cap \mathit{PreImage}(\emptyset, \mathcal{R}) = [\![\phi]\!] \cap \emptyset = \emptyset$. Therefore, $\emptyset$ is the least fixed point of G_ϕ.

Property 4.23

$$\mathsf{EG}\,\mathsf{EG}\,\phi \equiv \mathsf{EG}\,\phi$$

4.7.4 Fixed points and EU

Property 4.24 *For any Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ and any state formulas ϕ_1 and ϕ_2, define $U_{\phi_1\phi_2}(\mathcal{X}) = [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathit{PreImage}(\mathcal{X}, \mathcal{R}))$. Then $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$ is the least fixed point of $U_{\phi_1\phi_2}$.*

Proof: *First, note that $U_{\phi_1\phi_2}$ is monotonic. From Property 4.11, we have*

$$\begin{aligned}
[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!] &= [\![\phi_2 \vee (\phi_1 \wedge \mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2)]\!] \\
&= [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap [\![\mathsf{EX}\,\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]) \\
&= [\![\phi_2]\!] \cup ([\![\phi_1]\!] \cap \mathit{PreImage}([\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!], \mathcal{R}) \\
&= U_{\phi_1\phi_2}([\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!])
\end{aligned}$$

and therefore $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$ is a fixed point of $U_{\phi_1\phi_2}$.

Finally, we must show that it is the least *fixed point by showing that $[\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!] \subseteq \mathcal{X}$. Consider some $s \in [\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$. By definition, this says there exists a $j \geq 0$ and a path $(s_0 = s, s_1, s_2, \ldots)$ such that*

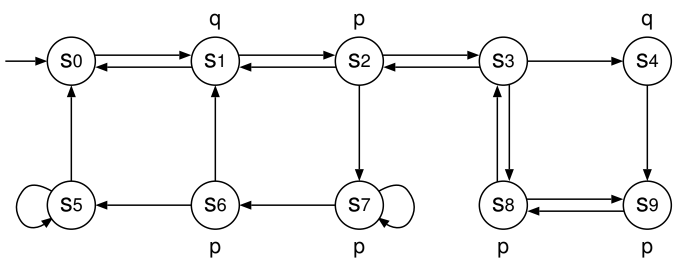
- *$s_j \models \phi_2$, and*
- *$\forall i < j, s_i \models \phi_1$.*

We show that $s \in \mathcal{X}$ by induction on j. In the base case, $j = 0$ and $s \models q$. But by definition of $U_{\phi_1\phi_2}$ and the fact that $\mathcal{X} = U_{\phi_1\phi_2}(\mathcal{X})$, we know $[\![\phi_2]\!] \subseteq \mathcal{X}$ and trivially $s \in \mathcal{X}$.

Now, assume it holds for $j \leq k$ and prove it holds for $j = k+1$. Considering the path $(s'_0 = s_1, s'_1 = s_2, \ldots, s'_k = s_{k+1})$, we have $s_1 \in [\![\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2]\!]$, and by the inductive hypothesis (since this path uses $j = k$), we have $s_1 \in \mathcal{X}$. But then $s_0 = s \in \mathcal{X}$, because $s_0 \in [\![\phi_1]\!]$ and $s_0 \in \mathit{PreImage}(\mathcal{X}, \mathcal{R})$.

Example 4.20

Show that $\mathcal{X} = \{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}$ is a fixed point for U_{pq} for the following Kripke structure:



(recall that $[\![\mathsf{E}\,p\,\mathsf{U}\,q]\!] = \{s_1, s_2, s_4, s_6, s_7\}$)

Solution:

$$\begin{aligned}
U_{pq}(\mathcal{X}) &= [\![q]\!] \cup ([\![p]\!] \cap \mathit{PreImage}(\{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}, \mathcal{R}) \\
&= \{s_1, s_4\} \cup (\{s_2, s_6, s_7, s_8, s_9\} \cap \{s_0, s_1, s_2, s_3, s_4, s_6, s_7, s_8, s_9\}) \\
&= \{s_1, s_2, s_4, s_6, s_7, s_8, s_9\}
\end{aligned}$$

Property 4.25

$$\mathsf{E}\,\phi_1\,\mathsf{U}\,(\mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2) \quad \equiv \quad \mathsf{E}\,\phi_1\,\mathsf{U}\,\phi_2$$

Property 4.26

$$\begin{aligned}
\mathsf{EF}\,\mathsf{EF}\,\phi &\equiv \mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,(\mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,\phi) \\
&\equiv \mathsf{E}\,\mathtt{tt}\,\mathsf{U}\,\phi \\
&\equiv \mathsf{EF}\,\phi
\end{aligned}$$

Property 4.27

$$\begin{aligned}
\mathsf{AG}\,\mathsf{AG}\,\phi &\equiv \mathsf{AG}\,\neg\mathsf{EF}\,\neg\phi \\
&\equiv \neg\mathsf{EF}\,\mathsf{EF}\,\neg\phi \\
&\equiv \neg\mathsf{EF}\,\neg\phi \\
&\equiv \mathsf{AG}\,\phi
\end{aligned}$$

4.8 Fairness

Suppose we have a system of three interacting processes, where each process

1. computes,
2. waits for the semaphore,
3. updates a shared data structure,
4. releases the semaphore,

and repeats those steps forever. Suppose we model the choice of "who gets the semaphore" when multiple processes are waiting as a non-deterministic choice. Now, suppose we would like to check that a process can always eventually enter the critical section, something like

$$\mathsf{EG}(\text{"process one waits"} \rightarrow \mathsf{AF}\,\text{"processs one updates"}).$$

It is likely that this property *will not* hold, because there will be a computation path where the semaphore alternates between processes two and three, and process one will wait forever. The problem here is that we would like to have an additional constraint, namely, that the semaphore operates "fairly". In CTL, the issue of fairness can be addressed using *fairness constraints*, which are sometimes[3] simply sets of states.

Definition 4.28 *A* fair path for constraint $\mathcal{C}$ *is a path* $(s_0, s_1, s_2, \ldots)$ *such that some states in* $\mathcal{C}$ *are visited infinitely often. Formally, there is an infinite set* $\{i_1, i_2, \ldots\}$ *where, for all* k, $s_{i_k} \in \mathcal{C}$.

We then treat the issue of fairness by quantifying over the *fair paths*, i.e.,

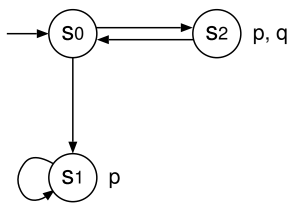
- $\mathsf{A}_{\mathcal{C}}$: for all fair paths for constraint $\mathcal{C}$.
- $\mathsf{E}_{\mathcal{C}}$: for some fair path for constraint $\mathcal{C}$.

For example, we have:

$M, s \models \mathsf{A}_{\mathcal{C}}\mathsf{X}\,\phi$, if and only if, for all fair paths $(s_0 = s, s_1, \ldots)$ for constraint $\mathcal{C}$, $s_1 \models \phi$.

Example 4.21

Consider the following Kripke structure:



For the constraint $\mathcal{C} = [\![\neg p]\!] = \{s_0\}$, we have

$$M, s_0 \models \mathsf{A}_{\mathcal{C}}\mathsf{X}\,(p \wedge q)$$

because the only fair path for constaint $\mathcal{C}$ is $(s_0, s_2, s_0, s_2, \ldots)$.

We only need the operator $\mathsf{E}_{\mathcal{C}}\mathsf{G}$, because all other "fair" operators can be expressed in terms of their unfair conterparts, and $\mathsf{E}_{\mathcal{C}}\mathsf{G}$. Specifically, we will use $\mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}$ for "there exists a fair path for constraint $\mathcal{C}$".

1. $\mathsf{E}_{\mathcal{C}}\mathsf{X}\,\phi \quad \equiv \quad \mathsf{EX}(\phi \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$
2. $\mathsf{E}_{\mathcal{C}}\,\phi_1\,\mathsf{U}\,\phi_2 \quad \equiv \quad \mathsf{E}(\phi_1\,\mathsf{U}\,(\phi_2 \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}))$

Since EX, EU, and EG are an adequate set of temporal operators, we are done.

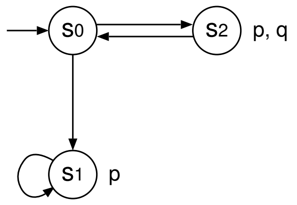
[3]There are other ways to specify fairness constraints that are also useful. We will see a different one, later.

4.8.1 Labeling algorithm for $\mathsf{E}_{\mathcal{C}}\mathsf{G}$

Recall that one algorithm for labeling $\phi = \mathsf{EG}\,\phi_1$ was to use strongly connected components (SCCs), based on the observation that, for ϕ_1 to occur in every state on an infinitely–long path, we need to have cycles of states satisfying ϕ_1. We only need a small modification if we want an infinitely–long *fair* path: we need to have SCCs where at least one state in the SCC is in set $\mathcal{C}$. We have the following algorithm.

1. Create a new graph M' by removing all states that do not satisfy ϕ_1, and any edges associated with those states.
2. Determine the SCCs for M'.
3. For each SCC **that contains at least one state in** $\mathcal{C}$,
   - if the SCC contains more than one state, then label all states in the SCC with ϕ
   - if the SCC contains only one state, and the state has an edge to itself, then label it with ϕ.
4. Any state that can reach (in M') a state labeled with ϕ should also be labeled with ϕ.

Example 4.22



For the above Kripke structure and constraint $\mathcal{C} = [\![\neg p]\!] = \{s_0\}$, which states satisfy $\mathsf{A}_{\mathcal{C}}\mathsf{X}(p \wedge q)$?

Solution: We have the equivalence

$$\neg\mathsf{A}_{\mathcal{C}}\mathsf{X}p \;\equiv\; \mathsf{E}_{\mathcal{C}}\mathsf{X}\neg p \;\equiv\; \mathsf{EX}(\neg p \wedge \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$$

which says that, $\mathsf{A}_{\mathcal{C}}\mathsf{X}\,p$ does not hold if there is a fair path with a next state satisfying $\neg p$. Rewriting our formula, we obtain

$$\mathsf{A}_{\mathcal{C}}\mathsf{X}(p \wedge q) \;\equiv\; \neg\mathsf{EX}(\neg(p \wedge q) \;\wedge\; \mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt})$$

Using the labeling algorithm for $\mathsf{E}_{\mathcal{C}}\mathsf{G}$, we determine SCCs and check for states in $\mathcal{C}$:

- SCC $\{s_0, s_2\}$ has one state, s_0, in $\mathcal{C}$
- SCC $\{s_1\}$ has no states in $\mathcal{C}$

Therefore, $[\![\mathsf{E}_{\mathcal{C}}\mathsf{G}\,\mathtt{tt}]\!] = \{s_0, s_2\}$. Also, we have $[\![\neg(p \wedge q)]\!] = \{s_0, s_1\}$. We next determine

$$\mathit{PreImage}(\{s_0, s_1\} \cap \{s_0, s_2\}, \mathcal{R}) \;=\; \mathit{PreImage}(\{s_0\}, \mathcal{R}) \;=\; \{s_2\}$$

Finally, taking the complement, we have

$$[\![\mathsf{A}_{\mathcal{C}}\mathsf{X}(p \wedge q)]\!] \;=\; \{s_0, s_1\}$$

Verify: We already determined $s_0 \in [\![\mathsf{A}_{\mathcal{C}}\mathsf{X}(p \wedge q)]\!]$ in the previous example. But what about s_1? This is a correct solution because *there are no fair paths from* s_1. Thus, the statement, "for all fair paths, the next state satisfies $p \wedge q$" is trivially true. If this is not as intended, then the issue is how to define $\mathsf{A}_{\mathcal{C}}$ when there are no fair paths. If the intended definition is, "there are some fair paths, and for all of them ...", then the equivalence

$$\neg\mathsf{A}_{\mathcal{C}}\mathsf{X}p \;\equiv\; \mathsf{E}_{\mathcal{C}}\mathsf{X}\neg p$$

does *not* hold.

Chapter 5

High–level Formalisms

We can verify properties of a system via the following steps.

1. Construct an appropriate Kripke structure.

2. Express the properties we want in a suitable logic.

3. Use an automatic tool to model check the Kripke structure against the properties.

We have seen how to do steps 2 and 3 above. What about step 1? This can be done in a few ways:

- By hand. That's what we have done so far, in class. However, step 1 can be difficult if the system is large and/or complex.

- Write a program to generate a Kripke structure for a particular system, and analyze it. While this works, for each new system to analyze, you need to write a new program, which means more time spent debugging.

- Use another, more compact and convenient model to describe the system. For this to be effective, we must be able to automatically construct a Kripke structure described by the model. That way, we can write, debug, test, etc., *one* program that reads a model as input and constructs its underlying Kripke structure.

  These compact models are called **high–level formalisms**.

5.1 Requirements of a formalism

What does a high–level formalism need to be able to do? For model checking, it must

1. Describe a finite set of states.

2. Provide the initial state(s) of the system.

3. Provide *formal* rules (thus the name "formalism") for changing states. I.e., if we are currently in state s, what state(s) can be reached from here in one step?

4. Be easier to use than describing a Kripke structure by hand. This is simply a practical requirement. This is the "high–level" part of "high–level formalisms".

Note that the first three requirements will allow us to construct a Kripke structure. We then write queries in terms of the high–level formalism. In particular, the atomic propositions will be expressed in terms of features in the high–level model.

There are many high–level formalisms. Why? If you think of a high–level formalism as a high–level language, then it is like the several high–level programming languages:

- certain tasks are easier in certain languages
- people have preferences
- they are easy to invent for a particular application

For us, the choice of formalism is mostly irrelevant. (Think of writing a compiler; this task is basically the same for a C compiler as for a Pascal compiler.)

5.2 Petri nets

Petri nets are one of many high–level formalisms. We will study these because

- They are graphical (easier to learn?)
- They are fairly powerful and expressive
- They have a rich underlying theory[1]
- They are fairly well–known
- Analysis is relatively easy (more features usually implies more difficult to analyze)

5.2.1 Informal introduction

A Petri net is a directed graph with two types of nodes:

Places, drawn as circles, which contain a non-negative integer number of **tokens**.

Transitions, drawn as rectangles or bars, which cause the tokens to "move".

Arcs connect places to transitions and transitions to places, never places to places or transitions to transitions. (This is a special type of graph called *bipartite*.)

So, let's see how Petri nets are a high-level formalism by checking the list of requirements:

1. How does a Petri net describe a discrete set of states?

   Each place can contain a non-negative integer number of tokens. The tokens cannot be distinguished. The state of the Petri net is completely described by its **marking**, which determines, for each place, how many tokens it contains. So, if $\mathcal{P}$ is the set of places, a marking $\mathbf{m}$ is a function $\mathbf{m} : \mathcal{P} \to \mathbb{N}$. Or, equivalently, a marking $\mathbf{m}$ is a vector of naturals, $\mathbf{m} \in \mathbb{N}^{|\mathcal{P}|}$, where $\mathbf{m}[p]$ is the number of tokens in place p. So, at most, the discrete set of states described by a Petri net is the set $\mathbb{N}^{|\mathcal{P}|}$.

[1] T. Murata. "Petri Nets: Properties, Analysis, and Applications", in *Proceedings of the IEEE*, 77 (4), April 1989, pages 541–580

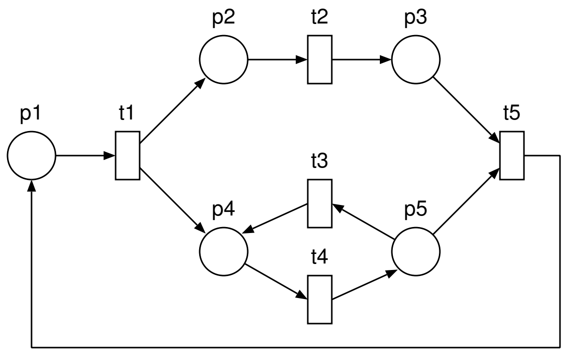
2. How to specify the initial state of a Petri net?

   This is done by giving the initial marking of the Petri net.

3. What are the rules for changing states in a Petri net?

   The marking of the net is changed by transitions. A transition is said to be **enabled** if all of its input places (the places with an arc to this transition) have at least one token. An enabled transition may **fire** by removing a token from each input place and adding a token to each output place. Note that tokens are not necessarily "conserved".

Example 5.1



$$\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$$

$$\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5\}$$

In the above example Petri net, suppose the current marking is

$$\begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \\ \left[\ 3 \right. & 0 & 0 & 0 & \left. 0\ \right] \end{array}$$

Which transitions are enabled in this marking?

t_1**: enabled** because its input places (p_1) contain tokens.

t_2**: disabled** because its input places (p_2) do not contain tokens.

t_3**: disabled** because its input places (p_5) do not contain tokens.

t_4**: disabled** because its input places (p_4) do not contain tokens.

t_5**: disabled** because its input places (p_3, p_5) do not contain tokens.

If t_1 fires, what is the new marking?

- Remove 1 token from p_1
- Add 1 token to p_2
- Add 1 token to p_4

This gives a new marking of

$$\begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \\ \left[\ 2 \right. & 1 & 0 & 1 & \left. 0\ \right] \end{array}$$

In the new marking, transitions $\{t_1, t_2, t_4\}$ are enabled. Any of these may fire. If t_1 fires again, the new marking is

$$\begin{array}{ccccccc} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ [& 1 & 2 & 0 & 2 & 0 &] \end{array}$$

and now transitions $\{t_1, t_2, t_4\}$ are enabled. If t_4 fires, the new marking is

$$\begin{array}{ccccccc} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ [& 1 & 2 & 0 & 1 & 1 &] \end{array}$$

and now transitions $\{t_1, t_2, t_3, t_4\}$ are enabled. If t_2 fires, the new marking is

$$\begin{array}{ccccccc} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ [& 1 & 1 & 1 & 1 & 1 &] \end{array}$$

and now all transitions are enabled. Note: in this marking,

- if t_3 fires, then t_5 will become disabled.
- if t_5 fires, then t_3 will become disabled.

This is called "conflict".

The above example illustrates that Petri nets can model the following.

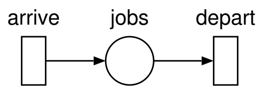
conflict: firing of one transition may disable another (e.g., t_3 and t_5).

choice: one place (e.g., p_1) as input to multiple transitions.

concurrency: firing of one transition does not affect firing of the other (e.g, t_2 and t_4).

synchronization: one transition (e.g., t_5) with multiple input places.

Example 5.2



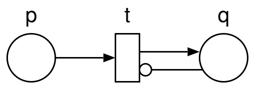
The above Petri net models a single–server service node with an unbounded queue. The number of tokens in place *jobs* is the number of customers in the node.

- Transition *arrive* has no input places; therefore it is always enabled, and it "creates" tokens. This is called a *source* transition.
- Transition *depart* has no output places; it "consumes" tokens. This is called a *sink* transition.

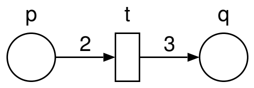
5.2.2 Petri net extensions

The Petri nets described above are "ordinary" Petri nets. Several extensions have been proposed (these are commonly used):

Inhibitor arcs: these allow the presence of tokens in places to disable a transition. Below, t is disabled if there is a token in q.



Arc weights: these specify a number of tokens to be added/removed or considered to inhibit a transition. Below, t is enabled iff p has at least 2 tokens; firing of t removes 2 tokens from p and adds 3 tokens to q.

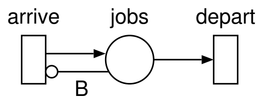


Transition guards: these are functions that must be true for a transition to be enabled.

Marking–dependent arc weights: arc weights can be functions of the current marking, instead of constants. These are also known as "self-modifying nets".

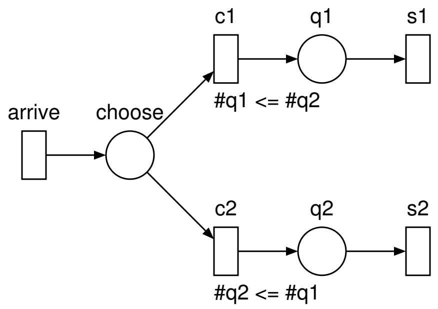
Color: Tokens have colors (called "Colored Petri Nets"). Colors are used to store data (i.e., distinguish tokens). There can be multiple "dimensions" of colors. This makes things *significantly* more complex, so we will ignore colors and study "uncolored" nets for a while.

Example 5.3



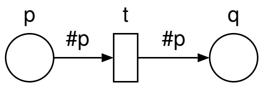
The above Petri net models a single–server service node with a bounded queue. For the given initial marking, the maximum number of tokens in place *jobs* is B, since transition *arrive* becomes disabled when there are B (or more) tokens in place *jobs*. If an initial marking specifies more than B tokens in place *jobs*, then transition *arrive* will remain disabled until transition *depart* fires enough times that fewer than B tokens appear in place *jobs*.

Example 5.4



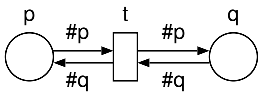
The above Petri net models the arrival of customers, where each customer joins either queue 1 or queue 2, based on which is shorter. This is done via the guard $\#q_1 \leq \#q_2$ on transition c_1, and the guard $\#q_2 \leq \#q_1$ on transition c_2, where $\#p$ means the number of tokens currently in place p. Therefore, c_1 is enabled only if the number of tokens in q_1 is not more than the number of tokens in q_2, and vice–versa. Note that if the queues have equal length, then both c_1 and c_2 are enabled.

Example 5.5



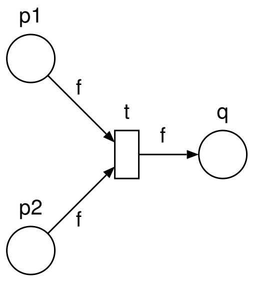
We can remove all tokens from a place, and add them to another place, using marking–dependent arc cardinalities, as above. When t fires, all tokens are removed from p and added to q. Note that, technically, t is enabled even when p is empty, but its firing in this case does not change the marking.

Example 5.6



We can exchange the number of tokens in two places, using marking–dependent arc cardinalities as above. When t fires, all tokens are removed from p and q, and the (previous) number of tokens in p is added to q, and the (previous) number of tokens in q is added to p.

Example 5.7



The above can be used to synchronize f jobs. If $f = 1$, then transition t is "ordinary" synchronization. If $f = 2$, then transition t synchronizes completing pairs of tasks, two at a time. If $f = \min(\#p_1, \#p_2)$, then transition t synchronizes as many completing pairs of tasks as possible.

5.2.3 Expressive power of Petri nets

Suppose we assign a symbol to each transition, and the firing of that transition produces the symbol. Then we can ask "what is the language generated by the Petri net"? Or, equivalently, a word is "accepted" if there exists a firing sequence that generates that word. The expressive power can be measured by answering the question, "what types of languages are accepted by Petri nets"? It turns out:

- Petri nets with inhibitor arcs are Turing equivalent.
- Ordinary Petri nets without inhibitor arcs are **not** Turing equivalent.
- Guards, marking-dependent weights, and inhibitor arcs are "equivalent" extensions, any of them will give you Turing equivalence.

However, if the number of tokens in each place is bounded (and therefore, the total number of markings that can be reached from the initial marking is *finite*), then the Petri net describes a finite state machine (regardless of which extensions are allowed), and the above extensions serve only to simplify the model description.

5.2.4 Formal definition

A Petri net is a tuple $(\mathcal{P}, \mathcal{T}, I, O, H, g, \mathbf{m}_0)$ where

- $\mathcal{P}$ is a finite set of places.
- $\mathcal{T}$ is a finite set of transitions, with $\mathcal{T} \cap \mathcal{P} = \emptyset$
- $I : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$ is a function to describe the marking-dependent input arc cardinalities
- $O : \mathcal{T} \times \mathcal{P} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$ is a function to describe the marking-dependent output arc cardinalities
- $H : \mathcal{P} \times \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N} \cup \{\infty\}$ is a function to describe the marking-dependent inhibitor arc cardinalities
- $g : \mathcal{T} \times \mathbb{N}^{|\mathcal{P}|} \to \{0, 1\}$ are the transition guard functions
- $\mathbf{m}_0 \in \mathbb{N}^{|\mathcal{P}|}$ is the initial marking

Definition 5.1 *(Enabling rule) Transition t is enabled in marking $\mathbf{m}$ if:*

$$g(t, \mathbf{m}) = true \qquad and \qquad \forall p \in \mathcal{P}, \quad I(p, t, \mathbf{m}) \leq \mathbf{m}[p] < H(p, t, \mathbf{m})$$

Definition 5.2 *(Firing rule) If transition t is enabled in marking $\mathbf{m}$, its firing leads to a new marking $\mathbf{n}$, with*

$$\forall p \in \mathcal{P}, \qquad \mathbf{n}[p] = \mathbf{m}[p] - I(p, t, \mathbf{m}) + O(t, p, \mathbf{m})$$

Chapter 6

Reachability

We now consider a fundamental question for all high–level formalisms, but phrase the question in terms of Petri nets.

The reachability problem: Given a Petri net with an initial marking $\mathbf{m}_0$, is it possible to (eventually) reach a specified marking $\mathbf{m}$?

This problem is known to be decidable[1], but requires exponential space and time in the worst case. We will look instead at a few "simple" algorithms.

6.1 Coverability

Definition 6.1 *Given a Petri net, a marking* $\mathbf{m}'$ covers *another marking* $\mathbf{m}$ *if*

$$\mathbf{m}'[p] \geq \mathbf{m}[p]$$

for all places $p \in \mathcal{P}$*.*

Note that a marking covers itself.

We will write $\mathbf{m}' \geq \mathbf{m}$ if $\mathbf{m}'$ covers $\mathbf{m}$.
We will write $\mathbf{m}' > \mathbf{m}$ if $\mathbf{m}'$ covers $\mathbf{m}$ and $\mathbf{m}' \neq \mathbf{m}$.

Definition 6.2 *Given a Petri net, a marking* $\mathbf{m}$ *is said to be* coverable *if there exists a marking* $\mathbf{m}' \geq \mathbf{m}$ *that is reachable from the initial marking* $\mathbf{m}_0$*.*

6.1.1 The coverability tree

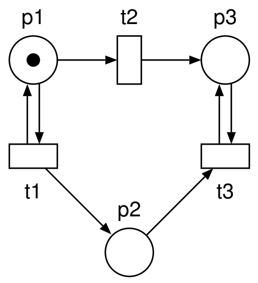
The *coverability tree* is a tree of markings that cover all markings reachable from the initial marking. Within these markings, we use a special symbol, ω, to indicate "unbounded number of tokens". For any integer n, we have $\omega > n$, $\omega + n = \omega - n = \omega$, and $\omega \geq \omega$. The coverability tree can be constructed using the following algorithm:

[1] I don't know which of the Petri net extensions are allowed in this decidability result.

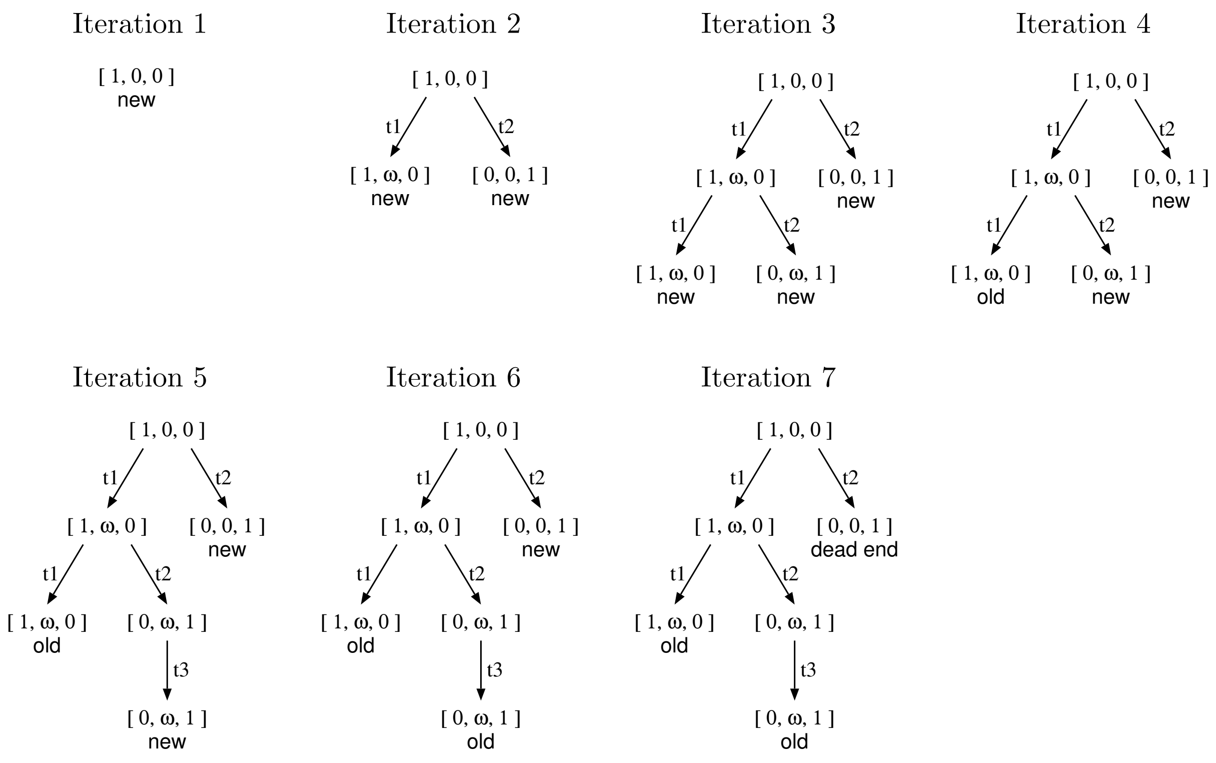
Algorithm 6.1 Coverability Tree Construction

```
Set $\mathbf{m}_0$ as the root of the tree; tag it as "new".
while $\exists$ "new" markings
    Select some "new" marking $\mathbf{m}$;
    Give $\mathbf{m}$ a blank tag;
    if $\exists \mathbf{m}'$, on the path from $\mathbf{m}_0$ to $\mathbf{m}$, with $\mathbf{m}' = \mathbf{m}$ then
        Tag $\mathbf{m}$ as "old";
        continue;
    endif
    if no transitions are enabled in $\mathbf{m}$ then
        Tag $\mathbf{m}$ as "dead end";
        continue;
    endif
    for each transition $t$ enabled in $\mathbf{m}$ do
        $\mathbf{m}' \leftarrow$ the marking reached from $\mathbf{m}$ by firing $t$
        if $\exists \mathbf{m}''$ on the path from $\mathbf{m}_0$ to $\mathbf{m}'$ with $\mathbf{m}' > \mathbf{m}''$
            for each $p$ such that $\mathbf{m}'[p] > \mathbf{m}''[p]$ do
                $\mathbf{m}'[p] \leftarrow \omega$;
            endfor
        endif
        $\mathbf{m}'$ is a new node in the tree;
        Draw an arc from $\mathbf{m}$ to $\mathbf{m}'$ with label $t$;
        Tag $\mathbf{m}'$ as "new";
    endfor
endwhile
```

Example 6.1



Consider the above Petri net, drawn with initial marking $[1, 0, 0]$. Note: t_1 can fire arbitrarily many times, until t_2 fires; then, t_3 can fire as many times as t_1 fired. Using the above algorithm, we can construct the coverability tree, drawn below at the beginning of each iteration of the while loop.



6.1.2 The coverability graph

The *coverability graph* is similar to the coverability tree. Nodes in the coverability graph correspond to the unique markings in the coverability tree. In the coverability graph, there is an edge from $\mathbf{m}$ to $\mathbf{m}'$, labeled with transition t, if

- t is enabled in marking $\mathbf{m}$, and
- the firing of t in $\mathbf{m}$ leads to marking $\mathbf{m}''$ with $\mathbf{m}' \geq \mathbf{m}''$.

Example 6.2

The coverability graph for the previous example is shown below.

$$\begin{tikzcd}
 & {[1,0,0]} \arrow[dl, "t1"'] \arrow[dr, "t2"] & \\
{[1,\omega,0]} \arrow[loop left, "t1"] \arrow[dr, "t2"] & & {[0,0,1]} \\
 & {[0,\omega,1]} \arrow[loop right, "t3"] &
\end{tikzcd}$$

The coverability graph (or tree) does *not* contain enough information to definitively solve the reachability problem:

- A marking $\mathbf{m}$ is *not* reachable if there is no marking $\mathbf{m}' \geq \mathbf{m}$ in the coverability graph.
- A marking $\mathbf{m}$ *may be* reachable if there is a marking $\mathbf{m}' \geq \mathbf{m}$ in the coverability graph.

The inability to say for certain that a marking is reachable is due to the loss of information that occurs due to the symbol ω.

6.2 The reachability graph

We can simplify the coverability graph algorithm by eliminating the symbol ω, and storing the actual markings reached. This gives us instead the *reachability graph*, whose set of nodes is equal to the markings that can be reached from the initial marking $\mathbf{m}_0$ in zero or more steps. This set of markings is called the *reachability set*. The reachability graph can be constructed using the following algorithm.

Algorithm 6.2 Reachability graph construction

```
$\mathcal{U} \leftarrow \{\mathbf{m}_0\}$;          // set of unexplored markings
$\mathcal{S} \leftarrow \{\mathbf{m}_0\}$;          // reachability set so far
$\mathcal{E} \leftarrow \emptyset$;                 // edges in the reachability graph
while $\mathcal{U} \neq \emptyset$
    Remove some marking $\mathbf{m}$ from $\mathcal{U}$;
    for each transition $t$ enabled in $\mathbf{m}$ do
        $\mathbf{m}' \leftarrow$ the marking reached from $\mathbf{m}$ by firing $t$;
        if $\mathbf{m}' \notin \mathcal{S}$ then
            Add $\mathbf{m}'$ to $\mathcal{S}$;
            Add $\mathbf{m}'$ to $\mathcal{U}$;
        endif
        Add edge from $\mathbf{m}$ to $\mathbf{m}'$ with label $t$ to $\mathcal{E}$;
    endfor
endwhile
```

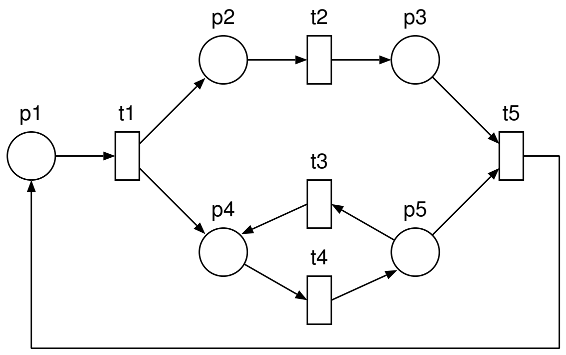
Note:

- The algorithm terminates iff the reachability set is finite.
- If markings can be added to and removed from sets $\mathcal{U}$ and $\mathcal{S}$ in constant time (not a realistic assumption), then the algorithm has complexity $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{T}|)$, if we check each transition in each marking.
- If markings are removed from $\mathcal{U}$ in FIFO order, we get "breadth–first" search.
- If markings are removed from $\mathcal{S}$ in LIFO order, we get "depth–first" search.
- The reachability graph contains enough information to answer any reachability question, and can be used to determine a firing sequence from $\mathbf{m}_0$ to any reachable marking $\mathbf{m}$.

Differences between the coverability graph and the reachability graph:

- The coverability graph can be used for unbounded Petri nets; the reachability graph will be infinite in this case.
- If the coverability graph contains no markings with ω, then it is equal to the reachability graph, and the reachability graph is finite.
- If the reachability graph is infinite, then the coverability graph will contain a marking with ω.

Example 6.3

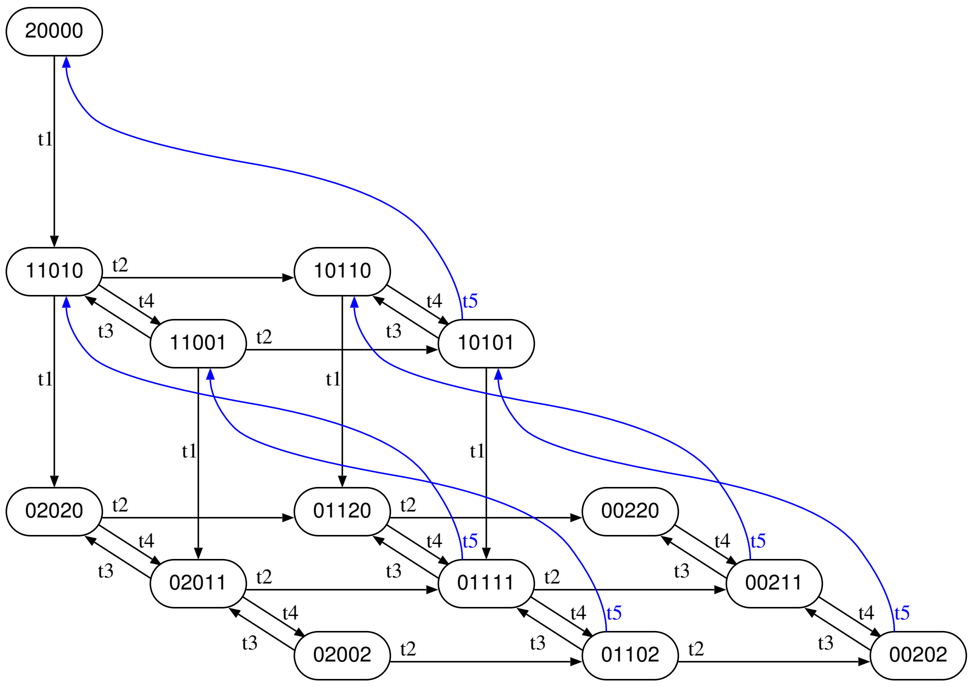


For the above Petri net, with initial marking

$$\begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \\ \left[\; 2 \right. & 0 & 0 & 0 & \left. 0 \;\right] \end{array}$$

construct the reachability graph.

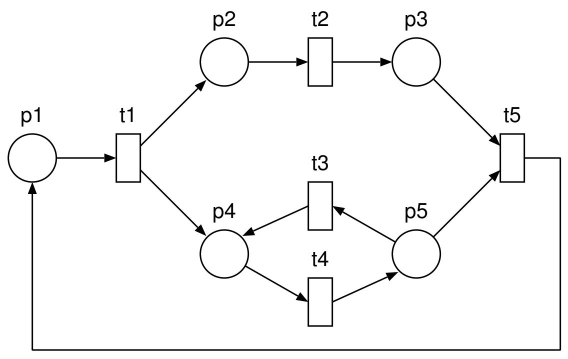
Using the reachability graph construction algorithm and some patience, we obtain the following graph.



6.3 State explosion

High level models (including Petri nets) often produce an extremely large number of reachable states, even for "small" models. This is known as the "state explosion problem". We will discuss ways to cope with this problem later in the semester.

Example 6.4



For the above Petri net, with initial marking

$$\begin{array}{ccccccc} & p_1 & p_2 & p_3 & p_4 & p_5 & \\ [& N & 0 & 0 & 0 & 0 &] \end{array}$$

how many reachable states are there?

In general, it is unknown how to answer this question without constructing the reachability graph. However, this Petri net is simple enough that we can answer this question by studying the Petri net, as follows. Note that, if transition t_1 fires n more times than transition t_5, then

- The sum of tokens in places p_2 and p_3 is exactly n. There are $n+1$ ways that this can occur.
- Similarly, the sum of tokens in places p_4 and p_5 is exactly n.
- The above two sums are completely independent.

Therefore, the number of reachable markings, given that transition t_1 has fired n more times than transition t_5, is $(n+1)^2$. For the given initial marking, the number of times t_1 may fire, before t_5 fires, is $0, 1, 2, \ldots, N$. The number of reachable markings is therefore

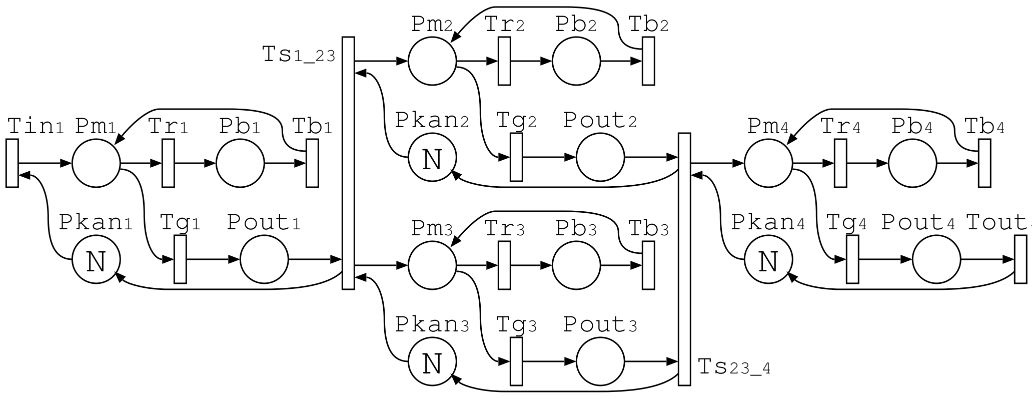
$$\sum_{n=0}^{N}(n+1)^2 \;=\; \sum_{n=0}^{N} n^2 + 2\sum_{n=0}^{N} n + \sum_{n=0}^{N} 1 \;=\; \cdots \;=\; \frac{(2N+3)(N+2)(N+1)}{6}$$

In particular, note that (as a sanity check):

- If $N = 0$, the number of markings is $(3 \cdot 2 \cdot 1)/6 = 1$
- If $N = 2$, the number of markings is $(7 \cdot 4 \cdot 3)/6 = 14$

The size of the reachability set grows as $\mathcal{O}(N^3)$.

Example 6.5



The above Petri net is a model of a kanban manufacturing system, comprised of four components. In each component, a part is processed, which is either successful and the part can move on to the next stage, or unsuccessful, which requires additional work to "un-do" the previous processing. Component 1 is used by components 2 and 3, and components 2 and 3 together are used by component 4. The initial marking specifies the initial number of raw parts N for each component. It can be shown that the number of reachable markings is exactly

$$|\mathcal{S}| = \frac{(N+1)^3(N+2)^3(N+3)^3(3N^2+12N+10)}{2160}$$

which grows as $\mathcal{O}(N^{11})$.

6.4 Model checking with Petri nets

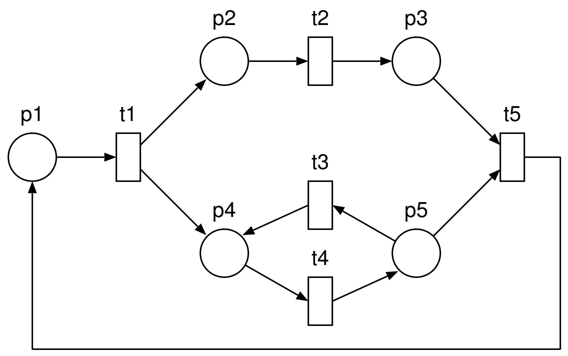
If the choice of which Petri net transition to fire is non-deterministic, and assuming the reachability graph is finite, then a (high–level) Petri net model can be used to describe a Kripke structure: the reachability graph becomes the Kripke structure. However, some care must be taken since the reachability graph may have markings with no outgoing edges; this can be handled by either

1. defining an atomic proposition for the deadlocked states, and adding self loops, or

2. modifying the model checking algorithms to work with finite paths.

Also, we must define the atomic propositions and label the states in the Kripke structure accordingly. Each atomic proposition p will be generated from a function $f_p : \mathbb{N}^{|\mathcal{P}|} \rightarrow \{0, 1\}$; this allows us to describe the atomic propositions in terms of the Petri net itself. Finally, we can write properties in our favorite logic.

Example 6.6

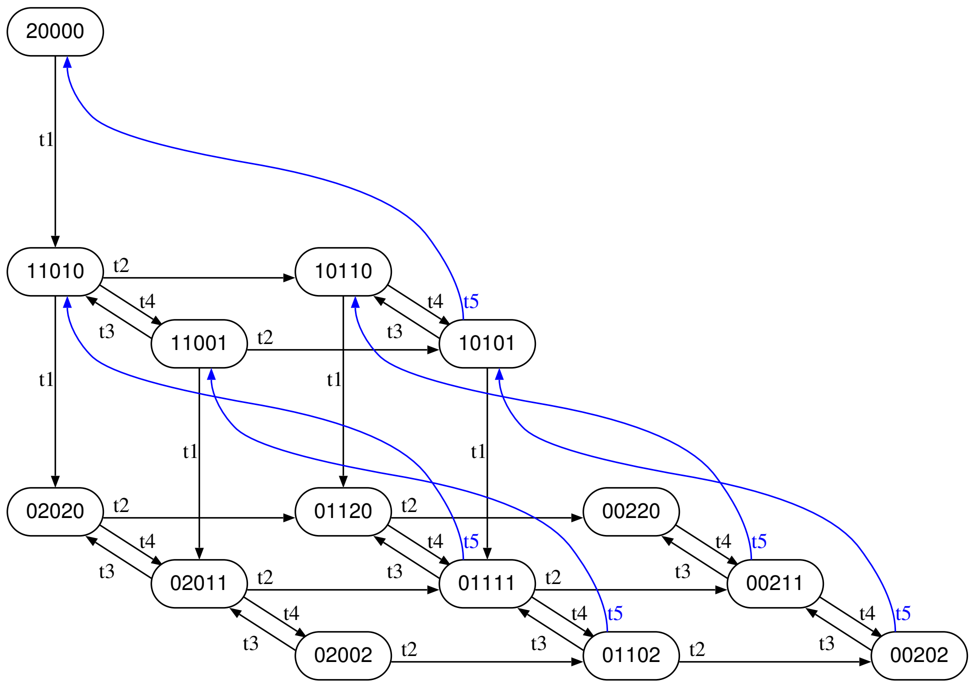


For the above Petri net, with initial marking

$$\begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \\ \left[\ 2\right. & 0 & 0 & 0 & \left. 0\ \right] \end{array}$$

is the property AF(Place p_1 is empty) satisfied?

First, we build the Kripke structure, which we obtain from the reachability graph:



Then, we build atomic proposition p corresponding to "Place p_1 is empty"; this can be done with the function $f_p(\mathbf{m}) = (\mathbf{m}[p_1] == 0)$. Rewriting the formula, we get $\mathsf{AF}\, p \equiv \neg\mathsf{EG}\neg p$. By inspection, we see that states satisfying atomic proposition $\neg p$ are $[\![\neg p]\!] = \{[20000], [11010], [11001], [10110], [10101]\}$. Since every state in the set $[\![\neg p]\!]$ can reach a state in $[\![\neg p]\!]$ in one step, we have that the expression $\mathsf{EG}(\neg p)$ is satisfied by exactly the states in $[\![\neg p]\!]$. Finally, taking the complement of $[\![\neg p]\!]$, we obtain the set of states satisfying AF(Place p_1 is empty), which is the set

$$\{[02020], [02011], [02002], [01120], [01111], [01102], [00220], [00211], [00202]\}.$$

Since this set does not contain the initial marking, we conclude that the Petri net does *not* satisfy AF(Place p_1 is empty).

Chapter 7

Decision diagrams

As discussed earlier, a high–level model can produce an extremely large reachability graph, easily containing millions of states or more. One way to combat this problem is to utilize techniques that can handle large numbers of states. *Decision diagrams* are one such technique.

7.1 Multi-value decision diagrams

A *multi–value decision diagram*, or MDD, is a data structure to represent *functions* over variables with finitely many possible values:

$$f : \mathcal{S}_K \times \cdots \times \mathcal{S}_1 \;\rightarrow\; \{0, \ldots, m-1\}$$

where sets $\mathcal{S}_k$ are finite, so for simplicity we will assume

$$\mathcal{S}_k = \{0, 1, \ldots, n_k - 1\}$$

A popular special case is *binary decision diagrams*, or BDDs[1], where $n_K = 2, \ldots, n_1 = 2, m = 2$.

7.1.1 Definition

- An MDD is a directed, acyclic graph with two types of nodes: *terminal* nodes and *non-terminal* nodes.
- Terminal nodes are labeled with values from the set $\{0, 1, \ldots, m-1\}$. (Actually, any set of values can be used.)
- Non-terminal nodes
  - are labeled with one of the function variables, x_k
  - contain n_k arcs to other nodes (the "decision" based on the value of x_k).
- Every MDD node represents some function of the form

$$f : \mathcal{S}_K \times \cdots \times \mathcal{S}_1 \;\rightarrow\; \{0, \ldots, m-1\}$$

[1]R. Bryant. "Graph–Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C–35 (8), August 1986, pages 677–691.

– Terminal node a represents the constant function:

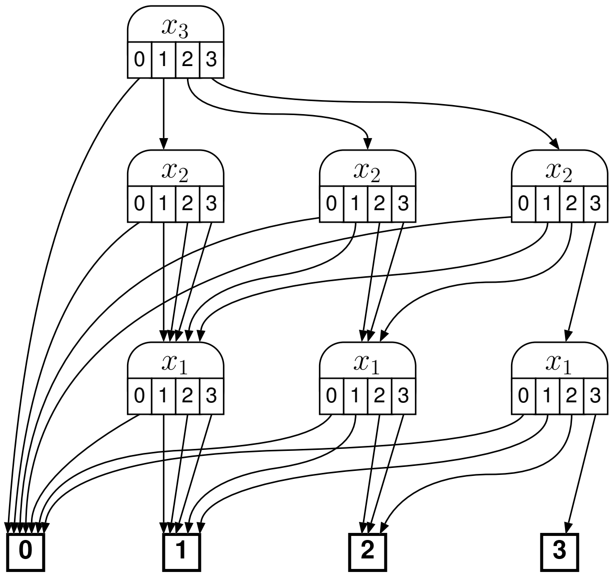
$$g(x_K, \ldots, x_1) = a$$

– A non-terminal node represents some function $f(x_K, \ldots, x_1)$. If the node is labeled with x_k, then the outgoing arc corresponding to value v goes to a node representing the function

$$f_{x_k=v}(x_K, \ldots, x_1) \ \equiv \ f(x_K, \ldots, x_{k+1}, v, x_{k-1}, \ldots, x_1)$$

- To evaluate a function for a particular assignment to the variables, start at the MDD node representing the function, and at each non-terminal node, follow the appropriate edge. The terminal node reached is the value of the function.

Example 7.1



Above, we have an MDD over $K = 3$ variables, with possible values

$$\mathcal{S}_3 = \mathcal{S}_2 = \mathcal{S}_1 = \{0, 1, 2, 3\}$$

The non-terminal node with label x_3 represents the function

$$f(x_3, x_2, x_1) = \min(x_3, x_2, x_1)$$

7.1.2 Terminology

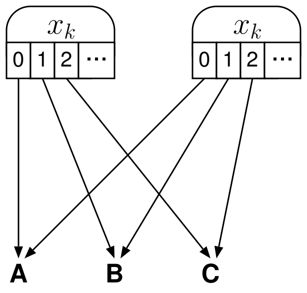
Definition 7.1 *An MDD is* ordered *if all paths through the MDD visit non-terminal nodes according to the same variable ordering.*

For an ordered MDD, we can define the *level* of a node as 0 for terminal nodes, or k for non-terminal nodes labeled with variable x_k. The ordering property guarantees that all outgoing arcs from a level–k node are to nodes at a level less than k.

Definition 7.2 *A non-terminal node is* redundant *if all of its outgoing arcs point to the same node.*

Note that a redundant node corresponds to a function that does not depend on the variable that labels the node.

Definition 7.3 *Two terminal nodes are* duplicates *if they have the same label. Two non-terminal nodes are* duplicates *if they have the same variable label, and they have the same outgoing arcs for each value:*



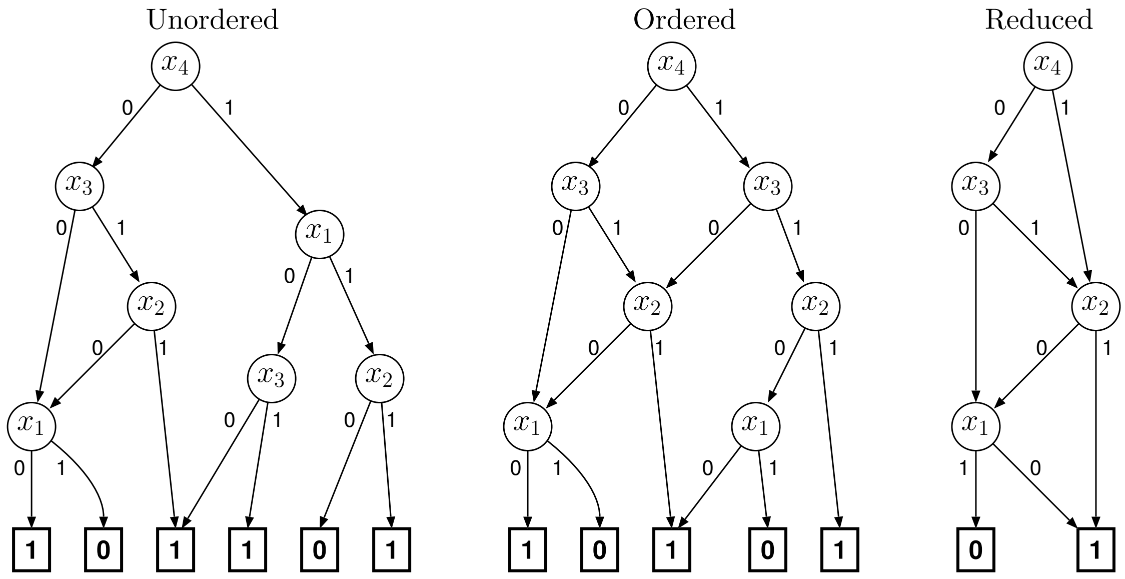
Definition 7.4 *A* reduced, ordered *MDD, or ROMDD, is an ordered MDD that contains no duplicate nodes and no redundant nodes.*

It can be shown that, for a fixed variable ordering, ROMDDs are a *canonical* form. This means that, for any given function and a fixed variable ordering, there is exactly one ROMDD representation of that function. As such, using ROMDDs, one can easily check if two functions are equivalent.

Example 7.2

The BDDs below are representations of the (boolean) function

$$f(x_4, x_3, x_2, x_1) \;=\; \neg x_1 + (x_4 + x_3) \cdot x_2$$



The reduced, ordered BDD shown above is *the* ROBDD representing $\neg x_1 + (x_4 + x_3) \cdot x_2$, for the variable ordering x_4, x_3, x_2, x_1.

7.1.3 Number of nodes and variable order

We will now look at two fundamental questions about ROMDDs.

1. ROMDDs are a canonical representation, *given a variable ordering.* This means that two different variable orders may give two different ROMDD representations. Does the choice of variable ordering affect the number of nodes in the ROMDD, and if so, how significantly?

2. We can use ROBDDs to solve the *Satisfiability* problem: build an ROBDD representing the formula to check; the formula is satisfiable if and only if the ROBDD representation is not terminal node zero. What is the worst–case complexity of ROBDDs? If they are efficient, then we could have $P = NP$, since the *Satisfiability* problem is known to be NP–complete.

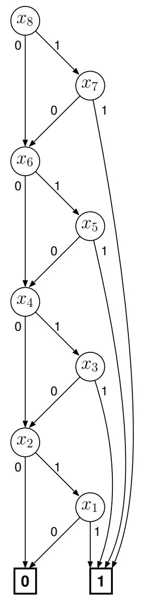
We will answer the first question with examples taken from Bryant's 1986 article.

Example 7.3

Build the ROBDD for the (boolean) function

$$f = x_{2N} \cdot x_{2N-1} + \cdots + x_4 \cdot x_3 + x_2 \cdot x_1$$

for the variable ordering $x_{2N}, \ldots, x_2, x_1$. You should obtain a BDD with $2N + 2$ nodes (including the terminal nodes). The case $N = 4$ is illustrated to the right.

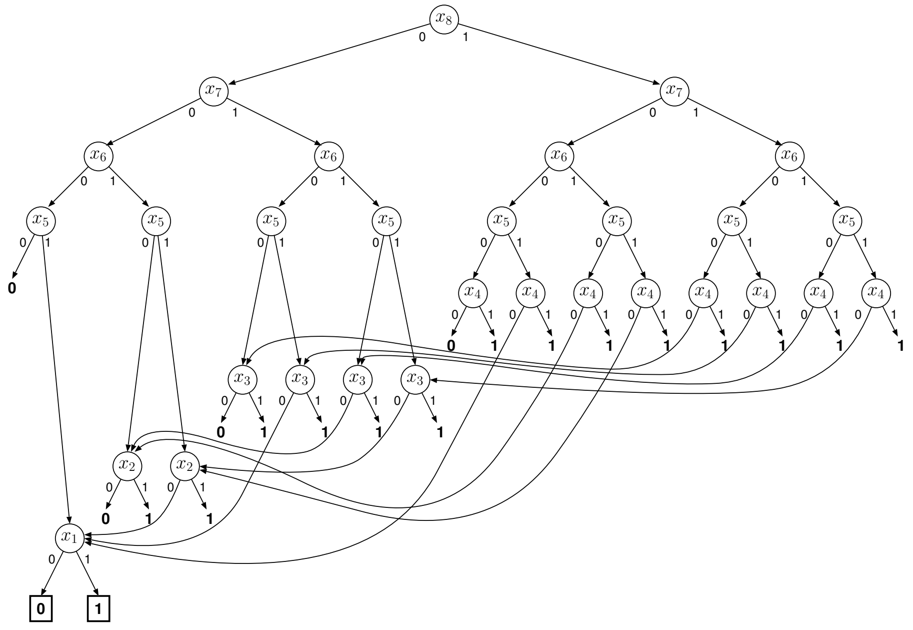


Example 7.4

Build the ROBDD for the (boolean) function

$$f = x_{2N} \cdot x_N + \cdots + x_{N+2} \cdot x_2 + x_{N+1} \cdot x_1$$

for the variable ordering $x_{2N}, \ldots, x_2, x_1$. Note this is the same function as the previous example, but with different variable ordering. The case $N = 4$ is illustrated below. We use several terminal nodes to clarify the illustration:



Assuming we merge all duplicate terminal nodes, the BDD will contain 2^{N+1} nodes.

As the above two examples illustrate, the size of a BDD can be *extremely* sensitive to the variable ordering. With one variable ordering, the number of BDD nodes grows as $\mathcal{O}(N)$, while with a different variable ordering, the number grows as $\mathcal{O}(2^N)$.

The second example also illustrates that ROBDDs can have exponential worst–case behavior if the variable ordering is not chosen wisely. But are there functions whose ROBDDs have an exponential number of nodes, *regardless* of the variable ordering? It has been shown[2] that there are functions of N variables for which the resulting ROBDD representation contains at least $\mathcal{O}(c^N)$ nodes, for some constant $c > 1$ (the value for c depends on the function; for example, the "hidden weighted bit function[3]" has $c \approx 1.14$). As such, ROBDDs cannot *always* solve the satisfiability problem in polynomial time, and require exponential time in the worst case.

While MDDs have poor worst–case performance, in practice, they are able to represent a large number of useful functions in a compact way. From now on, we will assume that all MDDs are *ordered* and *reduced*.

7.2 Operations

To use MDDs effectively, we must be able to construct new MDDs from old ones. In other words, we need algorithms to construct new functions from old ones according to some operation. These algorithms depend on the specific operation, but usually the algorithms share the following properties.

[2]R. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", in IEEE Transactions on Computers 40 (2), 1991.

[3]$f(x_N, \ldots, x_1) = x_{x_N + \cdots + x_1}$

- They are recursive. The algorithms typically recurse based on the node levels, and build new nodes at level k, whose downward arcs are determined via recursive calls.
- The MDD can be kept reduced during construction. Whenever a node is created, it is checked to see if it is a redundant node (in which case it can be eliminated), or if it is a duplicate of an existing node. This requires maintaining a *unique table* data structure (often a hash table) to discover duplicate nodes.

Typically, a generic recursive operation applied to MDD operand(s) proceeds as follows.

1. Handle the base cases, which occur when all the arguments are terminal nodes.
2. Handle any special cases that can be computed quickly.
3. Check if we have already computed this operation. Since MDD nodes may have several incoming arcs, we may generate the same recursive call several times. As such, we should maintain a *compute table* to remember results from each operation. This is important!
4. Recurse. Find the "top" variable x_n of the MDD operands, and build a node with label x_n. The downward pointers for this node are determined recursively.
5. Reduce the node we just built. That means eliminate it, if it is a redundant node; otherwise, check the uniqueness table to see if the new node is a duplicate of another (and if so, eliminate the new one).
6. Add the result to the compute table, and return the answer.

MDD libraries typically maintain an MDD *forest*, where all MDDs with the same variable order and node size are stored together.

- This allows different MDDs to share nodes.
- Functions are represented by pointers to nodes in the forest.
- Two functions are equivalent if and only if their pointers are equal.
- For manipulation (i.e., operators), new nodes are created as needed in the forest (and kept reduced).
- Garbage collection is a challenge: when nodes in the forest become disconnected, their memory should be (eventually) reclaimed.

7.2.1 Apply operation

A common operation for functions is to apply an operator to one or two functions. Specifically, given MDD representations for functions f and g, we want to construct the MDD representation for a function h defined as

$$h(x_K, \ldots, x_1) \;=\; f(x_K, \ldots, x_1) \oplus g(x_K, \ldots, x_1)$$

for some operator $\oplus$. Note that this describes many, but not all, interesting MDD operations. The recursive algorithm for "apply" will follow the basic steps outlined above:

1. Base cases: Recall that terminal nodes represent "constant" functions. It should be easy to determine $c_1 \oplus c_2$ for constants c_1, c_2.
2. Special cases. For example, $0 + g(x_K, \ldots, x_1) = g(x_K, \ldots, x_1)$.
3. Check the compute table for $f \oplus g$, if we already know the answer, return it immediately.
4. Recurse, based on the fact that
$$h_{x_n=i} = f_{x_n=i} \oplus g_{x_n=i}$$
5. Reduce the node we just created.
6. Add the result to the *compute table*.

A more detailed algorithm is given below. We use *Level*(f) to refer to the level of MDD node f, *Label*(f) to refer to the label of terminal node f, and $f[i]$ to refer to downward pointer i of non-terminal node f.

Algorithm 7.1 Apply $\oplus$

```
• Terminal case:
$k \leftarrow \max(Level(f), Level(g))$;
if $k = 0$ then
    return Terminal node with label $Label(f) \oplus Label(g)$;
endif
• Add special cases here.
• Check compute table:
if $\exists x$ such that $(f, \oplus, g, x) \in CT$ then        • If $\oplus$ commutes, check also for $(g, \oplus, f, x)$
    return $x$;
endif
• Build new node and recurse.
for each $i \in \{0, \ldots, n_k - 1\}$ do
    if $Level(f) = k$ then $f' \leftarrow f[i]$; else $f' \leftarrow f$; endif
    if $Level(g) = k$ then $g' \leftarrow g[i]$; else $g' \leftarrow g$; endif
    $h[i] \leftarrow Apply(f', \oplus, g')$;
endfor
• Check for redundant / duplicate nodes:
$h' \leftarrow Reduce(h)$;
• Add result to compute table:
$CT \leftarrow CT \cup \{(f, \oplus, g, h')\}$;
return $h'$;
```

Special cases for specific operators are given below:

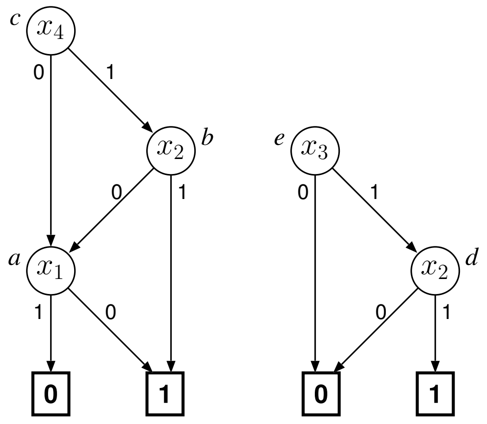
```
$+$ (addition) :
    if $f = 0$ then
        return $g$;
    endif
    **if** $g = 0$ **then**
        **return** $f$;
    **endif**

$-$ (subtraction) :
    **if** $f = g$ **then**
        **return** Terminal node $0$;
    **endif**
    **if** $g = 0$ **then**
        **return** $f$;
    **endif**

$\cdot$ (multiplication) :
    **if** $f = 0 \lor g = 0$ **then**
        **return** Terminal node $0$;
    **endif**
    **if** $f = 1$ **then**
        **return** $g$;
    **endif**
    **if** $g = 1$ **then**
        **return** $f$;
    **endif**

$+$ (boolean OR) :
    **if** $f = 1 \lor g = 1$ **then**
        **return** Terminal node $1$;
    **endif**
    **if** $f = g$ **then**
        **return** $f$;
    **endif**
    **if** $f = 0$ **then**
        **return** $g$;
    **endif**
    **if** $g = 0$ **then**
        **return** $f$;
    **endif**

$-$ (boolean, with $0 - 1 \equiv 0$) :
    **if** $f = 0 \lor g = 1 \lor f = g$ **then**
        **return** Terminal node $0$;
    **endif**
    **if** $g = 0$ **then**
        **return** $f$;
    **endif**
```

```
· (boolean AND) :
    if $f = 0 \vee g = 0$ then
        return Terminal node 0;
    endif
    if $f = g$ then
        return $f$;
    endif
    if $f = 1$ then
        return $g$;
    endif
    if $g = 1$ then
        return $f$;
    endif
```

A similar algorithm can be used for unary operators, i.e., for computing a function h where $h(x_K, \ldots, x_1) = \ominus f(x_K, \ldots, x_1)$.

Example 7.5

Suppose we have the following MDD forest, where $\mathcal{S}_4 = \mathcal{S}_3 = \mathcal{S}_2 = \mathcal{S}_1 = \{0, 1\}$. Again, we draw duplicate terminal nodes for clarity; in practice there is a single terminal node for each value.



In the forest,

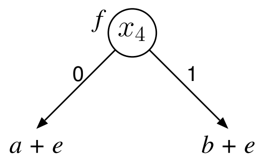
- node a represents the function $a(x_4, x_3, x_2, x_1) = \neg x_1$
- node b represents the function $b(x_4, x_3, x_2, x_1) = x_2 + \neg x_1$
- node c represents the function $c(x_4, x_3, x_2, x_1) = x_4 \cdot x_2 + \neg x_1$
- node d represents the function $d(x_4, x_3, x_2, x_1) = x_2$
- node e represents the function $e(x_4, x_3, x_2, x_1) = x_3 \cdot x_2$

To construct the MDD encoding the function

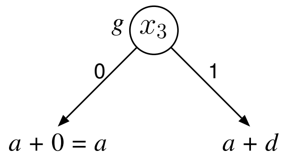
$$f(x_4, x_3, x_2, x_1) = c(x_4, x_3, x_2, x_1) + e(x_4, x_3, x_2, x_1) = x_3 \cdot x_2 + x_4 \cdot x_2 + \neg x_1$$

we call $Apply(c, +, e)$, which builds the following nodes:

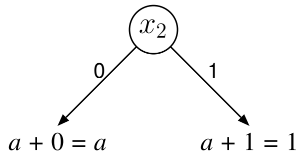
1. *Apply*$(c, +, e)$ will build a level 4 node:



2. *Apply*$(a, +, e)$ will build a level 3 node:

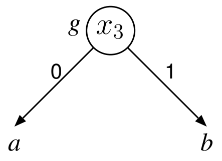


3. *Apply*$(a, +, d)$ will build a level 2 node:

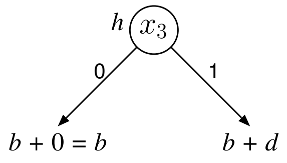


   Note that this node is a duplicate of node b; therefore, $a + d = b$.

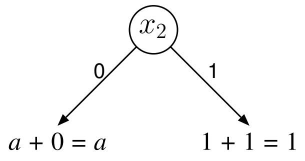
4. The constructed node for $g =$ *Apply*$(a, +, e)$ is:



5. *Apply*$(b, +, e)$ will build a level 3 node:

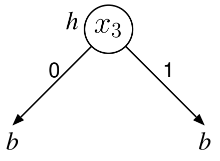


6. *Apply*$(b, +, d)$ will build a level 2 node:



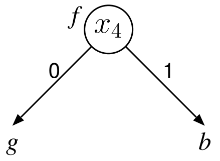
   But this node is a duplicate of node b; therefore, $b + d = b$.

7. The constructed node for $h =$ *Apply*$(b, +, e)$ is:



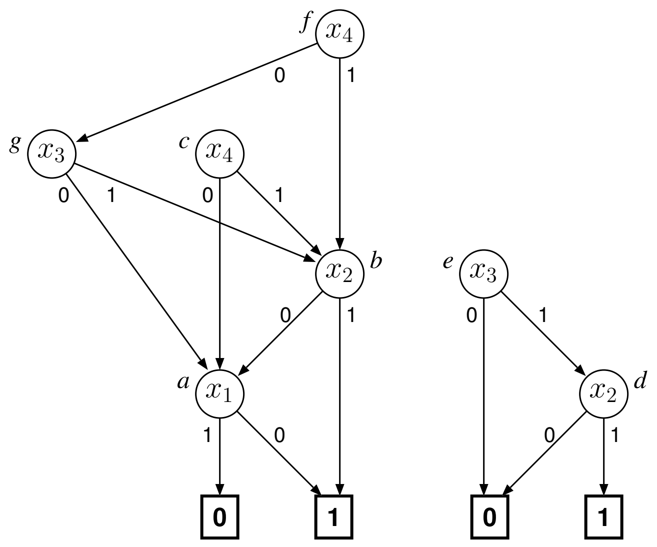
But this is a redundant node; therefore, $b + e = b$.

8. The constructed node for $f = \mathit{Apply}(c, +, e)$ is:



After performing the operation, we have the following forest, where the new nodes f and g represent the functions

$$\begin{aligned} f(x_4, x_3, x_2, x_1) &= x_3 \cdot x_2 + x_4 \cdot x_2 + \neg x_1 \\ g(x_4, x_3, x_2, x_1) &= a(x_4, x_3, x_2, x_1) + e(x_4, x_3, x_2, x_1) = \neg x_1 + x_3 \cdot x_2 \end{aligned}$$



7.2.2 Complexity of Apply operation

If a compute table is not used, the apply operation requires in the worst case to recurse until the terminal nodes are reached, and the construction of a node with label x_k generates n_k recursive calls. As such, the overall complexity will be

$$\mathcal{O}(n_K \cdots n_2 n_1)$$

which is usually prohibitively high. In particular, if all variables have the same number of values b, then the worst-case complexity is $\mathcal{O}(b^K)$.

However, using a compute table, any duplicate recursive calls to apply will not require any work (assuming $\mathcal{O}(1)$ time to check the compute table). Then, we can obtain the number of recursive calls that will require work in the worst case as the number of possible distinct calls to apply. For an apply function with a single MDD argument (e.g., logical negation), we can at most call apply for every node in the MDD. For an apply function with two MDD arguments, we can at most call apply for every possible pair of nodes (one taken from the left argument, one taken from the right argument). Then, the cost of each distinct call to apply is $\mathcal{O}(n_k)$ for a node created with variable x_k.

Property 7.5 *For unary operations, a call to* apply(p) *requires* $\mathcal{O}(\eta(p))$ *time, where* $\eta(p)$ *denotes the number of edges in the graph rooted at node* p*.*

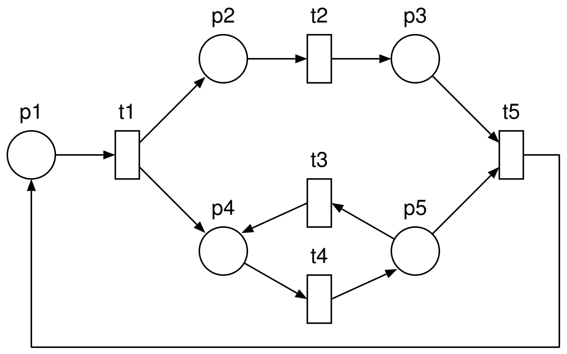
Property 7.6 *For binary operations, a call to* apply(p, q) *requires no more than* $\mathcal{O}(\nu(p) \cdot \nu(q) \cdot n)$ *time, where* $\nu(p)$ *denotes the number of nodes in the graph rooted at node* p*, and* $n = \max\{n_K, \ldots, n_1\}$.

7.3 Using MDDs for CTL model checking of Petri nets

To utilize MDDs for CTL model checking, we must be able to describe each state as a collection of K integer state variables. For a finite state machine, this implies that each state variable is bounded. For simplicity, for now we will assume that these bounds are known.

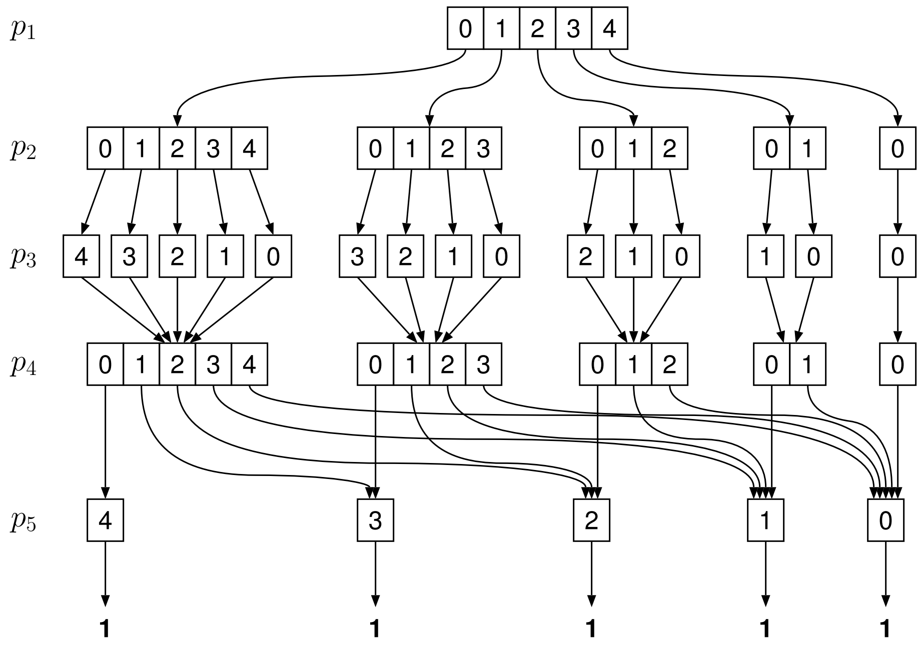
Following the notation used earlier, we must be able to represent a boolean vector $\mathbf{x}$ of dimension $|\mathcal{S}|$, where $\mathcal{S}$ represents the set of states in the state machine. Instead, we will represent boolean vectors $\mathbf{x}$ of dimension $|\mathcal{S}_K \times \cdots \times \mathcal{S}_1|$. Since we have $\mathcal{S} \subseteq \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, this will not be an issue as long as the elements of $\mathbf{x}$ corresponding to "unreachable" states are zeroes. If we index the vector $\mathbf{x}$ by the values of the K state variables, then it is straightforward to use an MDD to encode vector $\mathbf{x}$: simply encode the function $f(x_K, \ldots, x_1) = \mathbf{x}[(x_K, \ldots, x_1)]$. Equivalently, this can be interpreted as a set of states $\mathcal{X}$ with $(x_K, \ldots, x_1) \in \mathcal{X}$ if and only if $f(x_K, \ldots, x_1) = 1$. The element-wise logical operations on vectors necessary for CTL model checking, namely, $\neg\mathbf{x}$, $\mathbf{x} \wedge \mathbf{y}$, and $\mathbf{x} \vee \mathbf{y}$, can be performed using the apply operation discussed above.

Example 7.6



For the fork-join PN shown above, draw an MDD encoding the markings reachable from $[4, 0, 0, 0, 0]$.

Using $K = 5$ variables, each bounded by $n_k = 5$, we obtain the following MDD.



To conserve space, we write the variable labels to the left of the figure, and use a "sparse" representation for nodes. Any paths not displayed are assumed to go to terminal node 0. Looking at the MDDs rooted at variable p_2 in the figure, these correspond to "n tokens in places p_2 and p_3, and n tokens in places p_4 and p_5", with $n = 0$ at the right side of the figure, to $n = 4$ at the left side of the figure. Note that the MDD increases by $N + 4$ nodes and $3N + 5$ edges when N increases. As such, it can be shown that the MDD encoding the set of markings reachable from $[N, 0, 0, 0, 0]$, using the variable order shown above, has $(N^2 + 9N + 10)/2$ non-terminal nodes and $(3N^2 + 13N + 10)/2$ edges. Thus, the storage requirements for this MDD grows as $\mathcal{O}(N^2)$, while the number of reachable states grows as $\mathcal{O}(N^3)$.

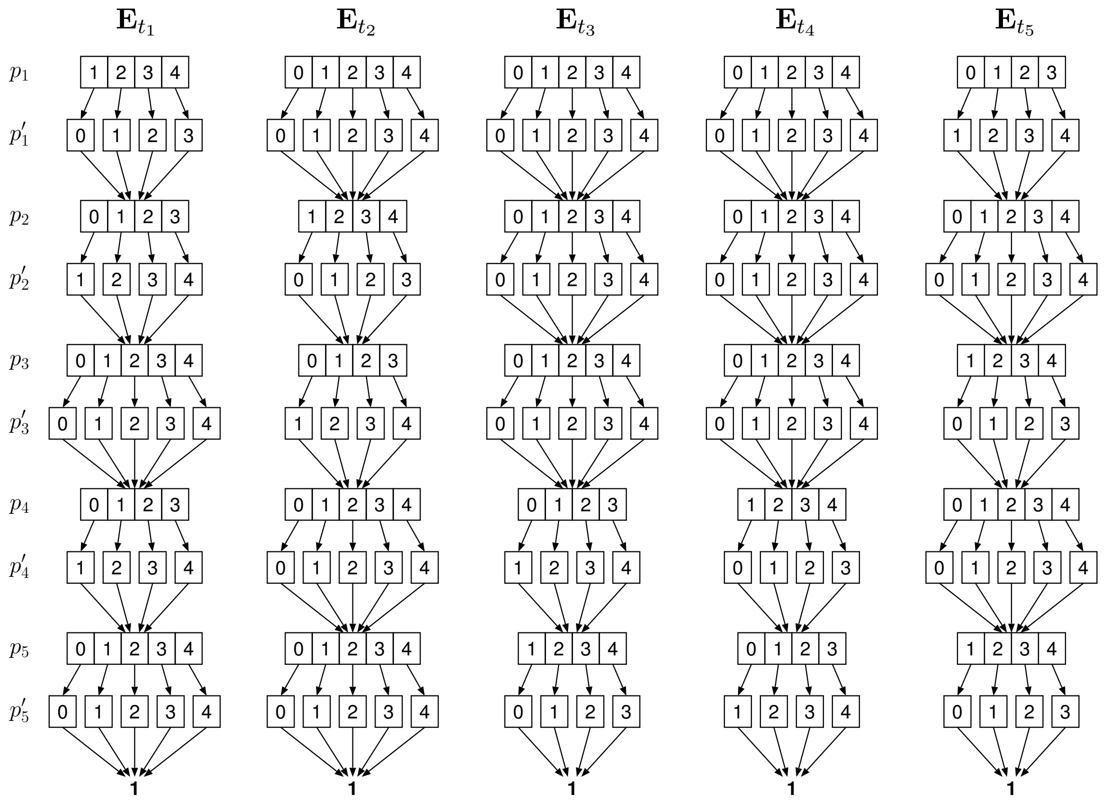
The edge matrix $\mathbf{E}$ can be represented as a boolean matrix of dimension $|\mathcal{S}_K^2 \times \cdots \times \mathcal{S}_1^2|$. Equivalently, this can be interpreted as a set of edges. While this is larger than the $|\mathcal{S}^2|$ matrix used earlier, this is not a problem as long as there are no edges from reachable states to unreachable ones. We do not care if there are any edges from unreachable states to unreachable or reachable states. To encode this as an MDD, we use twice as many state variables by creating a "primed" version of each state variable. Traditionally, the unprimed variables correspond to the source state, while primed variables correspond to the destination state. Thus, we encode the function $g(x_K, x'_K, \ldots, x_1, x'_1) = \mathbf{E}[(x_K, \ldots, x_1), (x'_K, \ldots, x'_1)]$. The "interleaved" variable order usually produces a more compact MDD than, say, the variable order $g(x_K, \ldots, x_1, x'_K, \ldots, x'_1)$. Again, we can use the apply operation to perform element-wise operations on sets of edges.

Example 7.7

For the fork-join Petri net and a bound of 5 tokens per place, we can construct the MDD encoding of $\mathbf{E}$ by recognizing that $\mathbf{E} = \mathbf{E}_{t_1} \vee \mathbf{E}_{t_2} \vee \mathbf{E}_{t_3} \vee \mathbf{E}_{t_4} \vee \mathbf{E}_{t_5}$, where $\mathbf{E}_t$ are the edges due to transition t. Each of these can be realized by an appropriate logical expression, based on Petri net firing rules. For example,

$$\begin{aligned}\mathbf{E}_{t_1}(p_1, p_1', \ldots, p_5, p_5') \quad = \quad & (p_1 > 0) \wedge (p_1' == p_1 - 1) \wedge (p_2' == p_2 + 1) \wedge \\ & (p_3' == p_3) \wedge (p_4' == p_4 + 1) \wedge (p_5' == p_5)\end{aligned}$$

From these we can obtain the following MDD forest (some nodes are duplicated, for clarity):



The overall set of edges $\mathbf{E}$ can be obtained by taking the set union (using apply) of $\mathbf{E}_{t_1}, \ldots, \mathbf{E}_{t_5}$. Note that $\mathbf{E}$ will contain edges from unreachable markings. For example, according to $\mathbf{E}_{t_1}$, there is an edge from marking $[4, 0, 4, 0, 4]$ to marking $[3, 1, 4, 1, 4]$; however, marking $[4, 0, 4, 0, 4]$ is not reachable from marking $[4, 0, 0, 0, 0]$.

All that remains are the pre-image and post-image operations. These can be obtained either by developing specialized recursive operations (similar to *apply*) just for pre-image and post-image, or by combining several simpler operations. For example, the post-image operation $\mathbf{x} \cdot \mathbf{E}$ can be performed by:

1. Treating $\mathbf{x}$ as a matrix with the "primed" variables having no effect, take the set intersection of $\mathbf{x}$ and $\mathbf{E}$. This will produce the set of edges in $\mathbf{E}$ with a source state in $\mathbf{x}$.

2. On the resulting set of edges, perform an "existential quantification" over the unprimed variables. This operation can be done recursively: for each MDD node corresponding to an unprimed variable, replace the node with the node encoding the set union of all its (existentially quantified) children. The result is the set of destination states, encoded using the primed variables.

3. Relabel the resulting set so that the primed variables are replaced by unprimed ones. This is a simple recursive operation.

A specialized algorithm essentially performs all three steps at once, which can be more efficient since fewer nodes must be created and destroyed.

7.3.1 Generating reachable states

Because we typically construct **E** from the high-level formalism and allow it to contain edges from unreachable states, we must generate the set of states reachable from the initial state(s) before we can perform CTL model checking. (Typically, this is the most expensive step!) This can be done entirely through MDD operations using the following simple algorithm:

Algorithm 7.2 Building the set of reachable states

```
$s \leftarrow$ initial states;      • $s$ stores known reachable states
$f \leftarrow s$;                   • $f$ stores newly discovered states
while ($f \neq \emptyset$) do
    $g \leftarrow f \cdot E$;       • Post-image operation
    $f \leftarrow g \setminus s$;   • $g$ stores everything reachable from $f$
    $s \leftarrow s \cup f$;
return $s$;
```

Interestingly, the size of an MDD is *not necessarily* related to the size of the set of states it encodes. For example, the MDD encoding of function $f(x_K, \ldots, x_1) = 0$, namely terminal node 0, is just as compact as the MDD encoding of function $f(x_K, \ldots, x_1) = 1$. The first function encodes the empty set, while the second function encodes the set $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$, the largest possible set of states for the given variables. As such, we could also use the algorithm:

Algorithm 7.3 Building the set of reachable states

```
$s \leftarrow$ initial states;      • $s$ stores known reachable states
$x \leftarrow \emptyset$;           • $x$ stores the previous set of reachable states
while ($s \neq x$) do
    $g \leftarrow s \cdot E$;       • $g$ stores everything reachable from $s$
    $x \leftarrow s$;
    $s \leftarrow s \cup g$;
return $s$;
```

Note that the comparison $s \neq x$ is equivalent to comparison of pointers if s and x are stored in the same MDD forest. Also, note that this algorithm uses fewer total MDD operations (a set difference is saved). While the post-image operation $s \cdot E$ is done on increasingly larger sets of states, note that

1. the time for this operation depends on the number of MDD nodes in s, which are *not necessarily* growing.
2. if s accumulates more states in such a way that only a few MDD nodes in the representation for s are updated at each iteration, then the recursive calls for $s \cdot E$ will often match entries in the compute table. A similar argument can be made for $s \cup g$.

Given the "symbolic" algorithm for performing the post-image operation, it is not difficult to modify the algorithm to instead perform the pre-image operation.

7.3.2 CTL model checking

Once we have obtained the MDD encoding for matrix $\mathbf{E}$, and the set of reachable states $\mathcal{S}$ encoded as an MDD, we are ready to perform CTL model checking. The explicit algorithms described earlier for CTL model checking can be adapted in a straightforward manner to the "symbolic" setting, using the "symbolic" algorithm for pre-image. The only change is due to the fact that our $\mathbf{E}$ matrix has "extra" rows and columns due to unreachable states. Thus, we must either

- set the rows and columns due to unreachable states to zero (by taking the set intersection $E \cap (\mathcal{S} \times \mathcal{S})$, where E is the set of edges corresponding to $\mathbf{E}$); or
- adapt the EX, EU, EG algorithms to work correctly for this larger $\mathbf{E}$ matrix, with appropriate use of set intersections with $\mathcal{S}$.

Both choices are reasonable.

Chapter 8

Linear Temporal Logic

Linear temporal logic, or LTL, is another commonly–used temporal logic. The name “linear”, as opposed to “branching”, refers to the fact that formulas describe conditions along a single, linear path, rather than allowing the possibility of the path to “branch” as it can in CTL. The temporal operators are the same, but there are subtle differences in the semantics. Outside reference: *Model Checking*, by Clarke, Grumberg, and Peled.

8.1 LTL syntax

LTL deals entirely with *path formulas*: formulas which can be, conceptually, verified along a single, infinitely–long path. An LTL path formula ψ has the following syntax:

$$\psi \ ::= \ \texttt{tt} \mid \texttt{ff} \mid p \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\,\psi \mid \mathsf{F}\,\psi \mid \mathsf{G}\,\psi \mid \psi\,\mathsf{U}\,\psi$$

where p is an atomic proposition. Given an LTL path formula ψ and a model to check, we say the model satisfies the formula if, for all paths starting in an initial state, the path formula ψ holds. For consistency, we will write an overall LTL formula as $\mathsf{A}\,\psi$, where ψ is a path formula. Thus, for example, $\mathsf{A}\,\mathsf{FGX}\,p$ is a valid LTL formula, but $\mathsf{AF}\,\mathsf{AG}\,p$ is not. However, it is common to omit the implied A path quantifier. Also, note that some people use $\bigcirc$ instead of X, $\Box$ instead of G, and $\Diamond$ instead of F. Thus, while we write $\mathsf{A}\,\mathsf{FGX}\,p$, others could write $\Diamond\Box\bigcirc\, p$ to denote the same formula.

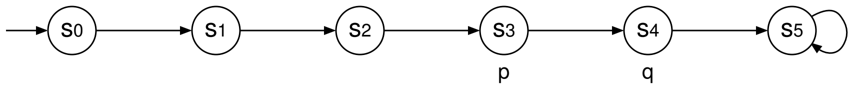
8.2 LTL semantics

We must give rules for which *paths* in a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ satisfy an LTL path formula ψ, which we write as $M, \pi \models \psi$. When a path π does not satisfy path formula ψ, we instead write $M, \pi \not\models \psi$. For the following, assume $\pi = (p_0, p_1, \ldots)$, and let π^i denote the suffix of π starting with p_i, i.e., $\pi^i = (p_i, p_{i+1}, \ldots)$.

1. $M, \pi \models \texttt{tt}$, for all paths.
2. $M, \pi \not\models \texttt{ff}$, for all paths.
3. $M, \pi \models p$, if and only if $p \in L(p_0)$.
4. $M, \pi \models \neg\psi$, if and only if $M, \pi \not\models \psi$.

5. $M, \pi \models \psi_1 \wedge \psi_2$, if and only if $M, \pi \models \psi_1$ and $M, \pi \models \psi_2$.
6. $M, \pi \models \psi_1 \vee \psi_2$, if and only if $M, \pi \models \psi_1$ or $M, \pi \models \psi_2$.
7. $M, \pi \models \mathsf{X}\psi$, if and only if $M, \pi^1 \models \psi$.
8. $M, \pi \models \mathsf{F}\psi$, if and only if there exists an $i \geq 0$ such that $M, \pi^i \models \psi$.
9. $M, \pi \models \mathsf{G}\psi$, if and only if $M, \pi^i \models \psi$ for all $i \geq 0$.
10. $M, \pi \models \psi_1 \mathsf{U}\, \psi_2$, if and only if there exists a $j \geq 0$ such that
    (a) $M, \pi^j \models \psi_2$, and
    (b) $M, \pi^i \models \psi_1$ for all $i < j$.

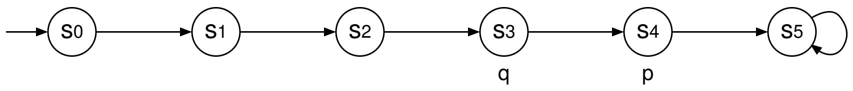
Example 8.1



Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?

Solution: There is only one path, $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, \ldots)$. First, check the path against the definition for U: using $j = 4$, we have $\pi^j \models q$. Then, check that $\pi^i \models \mathsf{F}\,p$ for all $i < j$. Since all of those hold, the answer is **yes**.

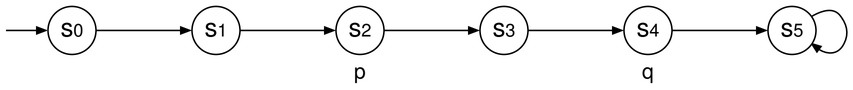
Example 8.2



Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?

Solution: This is similar to the previous example: examine the only possible path, $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, \ldots)$. Using $j = 3$, we have $\pi^j \models q$, and $\pi^i \models \mathsf{F}\,p$ for all $i < j$. So the answer is **yes**.

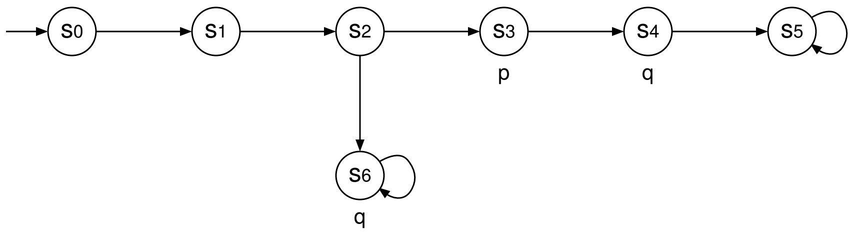
Example 8.3



Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?

Solution: Again, there is only one possible path, $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, \ldots)$. The only j where $\pi^j \models q$ is $j = 4$. But $\pi^3 = (s_3, s_4, s_5, s_5, \ldots)$ does not satisfy $\mathsf{F}\,p$. So the answer is **no**.

Example 8.4



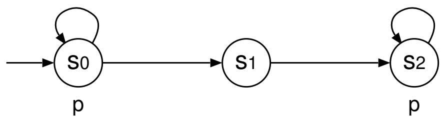
Does this model satisfy $\mathsf{A}\,(\mathsf{F}\,p)\,\mathsf{U}\,q$?

Solution: There are two paths to consider. For the path $\pi = (s_0, s_1, s_2, s_3, s_4, s_5, s_5, \ldots)$, the analysis is the same as Example 8.1, and the path satisfies $(\mathsf{F}\,p)\,\mathsf{U}\,q$.

Now consider the other path, $\pi = (s_0, s_1, s_2, s_6, s_6, \ldots)$. For $j \geq 3$, we have $\pi^j = (s_6, s_6, \ldots)$ and $\pi^j \models q$. Now, does $\pi^i \models \mathsf{F}\,p$ for all $i < j$? For $i = 0$, the answer is **no**: $\pi^0 = (s_0, s_1, s_2, s_6, s_6, \ldots)$, and none of those states satisfy p, so $\pi^0 \not\models \mathsf{F}\,p$.

Therefore, the answer is **no**.

Example 8.5



Does this model satisfy $\mathsf{A}\,\mathsf{F}\mathsf{G}\,p$?

Solution: The paths in this model can be described by the number of times the edge from s_0 to s_0 is traversed before going to s_1. Let

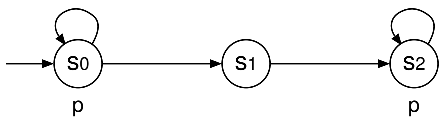
$$\begin{aligned}
\pi_0 &= (s_0, s_1, s_2, s_2, \ldots) \\
\pi_1 &= (s_0, s_0, s_1, s_2, s_2, \ldots) \\
\pi_2 &= (s_0, s_0, s_0, s_1, s_2, s_2, \ldots) \\
&\vdots
\end{aligned}$$

and finally, there is the possibility of never leaving s_0:

$$\pi_\infty = (s_0, s_0, s_0, \ldots)$$

For finite n, we have $\pi_n^{n+2} = (s_2, s_2, \ldots)$ and therefore $\pi_n^{n+2} \models \mathsf{G}\,p$, and $\pi_n \models \mathsf{F}\mathsf{G}\,p$. But the path π_∞ also satisfies $\mathsf{G}\,p$, and therefore $\pi_\infty \models \mathsf{F}\mathsf{G}\,p$. Therefore, all paths satisfy $\mathsf{F}\mathsf{G}\,p$, and the answer is **yes**.

Example 8.6



Does this model satisfy the CTL formula $\mathsf{AF}\,\mathsf{AG}\,p$?

Solution: Working inward, we first determine which states satisfy $\mathsf{AG}\,p$: only s_2. Next, we determine which states satisfy $\mathsf{AF}\,\mathsf{AG}\,p$, knowing that only $s_2 \models \mathsf{AG}\,p$. The answer is s_1 and s_2. Since the initial state s_0 is not in this set, the answer is **no**.

Property 8.1

$$\mathsf{AF}\,\mathsf{AG}\,p \quad \not\equiv \quad \mathsf{A}\,\mathsf{FG}\,p$$

Proof: from Example 8.5 and Example 8.6, there exists a Kripke structure that satisfies one formula but not the other.

The previous two examples illustrate the difference between CTL and LTL: in CTL, by nesting $\mathsf{AF}\,\mathsf{AG}\,p$, we must allow "branching" because we quantify over paths twice: for all paths, in the future, *for all paths*, globally. In contrast, the LTL nesting $\mathsf{A}\,\mathsf{FG}\,p$ (and any other nesting) only quantifies over paths once: for all paths, in the future, globally.

8.3 Equivalences

Note: these are *path formula* equivalences. To prove such an equivalence, we must show that for any path π, π satisfies the first path formula if and only if π satisfies the second path formula. To disprove an equivalence, it suffices to find a path that satisfies one formula but not the other.

8.3.1 Negations

Property 8.2

$$\neg\mathsf{A}\psi \;\equiv\; \mathsf{E}\neg\psi$$

Proof: *Follows from a similar property for $\forall$ and $\exists$.*

As a consequence of Property 8.2 and the fact that $\neg\psi$ is a valid path formula for any path formula ψ, it is possible to use LTL model checking to test for existence of a path satisfying a formula. Therefore, from now on, we will consider $\mathsf{E}\psi$ to be a valid LTL formula, where ψ is a path formula.

Property 8.3

$$\neg\mathsf{X}\psi \;\equiv\; \mathsf{X}\neg\psi$$

Proof:

$$\begin{aligned} \pi \models \neg\mathsf{X}\psi \quad &\Leftrightarrow \quad \pi \not\models \mathsf{X}\psi \\ &\Leftrightarrow \quad \pi^1 \not\models \psi \\ &\Leftrightarrow \quad \pi^1 \models \neg\psi \\ &\Leftrightarrow \quad \pi \models \mathsf{X}\neg\psi \end{aligned}$$

Property 8.4

$$\mathsf{F}\psi \;\equiv\; \mathtt{tt}\,\mathsf{U}\,\psi$$

Property 8.5

$$\neg\mathsf{G}\psi \;\equiv\; \mathsf{F}\neg\psi$$

We therefore have that $\neg$, $\wedge$, X, and U are an adequate set of operators for expressing LTL path formulas.

8.3.2 Conjunctions and disjunctions

Property 8.6

$$\begin{aligned} \mathsf{X}(\psi_1 \wedge \psi_2) &\equiv (\mathsf{X}\,\psi_1) \wedge (\mathsf{X}\,\psi_2) \\ \mathsf{X}(\psi_1 \vee \psi_2) &\equiv (\mathsf{X}\,\psi_1) \vee (\mathsf{X}\,\psi_2) \end{aligned}$$

Property 8.7

$$\mathsf{F}(\psi_1 \vee \psi_2) \equiv (\mathsf{F}\psi_1) \vee (\mathsf{F}\psi_2)$$

Property 8.8

$$\mathsf{G}(\psi_1 \wedge \psi_2) \equiv (\mathsf{G}\psi_1) \wedge (\mathsf{G}\psi_2)$$

Property 8.9

$$\begin{aligned} (\psi_1 \wedge \psi_2)\,\mathsf{U}\,\psi &\equiv (\psi_1\,\mathsf{U}\,\psi) \wedge (\psi_2\,\mathsf{U}\,\psi) \\ \psi\,\mathsf{U}\,(\psi_1 \vee \psi_2) &\equiv (\psi\,\mathsf{U}\,\psi_1) \vee (\psi\,\mathsf{U}\,\psi_2) \end{aligned}$$

8.3.3 Redundant nesting

Property 8.10

$$\mathsf{FF}\psi \equiv \mathsf{F}\psi$$

Property 8.11

$$\mathsf{GG}\psi \equiv \mathsf{G}\psi$$

Property 8.12

$$\psi_1\,\mathsf{U}\,(\psi_1\,\mathsf{U}\,\psi_2) \equiv \psi_1\,\mathsf{U}\,\psi_2$$

8.3.4 Recursion

Property 8.13

$$\mathsf{F}\,\psi \equiv \psi \vee \mathsf{XF}\,\psi$$

Property 8.14

$$\mathsf{G}\,\psi \equiv \psi \wedge \mathsf{XG}\,\psi$$

Property 8.15

$$\psi_1\,\mathsf{U}\,\psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \mathsf{X}\,(\psi_1\,\mathsf{U}\,\psi_2))$$

8.4 LTL model checking by tableau

A tableau–type algorithm for LTL model checking works by constructing a graph that encodes obligations along paths, and the resulting graph allows us to easily determine if the original Kripke structure satisfies the formula. We assume the original formula ψ is rewritten to contain only the operators $\neg$, $\wedge$, X, and U. Then, we build a list of relevant subformulas of ψ, called the *closure* of ψ and denoted $C(\psi)$, recursively as follows.

- $C(\mathtt{tt}) = C(\mathtt{ff}) = \emptyset$.
- If $\psi = p$, an atomic proposition, then $C(\psi) = \{p\}$.
- If $\psi = \neg\phi$, then $C(\psi) = C(\phi)$.
- If $\psi = \phi_1 \wedge \phi_2$, then $C(\psi) = \{\psi\} \cup C(\phi_1) \cup C(\phi_2)$.
- If $\psi = \mathsf{X}\,\phi$, then $C(\psi) = \{\psi\} \cup C(\phi)$.
- If $\psi = \phi_1 \,\mathsf{U}\, \phi_2$, then $C(\psi) = \{\psi\} \cup C(\phi_1) \cup C(\phi_2)$.

Definition 8.16 *The* length *of a path formula ψ, denoted as $|\psi|$, is given by*

$$|\psi| \quad \equiv \quad |C(\psi)|.$$

Note that $|\psi|$ is at most the total number of operators and propositions in the formula ψ.

8.4.1 Building the tableau graph

We build a tableau graph $T = (V, E)$ where a vertex in the graph is a pair $v = (s_v, L_v)$, where s_v is a state of the Kripke structure, and L_v is a labeling for each of the subformulas in $C(\psi)$, either that are known to hold, or are *obligated* to hold. We denote this as a set of subformulas that hold, so formally, we have $V \subseteq \mathcal{S} \times 2^{C(\psi)}$. We say a vertex $v = (s_v, L_v)$ satisfies ϕ, written $v \models \phi$, if

- $\phi \in L_v$, for $\phi \in C(\psi)$; or
- $\neg\phi \notin L_v$, for $\neg\phi \in C(\psi)$.

(We do this to reduce the size of C, to include "positives only"). Note that we implicitly have $v \models \mathtt{tt}$ and $v \not\models \mathtt{ff}$. We do not allow inconsistent labelings, according to the following rules. For any vertex $v = (s_v, L_v)$:

L0. If p is an atomic proposition, then $p \in L_v$ if and only if $p \in L(s_v)$.

L1. For $\phi = \phi_1 \wedge \phi_2$, $v \models \phi$ if and only if $v \models \phi_1$ and $v \models \phi_2$.

L2. If $v \models \phi_2$, then $v \models \phi_1 \,\mathsf{U}\, \phi_2$ (if this formula belongs to C).

L3. If $v \models \phi_1 \,\mathsf{U}\, \phi_2$, then $v \models \phi_1$ or $v \models \phi_2$.

We add edges to the tableau graph according to the following rules. We add an edge from vertex $v = (s_v, L_v)$ to vertex $v' = (s'_v, L'_v)$ if and only if:

E0. $(s_v, s'_v) \in \mathcal{R}$ (the edge matches one in the Kripke structure), and

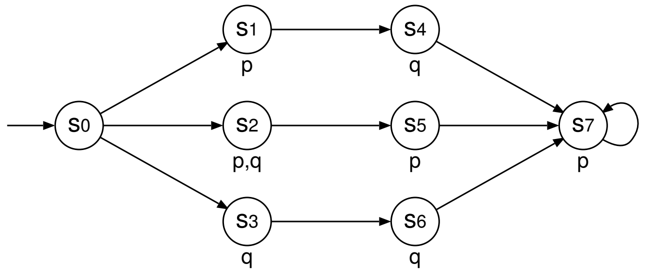
E1. for every formula of the form $\mathsf{X}\,\phi \in C(\psi)$,

$$\mathsf{X}\,\phi \in L_v \qquad \text{if and only if} \qquad v' \models \phi, \qquad \text{and}$$

E2. for every formula of the form $\phi_1 \,\mathsf{U}\, \phi_2 \in C(\psi)$, if $v \models \phi_1$ and $v \not\models \phi_2$, then

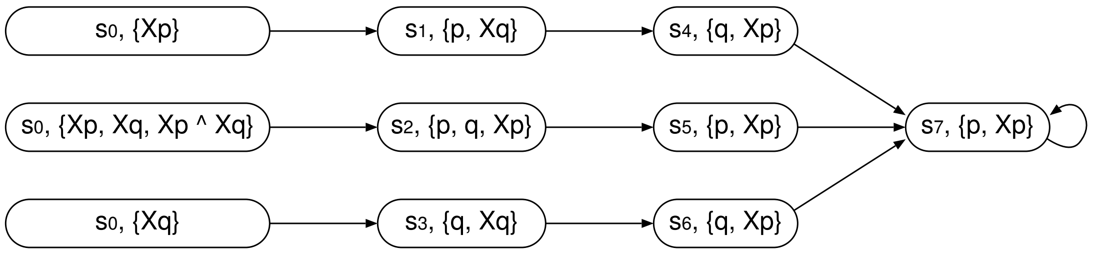
$$\phi_1 \,\mathsf{U}\, \phi_2 \in L_v \qquad \text{if and only if} \qquad \phi_1 \,\mathsf{U}\, \phi_2 \in L'_v.$$

Example 8.7

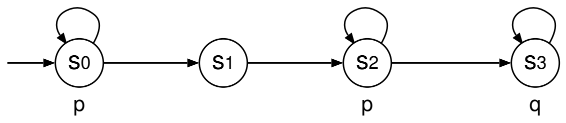


For the above Kripke structure, draw the tableau graph for formula $\psi = \mathsf{X}\,p \wedge \mathsf{X}\,q$.

Solution: First, note that we have $C(\psi) = \{p, q, \mathsf{X}\,p, \mathsf{X}\,q, \mathsf{X}\,p \wedge \mathsf{X}\,q\}$. We obtain the following tableau graph, where vertices with no outgoing edges are omitted from the drawing to save space.

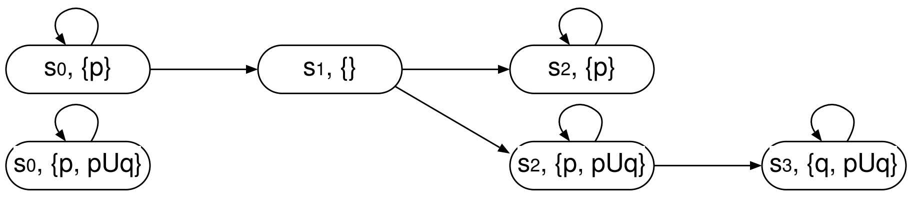


Example 8.8



For the above Kripke structure, draw the tableau graph for formula $\psi = p \,\mathsf{U}\, q$.

Solution: We have $C(\psi) = \{p, q, p \,\mathsf{U}\, q\}$. For states satisfying p, we can choose to label vertices with $p \,\mathsf{U}\, q$ or not. For states satisfying q, we must label with $p \,\mathsf{U}\, q$. For states satisfying neither p nor q, we cannot label with $p \,\mathsf{U}\, q$. Taking care with the edge rules, we can obtain the following tableau graph.



Property 8.17 *For a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$, the tableau graph for formula ψ has at most*

$$\begin{aligned} &2^{|\psi|} \cdot |\mathcal{S}| && \textit{states, and} \\ &2^{|\psi|} \cdot |\mathcal{R}| && \textit{edges.} \end{aligned}$$

8.4.2 Model checking with the tableau graph: theory

Suppose we have a tableau graph $T = (V, E)$ for Kripke structure M and formula ψ, and want to determine if $M, s \models \mathsf{E}\,\psi$ for some state s. What we would like to do is say something like

$$M, s \models \mathsf{E}\,\psi \quad \text{if and only if} \quad \exists v = (s, L_v) \in V, v \models \psi.$$

There is a small problem with this, though. It is best illustrated by example.

Example 8.9

From the Kripke structure in Example 8.7, we have $s_0 \models \mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$, and in the tableau graph, there is a vertex with state s_0 labeled with $\mathsf{X}\,p \wedge \mathsf{X}\,q$. No other states satisfy $\mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$, and no other vertices are labeled with $\mathsf{X}\,p \wedge \mathsf{X}\,q$. This works fine.

Example 8.10

From the Kripke structure in Example 8.8, we have $s_0 \not\models \mathsf{E}\,p\,\mathsf{U}\,q$, but in the tableau graph, there *is* a vertex with state s_0 labeled with $p\,\mathsf{U}\,q$.

Looking carefully at Example 8.10, we can see the problem: it is possible to have a vertex labeled with a formula $\phi_1\,\mathsf{U}\,\phi_2$, and yet have no path to a vertex satisfying ϕ_2.

Definition 8.18 *An* eventuality sequence *is an infinite path $\pi = (v_0, v_1, \ldots)$ in a tableau graph T such that, for any vertex $v_i = (s_i, L_i)$ in π, if there is a formula of the form $\phi_1\,\mathsf{U}\,\phi_2$ in L_i, then there is a vertex v_j in π with $v_j \models \phi_2$, with $i \leq j$.*

Note: by construction of the tableau graph (rule E2), on an eventuality sequence, once we reach a vertex labeled with a formula $\phi_1\,\mathsf{U}\,\phi_2$, all following vertices will be labeled with $\phi_1\,\mathsf{U}\,\phi_2$, until we reach a vertex satisfying ϕ_2. Because it is an eventuality sequence, it is guaranteed that we will eventually satisfy ϕ_2.

Example 8.11

Continuing Example 8.8, looking at the tableau graph, the infinite sequence

$$((s_0, \{p, p \mathsf{U} q\}), (s_0, \{p, p \mathsf{U} q\}, \ldots)$$

is not an eventuality sequence, because we never reach a vertex labeled with q. The infinite sequence

$$((s_0, \{p\}), (s_1, \emptyset), (s_2, \{p\}), (s_2, \{p\}), \ldots)$$

is an eventuality sequence, because there are no $\phi_1 \mathsf{U} \phi_2$ formulas that never reach ϕ_2. Also, the infinite sequence

$$((s_0, \{p\}), (s_1, \emptyset), (s_2, \{p, p \mathsf{U} q\}), (s_2, \{p, p \mathsf{U} q\}), (s_3, \{q, p \mathsf{U} q\}), (s_3, \{q, p \mathsf{U} q\}, \ldots)$$

is an eventuality sequence.

Property 8.19 *$M, s \models \mathsf{E}\,\psi$ if and only if the tableau graph for ψ contains an eventuality sequence starting from a vertex v with state s, and $v \models \psi$.*

Proof $\rightarrow$: *If $M, s \models \mathsf{E}\,\psi$, then there exists a path $\pi = (s_0 = s, s_1, s_2, \ldots)$ with $\pi \models \psi$. For all i, let $v_i = (s_i, L_i)$ where, for all $\phi \in C(\psi)$, $\phi \in L_i$ if and only if $\pi^i \models \phi$. Note v_i is a valid vertex.*

First, we must show that $(v_0, v_1, \ldots)$ is a path in the tableau graph. For each (v_i, v_{i+1}), rule E0 is satisfied because π is a path. Rule E1 is satisfied because $\pi^i \models \mathsf{X}\,\phi$ iff $\pi^{i+1} \models \phi$. Rule E2 is satisfied because if $\pi^i \models \phi_1$ and $\pi^i \not\models \phi_2$, then by Property 8.15 we have $\pi^i \models \phi_1 \mathsf{U} \phi_2$ if and only if $\pi^{i+1} \models \phi_1 \mathsf{U} \phi_2$.

Finally, we must show that $(v_0, v_1, \ldots)$ is an eventuality sequence:

$$\begin{array}{lll}
\phi_1 \mathsf{U} \phi_2 \in L_i & \rightarrow \quad \pi^i \models \phi_1 \mathsf{U} \phi_2 & \textit{(by construction of } L_i\textit{)} \\
& \rightarrow \quad \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j & \\
& \rightarrow \quad \exists j \geq i, \pi^j \in L_j, \pi^{i'} \in L_{i'}, \forall i \leq i' < j & \textit{(by construction of } L_i, \ldots, L_j\textit{)}
\end{array}$$

Proof $\leftarrow$: *Suppose there is an eventuality sequence $(v_0 = v, v_1, v_2, \ldots)$, where $v_i = (s_i, L_i)$. Let $\pi = (s_0, s_1, \ldots)$. We will prove the stronger claim that $\pi^i \models \psi$ if and only if $v_i \models \psi$, by induction on the structure of ψ.*

In the base case, $\psi = p$, an atomic proposition, and by labeling rule L0, $v_i \models p$ iff $s_i \models p$.

Now, assume it holds for all subformulas of ψ, and prove it holds for ψ.

If $\psi = \neg\phi$ *, then $\pi^i \models \psi$ iff $\pi^i \not\models \phi$. By the inductive hypothesis, $\pi^i \models \phi$ iff $v_i \models \phi$, and it follows that $\pi^i \models \neg\phi$ iff $v_i \models \neg\phi$.*

If $\psi = \phi_1 \wedge \phi_2$ *, then by the inductive hypothesis, $\pi^i \models \phi_1$ iff $v_i \models \phi_1$, and $\pi^i \models \phi_2$ iff $v_i \models \phi_2$. From labeling rule L1, $v_i \models \phi_1 \wedge \phi_2$ iff $v_i \models \phi_1$ and $v_i \models \phi_2$. It follows that $\pi^i \models \phi_1 \wedge \phi_2$ iff $v_i \models \phi_1 \wedge \phi_2$.*

If $\psi = \mathsf{X}\,\phi$ *, then we have the following.*

$$\begin{array}{lll}
\pi^i \models \mathsf{X}\,\phi & \Leftrightarrow \quad \pi^{i+1} \models \phi & \textit{(definition of } \mathsf{X}\textit{)} \\
& \Leftrightarrow \quad v_{i+1} \models \phi & \textit{(by inductive hypothesis)} \\
& \Leftrightarrow \quad v_i \models \mathsf{X}\,\phi & \textit{(by edge rule E1)}
\end{array}$$

If $\psi = \phi_1 \mathsf{U}\, \phi_2$ *, then we have*

$$\begin{array}{rcll}
v_i \models \phi_1 \mathsf{U}\, \phi_2 & \to & v_i \models \phi_2 \vee v_i \not\models \phi_2, v_i \models \phi_1 & \textit{(labeling rule L3)} \\
& \to & v_i \models \phi_2 \vee v_i \not\models \phi_2, v_i \models \phi_1, v_{i+1} \models \phi_1 \mathsf{U}\, \phi_2 & \textit{(edge rule E2)} \\
& \to & \pi^i \models \phi_2 \vee \pi^i \models \phi_1, v_{i+1} \models \phi_1 \mathsf{U}\, \phi_2 & \textit{(inductive hypothesis)} \\
& \to & \pi^i \models \phi_2 \vee (\pi^i \models \phi_1, \pi^{i+1} \models \phi_2 \vee (\pi^{i+1} \models \phi_1 \ldots)) & \\
& \to & \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j & \textit{(eventuality sequence)} \\
& \to & \pi^i \models \phi_1 \mathsf{U}\, \phi_2 &
\end{array}$$

$$\begin{array}{rcll}
\pi^i \models \phi_1 \mathsf{U}\, \phi_2 & \to & \exists j \geq i, \pi^j \models \phi_2, \pi^{i'} \models \phi_1, \forall i \leq i' < j & \\
& \to & \exists j \geq i, v_j \models \phi_2, v_{i'} \models \phi_1, \forall i \leq i' < j & \textit{(inductive hypothesis)} \\
& \to & \exists j \geq i, v_j \models \phi_2, v_j \models \phi_1 \mathsf{U}\, \phi_2, v_{i'} \models \phi_1, \forall i \leq i' < j & \textit{(labeling rule L2)} \\
& \to & \exists j \geq i, v_j \models \phi_2, v_{i'} \models \phi_1 \mathsf{U}\, \phi_2, \forall i \leq i' \leq j & \textit{(edge rule E2)} \\
& \to & v_i \models \phi_1 \mathsf{U}\, \phi_2 &
\end{array}$$

Example 8.12

Continuing Example 8.7, we have $s_0 \models \mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$, because in the tableau graph, we have the eventuality sequence

$$((s_0, \{\mathsf{X}\,p, \mathsf{X}\,q, \mathsf{X}\,p \wedge \mathsf{X}\,q\}), (s_2, \{p, q, \mathsf{X}\,p\}), (s_5, \{p, \mathsf{X}\,p\}), (s_7, \{p, \mathsf{X}\,p\}), \ldots)$$

which starts with a vertex for state s_0, labeled with $\mathsf{X}\,p \wedge \mathsf{X}\,q$.

Example 8.13

Continuing Example 8.8, we have $s_0 \not\models \mathsf{E}\,p\,\mathsf{U}\,q$, because there is no eventuality sequence starting with a vertex for state s_0 labeled with $p\,\mathsf{U}\,q$.

How do we find infinitely–long paths that satisfy a property in a finite graph? We look at the strongly connected components (SCCs). This motivates the following definition and theorem.

Definition 8.20 *A non-trivial*[1] *SCC in a tableau graph is* self–fulfilling *if for every vertex $v = (s_v, L_v)$ in the SCC, if there is a formula of the form $\phi_1 \mathsf{U}\, \phi_2$ in L_v, then there exists a vertex v' in the SCC with $v' \models \phi_2$.*

Property 8.21 *For any vertex v in a self–fulfilling SCC $\mathcal{C}$, there is an eventuality sequence starting with v.*

Proof: *We construct an eventuality sequence as follows. If v is not labeled with any formula of the form $\phi_1 \mathsf{U}\, \phi_2$, then choose some successor of v in $\mathcal{C}$, and repeat. Otherwise, for each such formula, there exists a vertex that satisfies ϕ_2. Add a path from v that visits each of these vertices. Repeat the process from the final vertex. Clearly, the resulting infinitely–long sequence is an eventuality sequence.*

Property 8.22 *There is an eventuality sequence starting from vertex v if and only if there is a path in the tableau graph from v to a self–fulfilling SCC.*

[1]A SCC is *non-trivial* if it contains more than one state, or is a single state with a loop.

Proof $\rightarrow$: *Suppose there is an eventuality sequence starting from v. Let $\mathcal{I}$ be the set of vertices that appear infinitely often in the sequence. Because the graph is finite, we must have $|\mathcal{I}| > 0$, and there is some non-trivial SCC $\mathcal{C}$ such that $\mathcal{I} \subseteq \mathcal{C}$. We will show that $\mathcal{C}$ is self–fulfilling, by contradiction. Suppose there exists a vertex $v' \in \mathcal{C}$, labeled with $\phi_1 \mathsf{U} \phi_2$, and no vertex exists in $\mathcal{C}$ satisfying ϕ_2. By construction of the tableau graph, all successors of v' must also be labeled with $\phi_1 \mathsf{U} \phi_2$, and therefore all vertices in $\mathcal{C}$ are labeled with $\phi_1 \mathsf{U} \phi_2$. But since $\mathcal{I} \subseteq \mathcal{C}$, if we look at the "end" of the eventuality sequence (after the point where all remaining vertices are in $\mathcal{I}$), we have vertices labeled with $\phi_1 \mathsf{U} \phi_2$ and no vertex satisfying ϕ_2. But this is impossible, in an eventuality sequence.*

Proof $\leftarrow$: *Suppose there is a path in the tableau graph from v to a self–fulfilling SCC $\mathcal{C}$. Consider some path $(v_0 = v, v_1, v_2, \ldots, v_j, v')$ where $v' \in \mathcal{C}$. From Property 8.21, there is an eventuality sequence $(v', v'_1, v'_2, \ldots)$. We show that $(v_0, \ldots, v_j, v', v'_1, v'_2, \ldots)$ is also an eventuality sequence. For any vertex v_i, if it is labeled with $\phi_1 \mathsf{U} \phi_2$, then either ϕ_2 holds for some vertex before reaching v', or not. If not, then v' must also be labeled with $\phi_1 \mathsf{U} \phi_2$, and since $(v', v'_1, v'_2, \ldots)$ is an eventuality sequence, there is some v'_j with $v'_j \models \phi_2$.*

Property 8.23 *$M, s \models \mathsf{E}\,\psi$ if and only if, in the tableau graph for ψ, there is a path from a vertex with state s, labeled with ψ, to a self–fulfilling SCC.*

Proof: *Follows immediately from Property 8.19 and Property 8.22.*

8.4.3 Model checking with the tableau graph: algorithm

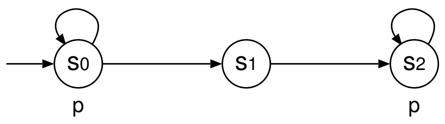
Based on the above discussion, we have the following algorithm for determining which states in a Kripke structure satisfy $\mathsf{E}\,\psi$ for an LTL path formula ψ.

1. Build the tableau graph $T = (V, E)$.
2. Determine the SCCs for T.
3. For each non-trivial SCC, determine if it is self–fulfulling or not:
   (a) Build a list of all formulas of the form $\phi_1 \mathsf{U} \phi_2$ that label vertices in the SCC.
   (b) For each formula in the list, check that some state in the SCC satisfies ϕ_2.
4. Label all self–fulfilling SCC vertices as *green*, and any vertex that can reach a green vertex as *green*.
5. $s \models \mathsf{E}\,\psi$ if and only if there is a green vertex (s, L) with $\psi \in L$.

What is the complexity of this algorithm? Step (1) is $\mathcal{O}(|T|) = \mathcal{O}(|V| + |E|)$, as is Step (2). Step (3) is $\mathcal{O}(|V|)$ in the worst case, Step (4) is $\mathcal{O}(|T|)$, and Step (5) gives us an answer for every state in $\mathcal{O}(|V|)$. Therefore, from Property 8.17, the (worst case) complexity to check every state in Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ against LTL formula ψ is

$$\mathcal{O}(2^{|\psi|}(|\mathcal{S}| + |\mathcal{R}|)).$$

Example 8.14



For the above Kripke structure, which states satisfy $\mathsf{A}\,\mathsf{F}\mathsf{G}\,p$?
Solution: First, rewrite the formula in terms of E and U:

$$\begin{aligned} \mathsf{A}\,\mathsf{F}\mathsf{G}\,p &\equiv \neg\mathsf{E}\neg\mathsf{F}\mathsf{G}\,p \\ &\equiv \neg\mathsf{E}\neg\mathsf{F}\neg\mathsf{F}\neg p \\ &\equiv \neg\mathsf{E}\neg\mathsf{F}\neg(\mathtt{tt}\,\mathsf{U}\,\neg p) \\ &\equiv \neg\mathsf{E}\neg(\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)) \end{aligned}$$

The subformulas of ψ are:

$$\begin{aligned} C(\psi) &= C(\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)) \\ &= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)\} \cup C(\mathtt{tt}) \cup C(\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)) \\ &= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p)\} \cup C(\mathtt{tt}\,\mathsf{U}\,\neg p) \\ &= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p), \mathtt{tt}\,\mathsf{U}\,\neg p\} \cup C(\mathtt{tt}) \cup C(\neg p) \\ &= \{\mathtt{tt}\,\mathsf{U}\,\neg(\mathtt{tt}\,\mathsf{U}\,\neg p), \mathtt{tt}\,\mathsf{U}\,\neg p, p\}. \end{aligned}$$

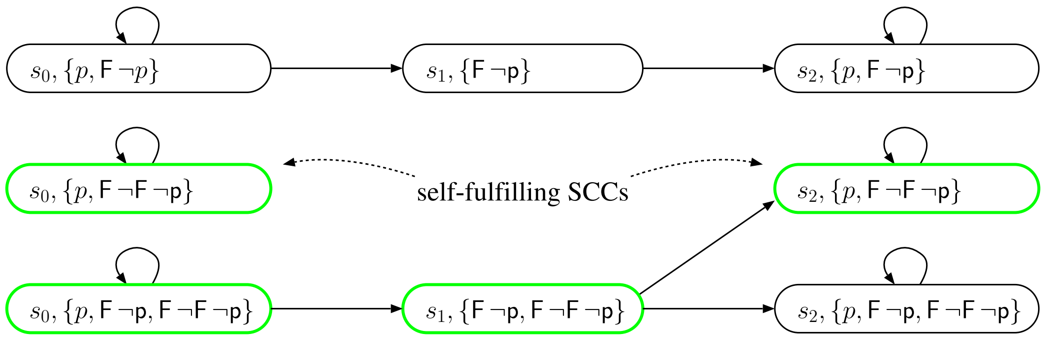
For shorthand, though, we will use $C(\psi) = \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$. Now we can build the tableau graph. First, we determine which vertices are in the graph according to the labeling rules, noting that $\neg p \rightarrow \mathsf{F}\,\neg p$ and $\neg\mathsf{F}\,\neg p \rightarrow \mathsf{F}\,\neg\mathsf{F}\,\neg p$. That leaves us with the vertices

$$\begin{aligned} V \;= \{ \quad & (s_0, \{p, \mathsf{F}\,\neg p\}), (s_0, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), (s_0, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), \\ & (s_1, \{\mathsf{F}\,\neg p\}), (s_1, \{\mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), \\ & (s_2, \{p, \mathsf{F}\,\neg p\}), (s_2, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}), (s_2, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}) \quad \}. \end{aligned}$$

Rewriting the U edge restriction from vertex v to v' for our specific formulas, we obtain:

1. If $v \models p$, then $\mathsf{F}\,\neg p$ labels both v and v', or neither.
2. If $v \models \mathsf{F}\,\neg p$, then $\mathsf{F}\,\neg\mathsf{F}\,\neg p$ labels both v and v', or neither.

These rules give us the following tableau graph.


$s_0, \{p, \mathsf{F}\,\neg p\}$
$s_1, \{\mathsf{F}\,\neg p\}$
$s_2, \{p, \mathsf{F}\,\neg p\}$
$s_0, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$
self-fulfilling SCCs
$s_2, \{p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$
$s_0, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$
$s_1, \{\mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$
$s_2, \{p, \mathsf{F}\,\neg p, \mathsf{F}\,\neg\mathsf{F}\,\neg p\}$

In the graph, the indicated SCCs are self–fulfilling because the vertices satisfy $\neg \mathsf{F} \neg p$. The others are not, because the vertices are all labeled with p, and therefore $\mathsf{F} \neg p$ cannot be satisfied. From the green vertices, we see that all states satisfy $\mathsf{E}\,\mathsf{F} \neg \mathsf{F} \neg p \equiv \mathsf{E}\,\mathsf{FG}\, p$ (there exists a path where, eventually, we never see $\neg p$), and no state satisfies $\mathsf{E} \neg \mathsf{FG}\, p$. Therefore, all states satisfy $\mathsf{A}\,\mathsf{FG}\, p$.

8.5 LTL model checking with Büchi Automata

Another algorithm for LTL model checking is based on Büchi Automata[2]. Before we can understand this algorithm, we must first cover some automata theory.

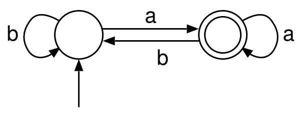
8.5.1 Büchi Automata

Basic idea

A Büchi Automaton is a finite state machine for accepting infinte words. The idea is similar to deterministic and non-deterministic finite automata, except those are languages of *finite* words. The idea is:

- Start in an initial state.
- Follow edges in finite automaton, according to "next input symbol"
- If we end up in an "accepting" state at the end of the input, then the word is accepted; otherwise it is not accepted.

Example 8.15



The above DFA accepts finite words over alphabet $\{a, b\}$ that end in "a". It is deterministic because in every state, there is at most one outgoing edge for each symbol.

Büchi automata are the same idea, except there's a catch: because words are infinitely long, we cannot "end up" in a state. We will need more interesting acceptance criteria. But first, some definitions and terminology.

Definition 8.24 *A (non-deterministic)* ω-automaton *is:*

- Σ*, a finite alphabet (symbols in the input words)*
- Q*, a finite set of states*
- Q_0*, a non-empty set of initial states with* $Q_0 \subseteq Q$*.*
- $\Delta \subseteq Q \times \Sigma \times Q$*, the transition relation*

[2]See for example: M.Vardi and P. Wolper. "An Automata–Theoretic Approach to Automatic Program Verification". In *LICS'1986*, pp. 332–344, 1986.

- *An acceptance condition, depending on the specific type of automaton (discussed below)*

Definition 8.25 *An* input *is an infinite word* $\alpha = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$.

Definition 8.26 *For an input word* $\alpha = \sigma_0\sigma_1\sigma_2 \ldots$*, a* run *is an infinite sequence of states* $\rho = q_0, q_1, q_2, \ldots \in Q^\omega$ *such that* $q_0 \in Q_0$ *and*

$$(q_i, \sigma_i, q_{i+1}) \in \Delta, \quad \forall i$$

Definition 8.27 *For a given run* $\rho = q_0, q_1, q_2, \ldots \in Q^\omega$*, the set of states visited infinitely often is denoted*

$$\mathit{Inf}(\rho) = \{q \mid q \in Q, \exists \textit{ infinitely many } i \textit{ such that } q_i = q\}$$

Note that the set $\mathit{Inf}(\rho)$ can never be empty because Q is finite and ρ is not.

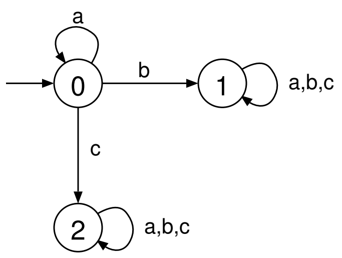
Standard Büchi automata

Definition 8.28 *A* (standard) Büchi automaton *is an* ω*-automaton with*

- F*, a finite set of states with* $F \subseteq Q$*. A run* ρ *is accepting if and only if* $\mathit{Inf}(\rho) \cap F \neq \emptyset$.

Example 8.16

Consider the (standard) Büchi automaton given by:



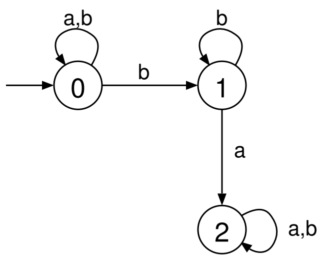
with $\Sigma = \{a, b, c\}$, $Q = \{0, 1, 2\}$, $Q_0 = \{0\}$, $F = \{1\}$. This is a deterministic automaton because no state has more than one outgoing edge for the same symbol. Consider the input word $\alpha = aaabccccc^\omega$. This produces the run $0, 0, 0, 0, 1, 1, 1, 1^\omega$. This run is accepting because state $1 \in F$ is visited infinitely many times. We can characterize the words that produce accepting runs as words where b eventually appears, and appears before any c.

Definition 8.29 *The* language accepted by *automaton* A*, denoted* $L(A)$*, is defined by*

$$L(A) = \{\alpha \mid \exists \textit{ an accepting run in } A \textit{ for } \alpha\}$$

Example 8.17

Consider the (standard) Büchi automaton given by:



with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2\}$, $Q_0 = \{0\}$, $F = \{1\}$. This is non-deterministic because in state 0, if symbol b appears, we can either stay in state 0 or go to state 1. Consider the input word $\alpha = babbbb^\omega$. There are many possible runs for this word, including:

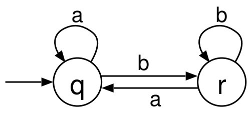
- $\rho = 0, 1, 2, 2, 2^\omega$, which *is not* accepting because $Inf(\rho) = \{2\}$ and so $Inf(\rho) \cap F = \emptyset$.
- $\rho = 0, 0, 0, 0, 0, 1, 1, 1, 1, 1^\omega$, which *is* accepting because $Inf(\rho) = \{1\}$ and so $Inf(\rho) \cap F \neq \emptyset$.

Because there is *some* accepting run for $babbbb^\omega$, we have $babbbb^\omega \in L(A)$. Note that

$$L(A) = \{\alpha \mid \alpha \text{ ends with only } b\text{'s}\}.$$

Example 8.18

Consider the (standard) Büchi automaton A given by:



with $\Sigma = \{a, b\}$, $Q = \{q, r\}$, $Q_0 = \{q\}$, $F = \{r\}$. What is $L(A)$?

Solution: State r is visited infinitely often if and only if the input word contains infinitely many b's. Therefore,

$$L(A) = \{\alpha \mid \alpha \text{ contains infinitely many } b\text{'s}\}.$$

Generalized Büchi automata

Definition 8.30 *A* generalized Büchi automaton *is an ω-automaton with*

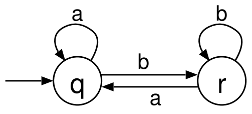
- *$\mathcal{F}$, a finite set of subsets of Q. A run ρ is accepting if and only if*

$$Inf(\rho) \cap F \neq \emptyset, \quad \forall F \in \mathcal{F}$$

Note that a standard Büchi automaton is the special case of a generalized Büchi automaton where $\mathcal{F} = \{F\}$.

Example 8.19

Consider the generalized Büchi automaton A given by:



with $\mathcal{F} = \{\{q\}, \{r\}\}$. What is $L(A)$?

Solution: For example, input word $\alpha = ababababab\ldots$ produces the run $\rho = q, r, q, r, q, r, q, r, \ldots$, with $\mathit{Inf}(\rho) = \{q, r\}$. Since $\mathit{Inf}(\rho) \cap \{q\} \neq \emptyset$ and $\mathit{Inf}(\rho) \cap \{r\} \neq \emptyset$, this is an accepting run and $\alpha \in L(A)$.

Notice that state q is visited infinitely often if and only if the input word contains infinitely many a's. State r is visited infinitely often if and only if the input word contains infinitely many b's. Since an accepting run must visit *both* state q and state r infinitely often,

$$L(A) = \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s and } b\text{'s}\}.$$

Also notice, this is different from using a standard Büchi automaton with $F = \{q, r\}$, which requires *either* that state q *or* state r is visited infinitely often.

Note: there are many other types of ω-automata, with various accepting conditions. For LTL model checking, we need only standard and generalized Büchi automata.

8.5.2 Some important algorithms for Büchi Automata

Language Emptiness

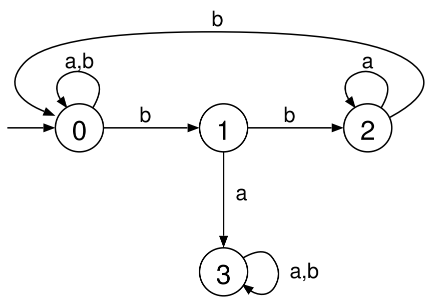
Definition 8.31 *The* language emptiness problem*: given a standard Büchi automaton A, determine if $L(A) = \emptyset$.*

This problem is *decidable*. Clearly, $L(A) \neq \emptyset$ if and only if there exists *some* accepting run ρ for some input word. An accepting run must visit some state $q \in F$ infinitely often. Since there are only finitely many states, this can happen only if there is a cycle containing q, and we can reach q from an initial state. Thus we have the following algorithm.

1. Treating the Büchi automaton as a directed graph (ignoring the symbols), determine the SCCs.
2. For any state in F that belongs to a non-trivial SCC, mark it as *green*.
3. For any state that can reach a green state, mark it as green.
4. $L(A) \neq \emptyset$ iff there exists a green state in Q_0.

Example 8.20

Consider the (standard) Büchi automaton A given by:



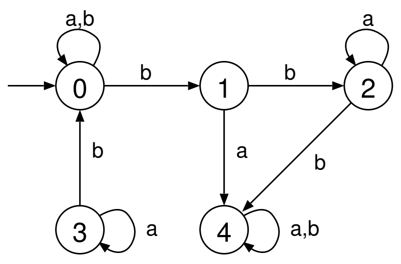
with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3\}$, $Q_0 = \{0\}$, $F = \{1\}$. Is $L(A) = \emptyset$?

Solution: Running the algorithm:

1. The non-trivial SCCs are $\{0, 1, 2\}$ and $\{3\}$.
2. State $1 \in F$ belongs to a non-trivial SCC, so mark it as green.
3. States 0 and 2 can reach 1, so mark those as green.
4. Since $0 \in Q_0$ and 0 is green, we conclude that $L(A) \neq \emptyset$.

Example 8.21

Consider the (standard) Büchi automaton A given by:



with $\Sigma = \{a, b\}$, $Q = \{0, 1, 2, 3, 4\}$, $Q_0 = \{0\}$, $F = \{1, 3\}$. Is $L(A) = \emptyset$?

Solution: Running the algorithm:

1. The non-trivial SCCs are $\{0\}$, $\{2\}$, $\{3\}$, and $\{4\}$.
2. State $3 \in F$ belongs to a non-trivial SCC, so mark it as green. (State 1 does not.)
3. No other states can reach a green state.
4. Since $\{0\} = Q_0$ and 0 is not green, we conclude that $L(A) = \emptyset$.

Converting from generalized to standard Büchi

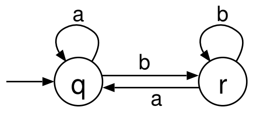
Given a generalized Büchi automaton A, can we build a standard Büchi automaton A' such that $L(A) = L(A')$?

Answer: YES. First, the idea.

- Suppose the generalized Büchi automaton has $\mathcal{F} = \{F_0, F_1, \ldots, F_{n-1}\}$ i.e., n sets of states that must be visited infinitely often. Then, our standard Büchi automaton state will be: which generalized Büchi automaton state we're in, plus a counter that goes from 0 to $n-1$.
- When the counter is i, if we visit a state in F_i, then increment the counter (modulo n) when we consume the next input symbol.
- When the counter wraps around to 0, we have seen a state in F_i for all i.
- Because each set F_i is finite, if the counter wraps around to 0 infinitely many times, then the generalized Büchi acceptance condition is met.

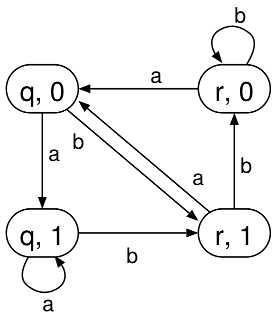
Example 8.22

Consider the generalized Büchi automaton A from Example 8.19:



with $\mathcal{F} = \{F_0, F_1\}$, $F_0 = \{q\}$, $F_1 = \{r\}$. Build an equivalent standard Büchi automaton.

Solution We can add a counter with values $\{0, 1\}$. If the counter is 0 and the state is q, then outgoing edges should set the counter to 1. If the counter is 1 and the state is r, then outgoing edges should set the counter to 0. That gives us:



Note: between any two visits to state $(q,0)$, we must visit state $(r,1)$ because all paths from $(q,0)$ to $(q,0)$ pass through $(r,1)$. Thus, if $(q,0)$ is visited infinitely often, so is $(r,1)$. So our standard Büchi automaton can use accepting condition $F = \{(q,0)\}$.

Given a generalized Büchi automaton $A = (\Sigma, Q, Q_0, \Delta, \mathcal{F})$ with $\mathcal{F} = \{F_0, \ldots, F_{n-1}\}$, we can build the standard Büchi automaton $A' = (\Sigma, Q', Q'_0, \Delta', F')$ with

- $Q' = Q \times \{0, 1, \ldots, n-1\}$, i.e., state plus the counter
- $Q'_0 = Q_0 \times \{0\}$, i.e., the counter starts at 0
- $F' = F_0 \times \{0\}$

- $(q_i, c), a, (q_j, d) \in \Delta'$ if and only if both:
  - $(q_i, a, q_j) \in \Delta$
  - If $q_i \in F_i$ and $c = i$, then $d = (c + 1) \bmod n$, otherwise $d = c$.

  where the first part of the rule handles edges in the original automaton and the second part of the rule handles the counter.

It can be shown that $L(A) = L(A')$.

Product construction

Given two standard Büchi automata

- $A_1 = (\Sigma, Q_1, Q_{01}, \Delta_1, F_1)$
- $A_2 = (\Sigma, Q_2, Q_{02}, \Delta_2, F_2)$

define the product automaton $A_1 \times A_2 = (\Sigma, Q, Q_0, \Delta, \mathcal{F})$ as

- $Q = Q_1 \times Q_2$
- $Q_0 = Q_{01} \times Q_{02}$
- $\mathcal{F} = \{F_1 \times Q_2, Q_1 \times F_2\}$
- $(q_1, q_2), a, (q_1', q_2') \in \Delta$ if and only if $q_1, a, q_1' \in \Delta_1$ and $q_2, a, q_2' \in \Delta_2$.

Note that this is a *generalized* Büchi automaton.

Property 8.32 *Word α produces run $\rho_1 = q_0, q_1, q_2, \ldots$ in A_1, and produces run $\rho_2 = r_0, r_1, r_2, \ldots$ in A_2, if and only if it produces run $\rho_1 \times \rho_2 = (q_0, r_0), (q_1, r_1), \ldots$ in $A_1 \times A_2$.*

Proof $\rightarrow$: *Let ρ_1 be a run for α in A_1, and ρ_2 be a run for α in A_2. By definition of a run, we have $(q_i, a, q_{i+1}) \in \Delta_1\ \forall i$ and $(r_i, a, r_{i+1}) \in \Delta_2\ \forall i$. By construction of Δ, we have $((q_i, r_i), a, (q_{i+1}, r_{i+1})) \in \Delta\ \forall i$. Thus $\rho_1 \times \rho_2$ is a run for α in $A_1 \times A_2$.*

Proof $\leftarrow$: *Let $\rho_1 \times \rho_2 = (q_0, r_0), (q_1, r_1), \ldots$ be a run for α in $A_1 \times A_2$. Then we have $((q_i, r_i), a, (q_{i+1}, r_{i+1})) \in \Delta\ \forall i$. But by definition of Δ, this says $(q_i, a, q_{i+1}) \in \Delta_1\ \forall i$ and $(r_i, a, r_{i+1}) \in \Delta_2\ \forall i$. Thus ρ_1 is a run for α in A_1, and ρ_2 is a run for α in A_2.*

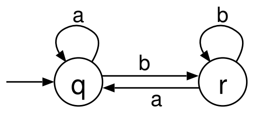
Property 8.33 *$L(A_1 \times A_2) = L(A_1) \cap L(A_2)$.*

Proof:

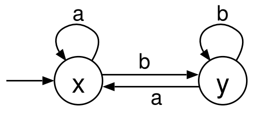
$$\begin{aligned}
\alpha \in L(A_1 \times A_2) &\leftrightarrow \textit{There is an accepting run } \rho_1 \times \rho_2 \textit{ on } \alpha \textit{ in } A_1 \times A_2 \\
&\leftrightarrow \textit{Run } \rho_1 \times \rho_2 \textit{ on } \alpha \textit{ in } A_1 \times A_2 \textit{ has} \\
&\quad\ \textit{Inf}(\rho_1 \times \rho_2) \cap (F_1 \times Q_2) \neq \emptyset \textit{ and} \\
&\quad\ \textit{Inf}(\rho_1 \times \rho_2) \cap (Q_1 \times F_2) \neq \emptyset \\
&\leftrightarrow \textit{Run } \rho_1 \textit{ on } \alpha \textit{ in } A_1 \textit{ has } \textit{Inf}(\rho_1) \cap F_1 \neq \emptyset \\
&\quad\ \textit{and run } \rho_2 \textit{ on } \alpha \textit{ in } A_2 \textit{ has } \textit{Inf}(\rho_2) \cap F_2 \neq \emptyset \\
&\leftrightarrow \textit{There is an accepting run } \rho_1 \textit{ on } \alpha \textit{ in } A_1, \textit{ and} \\
&\quad\ \textit{there is an accepting run } \rho_2 \textit{ on } \alpha \textit{ in } A_2 \\
&\leftrightarrow \alpha \in L(A_1) \textit{ and } \alpha \in L(A_2)
\end{aligned}$$

Example 8.23

Consider the Büchi automaton A_1 given by:



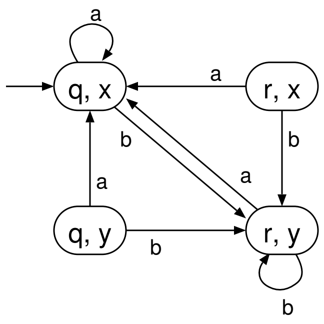
with $F = \{r\}$, and the Büchi automaton A_2 given by:



with $F = \{x\}$. It can be seen that

$$\begin{aligned} L(A_1) &= \{\alpha \mid \alpha \text{ contains infinitely many } b\text{'s}\} \\ L(A_2) &= \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s}\} \end{aligned}$$

The product automaton $A_1 \times A_2$ is given by:



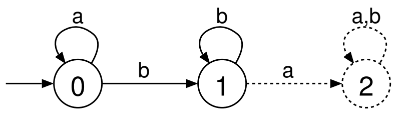
with $\mathcal{F} = \{\{rx, ry\}, \{qx, rx\}\}$, and note that

$$L(A_1 \times A_2) = \{\alpha \mid \alpha \text{ contains infinitely many } a\text{'s and } b\text{'s}\}.$$

Also notice that states (q, y) and (r, x) are unreachable! If we eliminate those states, we obtain a generalized Büchi automaton that is essentially the same as the one in Example 8.19.

Example 8.24

Consider the Büchi automaton A_3 given by:

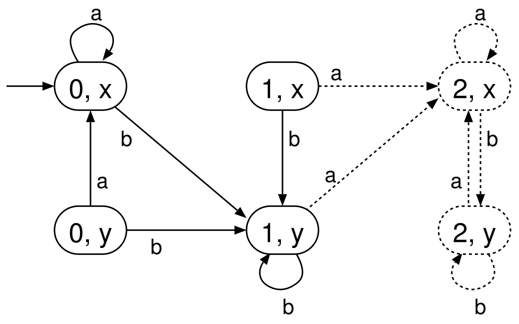


with $F = \{1\}$. It can be seen that

$$L(A_3) \;=\; \{\alpha \mid \alpha \text{ contains finitely many } a\text{'s, followed by infinitely many } b\text{'s}\}.$$

Note that we might draw this without state 2; any run in which symbol a is consumed in state 1 would either produce a finite run, or go to an imaginary "bad" state (in this case, state 2) from which it is impossible to reach any state in F. In any case, such a run would not be an accepting run. For this example, we include state 2 with dotted lines to show that its omission does not affect the result.

Using A_2 from Example 8.23, the product automaton $A_3 \times A_2$ is:



with $\mathcal{F} = \{\{\text{states with } 1\}, \{\text{states with } x\}\} = \{F_1, F_2\}$, where $F_1 = \{(1,x),(1,y)\}$ and if state 2 is omitted, $F_2 = \{(0,x),(1,x)\}$ otherwise $F_2 = \{(0,x),(1,x),(2,x)\}$. Now, state $(1,x)$ cannot be reached from the initial state, so to satisfy F_1 we must visit state $(1,y)$ infinitely often. But we cannot do this *and* visit state $(0,x)$ or $(2,x)$ infinitely often. So, there is no accepting run for this automaton. This makes sense, because

$$\begin{aligned} L(A_3 \times A_2) \;=\; \{\alpha \;\mid\; & \alpha \text{ contains finitely many } a\text{'s, followed by infinitely many } b\text{'s} \\ & \textit{and } \alpha \text{ contains infinitely many } a\text{'s}\} \;=\; \emptyset. \end{aligned}$$

8.5.3 LTL model checking

LTL model checking using Büchi automata is based on the following idea[3]:

1. We will use the alphabet $\Sigma = 2^{\mathcal{P}}$, i.e., each symbol is a subset of $\mathcal{P}$, the set of atomic propositions. The infinite words as input can be thought of as encoding the set of propositions that hold in each step.

2. Build a standard Büchi automaton A_M from the Kripke structure M, to encode the paths possible from the Kripke structure. More specifically we build A_M such that

$$\begin{aligned} \mathit{Lang}(A_M) \;=\; \{\alpha \;\mid\; & \text{There is a path } \pi \text{ through the Kripke structure that produces} \\ & \text{the sequence of sets of atomic propositions } \alpha\text{; formally} \\ & L(\pi_i) = \alpha_i, \forall i\}. \end{aligned}$$

We will discuss this in more detail, below.

[3]M.Vardi and P. Wolper. "An Automata–Theoretic Approach to Automatic Program Verification". In *LICS'1986*, pp. 332–344, 1986.

3. For path formula ψ, build a standard Büchi automaton A_ψ, to encode the paths (in any Kripke structure) that satisfy ψ. More specifically,

$$Lang(A_\psi) \;=\; \{\alpha \;\;|\;\; \text{If path } \pi \text{ has } L(\pi_i) = \alpha_i, \forall i \text{ then } \pi \models \psi\}$$

   We will discuss this in more detail, below. Note that the algorithm will give a *generalized* Büchi automaton, and we may need to run the conversion algorithm to get an equivalent *standard* Büchi automaton.

4. Build the product automaton $A_M \times A_\psi$. Note that $Lang(A_M \times A_\psi)$ corresponds to the set of paths both through the Kripke structure M and that satisfy ψ.

5. Do a language emptiness check on automaton $A_M \times A_\psi$. We have $M \models \mathsf{E}\,\psi$ if and only if $Lang(A_M \times A_\psi) \neq \emptyset$.

We have already discussed all the steps above, except for step (2) which is easy, and step (3) which is difficult.

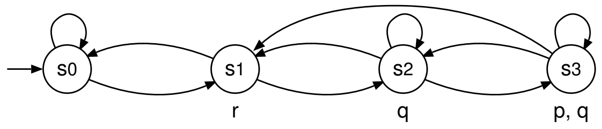
Kripke to Büchi conversion:

Informally, the idea is to make a copy of the Kripke structure, except we add a new state i which serves as the initial state, and edges in the Büchi automaton are labeled with the set of atomic propositions that hold in the destination state. Formally, given a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ with atomic propisitions $\mathcal{P}$, build Büchi automaton $A_M = (\Sigma, Q, Q_0, \Delta, F)$ with

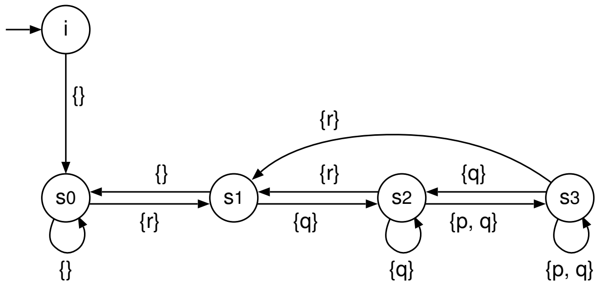
- $\Sigma = 2^{\mathcal{P}}$ (symbols are *sets* of atomic propositions)
- $Q = \mathcal{S} \cup \{i\}$ (i is a new initialization state)
- $Q_0 = \{i\}$
- $F = Q$ (every state is accepting)
- $(s, L(s'), s') \in \Delta$ if and only if $(s, s') \in \mathcal{R}$ or $s = i, s' \in \mathcal{S}_0$.

Example 8.25

Using Kripke structure for the CD player from Example 3.1 shown below,



we build the following Büchi automaton:



Since all states are accepting, all runs are accepting runs, and correspond to paths in the Kripke structure. For example, the Kripke structure path

$$s_0, s_1, s_2, s_1, s_2, s_1, s_2, \ldots$$

produces the sequence of sets

$$\{\}, \{r\}, \{q\}, \{r\}, \{q\}, \{r\}, \{q\}, \ldots$$

and note that this word is accepted by the Büchi automaton.

Path formula to Büchi conversion:

The idea here is:

- Use an algorithm that recursively processes formula ψ, and builds a graph
- From the graph, we will build a generalized Büchi automaton. Essentially, we will need to label the graph edges and determine the set $\mathcal{F}$.
- Formula ψ cannot contain F or G operators (we can rewrite these using U).
- Formula ψ must be written in "negation normal form", which means negations are allowed only directly before atomic propositions. Any formula can be written this way, using the following transformations:

$$\begin{aligned} \neg(\psi_1 \wedge \psi_2) &\equiv (\neg\psi_1) \vee (\neg\psi_2) \\ \neg(\psi_1 \vee \psi_2) &\equiv (\neg\psi_1) \wedge (\neg\psi_2) \\ \neg\mathsf{X}\,\psi &\equiv \mathsf{X}\,\neg\psi \\ \neg(\psi_1\,\mathsf{U}\,\psi_2) &\equiv (\neg\psi_1)\,\mathsf{R}\,(\neg\psi_2) \\ \neg(\psi_1\,\mathsf{R}\,\psi_2) &\equiv (\neg\psi_1)\,\mathsf{U}\,(\neg\psi_2) \end{aligned}$$

  Operator R is the "release" operator, which we have not seen before, and is defined as the dual of U. Formally,

$$\pi \models p\,\mathsf{R}\,q \quad \text{iff} \quad \forall j \geq 0, \text{if } \forall 0 \leq i < j, \pi^i \not\models p \text{ then } \pi^j \models q.$$

  and note that if we take the negation of this, we have[4]

$$\begin{aligned} \pi \not\models p\,\mathsf{R}\,q \quad &\text{iff} \quad \neg\big(\forall j \geq 0, (\forall 0 \leq i < j, \pi^i \not\models p) \rightarrow (\pi^j \models q)\big) \\ \pi \not\models p\,\mathsf{R}\,q \quad &\text{iff} \quad \exists j \geq 0,\ \neg\big((\forall 0 \leq i < j, \pi^i \not\models p) \rightarrow (\pi^j \models q)\big) \\ \pi \not\models p\,\mathsf{R}\,q \quad &\text{iff} \quad \exists j \geq 0, (\forall 0 \leq i < j, \pi^i \not\models p) \wedge (\pi^j \not\models q) \\ \pi \not\models p\,\mathsf{R}\,q \quad &\text{iff} \quad \exists j \geq 0, (\pi^j \models \neg q) \wedge (\forall 0 \leq i < j, \pi^i \models \neg p) \\ \pi \not\models p\,\mathsf{R}\,q \quad &\text{iff} \quad \pi \models \neg p\,\mathsf{U}\,\neg q. \end{aligned}$$

  Also, note that

$$\mathsf{G}\,q \equiv \neg\mathsf{F}\,\neg q \equiv \neg(\mathtt{tt}\,\mathsf{U}\,\neg q) \equiv \mathtt{ff}\,\mathsf{R}\,q$$

[4]Recall that $\neg(a \rightarrow b) \equiv a \wedge \neg b$.

which also follows easily from the definition of R above.

Since all the above transformations "push negations inside", we can apply them repeatedly until all negations are in front of atomic propositions.

Example 8.26

Write $\mathsf{FG}\,p$ in negative normal form, without operators F and G.

Solution:

$$\mathsf{FG}\,p \;\equiv\; \mathtt{tt}\,\mathsf{U}\,(\mathtt{ff}\,\mathsf{R}\,p)$$

Example 8.27

Write $\neg\mathsf{FG}\,p$ in negative normal form, without operators F and G.

Solution:

$$\neg\mathsf{FG}\,p \;\equiv\; \neg\,(\mathtt{tt}\,\mathsf{U}\,(\mathtt{ff}\,\mathsf{R}\,p)) \;\equiv\; \mathtt{ff}\,\mathsf{R}\,\neg(\mathtt{ff}\,\mathsf{R}\,p) \;\equiv\; \mathtt{ff}\,\mathsf{R}\,(\mathtt{tt}\,\mathsf{U}\,\neg p)$$

The translation algorithm builds a graph, where each node in the graph has the following data.

- **Old:** the set of (sub)formulas already processed for the node.
- **New:** the set of (sub)formulas still to process for the node.
- **Next:** the set of (sub)formulas that must hold in the next node in the graph.
- **Incoming:** the set of edges pointing to this node.

Note that for a node n, $n.Old \cup n.New$ is the set of formulas that should hold on paths starting here. The algorithm is given in detail below. The idea is to "expand" nodes, recursively. At each step, we will either

- replace a node with a new one (we will modify nodes "in place" instead); or
- split a node in half, which happens whenever there is an "OR". In this case we "clone" a node by copying all data, and after splitting we further expand the nodes, separately.

Algorithm 8.1 ψ to graph translation

```
nodeset translate(formula $f$)
{
    $n \leftarrow$ new node;
    $n.Old \leftarrow \emptyset$;
    $n.New \leftarrow \{f\}$;
    $n.Next \leftarrow \emptyset$;
    $n.Incoming \leftarrow \{init\}$;
    **return** expand($n, \emptyset$);
}
```

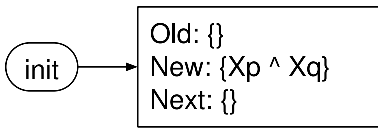
```
nodeset expand(node $n$, nodeset $N$)
{
    if $n.New == \emptyset$ then
        if $\exists m \in N, m.Old == n.Old \ \wedge \ m.Next == n.Next$ then
            // $m$ duplicates $n$; use $m$ instead and redirect edges
            $m.Incoming \leftarrow m.Incoming \cup n.Incoming$;
            // discard $n$ by not adding it to $N$ here
            return $N$;
        else
            // Add $n$ to the graph and keep processing
            $N \leftarrow N \cup \{n\}$;
            $n' \leftarrow$ new node;
            $n'.Old \leftarrow \emptyset$;
            $n'.New \leftarrow n.Next$;
            $n'.Next \leftarrow \emptyset$;
            $n'.Incoming \leftarrow \{n\}$;
            return expand($n'$, $N$);
        endif
    endif

    // $n.New$ is not empty; more processing to do

    choose some $f \in n.New$;
    $n.New \leftarrow n.New \setminus \{f\}$;
    if $f \in n.Old$ then
        // No need to process $f$ again
        return expand($n$, $N$);
    endif
    $n.Old \leftarrow n.Old \cup \{f\}$;

    // Remainder of algorithm: process $f$ based on the type of formula

    if $f \in \mathcal{P}$ or $\neg f \in \mathcal{P}$ or $f == \texttt{ff}$ or $f == \texttt{tt}$ then
        // $f$ is a trivial formula
        if $f == \texttt{ff}$ or $\neg f \in n.Old$ then
            // Logical impossibility, discard this node
            return $N$;
        else
            // No additional processing needed for this formula, keep expanding
            return expand($n$, $N$);
        endif

    elseif $f == g \wedge h$ then
        $n.New \leftarrow n.New \cup \{g, h\}$;
        return expand($n$, $N$);
    elseif $f == g \vee h$ then
        // Split
        $n' \leftarrow$ clone of $n$; // Copy all data from $n$ into new node $n'$
        $n.New \leftarrow n.New \cup \{g\}$;
        $n'.New \leftarrow n'.New \cup \{h\}$;
        return expand($n$, expand($n'$, $N$));

    elseif $f == \mathsf{X}\, g$ then
        $n.Next \leftarrow n.Next \cup \{g\}$;
        return expand($n$, $N$);

    elseif $f == g \,\mathsf{U}\, h$ then
        // Split, using recurrence    $g \,\mathsf{U}\, h \;\equiv\; h \vee (g \wedge \mathsf{X}(g \,\mathsf{U}\, h))$
        $n' \leftarrow$ clone of $n$;
        $n.New \leftarrow n.New \cup \{h\}$;
        $n'.New \leftarrow n'.New \cup \{g\}$;
        $n'.Next \leftarrow n'.Next \cup \{g \,\mathsf{U}\, h\}$;
        return expand($n$, expand($n'$, $N$));

    elseif $f == g \,\mathsf{R}\, h$ then
        // Split, using recurrence    $g \,\mathsf{R}\, h \;\equiv\; (g \wedge h) \vee (h \wedge \mathsf{X}(g \,\mathsf{R}\, h))$
        $n' \leftarrow$ clone of $n$;
        $n.New \leftarrow n.New \cup \{g, h\}$;
        $n'.New \leftarrow n'.New \cup \{h\}$;
        $n'.Next \leftarrow n'.Next \cup \{g \,\mathsf{R}\, h\}$;
        return expand($n$, expand($n'$, $N$));
    endif
}
```

Example 8.28

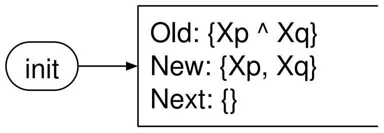
Run the translation algorithm on the formula $\psi = \mathsf{X}\, p \wedge \mathsf{X}\, q$.

Solution:

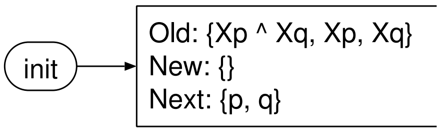
1. Start with a node "init", pointing to a new node n with $n.New = \{\mathsf{X}\, p \wedge \mathsf{X}\, q\}$, and then call expand() on that node:



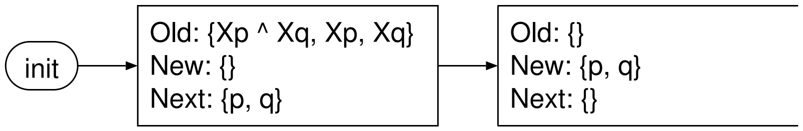
2. Choose formula $f = \mathsf{X}\, p \wedge \mathsf{X}\, q$ from $n.New$; remove it from $n.New$, add it to $n.Old$, and then process the "and": add $\mathsf{X}\, p$ and $\mathsf{X}\, q$ to $n.New$. That gives us:



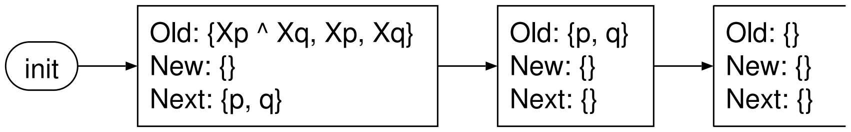
3. Choose (say) formula $f = \mathsf{X}\, p$ from $n.New$; remove it from $n.New$, add it to $n.Old$, and process the "X": add p to $n.Next$. Do the same for $\mathsf{X}\, q$. That gives us:



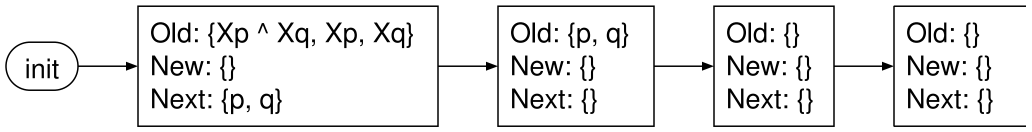
4. Since $n.New$ is now empty, we complete the node by adding it to the graph and adding a child node n' with $n'.New = n.Next$. That gives us:



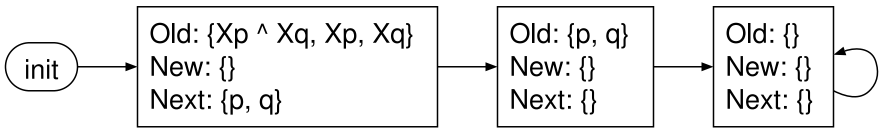
5. Choose formula $f = p$ and process it (just move it from New to Old). Do the same with $f = q$. Since New is now empty, complete the node by adding it to the graph and adding a child node n' with $n'.New = n.Next$. That gives us:



6. Since this node has an empty New, complete it by adding it to the graph and adding a child node. That gives us:



7. When this node is completed, we discover that it duplicates its parent node, so we discard it and redirect all incoming edges to the parent node. We terminate with the graph:



Now, how do we convert the graph into a Büchi automaton? Informally, the idea is:

- Label edges with all subsets of atomic propositions that are consistent with propositions and their negations listed in the destination node's *Old* set.
- Define an accepting condition similar to the tableau rule where until formulas must be eventually satisfied.

Formally, if we use Algorithm 8.1 to build a nodeset N for a formula ψ, we build the Büchi automaton $(\Sigma, Q, Q_0, \Delta, \mathcal{F})$ as follows.

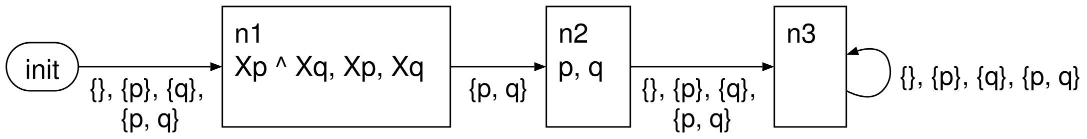
- $\Sigma = 2^{\mathcal{P}}$

- $Q = N \cup \{init\}$
- $Q_0 = \{init\}$
- $(q, A, q') \in \Delta$ if and only if $q \in q'.Incoming$ (i.e., there's an edge from q to q' in the graph) AND $A \subseteq \mathcal{P}$ is a set of propositions that satisfy the conjunction of all propositions and negated propositions in $q'.Old$.
- $\mathcal{F} = \{F_1, F_2, \ldots, F_u\}$ where each F_i is an accepting set for some until subformula and u is the total number of until subformulas. Specifically, for each subformula $g \,\mathsf{U}\, h$, add an accepting set
$$F_i = \{q \mid h \in q.Old \quad \vee \quad g \,\mathsf{U}\, h \notin q.Old\}$$
If there are no until subformulas, then use $\mathcal{F} = \{Q\}$.

Example 8.29

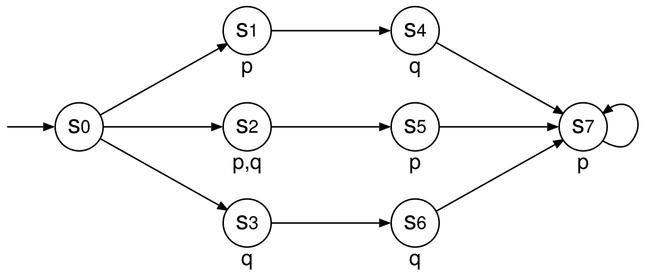
What is the Büchi automaton for $\psi = X\,p \wedge X\,q$?

Solution: Convert the graph obtained in Example 8.28 into a Büchi automaton. That gives us



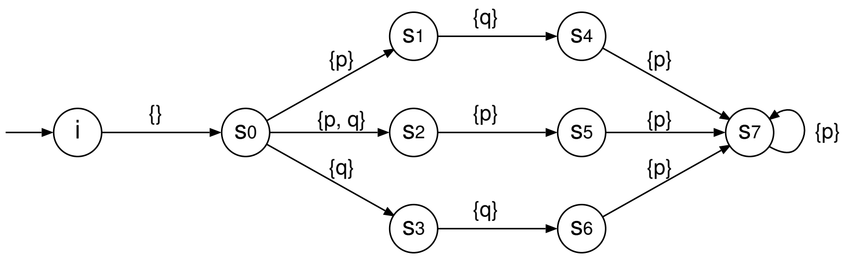
where each node is given a name and we show its *Old* set. Note that $\Sigma = \{\{\}, \{p\}, \{q\}, \{p, q\}\}$ and $\mathcal{F} = \{init, n1, n2, n3\}$ (making this a standard Büchi automaton) because there are no until formulas.

Example 8.30

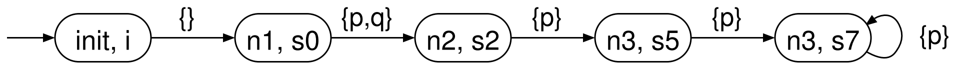


Does the above Kripke structure satisfy $\mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$?

Solution: First, convert the Kripke structure into a Büchi automaton:



Then, take the product of this and the Büchi automaton obtained in Example 8.29. If we start from the initial state and consider only states that can be reached, we obtain:



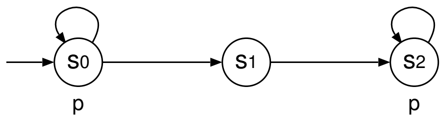
with $\mathcal{F} = \{F_\psi \times Q_K, Q_\psi \times F_K\}$ where Q_K, Q_ψ are the states in the automata for the Kripke structure and the formula ψ, and F_K, F_ψ are the accepting states in the automata for the Kripke structure and the formula ψ. In this case, $F_K = Q_K$ and $F_\psi = Q_\psi$ and so we have $\mathcal{F} = \{\text{set of all states}\}$. Therefore, this is also a standard Büchi automaton where F is the set of all states.

Finally, we check if the langage accepted by this automaton is empty. Running the algorithm:

1. There is one non-trivial SCC, namely the state (n_3, s_7).
2. This state belongs to F, so mark it as green.
3. All states can reach this one, so mark everything else as green.
4. The initial state is green, therefore $L(A) \neq \emptyset$.

Because the language is non-empty, there exists a path satisfying the formula. Therefore, the model satisfies $\mathsf{E}\,(\mathsf{X}\,p \wedge \mathsf{X}\,q)$.

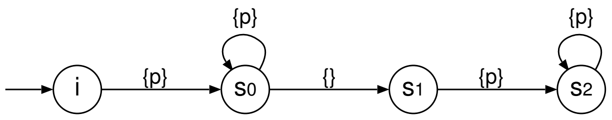
Example 8.31



Use Büchi automata to show this model satisfies $\mathsf{A\,FG}\,p$.

Solution: We know from Example 8.5 that the property holds. Following the steps:

1. The alphabet we need is $\Sigma = \{\{\}, \{p\}\}$.
2. Build a standard Büchi automaton A_M for the Kripke structure:



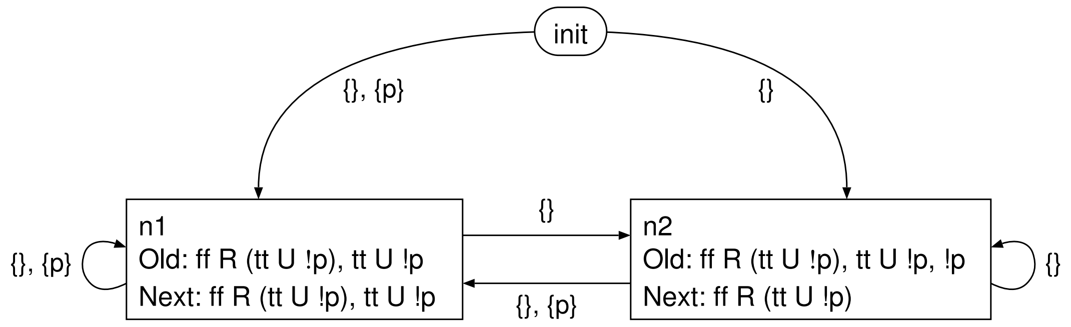
   Recall that F is the set of all states.
3. Build a standard Büchi automaton A_ψ. First, we need to convert our formula into an "existence" property:

$$\mathsf{A\,FG}\,p \;\equiv\; \neg\mathsf{E}\,\neg\mathsf{FG}\,p$$

   Now, write $\psi = \neg\mathsf{FG}\,p$ in negative normal form, without operators F and G. From Example 8.27, we have

$$\mathsf{A\,FG}\,p \;\equiv\; \neg\mathsf{E}\,\neg\mathsf{FG}\,p \;\equiv\; \neg\mathsf{E}\,\mathtt{ff}\,\mathsf{R}\,(\mathtt{tt}\,\mathsf{U}\,\neg p)$$

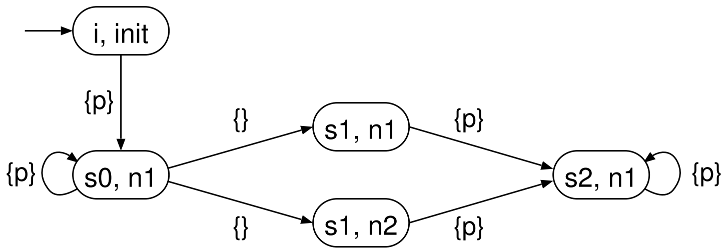
Next, run the translation algorithm for $\psi = \neg \mathsf{E}\, \mathtt{ff}\, \mathsf{R}\, (\mathtt{tt}\, \mathsf{U}\, \neg p)$. This gives us the automaton:



init
{}, {p}
{}
n1
Old: ff R (tt U !p), tt U !p
Next: ff R (tt U !p), tt U !p
{}
n2
Old: ff R (tt U !p), tt U !p, !p
Next: ff R (tt U !p)
{}, {p}
{}, {p}
{}

There is a single until formula, $\mathtt{tt}\, \mathsf{U}\, \neg p$, giving us an accepting set of states whose *Old* set either contains $\neg p$, or does not contain $\mathtt{tt}\, \mathsf{U}\, \neg p$. Therefore $\mathcal{F} = \{\{n_2\}\}$ or equivalently we have a standard Büchi automaton with $F = \{n_2\}$.

4. Build the product automaton $A_M \times A_\psi$, which gives us (if we eliminate unreachable states):



with acceptance condition $\mathcal{F} = \{\{\text{all states}\}, \{\text{states with } n_2\}\}$ which is equivalent to a standard Büchi automaton with $F = \{(s_1, n_2)\}$.

5. Is $L(A_M \times A_\psi)$ empty?

YES: there is no way to visit (s_1, n_2) infinitely often.
$\rightarrow$ the model *does not* satisfy $\mathsf{E}\,\psi \;=\; \mathsf{E}\,\neg \mathsf{FG}\, p$
$\rightarrow$ the model *does* satisfy $\neg\mathsf{E}\,\psi \;=\; \neg\mathsf{E}\,\neg\mathsf{FG}\, p \;=\; \mathsf{A}\,\mathsf{FG}\, p$

Complexity of LTL model checking with Büchi automata:

1. The size of the alphabet is $\mathcal{O}(2^{|\mathcal{P}|})$, which can be represented with $|\mathcal{P}|$ bits.
2. The size of A_M and the cost to build it is linear in the size of the Kripke structure: $\mathcal{O}(|\mathcal{S}|+|\mathcal{R}|)$.
3. It can be shown that the worst case size of A_ψ is $\mathcal{O}(2^{|\psi|})$.
4. The size of $A_M \times A_\psi$ (and the cost to build it) is at most the product of the sizes of A_M and A_ψ, which is at most $\mathcal{O}(2^{|\psi|}(|\mathcal{S}| + |\mathcal{R}|))$.
5. Checking for language emptiness can be done in time that is linear in the size of the automaton we are checking.

Therefore, using Büchi automata has the same worst-case complexity as using the tableau method.

8.6 Is a faster algorithm possible?

Both the tableau–based methods and Büchi–based methods for LTL model checking have worst-case complexity $\mathcal{O}(2^{|\psi|}(|\mathcal{S}|+|\mathcal{R}|))$. Can we do better? How "hard" is LTL model checking, anyway? We will look at a fragment of LTL in some detail, and summarize other results from the literature[5].

Definition 8.34 $\Phi(\mathcal{X})$ *denotes the set of all path formulas using operators from the set* $\mathcal{X}$.

For example, given any LTL path formula, we can find an equivalent one in the set $\Phi(\neg, \wedge, \mathsf{X}, \mathsf{U})$.

8.6.1 Model checking $\Phi(\neg, \wedge, \mathsf{F})$

Definition 8.35 *For any sequence of states* $\sigma = (s_0, s_1, s_2, \ldots)$, *define:*

Inf(σ)**:** *the set of states appearing infinitely often in* σ

tail(σ)**:** *the smallest* j *such that* $s_i \in \mathit{Inf}(\sigma)$, $\forall i \geq j$

size$(\sigma) =$ *tail*$(\sigma) + |\mathit{Inf}(\sigma)|$

Property 8.36 *For any paths* $\pi = (p_0, p_1, \ldots)$ *and* $\sigma = (s_0, s_1, \ldots)$ *with* $\mathit{tail}(\pi) = \mathit{tail}(\sigma)$, $p_0 = s_0$, $p_j = s_j\ \forall j < \mathit{tail}(\pi)$, *and* $\mathit{Inf}(\pi) = \mathit{Inf}(\sigma)$, *then for all* $\phi \in \Phi(\neg, \wedge, \mathsf{F})$,

1. $\forall i < \mathit{tail}(\pi)$,
$$\pi^i \models \phi \quad \textit{if and only if} \quad \sigma^i \models \phi.$$

2. $\forall i, i' \geq \mathit{tail}(\pi)$, *with* $p_i = s_{i'}$,
$$\pi^i \models \phi \quad \textit{if and only if} \quad \sigma^{i'} \models \phi.$$

Proof by induction on formula structure: *In the base case,* ϕ *is an atomic proposition, and the result holds trivially.*
Now, assume it holds for all subformulas of ϕ, *and prove it holds for* ϕ. *The cases* $\phi = \neg\phi_1$, $\phi = \phi_1 \wedge \phi_2$ *are trivial. Now, consider* $\phi = \mathsf{F}\,\phi_1$ *and suppose* $\pi^i \models \mathsf{F}\,\phi_1$. *Then by definition, there is a* $j \geq i$ *such that* $\pi^j \models \phi_1$.

1. *Suppose* $i < \mathit{tail}(\pi)$. *If* $j < \mathit{tail}(\pi)$, *then we have* $p_j = s_j$ *and by the inductive hypothesis,* $\sigma^j \models \phi_1$; *therefore,* $\sigma^i \models \mathsf{F}\,\phi_1$. *Otherwise,* $j \geq \mathit{tail}(\pi)$, *and we have* $p_j \in \mathit{Inf}(\pi) = \mathit{Inf}(\sigma)$. *Therefore, there exists a* $j' \geq \mathit{tail}(\pi)$ *such that* $s_{j'} = p_j$. *By the inductive hypothesis,* $\sigma^{j'} \models \phi_1$, *and since* $j' > i$, $\sigma^i \models \mathsf{F}\,\phi_1$.

2. *Suppose* $i \geq \mathit{tail}(\pi)$. *Then both* p_i *and* p_j *appear infinitely often in* π *and in* σ. *Therefore, for any* i' *such that* $p_i = s_{i'}$, *there exists* $j' \geq i'$ *such that* $p_j = s_{j'}$. *By the inductive hypothesis,* $\sigma^{j'} \models \phi_1$, *and therefore* $\sigma^{i'} \models \mathsf{F}\,\phi_1$.

[5]Some references:

- A. Sistla and E. Clarke, "The Complexity of Propositional Linear Temporal Logics". *Journal of the Association for Computing Machinery* 32 (3), pp. 733–749. 1985.
- M. Bauland *et al.*, "The Tractability of Model Checking for LTL: The Good, the Bad, and the Ugly Fragments". *ACM Transactions on Computational Logic* 12 (2). 2011.

Property 8.37 (Sistla and Clarke) *There is a path satisfying $\psi \in \Phi(\neg, \wedge, \mathsf{F})$ if and only if there is a path $\pi \models \psi$ with size$(\pi) \leq |\psi|$.*

Property 8.38 (Sistla and Clarke) *The problem of checking whether some path satisfies a formula ψ in $\Phi(\neg, \wedge, \mathsf{F})$ is in NP.*

Proof: *We give a nondeterministic algorithm that verifies if a path satisfies ψ, as follows.*

1. *Guess a finite sequence $(s_0, s_1, \ldots, s_n)$, these will form the first part of the path π.*
2. *Guess the states in Inf(π). From Property 8.36, the path will satisfy ψ or not, regardless of the order in which these states are visited. Also, note that Property 8.37 says that the total number of guesses in these first two steps is at most the size of formula ψ.*
3. *Verify that $(s_0, s_1, \ldots, s_n)$ is a valid path in the Kripke structure, and that there is an edge from s_n to some state in Inf(π).*
4. *Verify that the subgraph of the Kripke structure, containing only the states Inf(π), is strongly–connected.*
5. *Label the states in $(s_0, \ldots, s_n)$ and all states in Inf(π) using the following recursive algorithm.*
   (a) *For atomic proposition p, state s is labeled with p iff $p \in L(s)$.*
   (b) *For $\phi = \neg\phi_1$, label s with ϕ iff s is not labeled with ϕ_1.*
   (c) *For $\phi = \phi_1 \wedge \phi_2$, label s with ϕ iff s is labeled with ϕ_1 and ϕ_2.*
   (d) *For $\phi = \mathsf{F}\,\phi_1$, if some $s' \in$ Inf(π) is labeled with ϕ_1, then label all s with ϕ. If s_i is labeled with ϕ_1, then label all s_j with ϕ, for $j \leq i$.*
6. *The algorithm returns "yes" if s_0 is labeled with ψ.*

Note that the above algorithm requires polynomial time in the size of the Kripke structure, and of the formula ψ.

Property 8.39 (Sistla and Clarke) *The problem of checking whether some path satisfies a formula ψ in $\Phi(\neg, \wedge, \mathsf{F})$ is NP–Complete.*

Proof: *From Property 8.38, we know this problem is in NP. We complete the proof by showing how an existing NP–Complete problem (namely, 3SAT) can be transformed to checking a formula in $\Phi(\neg, \wedge, \mathsf{F})$.*

Suppose we are given a 3SAT instance with variables $x_1, x_2, \ldots, x_n$, and formula $f = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_i is the disjunction of exactly three literals:

$$C_i \quad = \quad l_{i,1} \vee l_{i,2} \vee l_{i,3}$$

where $l_{i,j}$ is either x_k or $\neg x_k$, for some $1 \leq k \leq n$.
We build a Kripke structure $M = (\mathcal{S}, \mathcal{S}_0, \mathcal{R}, L)$ as follows, with atomic propositions $c_1, \ldots, c_m$.

- $\mathcal{S} = \{y_0, x_0, x'_0, y_1, x_1, x'_1, \ldots, x_n, x'_n, y_n\}$
- $\mathcal{S}_0 = \{y_0\}$

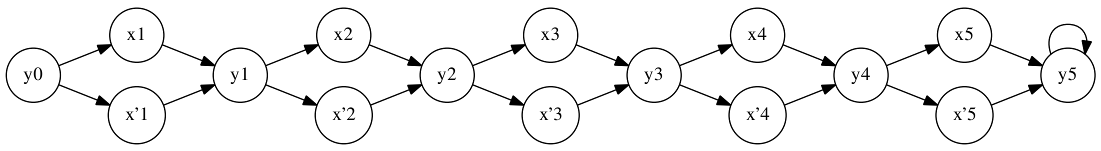


Figure 8.1: Kripke structure for a 3SAT instance with $n = 5$ variables

- $\mathcal{R} = \{(y_0, x_0), (y_0, x'_0), (x_0, y_1), (x'_0, y_1), \ldots, (y_{n-1}, x_n), (y_{n-1}, x'_n), (x_n, y_n), (x'_n, y_n), (y_n, y_n)\}$
- $L(y_i) = \emptyset$
- $c_j \in L(x_i)$ *if and only if* $l_{j,1} = x_i$ *or* $l_{j,2} = x_i$ *or* $l_{j,3} = x_i$ *(clause* C_j *contains* x_i*).*
- $c_j \in L(x'_i)$ *if and only if* $l_{j,1} = \neg x_i$ *or* $l_{j,2} = \neg x_i$ *or* $l_{j,3} = \neg x_i$ *(clause* C_j *contains* $\neg x_i$*).*

It can be proved that f *is satisfiable if and only if some path satisfies the formula*

$$\psi = \mathsf{F}\, c_1 \wedge \mathsf{F}\, c_2 \wedge \cdots \wedge \mathsf{F}\, c_m.$$

Note that this transformation is linear in the size of the 3SAT instance.

8.6.2 Summary of other results

Property 8.40 (Sistla and Clarke) *The problem of checking whether some path satisfies a formula* ψ *is PSPACE–complete for*

- $\psi \in \Phi(\neg, \wedge, \mathsf{F}, \mathsf{X})$
- $\psi \in \Phi(\neg, \wedge, \mathsf{U})$
- $\psi \in \Phi(\neg, \wedge, \mathsf{U}, \mathsf{X})$

Property 8.41 (Bauland *et al.*) *The problem of checking whether some path satisfies a formula* $\psi \in \Phi(\mathsf{U})$ *is NP–complete.*

8.7 Fairness

We do not need to do anything extra to deal with fairness in LTL, because fairness constraints can be written directly into the LTL path formula. For example, using the fairness constraints we discussed with CTL, we can specify a set of states $\mathcal{C}$ that we want to occur infinitely often by defining an atomic proposition c that is true on the states in $\mathcal{C}$. Then, the formula

$$\mathsf{GF}\, c$$

is satisfied only by paths where c holds infinitely often along the path. Thus, if we have an LTL path formula ψ, and want to know if it holds for some *fair* path, we can simply write

$$\mathsf{E}\,(\mathsf{GF}\, c \wedge \psi)$$

without needing any special quantifiers as we did for CTL. Or, if we want to quantify over all *fair* paths, we can write something like

$$\mathsf{A}\,(\mathsf{GF}\,c \to \psi)$$

which, again, will be true if there are no fair paths.

In fact, LTL allows us to specify more detailed fairness conditions. For example, suppose we have a Kripke structure that models three cooperating threads, then we can express the fairness constraint that all threads enter their critical sections infinitely often as

$$\mathsf{GF}\,c_1 \wedge \mathsf{GF}\,c_2 \wedge \mathsf{GF}\,c_3$$

where atomic proposition c_i holds whenever thread i is in its critical section. Or, we can write

$$\mathsf{GF}\,c_1 \to (\mathsf{GF}\,c_2 \wedge \mathsf{GF}\,c_3)$$

which says, if thread 1 enters its critical section infinitely often, then so do threads 2 and 3.

The above are examples of *strong fairness*. There is also a notion of *weak fairness*, which instead says

$$\mathsf{FG}\,p_1 \to \mathsf{GF}\,p_2$$

or, if p_1 eventually holds forever, then p_2 occurs infinitely often.

Property 8.42

$$\mathsf{FG}\,p \quad \to \quad \mathsf{GF}\,p$$

Proof:

$$\begin{aligned} \pi \models \mathsf{FG}\,p &\to \exists i \geq 0, \forall j \geq i, \pi^j \models p \\ &\to \forall j \geq 0, \exists k \geq j, \pi^k \models p \quad \textit{(if } j < i\textit{, use } k = i\textit{, else use } k = j\textit{)} \\ &\to \pi \models \mathsf{GF}\,p \end{aligned}$$

Property 8.43 *Strong fairness implies weak fairness.*

Proof: *Let $A = \mathsf{FG}\,p_1$, $B = \mathsf{GF}\,p_1$, and $C = \mathsf{GF}\,p_2$. Property 8.42 says $A \to B$, and is known to be true. Strong fairness says $B \to C$. Therefore, we have*

$$\mathsf{GF}\,p_1 \to \mathsf{GF}\,p_2 \quad \Leftrightarrow \quad B \to C \quad \Leftrightarrow \quad (A \to B)(B \to C) \quad \Rightarrow \quad (A \to C) \quad \Leftrightarrow \quad \mathsf{FG}\,p_1 \to \mathsf{GF}\,p_2$$

Chapter 9

CTL*

Previously, we saw the logic CTL, which dealt with state formulas and allowed multiple path quantifiers, and the logic LTL, which dealt with path formulas and allowed nesting of temporal operators. What about a logic that allows both, state formulas and path formulas? There is such a logic, called CTL*.

9.1 CTL* syntax

The CTL* syntax for a path formula ψ is similar to LTL:

$$\psi \ ::= \ \phi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\,\psi \mid \mathsf{F}\,\psi \mid \mathsf{G}\,\psi \mid \psi\,\mathsf{U}\,\psi$$

where ϕ is a state formula. The syntax for a state formula is:

$$\phi \ ::= \ \texttt{tt} \mid \texttt{ff} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathsf{A}\,\psi \mid \mathsf{E}\,\psi$$

where p is an atomic proposition, and ψ is a path formula. An "overall" CTL* formula is a state formula.

Note that the above syntax has some "conflicts". For example, in the formula $\mathsf{AF}\neg p$, according to the syntax rules, $\neg p$ can be either a path formula or a state formula. It turns out that the semantics are defined such that this does not matter, i.e., both interpretations will have the same meaning.

The syntax of CTL* is essentially to allow an LTL formula $\mathsf{E}\,\psi$ or $\mathsf{A}\,\psi$ to appear nested as a state formula, repeatedly. For example, the following is a valid CTL* formula.

$$\mathsf{A}\,\mathsf{GF}\,(\mathsf{E}\,a\,\mathsf{U}((\mathsf{AFG}\,b)\,\mathsf{U}\,(c\,\mathsf{U}\,d\,)\,)\,)$$

9.2 CTL* semantics

The semantics for a CTL* path formula are the same as the semantics for LTL path formulas, where any state formula ϕ is treated similar to an atomic proposition. As usual, we give the rules for when a path $\pi = (p_0, p_1, \ldots)$ satisfies a path formula.

1. $M, \pi \models \phi$, if and only if $p_0 \models \phi$.

2. $M, \pi \models \neg\psi$, if and only if $M, \pi \not\models \psi$.

3. $M, \pi \models \psi_1 \wedge \psi_2$, if and only if $M, \pi \models \psi_1$ and $M, \pi \models \psi_2$.

4. $M, \pi \models \psi_1 \vee \psi_2$, if and only if $M, \pi \models \psi_1$ or $M, \pi \models \psi_2$.

5. $M, \pi \models \mathsf{X}\psi$, if and only if $M, \pi^1 \models \psi$.

6. $M, \pi \models \mathsf{F}\psi$, if and only if there exists an $i \geq 0$ such that $M, \pi^i \models \psi$.

7. $M, \pi \models \mathsf{G}\psi$, if and only if $M, \pi^i \models \psi$ for all $i \geq 0$.

8. $M, \pi \models \psi_1 \mathsf{U}\, \psi_2$, if and only if there exists a $j \geq 0$ such that
   (a) $M, \pi^j \models \psi_2$, and
   (b) $M, \pi^i \models \psi_1$ for all $i < j$.

The semantics for state formulas are as follows. Again, we give the rules for when a state satisfies a state formula.

1. $M, s \models \texttt{tt}$, for all states s.

2. $M, s \not\models \texttt{ff}$, for all states s.

3. $M, s \models p$, if and only if $p \in L(s)$

4. $M, s \models \neg\phi$, if and only if $M, s \not\models \phi$

5. $M, s \models \phi_1 \wedge \phi_2$, if and only if $M, s \models \phi_1$ and $M, s \models \phi_2$

6. $M, s \models \phi_1 \vee \phi_2$, if and only if $M, s \models \phi_1$ or $M, s \models \phi_2$

7. $M, s \models \mathsf{A}\,\psi$, if and only if, for all paths $\pi = (p_0 = s, p_1, \ldots)$, $\pi \models \psi$.

8. $M, s \models \mathsf{E}\,\psi$, if and only if, for some path $\pi = (p_0 = s, p_1, \ldots)$, $\pi \models \psi$.

9.3 Expressive power of CTL, LTL, and CTL*

Note that any CTL formula ϕ is also a valid CTL* formula. It is easy to see that, for any valid CTL formula ϕ, the meaning of ϕ in CTL is equivalent to the meaning of ϕ in CTL*. Similarly, any LTL formula $\mathsf{E}\,\psi$ or $\mathsf{A}\,\psi$ is also a valid CTL* formula, and has the same meaning in both logics.

Given an arbitrary CTL* formula ϕ, there are four possibilities.

1. ϕ is a valid CTL formula, and is a valid LTL formula. $\phi = \mathsf{EX}\,p$ is an example of such a formula.

2. ϕ is a valid CTL formula, but not a valid LTL formula. $\phi = \mathsf{AF}\,\mathsf{AG}\,p$ is an example of such a formula.

3. ϕ is a valid LTL formula, but not a valid CTL formula. $\phi = \mathsf{A}\,\mathsf{FG}\,p$ is an example of such a formula.

4. ϕ is neither a valid CTL formula, nor is it a valid LTL formula. $\phi = \mathsf{A}\,\mathsf{FG}\,p \;\wedge\; \neg\mathsf{AFAG}\,p$ is an example of such a formula.

9.4 Comparing CTL and LTL formulas

Using CTL* semantics as a common framework, we can prove relationships between CTL and LTL formulas.

Property 9.1 *For any path formula ψ,*

$$\mathsf{EF}\,\mathsf{EG}\,\psi \quad \equiv \quad \mathsf{E}\,\mathsf{FG}\,\psi$$

Proof:

$$\begin{aligned}
s \models \mathsf{EF}\,\mathsf{EG}\,\psi &\Leftrightarrow \exists \pi = (s_0 = s, s_1, s_2, \ldots), \exists j \geq 0, s_j \models \mathsf{EG}\psi \\
&\Leftrightarrow \exists \pi = (s_0 = s, s_1, \ldots), \exists j \geq 0, \exists \pi' = (s'_0 = s_j, s'_1, \ldots), \pi' \models \mathsf{G}\,\psi \\
&\Leftrightarrow \exists j, \exists \rho = (s_0 = s, \ldots, s_j, s'_1, s'_2, \ldots), \rho^j \models \mathsf{G}\,\psi \\
&\Leftrightarrow \exists \rho = (s_0 = s, s_1, \ldots), \rho \models \mathsf{FG}\,\psi \\
&\Leftrightarrow s \models \mathsf{E}\,\mathsf{FG}\,\psi
\end{aligned}$$

Property 9.2 *For any path formula ψ,*

$$\mathsf{AF}\,\mathsf{AG}\,\psi \quad \rightarrow \quad \mathsf{A}\,\mathsf{FG}\,\psi$$

Proof:

$$\begin{aligned}
s \models \mathsf{AF}\,\mathsf{AG}\,\psi &\Leftrightarrow \forall \pi = (p_0 = s, p_1, p_2, \ldots), \exists j \geq 0, p_j \models \mathsf{AG}\psi \\
&\Leftrightarrow \forall \pi = (p_0 = s, p_1, \ldots), \exists j \geq 0, \forall \pi' = (p'_0 = p_j, p'_1, \ldots), \pi' \models \mathsf{G}\,\psi \\
&\rightarrow \forall \pi = (p_0 = s, p_1, \ldots), \exists j \geq 0, \pi^j \models \mathsf{G}\,\psi \quad \textit{(because } \pi^j \textit{ is one such path } \pi'\textit{)} \\
&\Leftrightarrow \forall \pi = (p_0 = s, p_1, \ldots), \pi \models \mathsf{FG}\,\psi \\
&\Leftrightarrow s \models \mathsf{A}\,\mathsf{FG}\,\psi
\end{aligned}$$

9.5 CTL* model checking

At first glance, it seems that the problem of model checking a CTL* formula must be impossibly complicated. However, it turns out to be just as hard as checking an LTL formula. We can determine the set of states satisfying a CTL* state formula ϕ using the following algorithm.

1. If ϕ is a state formula with no path quantifier A or E, then ϕ is a formula consisting only of atomic propositions and logical operators; determine which states satisfy ϕ in the obvious way, and terminate.

2. Find a subformula of ϕ with only one path quantifier A or E. Call this subformula ϕ'.

3. Subformula ϕ' will be an LTL formula. Use the tableau algorithm and determine which states satisfy the formula.

4. Create a new atomic proposition ν such that $s \models \nu$ if and only if $s \models \phi'$.

5. Remove ϕ' from ϕ, and replace it with atomic proposition ν.

6. Go to step (1).

Example 9.1

Using the above algorithm, we could check the formula

$$\phi = \mathsf{A}\,\mathsf{GF}\,(\mathsf{E}\,a\,\mathsf{U}((\mathsf{AFG}\,b)\,\mathsf{U}\,(c\,\mathsf{U}\,d\,)\,)\,)$$

as follows.

1. ϕ is not trivial, continue.
2. We get the subformula $\phi' = \mathsf{AFG}\,b$.
3. Find the states that satisfy ϕ' using LTL model checking.
4. Based on the results of (4), build atomic proposition ν.
5. Now, use formula
$$\phi = \mathsf{A}\,\mathsf{GF}\,(\mathsf{E}\,a\,\mathsf{U}(\nu\,\mathsf{U}\,(c\,\mathsf{U}\,d\,)\,)\,)$$
and repeat.

1. ϕ is not trivial, continue.
2. We get the subformula $\phi' = \mathsf{E}\,a\,\mathsf{U}(\nu\,\mathsf{U}\,(c\,\mathsf{U}\,d\,)\,)$.
3. Find the states that satisfy ϕ' using LTL model checking.
4. Based on the results of (4), build atomic proposition ν'.
5. Now, use formula
$$\phi = \mathsf{A}\,\mathsf{GF}\,\nu'$$
and repeat.

1. ϕ is not trivial, continue.
2. We get the subformula $\phi' = \phi = \mathsf{A}\,\mathsf{GF}\,\nu'$.
3. Find the states that satisfy ϕ' using LTL model checking.
4. Based on the results of (4), build atomic proposition ν''.
5. Now, use formula
$$\phi = \nu''$$
and repeat.

1. $\phi = \nu''$ is trivial; terminate.

Chapter 10

Review of Probability

10.1 Definition

Perform some random experiment. The sample space ω is the set of elements, where each element corresponds to one possible outcome of the experiment.

Example 10.1

Experiment: flip a coin.

$$\omega = \{H, T\} \quad \begin{array}{l} \text{where} \\ H = \text{the coin was heads} \\ T = \text{the coin was tails} \end{array}$$

Example 10.2

Experiment: roll a 6–sided die.

$$\omega = \{1, 2, 3, 4, 5, 6\} \quad \begin{array}{l} \text{where} \\ 1 = \text{the side with one dot was facing up} \\ 2 = \text{the side with two dots was facing up} \\ \vdots \\ 6 = \text{the side with six dots was facing up} \end{array}$$

10.2 Probability axioms and properties

Let $\mathcal{A}$ be some set of possible outcomes (or an "event"), with $\mathcal{A} \subseteq \omega$. A *probability* p is an assignment of a real number to the set $\mathcal{A}$, denoted $\Pr\{\mathcal{A}\}$, in such a way that the following axioms are satisfied:

Axiom 10.1

$$\Pr\{\emptyset\} = 0 \quad \textit{and} \quad \Pr\{\omega\} = 1$$

Axiom 10.2

$$0 \leq \Pr\{\mathcal{A}\} \leq 1$$

Axiom 10.3 *If $\mathcal{A}$ and $\mathcal{B}$ are disjoint (i.e., $\mathcal{A} \cap \mathcal{B} = \emptyset$), then*

$$\Pr\{\mathcal{A} \cup \mathcal{B}\} = \Pr\{\mathcal{A}\} + \Pr\{\mathcal{B}\}$$

From the above axioms, we can derive some useful properties.

Property 10.4 *If $\mathcal{A} \subseteq \mathcal{B}$, then*

$$\Pr\{\mathcal{B} \setminus \mathcal{A}\} = \Pr\{B\} - \Pr\{A\}$$

Proof: if $\mathcal{A} \subseteq \mathcal{B}$, then $\mathcal{B} = (\mathcal{B} \setminus \mathcal{A}) \cup \mathcal{A}$, and the two sets are disjoint. Using Axiom 10.3 we get

$$\begin{aligned} \Pr\{\mathcal{B}\} &= \Pr\{(\mathcal{B} \setminus \mathcal{A}) \cup \mathcal{A}\} \\ &= \Pr\{\mathcal{B} \setminus \mathcal{A}\} + \Pr\{\mathcal{A}\} \end{aligned}$$

Property 10.5 *If $\mathcal{A} \subseteq \mathcal{B}$, then*

$$\Pr\{\mathcal{A}\} \le \Pr\{\mathcal{B}\}$$

Proof: from Property 10.4, since $\Pr\{\mathcal{B} \setminus \mathcal{A}\} \ge 0$.

Property 10.6

$$\Pr\{\mathcal{A}^c\} = \Pr\{\omega \setminus \mathcal{A}\} = 1 - \Pr\{\mathcal{A}\}$$

Proof: immediate from Property 10.4.

Property 10.7 *If $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are disjoint, then*

$$\Pr\{\mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n\} = \Pr\{\mathcal{A}_1\} + \cdots + \Pr\{\mathcal{A}_n\}$$

Proof: apply Axiom 10.3 repeatedly (thanks to associativity of addition)

Property 10.8 *For any sets $\mathcal{A}$ and $\mathcal{B}$,*

$$\Pr\{\mathcal{A} \cup \mathcal{B}\} = \Pr\{\mathcal{A}\} + \Pr\{\mathcal{B}\} - \Pr\{\mathcal{A} \cap \mathcal{B}\}$$

Proof: $\mathcal{A} \setminus \mathcal{B}$ and $\mathcal{A} \cap \mathcal{B}$ are disjoint, and their union is $\mathcal{A}$; this gives

$$\begin{aligned} \Pr\{\mathcal{A}\} &= \Pr\{\mathcal{A} \setminus \mathcal{B} \cup \mathcal{A} \cap \mathcal{B}\} = \Pr\{\mathcal{A} \setminus \mathcal{B}\} + \Pr\{\mathcal{A} \cap \mathcal{B}\} \\ \Pr\{\mathcal{A}\} - \Pr\{\mathcal{A} \cap \mathcal{B}\} &= \Pr\{\mathcal{A} \setminus \mathcal{B}\} \end{aligned}$$

Further, since $\mathcal{A} \cup \mathcal{B} = \mathcal{A} \setminus \mathcal{B} \cup \mathcal{B}$ (another disjoint union), we get

$$\Pr\{\mathcal{A} \cup \mathcal{B}\} = \Pr\{\mathcal{A} \setminus \mathcal{B}\} + \Pr\{\mathcal{B}\}$$

Plugging in for $\Pr\{\mathcal{A} \setminus \mathcal{B}\}$ *completes the proof.*

10.3 Conditional probability

Conditional probability is defined as follows.

Definition 10.9 *For any $\mathcal{A}$, $\mathcal{B}$ with* $\Pr\{\mathcal{B}\} > 0$,

$$\Pr\{\mathcal{A}|\mathcal{B}\} = \frac{\Pr\{\mathcal{A} \cap \mathcal{B}\}}{\Pr\{\mathcal{B}\}}$$

The meaning is: the probability that $\mathcal{A}$ happened, given that $\mathcal{B}$ happened.

Since $\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A}$, we get

$$\begin{aligned}
\Pr\{\mathcal{B} \cap \mathcal{A}\} &= \Pr\{\mathcal{A} \cap \mathcal{B}\} \\
\Pr\{\mathcal{B}|\mathcal{A}\}\Pr\{\mathcal{A}\} &= \Pr\{\mathcal{A}|\mathcal{B}\}\Pr\{\mathcal{B}\} \\
\Pr\{\mathcal{B}|\mathcal{A}\} &= \frac{\Pr\{\mathcal{A}|\mathcal{B}\}\Pr\{\mathcal{B}\}}{\Pr\{\mathcal{A}\}}
\end{aligned}$$

Example 10.3

For the experiment: “roll a 6–sided die”, with $\omega = \{1, 2, 3, 4, 5, 6\}$.
Let $\mathcal{A}$ = “outcome is prime” = $\{2, 3, 5\}$.
Let $\mathcal{B}$ = “outcome less than 4” = $\{1, 2, 3\}$.

$$\begin{aligned}
\Pr\{\mathcal{A}\} &= \Pr\{\{2,3,5\}\} = 3/6 \\
\Pr\{\mathcal{B}\} &= \Pr\{\{1,2,3\}\} = 3/6 \\
\Pr\{\mathcal{A} \cap \mathcal{B}\} &= \Pr\{\{2,3\}\} = 2/6
\end{aligned}$$

$$\begin{aligned}
\Pr\{\mathcal{A}|\mathcal{B}\} &= \frac{\Pr\{\mathcal{A} \cap \mathcal{B}\}}{\Pr\{\mathcal{B}\}} = \frac{2/6}{3/6} = 2/3 \\
\Pr\{\mathcal{B}|\mathcal{A}\} &= \frac{\Pr\{\mathcal{A}|\mathcal{B}\}\Pr\{\mathcal{B}\}}{\Pr\{\mathcal{A}\}} = \frac{(2/3)(3/6)}{3/6} = 2/3 \\
&= \frac{\Pr\{\mathcal{B} \cap \mathcal{A}\}}{\Pr\{\mathcal{A}\}} = \frac{2/6}{3/6} = 2/3
\end{aligned}$$

10.4 Law of total probability

Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be a partition of ω.

$$\begin{aligned}
\Pr\{\mathcal{A}\} &= \Pr\{\mathcal{A} \cap \omega\} \\
&= \Pr\{\mathcal{A} \cap (\mathcal{B}_1 \cup \cdots \cup \mathcal{B}_n)\} \\
&= \Pr\{(\mathcal{A} \cap \mathcal{B}_1) \cup \cdots \cup (\mathcal{A} \cap \mathcal{B}_n)\} \\
&= \Pr\{\mathcal{A} \cap \mathcal{B}_1\} + \cdots + \Pr\{\mathcal{A} \cap \mathcal{B}_n\} \qquad \text{(from Axiom 10.3)} \\
&= \Pr\{\mathcal{A}|\mathcal{B}_1\}\Pr\{\mathcal{B}_1\} + \cdots + \Pr\{\mathcal{A}|\mathcal{B}_n\}\Pr\{\mathcal{B}_n\}
\end{aligned}$$

Example 10.4

From Example 10.3,

$$\begin{aligned}\text{Let } \mathcal{B}_1 &= \{1,2,3\}\\ \text{Let } \mathcal{B}_2 &= \{4,5\}\\ \text{Let } \mathcal{B}_3 &= \{6\}\end{aligned}$$

$$\begin{aligned}\Pr\{\text{“prime”}|\mathcal{B}_1\}\Pr\{\mathcal{B}_1\} &= (2/3)\cdot(3/6) &= 2/6\\ \Pr\{\text{“prime”}|\mathcal{B}_2\}\Pr\{\mathcal{B}_2\} &= (1/2)\cdot(2/6) &= 1/6\\ \Pr\{\text{“prime”}|\mathcal{B}_3\}\Pr\{\mathcal{B}_3\} &= 0\cdot(1/6) &= 0\end{aligned}$$

We can add these together to get $\Pr\{\text{“prime”}\}$:

$$\Pr\{\text{“prime”}\} = 2/6 + 1/6 + 0 = 3/6$$

We can also use the law of total probability with conditioning. Again, if $\mathcal{B}_1, \ldots, \mathcal{B}_n$ is a partition of ω, then

$$\Pr\{\mathcal{A}|\mathcal{C}\} = \Pr\{\mathcal{A}|\mathcal{B}_1,\mathcal{C}\}\Pr\{\mathcal{B}_1|\mathcal{C}\} + \cdots + \Pr\{\mathcal{A}|\mathcal{B}_n,\mathcal{C}\}\Pr\{\mathcal{B}_n|\mathcal{C}\}$$

10.5 Independence

Two events are independent if knowledge of one tells you nothing about the other.

Definition 10.10 *Events $\mathcal{A}$ and $\mathcal{B}$ are independent if*

$$\Pr\{\mathcal{A}|\mathcal{B}\} = \Pr\{\mathcal{A}\}$$

Otherwise they are dependent.

Note that if $\mathcal{A}$ and $\mathcal{B}$ are independent,

$$\Pr\{\mathcal{B}|\mathcal{A}\} = \frac{\Pr\{\mathcal{A}|\mathcal{B}\}\Pr\{\mathcal{B}\}}{\Pr\{\mathcal{A}\}} = \frac{\Pr\{\mathcal{A}\}\Pr\{\mathcal{B}\}}{\Pr\{\mathcal{A}\}} = \Pr\{\mathcal{B}\}$$

Property 10.11 *Events $\mathcal{A}$ and $\mathcal{B}$ are independent iff*

$$\Pr\{\mathcal{A}\cap\mathcal{B}\} = \Pr\{\mathcal{A}\}\Pr\{\mathcal{B}\}$$

Proof: $\rightarrow$

$$\begin{aligned}\Pr\{\mathcal{A}|\mathcal{B}\} &= \frac{\Pr\{\mathcal{A}\cap\mathcal{B}\}}{\Pr\{\mathcal{B}\}}\\ \Pr\{\mathcal{A}\} &= \frac{\Pr\{\mathcal{A}\cap\mathcal{B}\}}{\Pr\{\mathcal{B}\}}\\ \Pr\{\mathcal{A}\}\Pr\{\mathcal{B}\} &= \Pr\{\mathcal{A}\cap\mathcal{B}\}\end{aligned}$$

Proof: $\leftarrow$

$$\begin{aligned}
\Pr\{\mathcal{A}|\mathcal{B}\} &= \frac{\Pr\{\mathcal{A}\cap\mathcal{B}\}}{\Pr\{\mathcal{B}\}} \\
&= \frac{\Pr\{\mathcal{A}\}\cdot\Pr\{\mathcal{B}\}}{\Pr\{\mathcal{B}\}} \\
&= \Pr\{\mathcal{A}\}
\end{aligned}$$

Example 10.5

From Example 10.3,

$$\Pr\{\text{“prime”}|\text{“less than 4”}\} = 2/3 \qquad \neq \qquad \Pr\{\text{“prime”}\} = 3/6$$

Therefore, the events “prime” and “less than 4” are dependent.

Example 10.6

For the experiment: “flip a coin and roll a 6–sided die”.
Note $\omega = \{H1, H2, H3, H4, H5, H6, T1, T2, T3, T4, T5, T6\}$.
Let $\mathcal{A}$ = “coin was heads” = $\{H1, H2, H3, H4, H5, H6\}$.
Let $\mathcal{B}$ = “die was 2 or 5” = $\{H2, H5, T2, T5\}$.

$$\begin{aligned}
\Pr\{\mathcal{A}\} &= 6/12 \;=\; 1/2 \\
\Pr\{\mathcal{B}\} &= 4/12 \;=\; 1/3 \\
\Pr\{\mathcal{A}\cap\mathcal{B}\} &= \Pr\{\{H2, H5\}\} \;=\; 2/12 \;=\; 1/6 \\
\Pr\{\mathcal{A}|\mathcal{B}\} &= \frac{\Pr\{\mathcal{A}\cap\mathcal{B}\}}{\Pr\{\mathcal{B}\}} \;=\; \frac{1/6}{1/3} \;=\; 1/2
\end{aligned}$$

Since $\Pr\{\mathcal{A}|\mathcal{B}\} = \Pr\{\mathcal{A}\}$, events $\mathcal{A}$ and $\mathcal{B}$ are independent.

Example 10.7

Is an arbitrary event $\mathcal{A}$ independent of ω?

Chapter 11

Review of Random Variables

11.1 Definition

Conduct a random experiment with sample space ω. A *random variable* X is a function $X : \omega \to \mathcal{S}$.

Example 11.1

Experiment: flip a coin.

$$\begin{aligned}
\omega &= \{H, T\} \\
X(H) &= 1 \\
X(T) &= 0 \\
\\
\mathcal{S} &= \{0, 1\}
\end{aligned}$$

Note, $\mathcal{S}$ is finite.

Example 11.2

Experiment: roll a 6–sided die.

$$\begin{aligned}
\omega &= \{1, 2, 3, 4, 5, 6\} \\
X(1) &= 1 \\
&\vdots \\
X(6) &= 6 \\
\\
\mathcal{S} &= \{1, 2, 3, 4, 5, 6\}
\end{aligned}$$

Note, $\mathcal{S}$ is finite.

Example 11.3

Experiment: roll two 6–sided dice.

$$\begin{aligned} \omega &= \{(1,1),(1,2),\ldots,(6,6)\} \\ X &= \text{sum of up faces} \\ \mathcal{S} &= \{2,3,\ldots,12\} \end{aligned}$$

Note, $\mathcal{S}$ is finite.

Example 11.4

Experiment: flip a coin infinitely many times.

$$\begin{aligned} \omega &= \{H,T\}^{\infty} \\ X &= \text{number of tails before the first heads} \\ \mathcal{S} &= \{0,1,2,\ldots\} \end{aligned}$$

Note, $\mathcal{S}$ is discrete (countably infinite).

Example 11.5

Experiment: toss a pencil so that it lands flat on the floor.

$$\begin{aligned} \omega &= \mathbb{R}^4 \\ & \quad (x_1,y_1),(x_2,y_2) : \text{Coordinates of pencil endpoints} \\ X &= \text{angle we must rotate the pencil, clockwise, until it points north} \\ \mathcal{S} &= [0,2\pi) \end{aligned}$$

Note, $\mathcal{S}$ is continuous.

Notation and conventions:

- Random variables are usually denoted as capitals, e.g., X, Y, Z.
- We will talk about probabilities for random variables, with the understanding that this translates to the underlying experiment.
- An "instance" or "sample" of a random variable can be "generated" by conducting the experiment and applying the function to the outcome. I.e., if the outcome of the experiment is $a \in \omega$, then the "sample" is $X(a)$.
- We will eventually stop mentioning the random experiments. We will consider random variables on their own.

11.2 Discrete Random Variables

A discrete random variable is one whose set of possible values $\mathcal{S}$ is discrete. Discrete random variables can be completely specified by specifying

- The set of possible values, $\mathcal{S}$.
- The probability of each value in $\mathcal{S}$, called the probability distribution function (PDF).

Example 11.6

Experiment: flip a coin.

$$\begin{aligned} \omega &= \{H, T\} \\ X(H) &= 1 \\ X(T) &= 0 \end{aligned}$$

$$\begin{aligned} \Pr\{X = 0\} &= \Pr\{\{T\}\} \\ \Pr\{X = 1\} &= \Pr\{\{H\}\} \end{aligned}$$

If the coin is "fair", then we will have $\Pr\{X = 0\} = 1/2$

Example 11.7

Experiment: toss two dice

$$\begin{aligned} \omega &= \{1,2,3,4,5,6\}^2 \\ Y &= \text{sum of the up faces} \end{aligned}$$

$$\begin{aligned} \Pr\{Y = 2\} &= \Pr\{\{(1,1)\}\} \\ \Pr\{Y = 3\} &= \Pr\{\{(1,2),(2,1)\}\} \\ \Pr\{Y = 4\} &= \Pr\{\{(1,3),(2,2),(3,1)\}\} \\ &\vdots \\ \Pr\{Y = 12\} &= \Pr\{\{(6,6)\}\} \end{aligned}$$

Note that the set $\mathcal{S}$ forms a partition of ω based on the function X. I.e., let $\omega_n \subseteq \omega$ be the set of values a such that $X(a) = n$.

- $\omega_i \cap \omega_j = \emptyset$, for $i \neq j$
- $\omega = \bigcup_{i \in \mathcal{S}} \omega_i$

Property 11.1 *For any discrete random variable X,*

$$\sum_{i \in \mathcal{S}} \Pr\{X = i\} = 1$$

Proof: using Axiom 10.3,

$$\sum_{i \in \mathcal{S}} \Pr\{X = i\} = \sum_{i \in \mathcal{S}} \Pr\{\omega_i\} = \Pr\left\{\bigcup_{i \in \mathcal{S}} \omega_i\right\} = \Pr\{\omega\} = 1$$

We can also write the law of total probability for discrete random variables:

$$\begin{aligned}\Pr\{\mathcal{A}\} &= \sum_{i\in\mathcal{S}} \Pr\{\mathcal{A}|\omega_i\}\Pr\{\omega_i\} \\ &= \sum_{i\in\mathcal{S}} \Pr\{\mathcal{A}|X=i\}\Pr\{X=i\}\end{aligned}$$

11.2.1 Important discrete distributions

When talking about random variables, the underlying experiment is less important than the set of possible values $\mathcal{S}$ and the PDF. This information uniquely defines the "distribution" of the random variable. Several distributions have names. Some distributions have parameters.

Constant

A discrete random variable X is said to be $\mathit{Const}(n)$, written $X \sim \mathit{Const}(n)$, iff

$$\begin{aligned}\mathcal{S} &= \{n\} \\ \Pr\{X=n\} &= 1\end{aligned}$$

Bernoulli

$X \sim \mathit{Bernoulli}(p)$, with $0 \le p \le 1$, iff

$$\begin{aligned}\mathcal{S} &= \{0,1\} \\ \Pr\{X=1\} &= p \\ \Pr\{X=0\} &= 1-p\end{aligned}$$

Note that $\mathit{Bernoulli}(0) = \mathit{Const}(0)$ and $\mathit{Bernoulli}(1) = \mathit{Const}(1)$.

Equilikely

$X \sim \mathit{Equilikely}(a,b)$ iff

$$\begin{aligned}\mathcal{S} &= \{a,\ldots,b\} \\ \Pr\{X=n\} &= \frac{1}{b-a+1} \qquad \text{if } n \in \mathcal{S}\end{aligned}$$

If X is the upward face when a fair, 6–sided die is rolled, then $X \sim \mathit{Equilikely}(1,6)$. Special cases:

- $\mathit{Equilikely}(b,b) = \mathit{Const}(b)$
- $\mathit{Equilikely}(0,1) = \mathit{Bernoulli}(1/2)$

Geometric

$X \sim \mathit{Geom}(p)$, with $0 \leq p \leq 1$, iff

$$\begin{aligned} \mathcal{S} &= \{1, 2, \ldots\} \\ \Pr\{X = n\} &= (1-p)^{n-1}p \qquad \text{if } n \in \mathcal{S} \end{aligned}$$

Interpretation: $\mathit{Geom}(p)$ is the number of draws from $\mathit{Bernoulli}(p)$ before the first 1 is obtained. I.e., if $X_1 \sim \mathit{Bernoulli}(p), X_2 \sim \mathit{Bernoulli}(p), \ldots$ and $Y = \min\{i : X_i = 1\}$ then $Y \sim \mathit{Geom}(p)$.
Note! For students who took 455/555, this is *slightly different* than $\mathit{Geometric}(p)$.
Special case: $\mathit{Geom}(1) = \mathit{Const}(1)$.

Binomial

$Y \sim \mathit{Binomial}(n, p)$, with $n \in \{0, 1, 2, \ldots\}$ and $0 \leq p \leq 1$, iff $Y = \sum_{i=1}^{n} X_i$, where $X_i \sim \mathit{Bernoulli}(p)$. I.e., Binomial is the sum of independent Bernoulli random variables.

$$\begin{aligned} \mathcal{S} &= \{0, \ldots, n\} \\ \Pr\{Y = i\} &= \binom{n}{i} p^i (1-p)^{n-i} \qquad \text{if } i \in \mathcal{S} \end{aligned}$$

Note $\mathit{Binomial}(1, p) = \mathit{Bernoulli}(p)$, $\mathit{Binomial}(0, p) = \mathit{Const}(0)$, $\mathit{Binomial}(n, 1) = \mathit{Const}(n)$, $\mathit{Binomial}(n, 0) = \mathit{Const}(0)$.

Poisson

This can be viewed as the limiting case of Binomial:

$$\mathit{Poisson}(\lambda) = \lim_{n \to \infty} \mathit{Binomial}(n, \lambda/n)$$

with real–valued parameter $\lambda \geq 0$.
This gives the following PDF for $Z \sim \mathit{Poisson}(\lambda)$:

$$\begin{aligned} \mathcal{S} &= \{0, 1, 2, \ldots\} \\ \Pr\{Z = i\} &= \frac{\lambda^i}{i!} e^{-\lambda} \qquad \text{if } i \in \mathcal{S} \end{aligned}$$

11.2.2 Examples

Example 11.8

Suppose $X \sim \mathit{Equilikely}(a, b)$. What is $\Pr\{X < i\}$, for any integer i?

Solution: If $i < a$, then X cannot be less than i. Similarly, if $i > b$, then X must be less than i. So the only interesting case is $a \leq i \leq b$:

$$\begin{aligned} \Pr\{X < i\} &= \Pr\{X = a\} + \cdots + \Pr\{X = i - 1\} \\ &= \frac{1}{b-a+1} + \cdots + \frac{1}{b-a+1} \\ &= \frac{i-a}{b-a+1} \end{aligned}$$

Putting it all together, we get:

$$\Pr\{X < i\} = \begin{cases} 0 & \text{if } i < a \\ \frac{i-a}{b-a+1} & \text{if } a \le i \le b \\ 1 & \text{if } i > b \end{cases}$$

Example 11.9

Suppose $X \sim \textit{Geom}(p)$ and $Y \sim \textit{Geom}(p)$. What is $\Pr\{X = Y\}$?

Solution: Use the law of total probability:

$$\begin{aligned}
\Pr\{X = Y\} &= \sum_{i=1}^{\infty} \Pr\{X = Y | Y = i\} \Pr\{Y = i\} \\
&= \sum_{i=1}^{\infty} \Pr\{X = i\} \Pr\{Y = i\} \\
&= \sum_{i=1}^{\infty} (1-p)^{i-1} p \quad (1-p)^{i-1} p \\
&= p^2 \sum_{i=0}^{\infty} \left((1-p)^2\right)^i \\
& \quad \text{Trick: } (1-a)(1+a+a^2+a^3+\cdots) = 1, \text{ if } 0 \le a < 1 \\
&= p^2 \frac{1}{1-(1-p)^2} \\
\Pr\{X = Y\} &= \frac{p}{2-p}
\end{aligned}$$

Example 11.10

Suppose $X \sim \textit{Equilikely}(a, b)$ and $Y \sim \textit{Equilikely}(a, b)$. What is $\Pr\{X < Y\}$?

The solution is similar:

$$\begin{aligned}
\Pr\{X < Y\} &= \sum_{y=a}^{b} \Pr\{X < Y | Y = y\} \Pr\{Y = y\} \\
&= \sum_{y=a}^{b} \Pr\{X < y\} \Pr\{Y = i\} \\
&= \sum_{y=a}^{b} \frac{y-a}{b-a+1} \cdot \frac{1}{b-a+1} \\
&= \frac{1}{(b-a+1)^2} \sum_{y=a}^{b} y - a \\
&= \frac{1}{(b-a+1)^2} \sum_{i=0}^{b-a} i
\end{aligned}$$

$$\begin{aligned}
& \text{Trick: } 1+2+\cdots+n = n(n+1)/2 \\
&= \frac{1}{(b-a+1)^2} \cdot \frac{(b-a)(b-a+1)}{2} \\
\Pr\{X<Y\} &= \frac{b-a}{2(b-a+1)}
\end{aligned}$$

11.2.3 Expected value

Definition 11.2 *The expected value of a discrete random variable X is*

$$E[X] = \sum_{i \in \mathcal{S}} i \cdot \Pr\{X = i\}$$

The intuitive meaning of $E[X]$: if you were to "generate" or "sample" several values for X and take the average of those values, you expect that average to be close to $E[X]$. In fact, as the number of samples goes to infinity, the average will "converge" (in a probabilistic sense) to $E[X]$.

Example 11.11

What is $E[X]$ if $X \sim \mathit{Equilikely}(a,b)$?

$$\begin{aligned}
E[X] &= \sum_{i=a}^{b} i \cdot \Pr\{X = i\} \\
&= \sum_{i=a}^{b} \frac{i}{b-a+1} \\
&= \frac{1}{b-a+1} \sum_{i=a}^{b} i \\
&= \frac{1}{b-a+1} \left(\sum_{i=1}^{b} i - \sum_{i=1}^{a-1} i \right) \\
&= \frac{1}{b-a+1} \left(\frac{b(b+1)}{2} - \frac{(a-1)a}{2} \right) \\
&= \frac{b^2+b-ab+ab-a^2+a}{2(b-a+1)} \\
&= \frac{(b-a+1)(b+a)}{2(b-a+1)} \\
E[X] &= \frac{b+a}{2}
\end{aligned}$$

Property 11.3 *For constants $\alpha_1, \ldots, \alpha_n$ and discrete random variables $X_1, \ldots, X_n$ (from* any *distributions)*

$$E[\alpha_1 X_1 + \cdots + \alpha_n X_n] = \alpha_1 E[X_1] + \cdots + \alpha_n E[X_n]$$

Example 11.12

Let $\mu = E[X]$. The *variance* of X is $E[(X-\mu)^2]$. From Property 11.3, we have

$$\begin{aligned} E[(X-\mu)^2] &= E[X^2 - 2X\mu + \mu^2] \\ &= E[X^2] - 2\mu E[X] + \mu^2 \;=\; E[X^2] - 2\mu\mu + \mu^2 \\ &= E[X^2] - \mu^2 \end{aligned}$$

11.2.4 PDFs and CDFs

The *cumulative distribution function*, or CDF, of a random variable is given by

$$F(x) = \Pr\{X \le x\}$$

For a discrete random variable with possible values $\{a, a+1, \ldots\}$,

$$\begin{aligned} F(a) &= \Pr\{X = a\} \\ F(x) &= F(x-1) + \Pr\{X = x\}, \qquad x > a \end{aligned}$$

$$\Pr\{X = x\} = F(x) - F(x-1), \qquad x > a$$

11.3 Continuous Random Variables

A random variable is continuous if $\mathcal{S}$ is continuous. We will consider cases where $\mathcal{S}$ is a subset of reals, specifically, $\mathcal{S}$ is an interval.

By definition, the probability of a continuous random variable being equal to a particular value is zero[1].

Continuous random variables are specified by their possible values $\mathcal{S}$ and their CDF:

$$F(x) = \Pr\{X \le x\} = \Pr\{X < x\}$$

Analogous to the discrete case, we can also specify the probability *density* function, or PDF, for a continuous random variable:

$$\begin{aligned} \text{PDF:} \qquad f(x) &= \frac{d}{dx}F(x) \\ \text{CDF:} \qquad F(x) &= \int_{y<x} f(y)\, dy \end{aligned}$$

What is the intuitive meaning of the PDF, $f(x)$?

[1]Cases where this is not true lead to a distribution that is sometimes called "mixed". For instance, suppose X is the amount of time before a machine fails. Suppose there is a nonzero probability p that the machine has already failed. Thus, $\Pr\{X = 0\} = p$, and the set of possible values for X is $\mathcal{S} = [0, \infty)$, the nonnegative reals. We can use conditioning to eliminate these cases: let Y be the time before a machine fails, given that it is operational.

Property 11.4 *For any continuous random variable X with PDF $f(x)$,*

$$\Pr\{a \le X \le b\} = \int_a^b f(x)\ dx$$

Proof: from Property 10.4

$$\begin{aligned}
\Pr\{a \le X \le b\} &= \Pr\{(X \le b) \setminus (X < a)\} \\
&= \Pr\{X \le b\} - \Pr\{X < a\} \\
&= \Pr\{X \le b\} - \Pr\{X \le a\} \\
&= F(b) - F(a) \\
&= \int_a^b f(x)\ dx
\end{aligned}$$

We can write the "continuous version" of the rule that "probabilities sum to one":

Property 11.5 *For any continuous random variable,*

$$\int_{\mathcal{S}} f(x)\ dx = 1$$

Proof: directly from Property 11.4.

We can write the law of total probability for continuous random variables:

$$\Pr\{\mathcal{A}\} = \int_{\mathcal{S}} \Pr\{\mathcal{A}|X = x\} \cdot f(x)\ dx$$

Compare with the version for discrete random variables.

11.3.1 Important continuous distributions

Uniform

The continuous version of *Equilikely* is *Uniform*. $X \sim \mathit{Uniform}(a, b)$ iff

$$\begin{aligned}
\mathcal{S} &= (a, b) \\
f(x) &= \frac{1}{b-a}
\end{aligned}$$

Example 11.13

What is the CDF for $\mathit{Uniform}(a, b)$?

Solution:

$$\begin{aligned}
\Pr\{X \le x\} = F(x) &= \int_a^x f(y)\ dy \\
&= \int_a^x \frac{1}{b-a}\ dy \\
F(x) &= \frac{x-a}{b-a}
\end{aligned}$$

Exponential

$X \sim \mathit{Expo}(\lambda)$, for real–valued parameter $\lambda \geq 0$, iff

$$\begin{aligned} \mathcal{S} &= (0, \infty) \\ F(x) &= 1 - e^{-\lambda x} \end{aligned}$$

Example 11.14

What is the PDF of $\mathit{Expo}(\lambda)$?

Solution:

$$\begin{aligned} f(x) &= \frac{d}{dx}\ F(x) \\ &= \frac{d}{dx}\ (1 - e^{-\lambda x}) \\ f(x) &= \lambda e^{-\lambda x} \end{aligned}$$

Example 11.15

Suppose $X \sim \mathit{Uniform}(a, b)$ and $Y \sim \mathit{Uniform}(a, b)$. What is $\Pr\{X < Y\}$?

Solution: Use the law of total probability:

$$\begin{aligned} \Pr\{X < Y\} &= \int_a^b \Pr\{X < Y | Y = y\} \cdot f(y)\ dy \\ &= \int_a^b \Pr\{X < y\} \cdot f(y)\ dy \\ &= \int_a^b F(y) \cdot f(y)\ dy \\ &= \int_a^b \frac{y-a}{b-a} \cdot \frac{1}{b-a}\ dy \\ &= \frac{1}{(b-a)^2} \int_a^b y - a\ dy \\ &= \frac{1}{(b-a)^2} \left[y^2/2 - ya\right]_a^b \\ &= \frac{1}{(b-a)^2} \left[(b^2/2 - ba) - (a^2/2 - a^2)\right] \\ &= \frac{1}{(b-a)^2} \left[(b^2 - 2ab + a^2)/2\right] \\ \Pr\{X < Y\} &= 1/2 \end{aligned}$$

11.3.2 Expected value

Definition 11.6 *The expected value of a continuous random variable X is*

$$E[X] = \int_{\mathcal{S}} x \cdot f(x)\ dx$$

Compare this to the definition for discrete random variables. The intuitive meaning of $E[X]$ is the same as the discrete case.

Example 11.16

If $X \sim \textit{Expo}(\lambda)$, what is $E[X]$?

Solution:

$$\begin{aligned}
E[X] &= \int_0^\infty x \cdot f(x)\ dx \\
&= \int_0^\infty x\lambda e^{-\lambda x}\ dx \\
&= \lambda \int_0^\infty x e^{-\lambda x}\ dx \\
&\quad \text{Look up in table of integrals \ldots} \\
&= \lambda \left[\frac{e^{-\lambda x}}{\lambda^2}(-\lambda x - 1) \right]_0^\infty \\
&= \frac{1}{\lambda} \left[\left(\lim_{x\to\infty} e^{-\lambda x}(-\lambda x - 1) \right) - \left(e^0(0-1) \right) \right] \\
&= \frac{1}{\lambda} [0 - (-1)] \\
E[X] &= \frac{1}{\lambda}
\end{aligned}$$

Note: the parameter for *Expo* is the *rate*, not the *mean* (as it was in ComS 555).

Property 11.7 *For constants $\alpha_1, \ldots, \alpha_n$ and continuous random variables $X_1, \ldots, X_n$*

$$E[\alpha_1 X_1 + \cdots + \alpha_n X_n] = \alpha_1 E[X_1] + \cdots + \alpha_n E[X_n]$$

11.4 Independence

Definition 11.8 *Random variables X and Y are independent if*

$$\Pr\{X \le x | Y \le y\} = \Pr\{X \le x\}$$

for all $x \in \mathcal{S}_X, y \in \mathcal{S}_Y$. Otherwise they are dependent.

Property 11.9 *Random variables X and Y are independent iff*

$$\Pr\{X \le x, Y \le y\} = \Pr\{X \le x\} \cdot \Pr\{Y \le y\}$$

for all $x \in \mathcal{S}_x, y \in \mathcal{S}_y$.

Property 11.10 *Discrete random variables X and Y are independent iff*

$$\Pr\{X = x, Y = y\} = \Pr\{X = x\} \cdot \Pr\{Y = y\}$$

for all $x \in \mathcal{S}_x, y \in \mathcal{S}_y$.

Property 11.11 *Discrete random variables X and Y are independent iff*

$$\Pr\{X = x | Y = y\} = \Pr\{X = x\}$$

for all $x \in \mathcal{S}_x, y \in \mathcal{S}_y$.

Chapter 12

Stochastic processes

12.1 Definition

A stochastic process is a family of random variables

$$\{X(t), t \in \mathcal{T}\}$$

$\mathcal{T}$**:** the "index set". t is usually considered to be "time".

- Discrete $\mathcal{T}$: discrete–time stochastic process
- Continuous $\mathcal{T}$: continuous–time stochastic process

$\mathcal{S}$**:** possible values of $X(t)$ random variables.

- Called the "state space" of the stochastic process
- $\mathcal{S}$ can be continuous or discrete

Example 12.1

$X(t)$ = temperature at time t

- $\mathcal{T}$ is continuous
- $\mathcal{S}$ is continuous

Example 12.2

$Y(t)$ = number of spins of a coin during toss number t

- $\mathcal{T}$ is discrete
- $\mathcal{S}$ is discrete

Example 12.3

$Y(t)$ = number of customers in a queue at time t

- $\mathcal{T}$ is continuous

- $\mathcal{S}$ is discrete

Usually, the random variables in a stochastic process are related. In particular, the state of the process "now" may depend on the states observed at earlier times.

Example 12.4

- What is the temperature now?
- What is the temperature now, given that it was 50 degrees one hour ago?
- What is the temperature now, given that it was 50 degrees one hour ago and 45 degrees 2 hours ago?
- What is the temperature now, given that it was 50 degrees one hour ago, 45 degrees 2 hours ago, 35 degrees 3 hours ago?

12.2 Markov processes

A Markov process obeys the "memoryless" or "Markovian" property:

> the current state of the process depends only on the most recently observed state, not the entire past history.

In other words, knowledge of the current state is enough to predict future behavior; additional knowledge of the past does not help. Formally this property can be written as:

Property 12.1 *The Markovian property:*

$$\Pr\{X(t) \leq x | X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0\} = \Pr\{X(t) \leq x | X(t_n) = x_n\}$$

where $t > t_n > t_{n-1} > \cdots > t_0$ *are all times in* $\mathcal{T}$*, and* $x_n, x_{n-1}, \ldots, x_0$ *are all states in* $\mathcal{S}$*.*

Example 12.5

> The weather is not (usually) Markovian.

If $\mathcal{S}$ is discrete, then the Markov process is called a *Markov chain.*

$\mathcal{S}$	$\mathcal{T}$	Markov process
discrete	discrete	discrete–time Markov chain (DTMC)
discrete	continuous	continuous–time Markov chain (CTMC)

Chapter 13

Introduction to DTMCs

We will assume that $\mathcal{T} = \mathbb{N}$.

Property 13.1 *DTMC property:*

$$\Pr\{X(n+1) = x_{n+1} | X(n) = x_n, \ldots, X(0) = x_0\} = \Pr\{X(n+1) = x_{n+1} | X(n) = x_n\}$$

We will write this probability in shorthand as

$$\begin{aligned} P_{ij}(n) &= \Pr\{X(n+1) = j | X(n) = i\}, \qquad i, j \in \mathcal{S} \\ &= \text{Probability that the process is in state } j \text{ at time } n+1, \\ & \text{ given that it was in state } i \text{ at time } n. \end{aligned}$$

We will study an important special case: the probabilities do not change over time:

$$P_{ij}(n) = P_{ij}(0) = P_{ij}$$

These Markov chains are called *homogeneous*. Thus we have

$$\Pr\{X(n+1) = j | X(n) = i\} = P_{ij}$$

These probabilities are usually collected into a matrix, $\mathbf{P}$, with $\mathbf{P}[i,j] = P_{ij}$. $\mathbf{P}$ is called the *transition probability matrix.*

Property 13.2 *For any transition probability matrix* $\mathbf{P}$*,*

$$0 \le \mathbf{P}[i,j] \le 1$$

for all $i, j \in \mathcal{S}$*.*

Property 13.3 *For any transition probability matrix* $\mathbf{P}$*,*

$$\sum_{j \in \mathcal{S}} \mathbf{P}[i,j] = 1$$

for all $i \in \mathcal{S}$*. In other words, the rows of* $\mathbf{P}$ *sum to one. Why?*

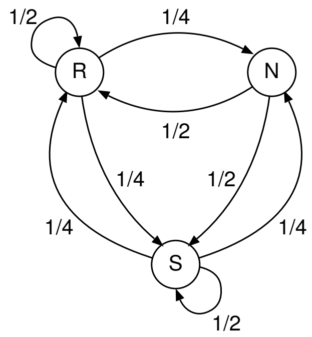
13.1 Example DTMCs

13.1.1 Land of Oz

This example is from *Finite Markov Chains*, by Kemeny and Snell.

$$\mathcal{S} = \{R, N, S\}$$

$$\mathbf{P} = \begin{array}{c} \\ R \\ N \\ S \end{array} \begin{array}{c} \begin{array}{ccc} R & N & S \end{array} \\ \left[\begin{array}{ccc} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{array}\right] \end{array}$$



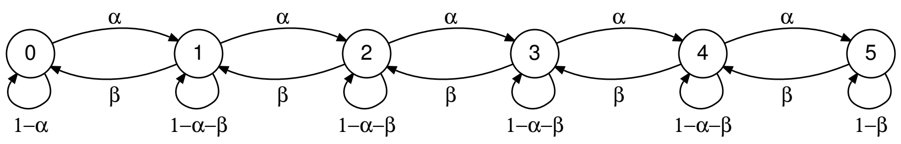
The weather in Oz is either *R*ainy, *N*ice, or *S*nowy. Tomorrow's weather depends only on today's weather, as given by transition probability matrix **P**. Note that there are never two nice days back to back. The DTMC can be drawn as a weighted, directed graph with incidence matrix **P**.

13.1.2 A "birth–death" chain

Suppose there is a small coffee house that can hold 5 customers. Every hour, we count the number of customers. Assume that the number of customers in the next hour either:

- Increases by one, unless it is already full, with probability α,
- Decreases by one, unless it is already empty, with probability β,
- or remains the same, otherwise.

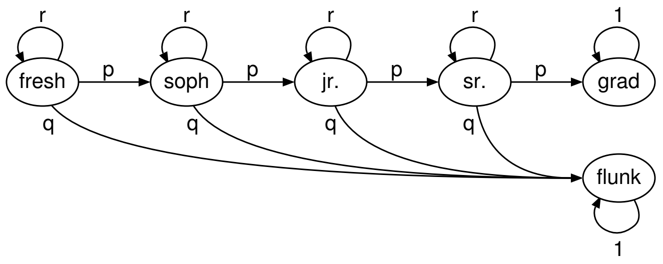
This stochastic process can be represented by the following DTMC, known as a "birth–death" Markov chain:



$$\mathcal{S} = \{0, 1, 2, 3, 4, 5\} \qquad \mathbf{P} = \begin{bmatrix} 1-\alpha & \alpha & 0 & 0 & 0 & 0 \\ \beta & 1-\alpha-\beta & \alpha & 0 & 0 & 0 \\ 0 & \beta & 1-\alpha-\beta & \alpha & 0 & 0 \\ 0 & 0 & \beta & 1-\alpha-\beta & \alpha & 0 \\ 0 & 0 & 0 & \beta & 1-\alpha-\beta & \alpha \\ 0 & 0 & 0 & 0 & \beta & 1-\beta \end{bmatrix}$$

13.1.3 University graduation

Each year, at a fictitious University, a student flunks out with probability q, passes with probability p, or has to repeat a year with probability r, where $p + q + r = 1$. This situation can be modeled with the following DTMC:



$\mathcal{S} = \{$*fresh*, *soph*, *jr*, *sr*, *grad*, *flunk*$\}$		
fresh	:	first year undergraduate
soph	:	second year undergraduate
jr	:	third year undergraduate
sr	:	fourth year undergraduate
grad	:	student has graduated
flunk	:	student has failed out

$$\mathbf{P} = \begin{array}{c|cccccc} & fr & so & jr & sr & gr & fl \\ \hline fr & r & p & 0 & 0 & 0 & q \\ so & 0 & r & p & 0 & 0 & q \\ jr & 0 & 0 & r & p & 0 & q \\ sr & 0 & 0 & 0 & r & p & q \\ gr & 0 & 0 & 0 & 0 & 1 & 0 \\ fl & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

13.2 Transient analysis of DTMCs

Question: If the process is in state i at time 0, what is the probability distribution of states at time 1?

I.e., what is $\Pr\{X(1) = j | X(0) = i\}$, for all j?

Want: a vector $\mathbf{p}_1$, of dimension $\mathcal{S}$, such that

$$\mathbf{p}_1[j] = \Pr\{X(1) = j | X(0) = i\}$$

Solution: $\mathbf{p}_1[j] = \mathbf{P}[i, j]$.
I.e., $\mathbf{p}_1$ is equal to row i of $\mathbf{P}$.

More interesting question: what is the probability distribution of states at time 1, given a distribution of states at time 0?

Have: vector $\mathbf{p}_0$, with $\mathbf{p}_0[i] = \Pr\{X(0) = i\}$

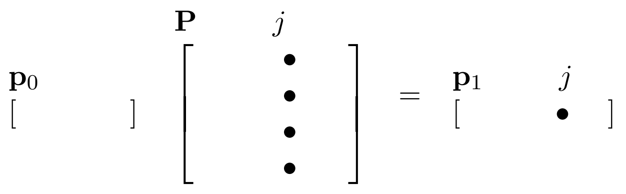
Want: vector $\mathbf{p}_1$, with $\mathbf{p}_1[j] = \Pr\{X(1) = j\}$

From the law of total probability, we have:

$$\mathbf{p}_1[j] \;=\; \Pr\{X(1) = j\} \;=\; \sum_{i \in \mathcal{S}} \Pr\{X(1) = j | X(0) = i\} \cdot \Pr\{X(0) = i\}$$

$$\begin{aligned} &= \sum_{i \in \mathcal{S}} \mathbf{P}[i,j] \cdot \mathbf{p}_0[i] \\ \mathbf{p}_1[j] &= \text{Dot product of: column } j \text{ of } \mathbf{P}\text{, and } \mathbf{p}_0 \end{aligned}$$

Picture of the above sum:



As a matrix equation, the above can be written as:

$$\mathbf{p}_1 = \mathbf{p}_0 \ \mathbf{P}$$

We are now ready to answer the following fundamental question.

Transient analysis question

Given the initial probability distribution $\mathbf{p}_0$, what is $\mathbf{p}_n$, the probability distribution at time n?

Transient analysis solution

Since the Markov chain is homogeneous, we know

$$\mathbf{p}_n = \mathbf{p}_{n-1} \ \mathbf{P}$$

for $n > 0$. We can use one of the following methods to compute $\mathbf{p}_n$:

Method 1 (used in practice):
start with $\mathbf{p}_0$,
multiply $\mathbf{p}_0$ by $\mathbf{P}$ to obtain $\mathbf{p}_1$,
multiply $\mathbf{p}_1$ by $\mathbf{P}$ to obtain $\mathbf{p}_2$,
etc.,
until we have obtained $\mathbf{p}_n$.

Method 2 (nice theoretical result):
Unfold the recursion:

$$\begin{aligned} \mathbf{p}_n &= \mathbf{p}_{n-1} \ \mathbf{P} \\ &= (\mathbf{p}_{n-2} \ \mathbf{P}) \ \mathbf{P} \\ &\vdots \\ \mathbf{p}_n &= \mathbf{p}_0 \ \mathbf{P}^n \end{aligned}$$

Example 13.1

Suppose it is a nice day in Oz. What will the weather be like in 3 days?

Solution:

$$\begin{aligned}
\mathbf{P} &= \begin{bmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix} \\
\mathbf{p}_0 &= \left[\ 0\ ,\ 1\ ,\ 0\ \right] \\
\\
\mathbf{p}_1 &= \mathbf{p}_0\mathbf{P} \\
&= \left[\ \tfrac{1}{2}\ ,\ 0\ ,\ \tfrac{1}{2}\ \right] \\
\\
\mathbf{p}_2 &= \mathbf{p}_1\mathbf{P} \\
&= \left[\ \tfrac{1}{2}\cdot\tfrac{1}{2}+0\cdot\tfrac{1}{2}+\tfrac{1}{2}\cdot\tfrac{1}{4}\ ,\ \tfrac{1}{2}\cdot\tfrac{1}{4}+0\cdot 0+\tfrac{1}{2}\cdot\tfrac{1}{4}\ ,\ \tfrac{1}{2}\cdot\tfrac{1}{4}+0\cdot\tfrac{1}{2}+\tfrac{1}{2}\cdot\tfrac{1}{2}\ \right] \\
&= \left[\ \tfrac{3}{8}\ ,\ \tfrac{2}{8}\ ,\ \tfrac{3}{8}\ \right] \\
\\
\mathbf{p}_3 &= \mathbf{p}_2\mathbf{P} \\
&= \left[\ \tfrac{3}{8}\cdot\tfrac{1}{2}+\tfrac{2}{8}\cdot\tfrac{1}{2}+\tfrac{3}{8}\cdot\tfrac{1}{4}\ ,\ \tfrac{3}{8}\cdot\tfrac{1}{4}+\tfrac{2}{8}\cdot 0+\tfrac{3}{8}\cdot\tfrac{1}{4}\ ,\ \tfrac{3}{8}\cdot\tfrac{1}{4}+\tfrac{2}{8}\cdot\tfrac{1}{2}+\tfrac{3}{8}\cdot\tfrac{1}{2}\ \right] \\
&= \left[\ \tfrac{13}{32}\ ,\ \tfrac{6}{32}\ ,\ \tfrac{13}{32}\ \right]
\end{aligned}$$

So, in 3 days,

- it will rain with probability 13/32
- it will be nice with probability 6/32
- it will snow with probability 13/32

Example 13.2

Suppose the weather for today in Oz has distribution

$$\mathbf{p}_0 = \left[\ 2/5\ ,\ 1/5\ ,\ 2/5\ \right]$$

i.e., today it will rain with probability 2/5, be nice with probability 1/5, and snow with probability 2/5. What is the distribution for tomorrow?

Solution:

$$\begin{aligned}
\mathbf{p}_1 &= \mathbf{p}_0\mathbf{P} \\
&= \left[\ \tfrac{2}{5}\cdot\tfrac{1}{2}+\tfrac{1}{5}\cdot\tfrac{1}{2}+\tfrac{2}{5}\cdot\tfrac{1}{4}\ ,\ \tfrac{2}{5}\cdot\tfrac{1}{4}+\tfrac{1}{5}\cdot 0+\tfrac{2}{5}\cdot\tfrac{1}{4}\ ,\ \tfrac{2}{5}\cdot\tfrac{1}{4}+\tfrac{1}{5}\cdot\tfrac{1}{2}+\tfrac{2}{5}\cdot\tfrac{1}{2}\ \right] \\
&= \left[\ \tfrac{2}{5}\ ,\ \tfrac{1}{5}\ ,\ \tfrac{2}{5}\ \right]
\end{aligned}$$

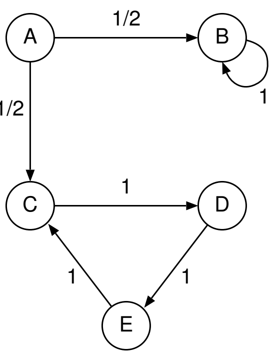
We will come back to this coincidence later.

Example 13.3

Consider the following DTMC:

$$\mathcal{S} = \{A, B, C, D, E\}$$

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



Suppose the initial state has distribution

$$\mathbf{p}_0 = \left[\ 1/3\ ,\ 1/3\ ,\ 1/3\ ,\ 0\ ,\ 0\ \right]$$

Compute the probability distribution at time $t = 4$.

Solution:

$$\begin{aligned} \mathbf{p}_1 &= \mathbf{p}_0\mathbf{P} \\ &= \left[\ 0\ ,\ \tfrac{1}{3}\cdot\tfrac{1}{2} + \tfrac{1}{3}\cdot 1\ ,\ \tfrac{1}{3}\cdot\tfrac{1}{2}\ ,\ \tfrac{1}{3}\cdot 1\ ,\ 0\ \right] \\ &= \left[\ 0\ ,\ \tfrac{1}{2}\ ,\ \tfrac{1}{6}\ ,\ \tfrac{1}{3}\ ,\ 0\ \right] \end{aligned}$$

$$\begin{aligned} \mathbf{p}_2 &= \mathbf{p}_1\mathbf{P} \\ &= \left[\ 0\ ,\ \tfrac{1}{2}\ ,\ 0\ ,\ \tfrac{1}{6}\ ,\ \tfrac{1}{3}\ \right] \end{aligned}$$

$$\begin{aligned} \mathbf{p}_3 &= \mathbf{p}_2\mathbf{P} \\ &= \left[\ 0\ ,\ \tfrac{1}{2}\ ,\ \tfrac{1}{3}\ ,\ 0\ ,\ \tfrac{1}{6}\ \right] \end{aligned}$$

$$\begin{aligned} \mathbf{p}_4 &= \mathbf{p}_3\mathbf{P} \\ &= \left[\ 0\ ,\ \tfrac{1}{2}\ ,\ \tfrac{1}{6}\ ,\ \tfrac{1}{3}\ ,\ 0\ \right] \\ &= \mathbf{p}_1 \end{aligned}$$

Since $\mathbf{p}_4 = \mathbf{p}_1$, we must have $\mathbf{p}_5 = \mathbf{p}_2$, $\mathbf{p}_6 = \mathbf{p}_3$, $\mathbf{p}_7 = \mathbf{p}_4 = \mathbf{p}_1$, ...

So, we have $\mathbf{p}_{n+1} = \mathbf{p}_{n \bmod 3 + 1}$

Example 13.4

What is $\Pr\{X(n) = j | X(0) = i\}$?

Using $\mathbf{p}_0 = [0, \ldots, 0, 1, 0, \ldots, 0]$, where the 1 corresponds to index i, then

$$\begin{aligned}
\Pr\{X(n) = j | X(0) = i\} &= \mathbf{p}_n[j] \\
&= (\mathbf{p}_0 \mathbf{P}^n)[j] \\
&= \mathbf{P}^n[i, j]
\end{aligned}$$

13.3 Multiplication on the right

We just saw that $\mathbf{x}\mathbf{P}^n$ gives the distribution at time n, where $\mathbf{x}$ is the distribution at time 0. What about $\mathbf{P}^n \mathbf{x}$?

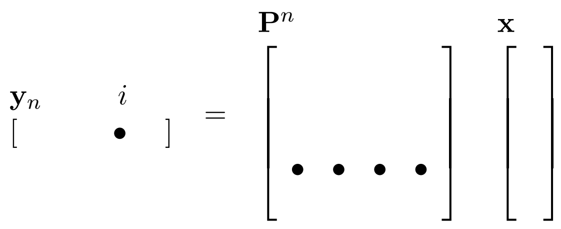
Consider the following. Let $\mathbf{x}$ be a vector of dimension $|\mathcal{S}|$. Define vector $\mathbf{y}_n$ as

$$\mathbf{y}_n[i] = E[\ \mathbf{x}[X(n)] \mid X(0) = i\ ].$$

Then $\mathbf{y}_n[i]$ is given by

$$\begin{aligned}
\mathbf{y}_n[i] &= E[\ \mathbf{x}[X(n)] \mid X(0) = i\ ] \\
&= \sum_{j \in \mathcal{S}} \mathbf{x}[j] \cdot \Pr\{X(n) = j | X(0) = i\} \\
&= \sum_{j \in \mathcal{S}} \mathbf{P}^n[i, j] \cdot \mathbf{x}[j] \\
\mathbf{y}_n[i] &= \text{Dot product of: row } i \text{ of } \mathbf{P}^n\text{, and } \mathbf{x}
\end{aligned}$$

Picture of the above sum:



As a matrix equation, the above can be written as:

$$\mathbf{y}_n = \mathbf{P}^n \mathbf{x}$$

which can be computed recursively by

$$\begin{aligned}
\mathbf{y}_0 &= \mathbf{x} \\
\mathbf{y}_n &= \mathbf{P}\mathbf{y}_{n-1}
\end{aligned}$$

Note that vector $\mathbf{y}_n$ is not a "probability vector", so its elements do not necessarily sum to one.

Example 13.5

In the land of Oz, what is the probability that it will be nice in 3 days, given knowledge of today's weather?

Solution:

$$
\begin{aligned}
\mathbf{P} &= \begin{bmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{2} \end{bmatrix} \\
\mathbf{y}_0 &= \left[\ 0\ ,\ 1\ ,\ 0\ \right] \\
\\
\mathbf{y}_1 &= \mathbf{P}\mathbf{y}_0 \\
&= \left[\ \frac{1}{4}\ ,\ 0\ ,\ \frac{1}{4}\ \right] \\
\\
\mathbf{y}_2 &= \mathbf{P}\mathbf{y}_1 \\
&= \left[\ \frac{1}{2}\cdot\frac{1}{4}+\frac{1}{4}\cdot 0+\frac{1}{4}\cdot\frac{1}{4}\ ,\ \frac{1}{2}\cdot\frac{1}{4}+0\cdot 0+\frac{1}{2}\cdot\frac{1}{4}\ ,\ \frac{1}{2}\cdot\frac{1}{4}+\frac{1}{4}\cdot 0+\frac{1}{4}\cdot\frac{1}{4}\ \right] \\
&= \left[\ \frac{3}{16}\ ,\ \frac{1}{4}\ ,\ \frac{3}{16}\ \right] \\
\\
\mathbf{y}_3 &= \mathbf{P}\mathbf{y}_2 \\
&\vdots \\
&= \left[\ \frac{13}{64}\ ,\ \frac{3}{16}\ ,\ \frac{13}{64}\ \right]
\end{aligned}
$$

So, in 3 days it will be nice

- with probability 13/64, if today is rainy
- with probability 3/16, if today is nice
- with probability 13/64, if today is snowy

13.4 Storage of Markov chains

Any of the data structures used to store a Kripke structure may also be used to store a Markov chain, with slight modifications (namely, we must store a probability with each edge). As with Kripke structures, the Markov chains we will use tend to be very large, but very sparse (i.e., containing mostly zero values). Using, say, a sparse representation where we maintain a linked list for each row of the matrix, we can exploit this sparseness. Note that there is no fundamental difference between storing a matrix "by rows" and storing it "by columns":

$$\mathbf{x}\mathbf{P} = \mathbf{P}^\top\mathbf{x}$$

To exploit sparseness in vector–matrix and matrix–vector multiplication, we must rewrite these operations in terms of generic "edges", rather than the typical doubly–nested loops that are used for full matrix storage.

Algorithm 13.1 Vector–matrix multiplication, computes $\mathbf{y} = \mathbf{x}\mathbf{P}$

```
$\mathbf{y} = \mathbf{0}$;
$\forall i, j$ such that $\mathbf{P}[i,j] \neq 0$ do
    $\mathbf{y}[j] = \mathbf{y}[j] + \mathbf{x}[i] \cdot \mathbf{P}[i,j]$;
End $\forall$
```

The order in which edges i, j are traversed should be whatever order is most efficient for the data structure used to store the matrix $\mathbf{P}$.

Algorithm 13.2 Matrix–vector multiplication, computes $\mathbf{y} = \mathbf{P}\mathbf{x}$

```
$\mathbf{y} = \mathbf{0}$;
$\forall i, j$ such that $\mathbf{P}[i,j] \neq 0$ do
    $\mathbf{y}[i] = \mathbf{y}[i] + \mathbf{P}[i,j] \cdot \mathbf{x}[j]$;
End $\forall$
```

The algorithms have the same complexity, which is equal to the complexity of traversing the edges. If the matrix $\mathbf{P}$ is sparse, and it is stored using a sparse data structure, then the above computations require $\mathcal{O}(|\mathcal{R}|)$ operations, where $\mathcal{R}$ is the set of edges with non-zero probability.

We will discuss advanced data structures for storing Markov chains later in the semester.

Chapter 14

DTMC properties

14.1 State classification for finite DTMCs

There is a fundamental difference between, say, the "land of Oz" DTMC and the "University graduation" DTMC. In the "land of Oz", for any state in $\mathcal{S}$, given enough time, there is a non-zero probability that the DTMC is in that state. In the "University graduation" DTMC, this is not true: once the DTMC reaches the state "grad", it can never go to the other states. The following definitions help to formalize this property. The definitions apply for *finite* Markov chains, i.e., Markov chains with a finite number of states.

Definition 14.1 *We say state j is* reachable *from state i, written $i \rightsquigarrow j$, if*

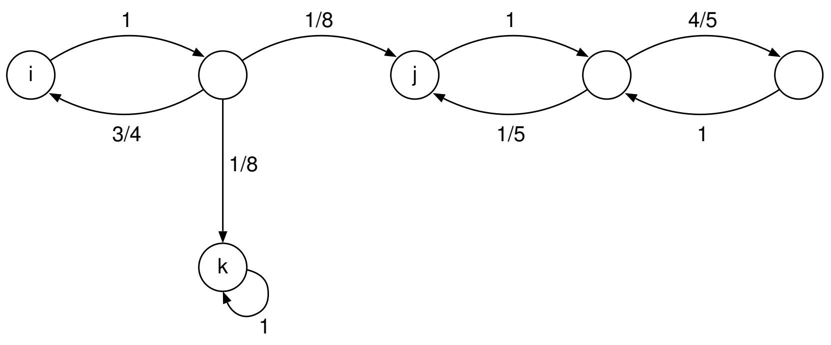
$$\exists t, \textit{such that} \quad \Pr\{X(t) = j | X(0) = i\} > 0$$

I.e., starting in state i now, it is possible to be in state j at some point in the future. If no such t exists, then state j is not reachable from state i, written $i \not\rightsquigarrow j$.

Property 14.2 *For a finite DTMC, $i \rightsquigarrow j$ iff there is a path in the DTMC (viewed as a graph) from state i to state j.*

Example 14.1

Consider the following DTMC for the next several examples.



In the above DTMC, we have $i \rightsquigarrow j$ and $i \rightsquigarrow k$.

Definition 14.3 *A state i is* transient *if*

$$\exists j \in \mathcal{S}, \textit{such that} \quad i \rightsquigarrow j \quad \textit{and} \quad j \not\rightsquigarrow i$$

I.e., from state i we can reach some state j, from which it is impossible to get back to state i.

Example 14.2

In the above DTMC, state i is transient, because $i \rightsquigarrow j$ but $j \not\rightsquigarrow i$.

Property 14.4 *If state i is transient, then*

1. *the probability that we never return to state i after leaving it is non-zero.*
2. $\lim_{n\to\infty} \Pr\{X(n) = i\} = 0$

Definition 14.5 *A state i is* recurrent *if it is not transient. Thus, a state i is recurrent if*

$$\forall j \ \textit{such that} \ i \rightsquigarrow j, \ j \rightsquigarrow i$$

Example 14.3

In the above DTMC, state j is recurrent, because from every state that we can reach from j, it is possible to get back to state j. Similarly, state k is recurrent.

Property 14.6 *If state i is recurrent, then*

1. *the probability that we never return state i after leaving it is zero. I.e., when we leave state i, we will eventually return with probability one.*
2. *We* **cannot say** *that* $\lim_{n\to\infty} \Pr\{X(n) = i\} \neq 0$

Definition 14.7 *A recurrent state is* absorbing *if it can reach only itself.*

Example 14.4

In the above DTMC, state k is absorbing.

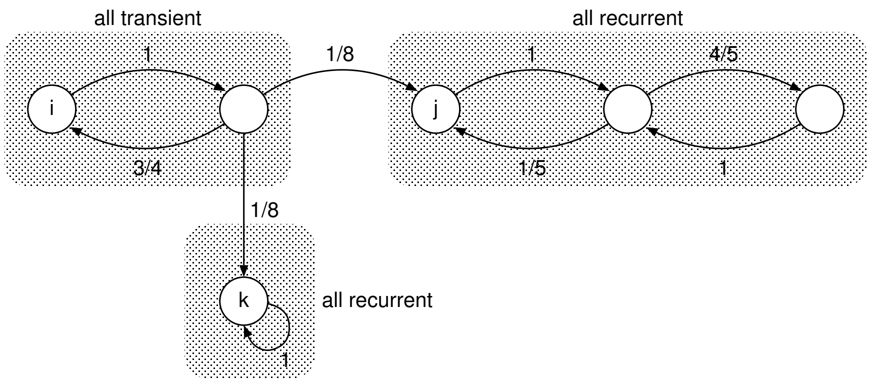
Definition 14.8 *We say states i and j are* mutually reachable *if $i \rightsquigarrow j$ and $j \rightsquigarrow i$.*

We can partition the state space $\mathcal{S}$ of a Markov chain into classes, where all states within a class are mutually reachable. These are exactly the same as strongly connected components (SCCs) from classical graph theory. Within a given class, either

- all states are transient; or
- all states are recurrent (called a *recurrent class*).

Example 14.5

The partition for the above example DTMC is:



Definition 14.9 *A Markov chain is* irreducible *if $\mathcal{S}$ is a recurrent class, i.e., all states are mutually reachable. Otherwise it is* reducible.

Example 14.6

The above example DTMC is reducible, since $\mathcal{S}$ is not a recurrent class.

Example 14.7

The "land of Oz" DTMC is irreducible.

14.2 Periodicity

14.2.1 Definition and properties

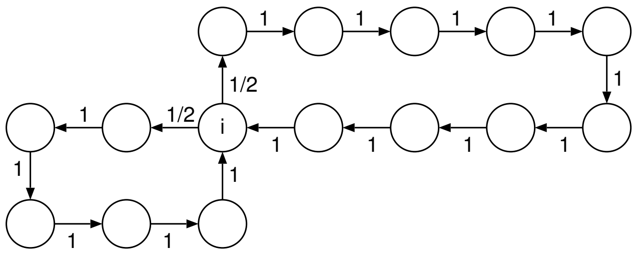
In a DTMC, if state i is recurrent, then by definition, for any state j such that $i \rightsquigarrow j$, $j \rightsquigarrow i$. That means there must exist some integer $n > 0$ such that

$$\Pr\{X(n) = i | X(0) = i\} > 0$$

We say n is a possible *return time* for state i.

Example 14.8

Consider the following DTMC for the next several examples.



6 is a return time for state i, because

$$\Pr\{X(6) = i | X(0) = i\} = \frac{1}{2}$$

Property 14.10 *If n_1 and n_2 are return times for a state i, then $n_1 + n_2$ is also a return time for state i.*

Example 14.9

For the above DTMC, 6 is a return time for state i (around the short loop). Also, 10 is a return time for state i (around the long loop). Thus, 12 is a return time for state i (around the short loop twice), 16 is a return time (short loop then long loop), ...

Property 14.11 *A recurrent state i has infinitely many return times.*

Definition 14.12 *Let $\mathcal{N}$ be the set of all return times for state i. $k = \gcd(\mathcal{N})$ is called the* period *of state i.*

Note that all return times for state i are multiples of its period. But, not all multiples of the period are necessarily return times.

Example 14.10

For the above DTMC, the return times for state i are

$$\mathcal{N} = \{6, 10, 12, 16, 18, 20, 22, 24, \ldots\}$$

Since the gcd of the integers in this set is 2, state i has period 2.

Property 14.13 *There exists an N such that, for all $n > N$, $n \cdot k$ is a return time for state i, where k is the period of state i.*

Proof: interesting and non-trivial, but omitted.

Property 14.14 *All states in a recurrent class have the same period.*

Proof (sketch):

Consider mutually reachable states i and j. Find a clever return time for i via state j, which is also a return time for state j. From this path length, show that the period of i must be a multiple of the period for j and vice versa.

Proof:

Suppose i has period k, and j has period k'. If i and j are in the same class, then they are mutually reachable. Suppose i can reach j in m steps, and j can reach i in n steps. Clearly, $m + n$ is a possible return time for both state i and state j. That means $m + n$ is a multiple of both k and k'.

From property 14.13, $\exists$ a prime p_1 such that $p_1 \cdot k$ is a return time for state i, where state j is visited at least once. Let t be the path up to state j, and u be the path after state j. Similarly, there is a prime $p_2 \neq p_1$ such that $p_2 \cdot k'$ is a return time for state j, where state i is visited at least once. Let v be the path up to state i, and w be the path after state i.

Since the path t, v, w, u starts and finishes at state i, $p_1 \cdot k + p_2 \cdot k'$ is a return time for state i, and must be a multiple of k. Since p_2 is prime, this implies k' is a multiple of k.

Similarly, the path v, t, u, w starts and finishes at state j, so $p_1 \cdot k + p_2 \cdot k'$ is a return time for state j and must be a multiple of k'. Since p_1 is prime, this implies k is a multiple of k'.

Thus, we must have $k' = k$.

So, we can refer to the period of a recurrent class, since all states in that class must have the same period.

Example 14.11

The DTMC in the above example consists of a single recurrent class (and therefore is irreducible). Thus, all states in the DTMC have period 2.

Definition 14.15 *A DTMC is* aperiodic *if all its states have period 1.*

Example 14.12

Our running example DTMC has two "loops", one with length 6, and the other with length 10. If we add one more state to the longer loop, we obtain a loop with length 11. Since 6 and 11 are relatively prime, state i will now have period 1. Thus, the new DTMC is aperiodic.

14.2.2 Algorithm to determine the period

Definition 14.16 *A* cycle *is a sequence of states $(s_0, s_1, \ldots, s_n)$ with $s_n = s_0$, and $\mathbf{P}[s_i, s_{i+1}] > 0$ for all $0 \le i < n$. We say the cycle has* length n.

Definition 14.17 *A* simple cycle *is a cycle $(s_0, s_1, \ldots, s_n)$ where $s_1, \ldots, s_n$ are distinct.*

Property 14.18 *The period of a recurrent class is*

$$k = gcd(n_1, \ldots, n_c)$$

where c is the number of simple cycles, and n_i is the length of simple cycle i.

Property 14.19 *If n_i and n_j are the lengths of cycles, then $n_i - n_j$ must be a multiple of the period.*

Our algorithm to find the period of a recurrent class is based on Property 14.19, and a clever trick to determine differences of cycle lengths.

Algorithm 14.1 Determine the period of a recurrent class

1. Select an arbitrary "starting state" s_0 in the recurrent class. We will determine its period, and hence, the period of all states in the class.

2. For each state s_i in the recurrent class, determine $distance(s_i)$, the length (in number of edges) of a shortest path from s_0 to s_i.
3. If $\mathbf{P}[s_i, s_j] > 0$ and $distance(s_i) \geq distance(s_j)$, then add

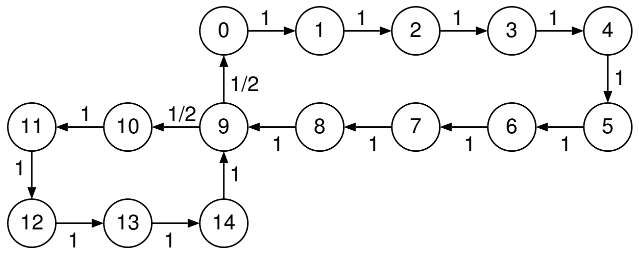
$$distance(s_i) - distance(s_j) + 1$$

to a set of integers.
4. The gcd of the set of integers gives the period.

Since these states belong to a recurrent class, whenever $distance(s_i) \geq distance(s_j)$ and $\mathbf{P}[s_i, s_j] > 0$, we have that $distance(s_j) + x$ is a return time for s_0 via state s_j and following some path back, and so is $distance(s_i) + 1 + x$, via state s_i, then s_j and the same path back. The difference of these cycle lengths, $distance(s_i) + 1 - distance(s_j)$, must be a multiple of the period.

Example 14.13

Running the algorithm on our example DTMC, we obtain the following distances (the start state, with distance 0, is chosen arbitrarily):



There are two cases with $distance(s_i) \geq distance(s_j)$.

- The edge from distance 9 to distance 0
- The edge from distance 14 to distance 9

This gives us the set of integers $\mathcal{L} = \{9 - 0 + 1, 14 - 9 + 1\} = \{10, 6\}$. Taking the gcd of these integers, we obtain a period of 2.

What is the complexity of Algorithm 14.1? Consider each step of the algorithm.

- Step 1: Select a starting state. $O(1)$
- Step 2: Determine distances. Can use a breadth-first search, requires $O(|\mathcal{E}|)$ time and $O(|\mathcal{S}|)$ memory.
- Steps 3 and 4: Find edges with property, accumulate gcd. $O(|\mathcal{E}|)$ time

Overall complexity: $O(|\mathcal{E}|)$ running time, requires $O(|\mathcal{S}|)$ auxiliary storage.

Chapter 15

Ergodic DTMCs

We saw earlier how to perform transient analysis, i.e., to compute the probability distribution at time n, stored as the vector $\mathbf{p}_n$. In this chapter we will begin to address the following fundamental questions:

1. When does $\lim_{n\to\infty} \mathbf{p}_n$ exist?
2. If the limit exists, how can it be computed?

When the limit exists, it is called the *steady–state* or *stationary* distribution of the DTMC.

Definition 15.1 *A DTMC is* ergodic *if*

- *it is irreducible, and*
- *it is aperiodic.*

From previous definitions, that means that

- Every pair of states i and j are mutually reachable.
- Every state has period 1.

Property 15.2 *In an ergodic DTMC, for all states i,*

$$\exists N \textit{ such that } \Pr\{X(t) = i | X(0) = i\} > 0, \forall t > N$$

Proof: follows directly from the fact that state i has period 1.

Property 15.3 *In an ergodic DTMC, for all pairs of states i and j,*

$$\exists N \textit{ such that } \Pr\{X(t) = j | X(0) = i\} > 0, \forall t > N$$

Proof: follows from the above property, and the fact that states i and j are mutually reachable.

We can restate the above property as follows:

Property 15.4 *In an ergodic DTMC, there exists an N such that all entries of $\mathbf{P}^t$ are greater than zero, for all $t > N$.*

This is crucial to proving the main result about ergodic DTMCs, but requires the following other theorem first.

Property 15.5 *Given a transition probability matrix* $\mathbf{P}$ *with smallest entry* $\epsilon > 0$*, a (column) vector* $\mathbf{x}$ *with largest element* $M_0 \leq 1$ *and smallest element* $m_0 \geq 0$*, the (column) vector* $\mathbf{y} = \mathbf{P}\mathbf{x}$ *has largest element* M_1 *and smallest element* m_1 *where*

1. $M_1 \leq M_0$
2. $m_1 \geq m_0$
3. $M_1 - m_1 \leq (1 - 2\epsilon)(M_0 - m_0)$

Proof:

1. *Let* $\mathbf{x}'$ *be a vector with all elements equal to* M_0*, except for one element equal to* m_0*, at the same position as in* $\mathbf{x}$*. Note* $\mathbf{x} \leq \mathbf{x}'$*.*

   Let $\mathbf{y}' = \mathbf{P}\mathbf{x}'$*. Each element of* $\mathbf{y}'$ *has the form*

   $$a \cdot m_0 + (1 - a)M_0$$

   where a *is one entry in* $\mathbf{P}$*, and* $(1-a)$ *is from the remainder of that row. The largest element of* $\mathbf{y}'$ *occurs when* a *is smallest. Since* $\mathbf{y} \leq \mathbf{y}'$*, we have*

   $$M_1 \leq \epsilon \cdot m_0 + (1 - \epsilon)M_0 = M_0 - \epsilon(M_0 - m_0)$$

2. *(sketch): similar to proof of (1), but use* $-\mathbf{x}$ *and obtain*

   $$-m_1 \leq -m_0 - \epsilon(-m_0 + M_0)$$

3. *Add the inequalities obtained above together:*

   $$\begin{aligned} M_1 - m_1 &\leq (M_0 - \epsilon(M_0 - m_0)) - (-m_0 - \epsilon(-m_0 + M_0)) \\ &\leq M_0 - m_0 - 2\epsilon(M_0 - m_0) \\ &\leq (1 - 2\epsilon)(M_0 - m_0) \end{aligned}$$

We are now ready to prove the main result for ergodic DTMCs.

Property 15.6 *If* $\mathbf{P}$ *is the transition probability matrix of an ergodic DTMC, then*

1. $\lim_{n\to\infty} \mathbf{P}^n$ *exists, call it* $\mathbf{P}^\infty$
2. $\mathbf{P}^\infty$ *has rows all equal to the same probability vector* $\boldsymbol{\rho}$
3. *All elements of* $\boldsymbol{\rho}$ *are positive*

Proof sketch:

- *Since the DTMC is ergodic, $\exists N$ such that $\mathbf{P}^m$ has all nonzero entries, for all $m > N$. This proves (3).*

- *Consider $\mathbf{P}^{m+1} = \mathbf{P}^m\mathbf{P}$, for $m > N$. From property 15.5 we know that the difference between the largest and smallest elements of column j of $\mathbf{P}^{m+1}$ is less than the difference between the largest and smallest elements of column j of $\mathbf{P}^m$.*

- *Thus, in the limit as $m \to \infty$, elements of column j of $\mathbf{P}^m$ have no difference. This proves (1) and (2).*

We can now answer the fundamental questions from the start of this chapter.

Property 15.7 *Given an ergodic DTMC, for any initial probability distribution $\mathbf{p}_0$,*

$$\begin{aligned}
\lim_{n\to\infty} \mathbf{p}_n &= \lim_{n\to\infty} \mathbf{p}_0\mathbf{P}^n \\
&= \mathbf{p}_0 \lim_{n\to\infty} \mathbf{P}^n \\
&= \mathbf{p}_0\mathbf{P}^\infty \\
&= \sum_{\forall i} \mathbf{p}_0[i] \cdot \boldsymbol{\rho} \\
&= \boldsymbol{\rho}
\end{aligned}$$

Property 15.8 *$\boldsymbol{\rho}$ is the* unique *probability vector satisfying*

$$\boldsymbol{\rho}\mathbf{P} = \boldsymbol{\rho}$$

Proof: for any probability vector $\mathbf{x}$ such that $\mathbf{x}\mathbf{P} = \mathbf{x}$, repeated substitution gives us $\mathbf{x}\mathbf{P}^n = \mathbf{x}$. From the previous property, as $n \to \infty$ we get $\mathbf{x}\mathbf{P}^n \to \boldsymbol{\rho}$. Therefore $\mathbf{x} = \boldsymbol{\rho}$.

15.1 Computing the steady–state distribution

For an ergodic DTMC, we have a unique steady–state distribution $\boldsymbol{\rho}$. Given an ergodic DTMC with matrix $\mathbf{P}$, how do we compute $\boldsymbol{\rho}$ in practice?

15.1.1 Power method

From property 15.7, $\boldsymbol{\rho} = \lim_{n\to\infty} \mathbf{p}_0\mathbf{P}$. Thus, we can use any initial $\mathbf{p}_0$ and perform transient analysis to compute $\mathbf{p}_n$ for a very large n.

- Better to use a "uniform" initial distribution, i.e., $\mathbf{p}_0[i] = 1/|\mathcal{S}|$, than start in a particular state.

- Solution requires (only) vector–matrix multiplication.

- Requires 2 vectors (one for $\mathbf{p}_n$, one for $\mathbf{p}_{n-1}$).

- Can keep going until vectors "converge".

- Downside: may converge slowly.

15.1.2 Linear algebra

From property 15.8,

$$\begin{aligned} \boldsymbol{\rho}\mathbf{P} &= \boldsymbol{\rho} \\ \boldsymbol{\rho}\mathbf{P} - \boldsymbol{\rho} &= \mathbf{0} \\ \boldsymbol{\rho}(\mathbf{P} - \mathbf{I}) &= \mathbf{0} \end{aligned}$$

Solve the above linear system for $\boldsymbol{\rho}$. This can be done in various ways:

- By hand. Usually unpleasant, especially as $|\mathcal{S}|$ grows.
- Direct approaches (Gaussian elimination)
- Indirect approaches

We will discuss these methods soon.

15.2 Examples

Example 15.1

Compute the steady–state distribution for the Land of Oz DTMC.

Solve $\boldsymbol{\rho}(\mathbf{P} - \mathbf{I}) = \mathbf{0}$, by hand.

$$\mathbf{P} - \mathbf{I} = \begin{bmatrix} -1/2 & 1/4 & 1/4 \\ 1/2 & -1 & 1/2 \\ 1/4 & 1/4 & -1/2 \end{bmatrix}$$

Let $\boldsymbol{\rho} = [r, n, s]$. Multiply out $\boldsymbol{\rho}(\mathbf{P} - \mathbf{I}) = \mathbf{0}$ to obtain 3 equations (one equation per column of $\mathbf{P}$):

$$\begin{array}{rcrcrcl} -1/2\ r & + & 1/2\ n & + & 1/4\ s & = & 0 \\ 1/4\ r & - & n & + & 1/4\ s & = & 0 \\ 1/4\ r & + & 1/2\ n & - & 1/2\ s & = & 0 \end{array}$$

If we add the equations together, we get $0 = 0$. This will *always* be the case, for *every* DTMC. (Why?) Thus, we have one equation too few. This makes sense, since $a\boldsymbol{\rho}\mathbf{P} = a\boldsymbol{\rho}$, for any scalar a. Since we want $\boldsymbol{\rho}$ to be a *probability* vector, we have one more equation:

$$r + n + s = 1$$

Eq(3) - Eq(2) gives us

$$\begin{aligned} 3/2n - 3/4s &= 0 \\ 3/2n &= 3/4s \\ 2n &= s \end{aligned}$$

Plug this into Eq(3):

$$\begin{aligned}
1/4r + 1/2n - 1/2(2n) &= 0 \\
1/4r &= 1/2n \\
r &= 2n
\end{aligned}$$

Now, using $r + n + s = 1$, we obtain:

$$\begin{aligned}
2n + n + 2n &= 1 \\
n &= 1/5
\end{aligned}$$

Thus, the steady–state distribution is

$$\boldsymbol{\rho} = \left[\ 2/5\ ,\ 1/5\ ,\ 2/5\ \right]$$

Example 15.2

Consider the DTMC with $\mathcal{S} = \{A, B\}$ and

$$\mathbf{P} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

This DTMC is irreducible, but has period 2. What happens if we try to solve $\boldsymbol{\rho}(\mathbf{P} - \mathbf{I}) = \mathbf{0}$? We have the following set of equations:

$$\left[\ a\ ,\ b\ \right] \cdot \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} = \left[\ 0\ ,\ 0\ \right]$$

Multiplying out the above, we obtain

$$\begin{aligned}
-a + b &= 0 \\
a - b &= 0
\end{aligned}$$

Both equations imply $a = b$. Since $a + b = 1$, we obtain

$$\boldsymbol{\rho} = \left[\ 1/2\ ,\ 1/2\ \right]$$

What happened?

- Just because a solution exists to the system $\boldsymbol{\rho}(\mathbf{P} - \mathbf{I}) = \mathbf{0}$, does not mean that $\boldsymbol{\rho}$ is the steady–state vector; only if the DTMC is ergodic is this true.
- I.e., **If** the DTMC is ergodic, **then** the above equation may be used to determine the steady–state vector.
- In this particular case, if $\mathbf{p}_0 = [1, 0]$, then we have

$$\begin{aligned}
\mathbf{p}_0 &= [1, 0] \\
\mathbf{p}_1 &= [0, 1] \\
\mathbf{p}_2 &= [1, 0] \\
&\vdots
\end{aligned}$$

  and clearly $\lim_{n\to\infty} \mathbf{p}_n$ does not exist.

- But, if $\mathbf{p}_0 = \left[\ 1/2\ ,\ 1/2\ \right]$ then we have

$$\begin{aligned}\mathbf{p}_0 &= [1/2, 1/2] \\ \mathbf{p}_1 &= [1/2, 1/2] \\ &\vdots\end{aligned}$$

  and clearly $\lim_{n\to\infty} \mathbf{p}_n = \boldsymbol{\rho}$.

- As this example illustrates, in general, the long–term behavior of a DTMC depends on the initial distribution.

Chapter 16

Absorbing DTMCs

We will now study absorbing DTMCs, where every state is either transient or absorbing:

- $\mathcal{Z}$: set of transient states
- $\mathcal{A}$: set of absorbing states
- $\mathcal{S} = \mathcal{Z} \cup \mathcal{A}$

The following quantities may be of interest when studying absorbing DTMCs:

- Which absorbing state will the DTMC end up in, if we know the initial state (or initial distribution)? I.e., determine $\lim_{n\to\infty} \Pr\{X(n) = i\}$, for all $i \in \mathcal{A}$, given $\mathbf{p}_0$.
- How long, on average, does it take before the DTMC reaches an absorbing state? (This is called the "mean time to absorption") Again, this will depend on the initial state (or distribution).

16.1 Structure of absorbing DTMCs

Given the transition probability matrix $\mathbf{P}$ for an absorbing DTMC, we can re-order the states so that the transient states come first, then the absorbing states. This will produce the following block structure:

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}] & \mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

transient–transient: arcs from transient states to transient states are represented by the block $\mathbf{P}[\mathcal{Z},\mathcal{Z}]$, a square matrix of dimension $|\mathcal{Z}|$.

transient–absorbing: arcs from transient states to absorbing states are represented by the block $\mathbf{P}[\mathcal{Z},\mathcal{A}]$. Note that $\mathbf{P}[\mathcal{Z},\mathcal{A}] > \mathbf{0}$, i.e., it contains at least one non-zero entry.

absorbing–transient: there are no arcs from absorbing states to transient states; this produces the $\mathbf{0}$ block.

absorbing–absorbing: there are arcs from absorbing states to absorbing states, but only self loops. This gives an identity matrix of dimension $|\mathcal{A}|$.

Property 16.1 *For any $a \in \mathcal{A}$ in an absorbing DTMC,*

$$\Pr\{X(n+1)=a\} \geq \Pr\{X(n)=a\}$$

Proof: let $\mathbf{p}_n$ be any probability distribution at time n. Look at entry a of $\mathbf{p}_{n+1}$:

$$\begin{aligned}
\mathbf{p}_{n+1}[a] &= (\mathbf{p}_n\mathbf{P})[a] \\
&= \mathbf{p}_n[\mathcal{Z}]\mathbf{P}[\mathcal{Z},a] + \mathbf{p}_n[\mathcal{A}]\mathbf{P}[\mathcal{A},a] \\
&= \mathbf{p}_n[\mathcal{Z}]\mathbf{P}[\mathcal{Z},a] + \mathbf{p}_n[a]
\end{aligned}$$

Property 16.2 *In an absorbing DTMC,*

$$\forall i \in \mathcal{S}, \quad i \rightsquigarrow a \textit{ for some } a \in \mathcal{A}$$

Proof:

If $i \in \mathcal{A}$, then the property holds trivially. Otherwise i is transient, and by definition, $\exists i_1 \in \mathcal{S} \setminus \{i\}$ such that $i \rightsquigarrow i_1, i_1 \not\rightsquigarrow i$.

If $i_1 \in \mathcal{A}$, then the property holds trivially. Otherwise i_1 is transient, and by definition, $\exists i_2 \in \mathcal{S} \setminus \{i, i_1\}$ such that $i_1 \rightsquigarrow i_2, i_2 \not\rightsquigarrow i_1$.

Repeat the above argument, and note that the set of states from which we can select the next state is $\mathcal{S} \setminus \{i, i_1, i_2, \ldots, i_n\}$ after n iterations of the argument. Eventually, we will have to select an absorbing state, since the set $\mathcal{S}$ is finite.

Property 16.3 $\mathbf{P}^n[\mathcal{Z},\mathcal{Z}] = \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n$

Proof by induction on n. Trivially it holds for $n = 1$. For $n > 1$, we get

$$\begin{aligned}
\mathbf{P}^n[\mathcal{Z},\mathcal{Z}] &= (\mathbf{P}^{n-1}\mathbf{P})[\mathcal{Z},\mathcal{Z}] \\
&= \mathbf{P}^{n-1}[\mathcal{Z},\mathcal{Z}]\mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{P}^{n-1}[\mathcal{Z},\mathcal{A}]\mathbf{0} \\
&= \mathbf{P}[\mathcal{Z},\mathcal{Z}]^{n-1}\mathbf{P}[\mathcal{Z},\mathcal{Z}]
\end{aligned}$$

Property 16.4 *Let $\mathbf{A}$ be a matrix with non-negative elements, with largest rowsum $\alpha < 1$. Then,*

$$\lim_{n\to\infty} \mathbf{A}^n = \mathbf{0}$$

Proof (sketch): show by induction that the largest rowsum of $\mathbf{A}^n$ is at most α^n. Since $\alpha < 1$, we know $\lim_{n\to\infty} \alpha^n = 0$, and since the rowsums of $\mathbf{A}^n$ go to zero, the elements must also go to zero.

Property 16.5

$$\lim_{n\to\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n = \mathbf{0}$$

From properties 16.1 and 16.2, there exists an m such that every row of $\mathbf{P}^m[\mathcal{Z},\mathcal{A}]$ contains at least one non-zero entry. Thus, every row of $\mathbf{P}^m[\mathcal{Z},\mathcal{Z}] = \mathbf{P}[\mathcal{Z},\mathcal{Z}]^m$ sums to less than one. Now, we have

$$\begin{aligned}\lim_{n\to\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n &= \lim_{n\to\infty} (\mathbf{P}[\mathcal{Z},\mathcal{Z}]^m)^n \\ &= \mathbf{0}\end{aligned}$$

where the last step is due to property 16.4.

Property 16.6 *In an absorbing DTMC,*

1. $\lim_{n\to\infty} \Pr\{X(n) \in \mathcal{Z}\} = 0$
2. $\lim_{n\to\infty} \Pr\{X(n) \in \mathcal{A}\} = 1$

16.2 Fundamental matrix

Property 16.7 *(The fundamental theorem for absorbing DTMCs):*

1. $\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}]$ *has an inverse*
2. $(\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}])^{-1} = \sum_{k=0}^{\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k$

Proof: Note that

$$\begin{aligned}(\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}]) \sum_{k=0}^{n} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k &= \sum_{k=0}^{n} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k - \mathbf{P}[\mathcal{Z},\mathcal{Z}] \sum_{k=0}^{n} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k \\ &= \sum_{k=0}^{n} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k - \sum_{k=1}^{n+1} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k \\ &= \mathbf{P}[\mathcal{Z},\mathcal{Z}]^0 - \mathbf{P}[\mathcal{Z},\mathcal{Z}]^{n+1}\end{aligned}$$

Now, take the limit as $n \to \infty$. Since $\lim_{n\to\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n = \mathbf{0}$, we get

$$(\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}]) \sum_{k=0}^{\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^k = \mathbf{I}$$

Definition 16.8 $\mathbf{N} = (\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}])^{-1}$ *is called the Fundamental matrix.*

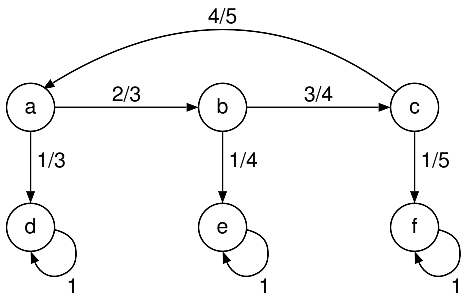
What is the meaning of $\mathbf{N}[i,j]$? (Note $i, j \in \mathcal{Z}$)

$$\begin{aligned}\mathbf{N}[i,j] &= \sum_{k=0}^{\infty} (\mathbf{P}[\mathcal{Z},\mathcal{Z}]^k)[i,j] \\ &= \sum_{k=0}^{\infty} \mathbf{P}^k[i,j]\end{aligned}$$

$$\begin{aligned}
&= \sum_{k=0}^{\infty} \Pr\{X(k)=j|X(0)=i\} \\
&= \sum_{k=0}^{\infty} E[1, \text{ if } X(k)=j, \text{ 0 otherwise, given } X(0)=i] \\
&= \sum_{k=0}^{\infty} E[\text{number of visits to state } j \text{ at time } k, \text{ given } X(0)=i] \\
&= E\left[\sum_{k=0}^{\infty} \text{number of visits to state } j \text{ at time } k, \text{ given } X(0)=i\right] \\
\mathbf{N}[i,j] &= E[\text{total number of visits to state } j, \text{ given } X(0)=i]
\end{aligned}$$

Example 16.1

Consider the following absorbing DTMC for the next few examples:



The transient states are: $\mathcal{Z} = \{a, b, c\}$
The absorbing states are: $\mathcal{A} = \{d, e, f\}$
The transition probability matrix is:

$$\mathbf{P} = \left[\begin{array}{ccc|ccc}
0 & 2/3 & 0 & 1/3 & 0 & 0 \\
0 & 0 & 3/4 & 0 & 1/4 & 0 \\
4/5 & 0 & 0 & 0 & 0 & 1/5 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1
\end{array}\right]$$

The fundamental matrix is:

$$\mathbf{N} = (\mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}])^{-1} = \begin{bmatrix} 1 & -2/3 & 0 \\ 0 & 1 & -3/4 \\ -4/5 & 0 & 1 \end{bmatrix}^{-1} = \cdots = \begin{bmatrix} 5/3 & 10/9 & 5/6 \\ 1 & 5/3 & 5/4 \\ 4/3 & 8/9 & 5/3 \end{bmatrix}$$

So, if the DTMC starts in state a, the expected number of visits to state a is 5/3.

16.3 Expected number of visits to each state

Suppose we want to compute a vector $\mathbf{n}$, where $\mathbf{n}[j]$ is the expected number of visits to state j, given an initial probability distribution $\mathbf{p}_0$. What is $\mathbf{n}[j]$? Solution: condition on the initial state:

$$\begin{aligned} \mathbf{n}[j] &= \sum_{i\in\mathcal{Z}} E[\text{visits to } j|\text{start in } i] \cdot \Pr\{\text{start in } i\} \\ &= \sum_{i\in\mathcal{Z}} \mathbf{N}[i,j] \cdot \mathbf{p}_0[i] \\ &= \mathbf{N}[\mathcal{Z},j] \cdot \mathbf{p}_0[\mathcal{Z}] \end{aligned}$$

where the last equation is the dot product of column j of $\mathbf{N}$ with the transient portion of $\mathbf{p}_0$. Thus, we have

Property 16.9

$$\mathbf{n} = \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N}$$

is a vector that counts the expected number of visits to each transient state, given an initial probability distribution.

Property 16.10

$$\mathbf{n} = \sum_{n=0}^{\infty} \mathbf{p}_n[\mathcal{Z}]$$

where $\mathbf{p}_n$ is the probability distribution at time n.

Since the DTMC never visits any transient states once an absorbing state has been reached, and since time starts at 0, the total time the DTMC spends in transient states is equal to the time at which the DTMC first reaches an absorbing state. Thus, we have the following

Property 16.11 *The mean time to absorption of an absorbing DTMC can be found by summing the elements of* $\mathbf{n}$.

Example 16.2

For the previous absorbing DTMC, suppose

$$\mathbf{p}_0 = \left[\ 1/3\ ,\ 0\ ,\ 1/3 \mid 1/3\ ,\ 0\ ,\ 0\ \right]$$

Then $\mathbf{p}_0[\mathcal{Z}] = \left[\ 1/3\ ,\ 0\ ,\ 1/3\ \right]$ and we can compute $\mathbf{n}$ as

$$\begin{aligned} \mathbf{n} &= \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N} \\ &= \left[\ 1/3 \cdot 5/3 + 1/3 \cdot 4/3\ ,\ 1/3 \cdot 10/9 + 1/3 \cdot 8/9\ ,\ 1/3 \cdot 5/6 + 1/3 \cdot 5/3\ \right] \\ &= \left[\ 1\ ,\ 2/3\ ,\ 5/6\ \right] \end{aligned}$$

The mean time to absorption is thus

$$1 + 2/3 + 5/6 = 5/2$$

16.4 Computational considerations

How is the vector $\mathbf{n}$ computed in practice? Mathematicians use

$$\mathbf{n} = \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N}$$

but computationally this is difficult because construction of $\mathbf{N}$ requires inverting a matrix.

- The DTMC is typically extremely large and sparse, which means $\mathbf{P}[\mathcal{Z}, \mathcal{Z}]$ is large and sparse.
- The matrix $\mathbf{N}$ is usually *not* sparse, even if $\mathbf{P}[\mathcal{Z}, \mathcal{Z}]$ is sparse. Thus, the straightforward approach is expensive in terms of storage.
- Computation of $\mathbf{N}$ using matrix inversion is expensive in terms of computation time.

Fortunately, we can eliminate these issues by rewriting the above equation:

$$\begin{aligned}
\mathbf{n} &= \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N} \\
\mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) &= \mathbf{p}_0[\mathcal{Z}] \cdot \mathbf{N} \cdot (\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) \\
\mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) &= \mathbf{p}_0[\mathcal{Z}] \\
-\mathbf{n}(\mathbf{I} - \mathbf{P}[\mathcal{Z}, \mathcal{Z}]) &= -\mathbf{p}_0[\mathcal{Z}] \\
\mathbf{n}(\mathbf{P}[\mathcal{Z}, \mathcal{Z}] - \mathbf{I}) &= -\mathbf{p}_0[\mathcal{Z}]
\end{aligned}$$

The vector $\mathbf{n}$ can be computed by solving the above linear system. Benefits of this approach:

- We need to store only the matrix $\mathbf{P} - \mathbf{I}$, which can exploit sparse (or other) matrix representations.
- The linear equations are similar to those required for steady–state analysis of ergodic DTMCs.

Example 16.3

To compute $\mathbf{n}$ for our previous example DTMC by solving the linear equations, let $\mathbf{n} = [a, b, c]$, and multiply out the matrix equation

$$[a, b, c] \cdot \begin{bmatrix} -1 & 2/3 & 0 \\ 0 & -1 & 3/4 \\ 4/5 & 0 & -1 \end{bmatrix} = [-1/3, 0, -1/3]$$

to obtain the following system of equations:

$$\begin{array}{rcrcrcl}
-\ a & & & + & 4/5\ c & = & -1/3 \\
2/3\ a & - & b & & & = & 0 \\
 & & 3/4\ b & - & c & = & -1/3
\end{array}$$

Multiplying the second equation by 3/5 and the third equation by 4/5 gives us

$$\begin{array}{rcrcrcl}
-\ a & & & + & 4/5\ c & = & -1/3 \\
2/5\ a & - & 3/5\ b & & & = & 0 \\
 & & 3/5\ b & - & 4/5\ c & = & -4/15
\end{array}$$

Adding all three equations together, we obtain

$$\begin{aligned} -3/5\ a &= -9/15 \\ a &= 1 \end{aligned}$$

Plugging into the first equation gives us

$$\begin{aligned} -1 + 4/5\ c &= -1/3 \\ 4/5\ c &= 2/3 \\ c &= 5/6 \end{aligned}$$

and plugging into the second equation gives us

$$\begin{aligned} 2/3 - b &= 0 \\ 2/3 &= b \end{aligned}$$

Thus, we obtain $\mathbf{n} = [\ 1\ ,\ 2/3\ ,\ 5/6\ ]$.

16.5 Limiting distribution

Suppose we want to know the probability that the DTMC eventually reached each of the absorbing states. In other words, we want to compute

$$\lim_{n\to\infty} \Pr\{X(n) = i\}$$

for all absorbing states $i \in \mathcal{A}$. (Recall that the above probabilities are zero for transient states.)

Let's multiply $\mathbf{P}$ out to see what happens to $\mathbf{P}^n$:

$$\begin{aligned}
\mathbf{P} &= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}] & \mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\
\mathbf{P}^2 &= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}] & \mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}] & \mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^2 + \mathbf{P}[\mathcal{Z},\mathcal{A}]\cdot\mathbf{0} & \mathbf{P}[\mathcal{Z},\mathcal{Z}]\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{P}[\mathcal{Z},\mathcal{A}]\cdot\mathbf{I} \\ \mathbf{0}\cdot\mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I}\cdot\mathbf{0} & \mathbf{0}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{I}^2 \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^2 & (\mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I})\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\
\mathbf{P}^4 &= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^2 & (\mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I})\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix}^2 \\
&= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^4 + (\ldots)\cdot\mathbf{0} & (\mathbf{P}[\mathcal{Z},\mathcal{Z}]^2 + \mathbf{I})\cdot(\mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I})\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^4 & (\mathbf{P}[\mathcal{Z},\mathcal{Z}]^3 + \mathbf{P}[\mathcal{Z},\mathcal{Z}]^2 + \mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I})\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix}
\end{aligned}$$

Hopefully, it is clear that $\mathbf{P}^n$ is

$$\mathbf{P}^n = \begin{bmatrix} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n & (\mathbf{P}[\mathcal{Z},\mathcal{Z}]^{n-1} + \cdots + \mathbf{P}[\mathcal{Z},\mathcal{Z}] + \mathbf{I}) \cdot \mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

Now, take the limit as $n \to \infty$. What do we get?

- From earlier, $\lim_{n\to\infty} \mathbf{P}[\mathcal{Z},\mathcal{Z}]^n = \mathbf{0}$.
- What is $\lim_{n\to\infty}(\mathbf{P}[\mathcal{Z},\mathcal{Z}]^n + \cdots + \mathbf{I})$? This is the definition of the fundamental matrix.

Thus, we get

$$\mathbf{P}^\infty = \lim_{n\to\infty} \mathbf{P}^n = \begin{bmatrix} \mathbf{0} & \mathbf{N}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix}$$

Let $\mathbf{p}_\infty = \lim_{n\to\infty} \mathbf{p}_n$. From transient analysis, we know

$$\begin{aligned} \mathbf{p}_\infty &= \lim_{n\to\infty} \mathbf{p}_n \\ &= \lim_{n\to\infty} \mathbf{p}_0 \mathbf{P}^n \\ &= \mathbf{p}_0 \lim_{n\to\infty} \mathbf{P}^n \\ &= [\ \mathbf{p}_0[\mathcal{Z}]\ ,\ \mathbf{p}_0[\mathcal{A}]\ ] \cdot \begin{bmatrix} \mathbf{0} & \mathbf{N}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \\ &= [\ \mathbf{0}\ ,\ \mathbf{p}_0[\mathcal{Z}]\cdot\mathbf{N}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{p}_0[\mathcal{A}]\ ] \\ \mathbf{p}_\infty &= [\ \mathbf{0}\ ,\ \mathbf{n}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{p}_0[\mathcal{A}]\ ] \end{aligned}$$

Thus, we can determine which absorbing state is finally reached by multiplying $\mathbf{n}$ by $\mathbf{P}[\mathcal{Z},\mathcal{A}]$, which corresponds to the chance of entering each of the absorbing states each time we visit a transient state, and adding $\mathbf{p}_0[\mathcal{A}]$, which is the probability that we started in each absorbing state.

Example 16.4

For the example DTMC used previously, we determined

$$\mathbf{n} = [\ 1\ ,\ 2/3\ ,\ 5/6\ ]$$

from an initial distribution of

$$\mathbf{p}_0 = \left[\ 1/3\ ,\ 0\ ,\ 1/3 \mid 1/3\ ,\ 0\ ,\ 0\ \right]$$

What is $\mathbf{p}_\infty$?

For this DTMC we have

$$\mathbf{P}[\mathcal{Z},\mathcal{A}] = \begin{bmatrix} 1/3 & 0 & 0 \\ 0 & 1/4 & 0 \\ 0 & 0 & 1/5 \end{bmatrix}$$

so we obtain

$$\begin{aligned} \mathbf{n}\cdot\mathbf{P}[\mathcal{Z},\mathcal{A}] &= \left[\ 1\cdot 1/3\ ,\ 2/3\cdot 1/4\ ,\ 5/6\cdot 1/5\ \right] \\ &= \left[\ 1/3\ ,\ 1/6\ ,\ 1/6\ \right] \end{aligned}$$

Thus, we have

$$\begin{aligned}\mathbf{p}_\infty &= \left[\ 0\ ,\ 0\ ,\ 0 \mid 1/3+1/3\ ,\ 1/6+0\ ,\ 1/6+0\ \right] \\ &= \left[\ 0\ ,\ 0\ ,\ 0 \mid 2/3\ ,\ 1/6\ ,\ 1/6\ \right]\end{aligned}$$

Thus, the DTMC is absorbed into state d with probability 2/3, into state e with probability 1/6, and into state f with probability 1/6.

Note that for absorbing DTMCs, the limiting distribution depends on the initial distribution.

Part II

Advanced topics

Chapter 17

Introduction to CTMCs

We will assume that $\mathcal{T} = \mathbb{R}^* = [0, \infty)$, the non-negative reals.

Property 17.1 *CTMC property:*

$$\Pr\{X(t) = x | X(t_n) = x_n, \ldots, X(t_0) = x_0\} = \Pr\{X(t) = x | X(t_n) = x_n\}$$

where $t > t_n > \cdots > t_0$ are all times in $\mathcal{T}$, and $x, x_n, \ldots, x_0$ are all states in $\mathcal{S}$.

Again, we will only consider *homogeneous* CTMCs, where

$$\Pr\{X(t+h) = j | X(t) = i\} = \Pr\{X(h) = j | X(0) = i\}$$

for $h \geq 0$; i.e., the probabilities depend on the *time difference*, not the actual time. For shorthand, we will denote this as a matrix

$$\mathbf{P}_h[i,j] = \Pr\{X(h) = j | X(0) = i\}$$

For now, we will assume that *transitions between states are not arbitrarily fast*[1]:

$$\lim_{h \to 0} \mathbf{P}_h = \mathbf{I}$$

I.e., as the time difference goes to zero, the probability of changing states goes to zero.

In the discrete case, we discussed how the process evolves with every "clock tick", i.e., as time advances by discrete steps. Since time is now continuous, to discuss how the process evolves at an infinitesimally–small time instant, we must take the derivative of $\mathbf{P}_h$. Since we are considering the homogeneous case, this derivative must be independent of the time instant, so we will look at time zero. In particular, let $\mathbf{Q}$ be the derivative of $\mathbf{P}_h$ at time 0:

$$\begin{aligned} \mathbf{Q} &= \lim_{h \to 0} \frac{\mathbf{P}_h - \mathbf{P}_0}{h} \\ &= \lim_{h \to 0} \frac{\mathbf{P}_h - \mathbf{I}}{h} \end{aligned}$$

The units of each element $\mathbf{Q}[i,j]$ are "probability per time unit"; or, think of it as a "flow of probability from state i to state j"; or, think of it as "how does the probability of going from state i to state j increase over time".

Let's examine the entries of matrix $\mathbf{Q}$.

[1]We will discuss how to relax this restriction later.

- The "off–diagonal" elements:

$$\mathbf{Q}[i,j] = \lim_{h\to 0} \frac{\mathbf{P}_h[i,j] - 0}{h}$$

  Since $\mathbf{P}_h[i,j] \geq 0$, we must have $\mathbf{Q}[i,j] \geq 0$. This makes sense: because $\mathbf{P}_0[i,j]$ is zero (the smallest possible probability value), $\mathbf{P}_h[i,j]$ cannot decrease.

- The "diagonal" elements:

$$\mathbf{Q}[i,i] = \lim_{h\to 0} \frac{\mathbf{P}_h[i,i] - 1}{h}$$

  Since $\mathbf{P}_h[i,i] \leq 1$, we must have $\mathbf{Q}[i,i] \leq 0$. This also makes sense: because $\mathbf{P}_0[i,i]$ is one (the largest possible probability value), $\mathbf{P}_h[i,i]$ cannot increase.

For any value h, though, we must have that the rows of $\mathbf{P}_h$ sum to one. Therefore the rows of $\mathbf{P}_h - \mathbf{I}$ sum to zero. Thus, the rows of $\mathbf{Q}$ sum to zero. Therefore, we must have that

Property 17.2

$$\mathbf{Q}[i,i] = -\sum_{j\neq i} \mathbf{Q}[i,j]$$

for all $i \in \mathcal{S}$.

This property makes sense: if the probability of going to some other state from state i increases with time, then the probability of remaining in state i must decrease with time, at the same rate.

Definition 17.3 *The matrix*

$$\mathbf{Q} = \lim_{h\to 0} \frac{\mathbf{P}_h - \mathbf{I}}{h}$$

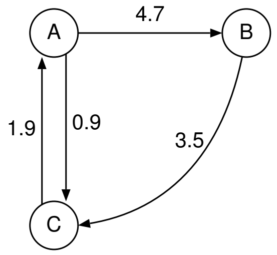
is called the infinitesimal generator matrix.

Sometimes we use the *transition rate matrix* $\mathbf{R}$:

$$\mathbf{R}[i,j] = \begin{cases} \mathbf{Q}[i,j] & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

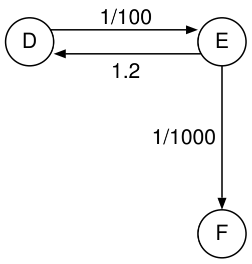
CTMCs are usually drawn as weighted, directed graphs with incidence matrix $\mathbf{R}$.

Example 17.1



$$\mathbf{R} = \begin{bmatrix} 0 & 4.7 & 0.9 \\ 0 & 0 & 3.5 \\ 1.9 & 0 & 0 \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} -5.6 & 4.7 & 0.9 \\ 0 & -3.5 & 3.5 \\ 1.9 & 0 & -1.9 \end{bmatrix}$$

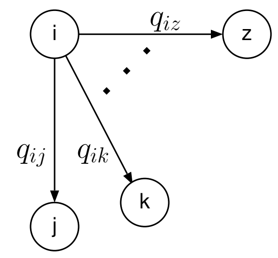
Example 17.2



This CTMC might represent a failure / repair model: state D represents a working state, state E represents a failed state, and state F represents a failed state that cannot be repaired. Note that F is an absorbing state.

17.1 Intuitive meaning of the rates

Consider a particular state i, with several non-zero rates of transition to other states:



To understand the meaning of these rates, let's determine

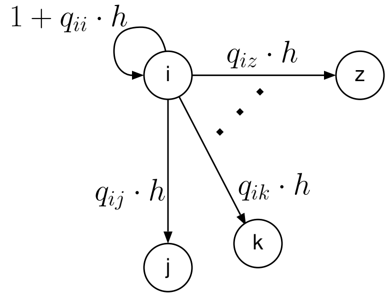
1. How long does the CTMC remain in state i, before changing states? This is a random variable.
2. Once leaving state i, which state will we enter? This is also random, so what we really want is the probability of entering each state.

How long do we stay in state i

Suppose we observe the state of the CTMC every h time units, for small h. This gives us a DTMC, where the time step for the DTMC is "observation number" of the CTMC. Since h is small, we can estimate the probability of going from state i to state j in one observation as $\mathbf{P}_h[i,j] \approx q_{ij} \cdot h$, for $i \neq j$. The probability of remaining in state i after one observation is

$$1 - (q_{ij} \cdot h + \cdots + q_{iz} \cdot h) \;=\; 1 + q_{ii} \cdot h \;\approx\; \mathbf{P}_h[i,i]$$

Thus, we obtain the following as the "observation DTMC".



Let random variable M equal the time spent in state i, before the first state change. Let $t = n \cdot h$, i.e., the CTMC time t corresponds to observation n of the DTMC. What is the CDF of M?

$$\begin{aligned}
\Pr\{M < t\} &= \Pr\{\text{CTMC left state } i \text{ before time } t\} \\
&\approx \Pr\{\text{DTMC left state } i \text{ before observation } n\} \\
&\approx 1 - \Pr\{\text{DTMC still in state } i \text{ at observation } n\} \\
&\approx 1 - (1 + q_{ii} \cdot h)^n \\
&\approx 1 - (1 + q_{ii} \cdot t/n)^n
\end{aligned}$$

To obtain the exact value of the probability, we take the limit as $h \to 0$, and therefore $n \to \infty$, keeping $n \cdot h = t$ fixed:

$$\begin{aligned} \Pr\{M < t\} &= \lim_{n\to\infty} 1 - (1 + q_{ii} \cdot t/n)^n \\ &= 1 - e^{q_{ii} \cdot t} \end{aligned}$$

Thus, $M \sim \mathit{Expo}(-q_{ii}) = \mathit{Expo}(q_{ij} + \cdots + q_{iz})$.

Which state do we go to

Using the "observation DTMC", we can make all states except state i absorbing, to determine what state is reached after leaving state i. The only transient state is i, and we have

$$\begin{aligned} \mathbf{P}[\mathcal{Z},\mathcal{Z}] &= [\ 1 + q_{ii} \cdot h\ ] \\ \mathbf{I} - \mathbf{P}[\mathcal{Z},\mathcal{Z}] &= [\ -q_{ii} \cdot h\ ] \\ \mathbf{N} &= \left[\ -\frac{1}{q_{ii} \cdot h}\ \right] \\ \mathbf{P}[\mathcal{Z},\mathcal{A}] &= [\ q_{ij} \cdot h\ ,\ \ldots\ ,\ q_{iz} \cdot h\ ] \\ \mathbf{N} \cdot \mathbf{P}[\mathcal{Z},\mathcal{A}] &= \left[\ \frac{q_{ij}}{-q_{ii}}\ ,\ \ldots\ ,\ \frac{q_{iz}}{-q_{ii}}\ \right] \end{aligned}$$

Recall that $\mathbf{N} \cdot \mathbf{P}[\mathcal{Z},\mathcal{A}]$ gives the probability of being absorbed into each absorbing state. Now, take the limit as $h \to 0$. Since the vector does not depend on h, this is trivial. Thus, in the CTMC, once we leave state i, we go to state j with probability

$$\frac{q_{ij}}{-q_{ii}} = \frac{q_{ij}}{q_{ij} + \cdots + q_{iz}}$$

Summary of CTMC behavior

Recall: if we have $X_j \sim \mathit{Expo}(q_{ij}), X_k \sim \mathit{Expo}(q_{ik}), \ldots, X_z \sim \mathit{Expo}(q_{iz})$ then $X = \min(X_j, X_k, \ldots X_z)$ has distribution $\mathit{Expo}(q_{ij} + q_{ik} + \cdots + q_{iz})$. And, $X = X_j$ with probability

$$\frac{q_{ij}}{q_{ij} + q_{ik} + \cdots + q_{iz}}$$

Therefore, we have the following.

- As soon as the CTMC enters state i, look at all the outgoing arcs to states $j, k, \ldots, z$ and sample a random time $X_j \sim \mathit{Expo}(q_{ij})$, $X_k \sim \mathit{Expo}(q_{ik})$, ..., $X_z \sim \mathit{Expo}(q_{iz})$ for each arc.
- Whichever time is smallest "wins", and after that amount of time, the CTMC will change to that state. E.g., if X_m is the smallest, then after X_m time the CTMC switches from state i to state m.
- The CTMC will remain in state i for time

$$\min(X_j, X_k, \ldots, X_z) \sim \mathit{Expo}(-q_{ii})$$

- The CTMC will switch to state j when $X_j = \min(X_j, X_k, \ldots, X_z)$, which occurs with probability
$$\frac{q_{ij}}{q_{ij} + q_{ik} + \cdots + q_{iz}}$$

Chapter 18

Analyzing CTMCs

18.1 Transient analysis

Suppose we have a CTMC, and an initial distribution, and we want to know the probability distribution at a particular (finite) time t:

$$\boldsymbol{\pi}_t[i] = \Pr\{X(t) = i\}$$

In the discrete–time case, we found the distribution by looking at the change in probability as one time step passed. In the continuous–time case, we again will use the continuous analog: differentiation. Thus, we will work towards finding

$$\lim_{h \to 0} \frac{\Pr\{X(t+h) = i\} - \Pr\{X(t) = i\}}{h}$$

First, we will derive an expression for $\Pr\{X(t+h) = i\}$, conditioned on an observation of the state at time t. In the following discussion, we use the notation $i \xrightarrow{t} j$ to mean that the CTMC changed state from i to j at time t. We can enumerate all the ways the CTMC can be in state i at time $t+h$, based on the number of state changes since the last observation at time t:

$$\begin{aligned}
\Pr\{X(t+h) = i\} \quad = \quad & \Pr\{X(t) = i, \text{and the CTMC did not change state}\} \\
& + \\
& \sum_{j \neq i} \Pr\left\{X(t) = j, j \xrightarrow{t_1} i, \text{and we remain in } i\right\} \\
& \qquad \text{for } t < t_1 < t+h \\
& + \\
& \sum_{k \neq j} \sum_{j \neq i} \Pr\left\{X(t) = k, k \xrightarrow{t_1} j, j \xrightarrow{t_2} i, \text{and we remain in } i\right\} \\
& \qquad \text{for } t < t_1 < t_2 < t+h \\
& + \\
& \vdots
\end{aligned}$$

This raises an interesting question: how many state changes are possible in a short time h? More practically, we want to know, for what n does

$$\lim_{h \to 0} \frac{\Pr\{\text{The CTMC changes state } n \text{ times in time } h\}}{h}$$

become zero, so we can limit our work on the above sums.

Example 18.1

Suppose we want to know the time required for a CTMC to change from state i to state j. This can be modeled by a random variable $X \sim \mathit{Expo}(\lambda)$. Thus, we can compute:

$$\begin{aligned}
\lim_{h\to 0} \frac{\Pr\{X < h\}}{h} &= \lim_{h\to 0} \frac{1 - e^{-\lambda h}}{h} \\
&= \lim_{h\to 0} \frac{\lambda e^{-\lambda h}}{1} \\
&= \lambda
\end{aligned}$$

Example 18.2

Suppose we want to know the time required for a CTMC to change from state i to state j, then from state j to state k. This can be modeled by a random variable $X = X_1 + X_2$, where $X_1 \sim \mathit{Expo}(\lambda)$, $X_2 \sim \mathit{Expo}(\mu)$. First, we need to know the CDF of X:

$$\begin{aligned}
\Pr\{X < t\} &= \Pr\{X_1 + X_2 < t\} \\
&= \int_0^t \Pr\{X_1 + X_2 < t | X_2 = x\} \cdot \mu e^{-\mu x}\, dx \\
&= \int_0^t \left(1 - e^{-\lambda(t-x)}\right) \cdot \mu e^{-\mu x}\, dx \\
&\vdots \\
\Pr\{X < t\} &= 1 - \frac{\lambda e^{-\mu t} - \mu e^{-\lambda t}}{\lambda - \mu} \qquad \text{for } \lambda \neq \mu
\end{aligned}$$

Now, we can compute the limit, as in the previous example:

$$\begin{aligned}
\lim_{h\to 0} \frac{\Pr\{X < h\}}{h} &= \lim_{h\to 0} \frac{\lambda - \mu - \lambda e^{-\mu h} + \mu e^{-\lambda h}}{(\lambda - \mu)h} \\
&= \lim_{h\to 0} \frac{\lambda\mu e^{-\mu h} - \lambda\mu e^{-\lambda h}}{(\lambda - \mu)} \\
&= 0
\end{aligned}$$

The case $\lambda = \mu$ is left as an exercise.

From the previous two examples, we see that we do not need to consider two or more state changes in the CTMC in our derivation of $\boldsymbol{\pi}_{t+h}[i] = \Pr\{X(t+h) = i\}$, since these have a limit of zero.

$$\begin{aligned}
\lim_{h\to 0} \frac{\boldsymbol{\pi}_{t+h}[i]}{h} &= \lim_{h\to 0} \frac{\boldsymbol{\pi}_t[i] \cdot \Pr\{X(t+h) = i | X(t) = i\}}{h} + \\
&\quad \lim_{h\to 0} \frac{\sum_{j\neq i} \boldsymbol{\pi}_t[j] \cdot \Pr\{X(t+h) = i | X(t) = j\}}{h} \\
&= \lim_{h\to 0} \frac{\sum_{\forall j} \boldsymbol{\pi}_t[j] \cdot \Pr\{X(t+h) = i | X(t) = j\}}{h} \\
&= \lim_{h\to 0} \frac{\sum_{\forall j} \boldsymbol{\pi}_t[j] \cdot \mathbf{P}_h[j, i]}{h}
\end{aligned}$$

As usual, the above sum reduces to vector–matrix multiplication, and we have

$$\begin{aligned}
\lim_{h\to 0}\frac{\boldsymbol{\pi}_{t+h}}{h} &= \lim_{h\to 0}\frac{\boldsymbol{\pi}_t\mathbf{P}_h}{h}\\
\lim_{h\to 0}\frac{\boldsymbol{\pi}_{t+h}-\boldsymbol{\pi}_t}{h} &= \lim_{h\to 0}\frac{\boldsymbol{\pi}_t\mathbf{P}_h-\boldsymbol{\pi}_t}{h}\\
&= \boldsymbol{\pi}_t\lim_{h\to 0}\frac{\mathbf{P}_h-\mathbf{I}}{h}\\
\frac{d}{dt}\boldsymbol{\pi}_t &= \boldsymbol{\pi}_t\mathbf{Q}
\end{aligned}$$

We have a differential equation!

Example 18.3

The equations for transient analysis of DTMCs can be rewritten as

$$\begin{aligned}
\mathbf{p}_{n+1} &= \mathbf{p}_n\mathbf{P}\\
\mathbf{p}_{n+1}-\mathbf{p}_n &= \mathbf{p}_n\mathbf{P}-\mathbf{p}_n\\
\mathbf{p}_{n+1}-\mathbf{p}_n &= \mathbf{p}_n(\mathbf{P}-\mathbf{I})
\end{aligned}$$

which is the discrete version of the above differential equation.

Example 18.4

Suppose we have the differential equation

$$\frac{d}{dx}y = y\cdot q$$

and we want to know y as a function of x. In other words, for what $y = f(x)$ does the above hold? While solving differential equations is not a prerequisite for this course, we can "guess" a solution and verify it using calculus (which *is* a prerequisite):

$$\begin{aligned}
y &= c\cdot e^{qx} \qquad c \text{ is a constant}\\
\frac{d}{dx}y &= \frac{d}{dx}\,c\cdot e^{qx}\\
&= cq\cdot e^{qx}\\
&= y\cdot q
\end{aligned}$$

Thus, $f(x) = c\,e^{qx}$ is a solution. Note that, since a differential equation simply specifies the "slope" of the function, we cannot solve for the constant c unless we have more information, in particular one point of the function, say $y_0 = f(0)$:

$$y_0 \;=\; f(0) \;=\; c\,e^0 \;=\; c$$

Thus, if the point y_0 is known, then the solution is

$$y = y_0\cdot e^{qx}$$

Note the previous example has the same form as our differential equation for $\boldsymbol{\pi}_t$. Therefore, the solution must be

Property 18.1 *(CTMC distributions at finite time t)*

$$\boldsymbol{\pi}_t = \boldsymbol{\pi}_0 \cdot e^{\mathbf{Q}t}$$

The above equation uses the *matrix exponential*, which can be computed using the Taylor expansion for e^x:

$$\begin{aligned}\boldsymbol{\pi}_t &= \boldsymbol{\pi}_0 \cdot e^{\mathbf{Q}t} \\ &= \boldsymbol{\pi}_0 \cdot \sum_{n=0}^{\infty} \frac{t^n}{n!} \mathbf{Q}^n\end{aligned}$$

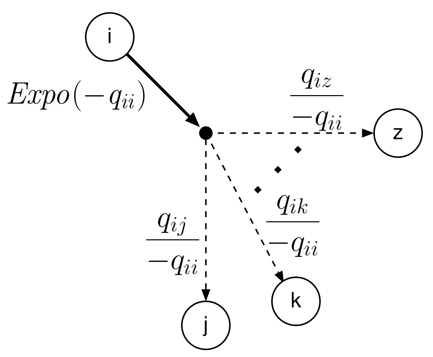
The above is useful mathematically, but it is impractical for computing $\boldsymbol{\pi}_t$ for two important reasons:

1. If $\mathbf{Q}$ is large and sparse, as n increases, $\mathbf{Q}^n$ will become less and less sparse. So computing $\mathbf{Q}^n$ will not be possible for large CTMCs.
2. The matrix $\mathbf{Q}$ contains positive and negative elements. That means that computing $\mathbf{Q}^n$ will require lots of floating–point subtractions.

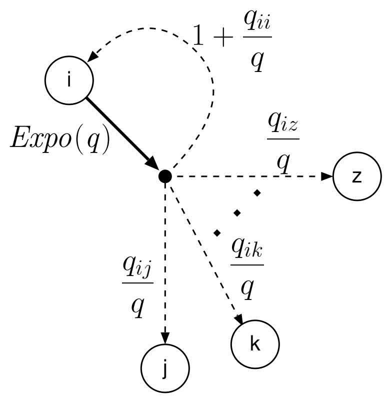
There is another approach that does not suffer from the above two problems.

18.1.1 Uniformization

As discussed earlier, a CTMC remains in state i for *Expo*$(-q_{ii})$ time, before changing states. Then, it changes to state j with probability $-q_{ij}/q_{ii}$. In the method of *uniformization*, we adjust the CTMC so that the time spent in state i before changing states is *Expo*(q). To achieve this, while still keeping the original behavior of the CTMC, we allow the CTMC to return to state i with the appropriate probability, and adjust the probabilities of changing states, as follows.



Original CTMC



Uniformized CTMC

First, note that the outgoing probabilities on the dotted arcs of the "uniformized CTMC" sum to one:

$$1 + \frac{q_{ii}}{q} + \frac{q_{ij}}{q} + \cdots + \frac{q_{iz}}{q} \;=\; 1 + \frac{\sum_k q_{ik}}{q} \;=\; 1 + \frac{0}{q} \;=\; 1$$

Next, note that the expected rate of leaving state i in the uniformized CTMC

$$q \cdot \Pr\{\text{We do not loop back to state } i\} \;=\; q \cdot \left(1 - \left(1 + \frac{q_{ii}}{q}\right)\right) \;=\; -\,q_{ii}$$

is the same as in the original CTMC. Finally, the expected rate of going from state i to state j in the uniformized CTMC

$$q \cdot \Pr\{\text{We switch to state } j\} \;=\; q \cdot \frac{q_{ij}}{q} \;=\; q_{ij}$$

is the same as in the original CTMC.

The above transformation works only if the above values on the dotted arcs are indeed probabilities, which implies $q \geq q_{ij}$ and $q \geq -q_{ii}$. Thus, we must have q at least as large in magnitude as the largest element of matrix $\mathbf{Q}$. There is no other restriction as to how we select q.

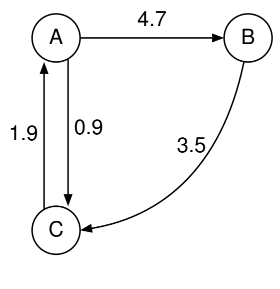
Now, the time between state changes in the uniformized CTMC is *Expo*(q), regardless of which state the CTMC is currently in. If we ignore (or treat separately) the time required to change state, and look only at the state changes, we obtain a DTMC. The transition probabilities for this DTMC correspond exactly to the probabilities on the dotted arcs in the above picture. We therefore can obtain a uniformized DTMC using the matrix equation

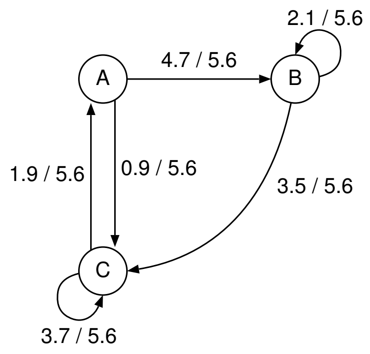
$$\mathbf{P} = \frac{1}{q}\mathbf{Q} + \mathbf{I}$$

Note that time in the DTMC corresponds to "attempt number to change state" (at rate *Expo*(q)).

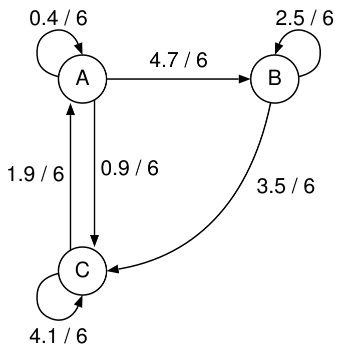
Example 18.5

For the CTMC below (left), we can obtain the following uniformized DTMCs, using various values of q:





$q = 5.6$



$q = 6.0$

Original CTMC

Uniformized DTMCs

Note that the smallest possible value of q is 5.6.

Note we can rewrite the uniformization equation as

$$\mathbf{Q} = q \cdot (\mathbf{P} - \mathbf{I})$$

Let's substitute this into the matrix exponential and see what happens:

$$\begin{aligned}
\boldsymbol{\pi}_t &= \boldsymbol{\pi}_0 \cdot e^{\mathbf{Q}t} \\
&= \boldsymbol{\pi}_0 \cdot e^{qt(\mathbf{P}-\mathbf{I})} \\
&= \boldsymbol{\pi}_0 \cdot e^{qt\mathbf{P}} \cdot e^{-qt\mathbf{I}} \\
&= \boldsymbol{\pi}_0 \cdot \left(\sum_{n=0}^{\infty} \frac{(qt)^n}{n!}\mathbf{P}^n\right) \cdot \left(\sum_{n=0}^{\infty} \frac{(-qt)^n}{n!}\mathbf{I}^n\right) \\
&= \left(\sum_{n=0}^{\infty} \frac{(qt)^n}{n!}\boldsymbol{\pi}_0\mathbf{P}^n\right) \cdot \left(e^{-qt}\mathbf{I}\right) \\
\boldsymbol{\pi}_t &= \sum_{n=0}^{\infty} e^{-qt}\frac{(qt)^n}{n!}\ \mathbf{p}_n
\end{aligned}$$

Thus we have

Property 18.2 *Transient analysis using uniformization*

$$\boldsymbol{\pi}_t = \sum_{n=0}^{\infty} e^{-qt}\frac{(qt)^n}{n!}\ \mathbf{p}_n$$

In the above equation:

- $\mathbf{p}_n$ is the distribution of the uniformized DTMC at time n using an initial distribution of $\boldsymbol{\pi}_0$.
- $e^{-qt}\frac{(qt)^n}{n!}$ is the PDF of *Poisson*: if $Y_{qt} \sim \textit{Poisson}(qt)$, then

$$\Pr\{Y_{qt} = n\} = e^{-qt}\frac{(qt)^n}{n!}$$

Why do we obtain the Poisson distribution?

Property 18.3 *Let $X_1, X_2, \ldots$ be an iid sequence of Expo(q) random variables. Let (random variable) N be the largest n such that*

$$X_1 + X_2 + \cdots + X_n < t$$

Then $N \sim \textit{Poisson}(qt)$.

For a proof of Property 18.3, see for example *Discrete–Event Simulation: A First Course* by Leemis and Park.

The intuitive meaning of Property 18.2 is therefore:

$$\begin{aligned}
\boldsymbol{\pi}_t &= \sum_{n=0}^{\infty} e^{-qt}\frac{(qt)^n}{n!}\ \mathbf{p}_n \\
&= \sum_{n=0}^{\infty} \Pr\{Y_{qt} = n\} \cdot \mathbf{p}_n \\
&= \sum_{n=0}^{\infty} \Pr\{\text{The CTMC changes state exactly } n \text{ times before time } t\} \cdot \mathbf{p}_n
\end{aligned}$$

How can we compute $\boldsymbol{\pi}_t$ in practice, using Property 18.2?

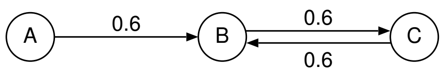
1. Decide the uniformization parameter q.

2. Compute the Poisson distribution for parameter qt. There is an efficient algorithm to do this with high accuracy, due to Fox and Glynn[1]. The algorithm gives left and right "truncation points", l and r, so that
$$\Pr\{l \le Y_{qt} \le r\} \ge 1 - \epsilon$$
where $Y_{qt} \sim \mathit{Poisson}(qt)$ and ϵ is a user–desired precision. The algorithm also computes $\Pr\{Y_{qt} = n\}$ for all $l \le n \le r$.

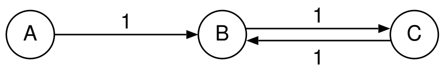
3. Compute
$$\boldsymbol{\pi}_t \approx \sum_{n=l}^{r} \Pr\{Y_{qt} = n\} \cdot \mathbf{p}_n$$
and the resulting approximation to $\boldsymbol{\pi}_t$ will have elements that sum to at least $1 - \epsilon$.

Example 18.6

For the following CTMC, if the CTMC is initially in state A, what is the probability distribution at time $t = 10$?



First, we produce a uniformized DTMC, using $q = 0.6$:



To compute $\boldsymbol{\pi}_{10}$ with $q = 0.6$, we need the PDF of $Y_{6.0} \sim \mathit{Poisson}(6.0)$ The table below shows how the approximation to $\boldsymbol{\pi}_{10}$ converges as the number of terms in the sum in Property 18.2 increases. Note, the column $\Pr\{Y_{6.0} > n\}$ gives an indication as to the achieved precision of our approximation to $\boldsymbol{\pi}_{10}$. The estimate of $\boldsymbol{\pi}_{10}$ for n is given by the estimate of $\boldsymbol{\pi}_{10}$ for $n - 1$, plus $\mathbf{p}_n$ multiplied by $\Pr\{Y_{6.0} = n\}$.

[1]B. L. Fox and P. W. Glynn. "Computing Poisson Probabilities", in *Communications of the ACM* 31 (4), April 1988, pages 440–445

n	$\mathbf{p}_n$	$\Pr\{Y_{6.0} = n\}$	$\Pr\{Y_{6.0} > n\}$	Estimate of $\boldsymbol{\pi}_{10}$
0	[1, 0, 0]	0.00247875	0.99752125	[0.00247875, 0.0 , 0.0]
1	[0, 1, 0]	0.01487251	0.98264873	[0.00247875, 0.01487251, 0.0]
2	[0, 0, 1]	0.04461754	0.93803120	[0.00247875, 0.01487251, 0.04461754]
3	[0, 1, 0]	0.08923508	0.84879612	[0.00247875, 0.10410759, 0.04461754]
4	[0, 0, 1]	0.13385262	0.71494350	[0.00247875, 0.10410759, 0.17847016]
5	[0, 1, 0]	0.16062314	0.55432036	[0.00247875, 0.26473073, 0.17847016]
6	[0, 0, 1]	0.16062314	0.39369722	[0.00247875, 0.26473073, 0.33909330]
7	[0, 1, 0]	0.13767698	0.25602024	[0.00247875, 0.40240771, 0.33909330]
8	[0, 0, 1]	0.10325773	0.15276251	[0.00247875, 0.40240771, 0.44235103]
9	[0, 1, 0]	0.06883849	0.08392402	[0.00247875, 0.47124620, 0.44235103]
10	[0, 0, 1]	0.04130309	0.04262092	[0.00247875, 0.47124620, 0.48365412]
11	[0, 1, 0]	0.02252896	0.02009196	[0.00247875, 0.49377516, 0.48365412]
12	[0, 0, 1]	0.01126448	0.00882748	[0.00247875, 0.49377516, 0.49491860]
13	[0, 1, 0]	0.00519899	0.00362849	[0.00247875, 0.49897415, 0.49491860]
14	[0, 0, 1]	0.00222814	0.00140035	[0.00247875, 0.49897415, 0.49714674]
15	[0, 1, 0]	0.00089126	0.00050910	[0.00247875, 0.49986541, 0.49714674]
16	[0, 0, 1]	0.00033422	0.00017488	[0.00247875, 0.49986541, 0.49748096]
17	[0, 1, 0]	0.00011796	0.00005692	[0.00247875, 0.49998337, 0.49748096]
18	[0, 0, 1]	0.00003932	0.00001760	[0.00247875, 0.49998337, 0.49752028]
19	[0, 1, 0]	0.00001242	0.00000518	[0.00247875, 0.49999578, 0.49752028]
20	[0, 0, 1]	0.00000373	0.00000146	[0.00247875, 0.49999578, 0.49752401]
21	[0, 1, 0]	0.00000106	0.00000039	[0.00247875, 0.49999685, 0.49752401]
22	[0, 0, 1]	0.00000029	0.00000010	[0.00247875, 0.49999685, 0.49752430]
23	[0, 1, 0]	0.00000008	0.00000002	[0.00247875, 0.49999692, 0.49752430]
24	[0, 0, 1]	0.00000002	0.00000001	[0.00247875, 0.49999692, 0.49752432]
25	[0, 1, 0]	0.00000000	0.00000000	[0.00247875, 0.49999693, 0.49752432]

18.2 Irreducible CTMCs

As with DTMCs, CTMC states can be classified into transient and recurrent. Recall, our earlier definition for $i \rightsquigarrow j$ was in terms of "probability of reaching a state". Therefore, all the discussion for DTMCs applies also to CTMCs. Namely, for a finite CTMC, we can determine recurrent classes and transient states by examining the strongly–connected components of the CTMC graph. As with DTMCs, we say a CTMC is *irreducible* if all pairs of states are mutually reachable (i.e., $\mathcal{S}$ is a recurrent class).

Recall that each recurrent class in a DTMC has a period, based on the possible return times for each state in the recurrent class. For a CTMC, though, if $i \rightsquigarrow j$ then $\Pr\{X(t) = j | X(i) = 0\} > 0$ for *all* positive t. That is because, for any path $i \to i_1 \to i_2 \to \cdots \to j$, the time to go from state i to state j is the sum of (independent) *Expo* random variables, and therefore the possible times are between 0 and infinity (of course, some of these times may be extremely improbable). Thus, the set of return times for any recurrent state is always equal to the interval $(0, \infty)$. As such, there is no notion of "period" for CTMCs.

Definition 18.4 *A CTMC is* ergodic *if it is irreducible.*

Suppose we have an irreducible CTMC. Can we determine

$$\boldsymbol{\pi} = \lim_{t\to\infty} \boldsymbol{\pi}_t$$

18.2.1 Argument 1

Using uniformization we can obtain a DTMC. Note that as time goes to infinity, the number of state changes of the CTMC must also go to infinity. So, the steady–state distribution of the uniformized DTMC is the same as the steady–state distribution of the original CTMC. Therefore, we have

$$\begin{aligned}
\boldsymbol{\pi}(\mathbf{P} - \mathbf{I}) &= \mathbf{0} \\
\boldsymbol{\pi}((\mathbf{Q}/q + \mathbf{I}) - \mathbf{I}) &= \mathbf{0} \\
\boldsymbol{\pi}(\mathbf{Q}/q) &= \mathbf{0} \\
\boldsymbol{\pi}\mathbf{Q} &= \mathbf{0}
\end{aligned}$$

18.2.2 Argument 2

The steady–state distribution does not change over time. Thus, in steady–state, we have

$$\frac{d}{dt}\boldsymbol{\pi}_t = \mathbf{0}$$

Plugging this into the differential equation we obtain

$$\begin{aligned}
\frac{d}{dt}\boldsymbol{\pi}_t &= \boldsymbol{\pi}_t\mathbf{Q} \\
\mathbf{0} &= \boldsymbol{\pi}_t\mathbf{Q}
\end{aligned}$$

18.2.3 Argument 3

In steady–state, the probability flow out of a state must equal the probability flow into a state. The probability flow out of state i is

$$\boldsymbol{\pi}[i] \cdot \sum_{j\neq i} \mathbf{Q}[i,j]$$

and the probability flow into state i is

$$\sum_{j\neq i} \boldsymbol{\pi}[j]\mathbf{Q}[j,i]$$

Setting these equal, we obtain

$$\begin{aligned}
\sum_{j\neq i} \boldsymbol{\pi}[j]\mathbf{Q}[j,i] &= \boldsymbol{\pi}[i] \cdot \sum_{j\neq i} \mathbf{Q}[i,j] \\
\sum_{j\neq i} \boldsymbol{\pi}[j]\mathbf{Q}[j,i] &= \boldsymbol{\pi}[i] \cdot -\mathbf{Q}[i,i] \\
\sum_{\forall j} \boldsymbol{\pi}[j]\mathbf{Q}[j,i] &= 0
\end{aligned}$$

which says that the dot product of $\boldsymbol{\pi}$ and column i of $\mathbf{Q}$ is 0. Since this holds for all columns, the above equations can be written as

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0}$$

As such, the above equation is sometimes called the "flow balance equations".

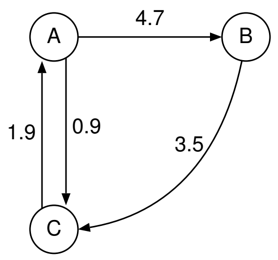
Property 18.5 *For an irreducible CTMC, the steady–state distribution is the unique probability vector $\boldsymbol{\pi}$ satisfying*

$$\boldsymbol{\pi}\mathbf{Q} \;=\; \mathbf{0}$$

As with DTMCs, note that the steady–state distribution of an ergodic CTMC does not depend on the initial distribution.

Example 18.7

What is the steady–state distribution for the following CTMC?



We can write the flow balance equations by looking at the CTMC (or, equivalently, write down the matrix $\mathbf{Q}$ and multiply out the equations):

$$\begin{aligned} 1.9 \cdot C &= 5.6 \cdot A \\ 4.7 \cdot A &= 3.5 \cdot B \\ 0.9 \cdot A + 3.5 \cdot B &= 1.9 \cdot C \end{aligned}$$

and of course we also require $A + B + C = 1$. The first equation gives us

$$C = \frac{5.6}{1.9}A$$

and the second equation gives us

$$B = \frac{4.7}{3.5}A$$

Note that plugging these into the third equation gives us $0 = 0$, so we instead use

$$\begin{aligned} A + B + C &= 1 \\ A + \frac{4.7}{3.5}A + \frac{5.6}{1.9}A &= 1 \\ A &= \frac{1}{1 + \frac{4.7}{3.5} + \frac{5.6}{1.9}} \approx 0.189028 \end{aligned}$$

Thus we obtain

$$\begin{aligned} C &= \frac{5.6}{1.9}A \approx 0.557135 \\ B &= \frac{4.7}{3.5}A \approx 0.253837 \end{aligned}$$

and therefore we have $\boldsymbol{\pi} = [\ 0.189028,\ 0.253837,\ 0.557135\ ]$.

18.3 Absorbing CTMCs

In an absorbing CTMC, as with the discrete–time case, we have that all states are either transient or absorbing:

- $\mathcal{Z}$ is the set of transient states
- $\mathcal{A}$ is the set of abosrbing states
- $\mathcal{S} = \mathcal{Z} \cup \mathcal{A}$

How can we compute the usual quantities:

1. The expected time spent in each transient state before absorption, as a vector $\boldsymbol{\sigma}$.
2. The probability of being absorbed into each absorbing state.

Note that when we organize the rows and columns of $\mathbf{Q}$ for an absorbing CTMC, we obtain the following block structure:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}[\mathcal{Z},\mathcal{Z}] & \mathbf{Q}[\mathcal{Z},\mathcal{A}] \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$$

Note that the absorbing states have no outgoing arcs, so the corresponding rows of $\mathbf{Q}$ are entirely 0. Compare this to the DTMC case.

18.3.1 Time spent in each transient state

We can obtain the expected number of visits to each transient state in the uniformized CTMC by studying the uniformized DTMC:

$$\mathbf{n}(\mathbf{P}[\mathcal{Z},\mathcal{Z}] - \mathbf{I}) = -\mathbf{p}_0[\mathcal{Z}]$$

But how does this relate to the expected *time* spent in each state? In the discrete case, the time spent in each state is $\sim$ *Const*(1). In the uniformized CTMC, the time spent in each state is $\sim$ *Expo*(q). Since the expected value of an *Expo*(q) random variable is $1/q$, we have

$$\boldsymbol{\sigma} = \mathbf{n}/q$$

Therefore, we can rewrite the above linear system as

$$\begin{aligned} \mathbf{n}(\mathbf{P}[\mathcal{Z},\mathcal{Z}] - \mathbf{I}) &= -\mathbf{p}_0[\mathcal{Z}] = -\boldsymbol{\pi}_0[\mathcal{Z}] \\ q\boldsymbol{\sigma}(\mathbf{P}[\mathcal{Z},\mathcal{Z}] - \mathbf{I}) &= -\boldsymbol{\pi}_0[\mathcal{Z}] \\ q\boldsymbol{\sigma}((\mathbf{Q}[\mathcal{Z},\mathcal{Z}]/q + \mathbf{I}) - \mathbf{I}) &= -\boldsymbol{\pi}_0[\mathcal{Z}] \\ \boldsymbol{\sigma}\mathbf{Q}[\mathcal{Z},\mathcal{Z}] &= -\boldsymbol{\pi}_0[\mathcal{Z}] \end{aligned}$$

Property 18.6 *The vector* $\boldsymbol{\sigma}$*, which holds the expected time spent in each transient state, is the unique solution to*

$$\boldsymbol{\sigma}\mathbf{Q}[\mathcal{Z},\mathcal{Z}] \;=\; -\boldsymbol{\pi}_0[\mathcal{Z}]$$

where $\boldsymbol{\pi}_0$ *is the initial probability distribution.*

Property 18.7

$$\boldsymbol{\sigma} \;=\; \int_0^\infty \boldsymbol{\pi}_t[\mathcal{Z}]\; dt$$

where $\boldsymbol{\pi}_t$ *is the probability distribution at time* t*. Proof:*

$$\begin{aligned}
\int_0^\infty \boldsymbol{\pi}_t[\mathcal{Z}]\; dt &= \int_0^\infty \boldsymbol{\pi}_0[\mathcal{Z}]e^{\mathbf{Q}[\mathcal{Z},\mathcal{Z}]t}\; dt \\
&= \left[\boldsymbol{\pi}_0[\mathcal{Z}]e^{\mathbf{Q}[\mathcal{Z},\mathcal{Z}]t}\mathbf{Q}[\mathcal{Z},\mathcal{Z}]^{-1}\right]_0^\infty \\
&= \left[\boldsymbol{\pi}_t[\mathcal{Z}]\mathbf{Q}[\mathcal{Z},\mathcal{Z}]^{-1}\right]_0^\infty \;=\; (\boldsymbol{\pi}_\infty[\mathcal{Z}] - \boldsymbol{\pi}_0[\mathcal{Z}])\mathbf{Q}[\mathcal{Z},\mathcal{Z}]^{-1} \\
&= -\boldsymbol{\pi}_0[\mathcal{Z}]\mathbf{Q}[\mathcal{Z},\mathcal{Z}]^{-1} \;=\; \boldsymbol{\sigma}
\end{aligned}$$

18.3.2 Probability of reaching each absorbing state

From an earlier discussion, recall that the limiting distribution of the uniformized DTMC is the same as the limiting distribution for the original CTMC. Thus, the probability of reaching each absorbing state is

$$\mathbf{p}_\infty[\mathcal{A}] \;=\; \mathbf{n}\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{p}_0[\mathcal{A}]$$

We can rewrite this in terms of the matrix $\mathbf{Q}$ using the uniformization equation:

$$\begin{aligned}
\mathbf{p}_\infty[\mathcal{A}] &= \mathbf{n}\mathbf{P}[\mathcal{Z},\mathcal{A}] + \mathbf{p}_0[\mathcal{A}] \\
&= \mathbf{n}(\mathbf{Q}/q + \mathbf{I})[\mathcal{Z},\mathcal{A}] + \mathbf{p}_0[\mathcal{A}] \\
&= \mathbf{n}\mathbf{Q}[\mathcal{Z},\mathcal{A}]/q + \mathbf{p}_0[\mathcal{A}] \\
\boldsymbol{\pi}_\infty[\mathcal{A}] &= \boldsymbol{\sigma}\mathbf{Q}[\mathcal{Z},\mathcal{A}] + \boldsymbol{\pi}_0[\mathcal{A}]
\end{aligned}$$

And of course we have $\boldsymbol{\pi}_\infty[\mathcal{Z}] = \mathbf{0}$.

18.4 Reducible CTMCs and measures

The discussion in a previous lecture about how to analyze reducible DTMCs applies equally well to reducible CTMCs, except with CTMCs we do not have to worry about the period for each recurrent class. Thus, given a reducible CTMC and its initial distribution, we can *always* find the limiting distribution: first find the probability of reaching each recurrent class (using the equations for absorbing CTMCs), then find the steady–state distribution for each recurrent class.

Similarly, the discussion about "reward measures" for DTMCs applies also to CTMCs. In this case, the rewards are accumulated based on the *amount of time* spent in each state, rather than the *number of visits* to each state[2]. To express rewards in terms of the number of visits to each state of a CTMC, we must first convert the CTMC into a DTMC, but rather than using uniformization to do this, we must use another technique. We will discuss this later.

[2]For a DTMC, these are equal: the time spent in a state is equal to the number of visits to the state, since the time per visit is always 1.

Chapter 19

Stochastic Petri Nets

So far, the Petri nets we discussed can be used for reachability questions. What about for measuring performance? We need to somehow assign stochastic behavior to the Petri net. We call this a "stochastic Petri net", or SPN.

19.1 Extending Petri nets to include stochastic behavior

Stochastic behavior is usually associated with transition firings. Since several transitions may be enabled in a given marking, we must specify how to choose between them, how time elapses, etc. This is usually done by specifying the following information.

19.1.1 Pre-selection priorities

If multiple transitions are enabled simultaneously, these priorities specify which ones may fire, or probabilities of doing so.

For example, we could assign an integer priority to each transition, which says that if transition t is enabled in a given marking $\mathbf{m}$, it cannot fire if another transition with higher priority[1] is also enabled in marking $\mathbf{m}$.

19.1.2 Firing distributions

A distribution is assigned to each transition, which specifies the random amount of time required to fire the transition. In general, we can allow the distribution to depend on the current marking. If more than one transition is enabled, then normally a *race semantic* is assumed, meaning that whichever transition has the shortest firing time will fire first, and will fire after that much time has elapsed. Conceptually, this says that each transition has its own internal clock, which is set to a random time (according to its distribution) when it becomes enabled, and counts down until it reaches zero, at which point the transition fires. (More about this later.) We must address several important questions:

1. What happens to the remaining time of a transition, if it becomes disabled due to another transition firing?

[1]As usual, there is no agreement if greater or lesser integers represent higher priority.

This is addressed by setting the appropriate policy, per transition, possibly depending on the current marking:

Resample: The next time the transition is enabled, throw away the previous remaining firing time and "start over" using a newly sampled firing time. This policy is useful for modeling a true "race" between activities, where the losing activity is discarded.

Resume: The next time the transition is enabled, "continue" with whatever is leftover from the previous firing time. This policy is useful for modeling service interruptions, where an interruption does not increase the time for an activity. Example: the activity is "drilling a hole through a sheet of metal", and an interruption is "loss of electricity to the drill".

Restart: Like "Resample", except the previously–sampled time is used again. Useful for service interruptions where the service must "start over" if it does not complete successfully. (This policy usually requires simulation, rather than numerical analysis.)

2. What if a transition becomes enabled a second time? Do we use a second clock for the transition, and take the minumum of the two clocks as the next firing time (the two enablings proceed in parallel), or do we wait until after the transition fires to consider the second enabling (the second enabling is added to a queue)?

19.1.3 Post-selection priorities

If we have discrete firing time distributions, and we use a race semantic, then it is possible that two enabled transitions will have the same firing time. What happens in this case? (Note this issue goes away if we have only continuous firing time distributions, since the probability of equal firing times will be zero.)

Again, we can set priorities or probabilities to resolve this issue. Note that, since the transitions may be in conflict (the winner will cause the loser to become disabled), the firing time for the "losing" transition will be determined by the resample/resume/restart policy.

19.1.4 Analysis

We can always use discrete–event simulation to analyze a stochastic Petri net, regardless of our choice in firing distributions, priorities, and disabling policies. Numerical analysis is also possible in certain cases. Specifically, the type of stochastic process described by the stochastic Petri net will depend on the firing distributions, priorities, disabling policies, and the model itself. We will start with a simple case, and consider cases with increasing complexity later.

19.2 Exponential firing times

Suppose we have a Petri net where the firing time distributions are all *Expo*. (To be confusing, some people use this as the definition of "stochastic Petri net".) This simplifies things considerably.

1. If $X \sim \mathit{Expo}(\lambda)$, it can be shown that

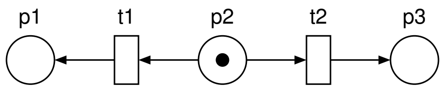
$$\Pr\{X < t + h | X > h\} \;=\; \Pr\{X < t\}$$

which implies that, if the transition has not fired by time h, then its remaining firing time has distribution *Expo*(λ). This says that the policies "resample" and "resume" are equivalent!

2. Because the minimum of two *Expo*(λ) firing times has distribution *Expo*(2λ), if we want to have a "separate clock" for each enabling of the transition, we can simply use a marking–dependent firing time.

3. No simultaneous firings are possible.

Consider the following tiny SPN:



t_1	has firing distribution	*Expo*(λ_1)
t_2	has firing distribution	*Expo*(λ_2)

From the specified initial marking, the Petri net

will fire t_1 first if *Expo*$(\lambda_1) <$ *Expo*(λ_2), putting a token in place p_1
will fire t_2 first if *Expo*$(\lambda_2) <$ *Expo*(λ_1), putting a token in place p_3

But this behavior is the same as the following CTMC:

$$\begin{tikzcd}100 & 010 \arrow[l, "\lambda 1"'] \arrow[r, "\lambda 2"] & 001\end{tikzcd}$$

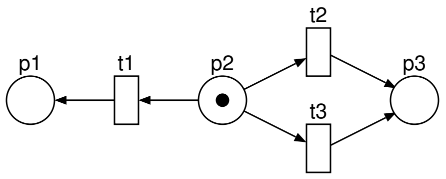
Therefore, if all firing distributions are *Expo*,

1. the underlying stochastic process is a CTMC.

2. the CTMC is equivalent to the reachability graph, where transition names are replaced by transition firing rates (the *Expo* parameter).

19.2.1 Multiple transitions between markings

What about multiple edges between the same pair of markings in the reachability graph? Consider the following SPN:



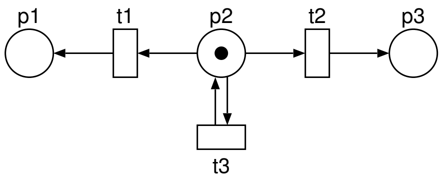
t_1	has firing distribution	*Expo*(λ_1)
t_2	has firing distribution	*Expo*(λ_2)
t_3	has firing distribution	*Expo*(λ_3)

Due to the race semantics, transitions t_2 and t_3 can be merged into a single transition, with firing distribution $\min(\textit{Expo}(\lambda_2), \textit{Expo}(\lambda_3)) \equiv \textit{Expo}(\lambda_2 + \lambda_3)$.

So, if the reachability graph has a single edge labeled with several transitions, and the firing distributions are all *Expo*, the corresponding edge in the underlying CTMC is obtained by summing the rates of those transitions.

19.2.2 Self loops

What about self loops within the reachability graph? Consider the following SPN:



Thanks to the memoryless property of the *Expo* distribution, firing t_3 and causing t_1 and t_2 to resample their firing times is statistically equivalent to not firing t_3 and having t_1 and t_2 continue their firing times. As such, the above SPN has the same behavior as the following CTMC:

$$\begin{tikzcd} 100 & 010 \arrow[l, "\lambda 1"'] \arrow[r, "\lambda 2"] & 001 \end{tikzcd}$$

Therefore, self loops in the reachability graph may be discarded in the CTMC.

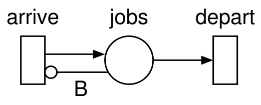
19.2.3 Measures

Simple reward measures can be defined in terms of the Petri net structure. For example, we can ask "what is the steady-state expected number of tokens in place p", or "what is the probability that transition t is enabled at time 15". The first question can be answered by constructing the underlying Markov chain, determining the steady-state distribution, and using a reward function where, for a given state of the Markov chain, the reward value is given by the number of tokens in place p. Similarly, the second question can be answered by constructing the underlying Markov chain, determining the distribution at time 15, and using a reward function where the reward value is 1 iff transition t is enabled in the given state.

19.2.4 Examples

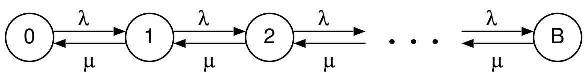
Example 19.1

Consider a bounded queue, containing at most B customers, with one server. Assume the time between arrivals is $Expo(\lambda)$, and the service time for each customer is $Expo(\mu)$. This can be modeled using the following simple Petri net:



Transition	**Firing distribution**
arrive	$Expo(\lambda)$
depart	$Expo(\mu)$

This stochastic Petri net describes the following CTMC:



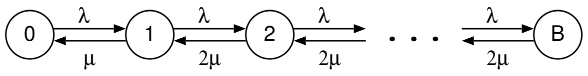
We can ask, "how many customers are in the system, on average, in steady state", using a reward function of "number of tokens in place *jobs*".

Example 19.2

Suppose, like the previous example, we have a bounded queue, except there are *two* servers, and the service time is the same regardless of the server used. This can be modeled using the same Petri net, with different firing distributions:

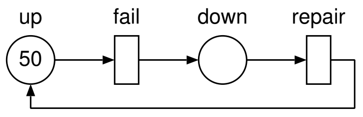
Transition	**Firing distribution**	
arrive	*Expo*(λ)	
depart	*Expo*(μ)	if $\#jobs < 2$
	Expo(2μ)	otherwise

This stochastic Petri net describes the following CTMC:



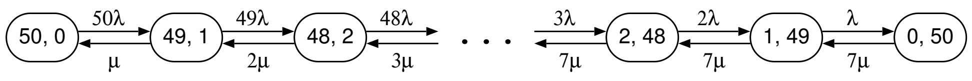
Example 19.3

Suppose we have 50 machines, each of which fails after *Expo*(λ) time. Failed machines are repaired by one of 7 technicians, which requires *Expo*(μ) time. We can model this using the following simple Petri net:



Transition	**Firing distribution**
fail	*Expo*$(\#up \cdot \lambda)$
repair	*Expo*$(\min(7, \#down) \cdot \mu)$

This stochastic Petri net describes the following CTMC:



To obtain the expected number of working machines in steady-state, we use the reward function "number of tokens in place *up*". In the above CTMC, this would assign a reward of 50 to state $(50, 0)$, a reward of 49 to state $(49, 1)$, etc. Once the steady-state distribution $\boldsymbol{\pi}$ is determined for the CTMC, we can apply the reward function to obtain the desired measure.

19.3 SPNs in Smart

A stochastic Petri net can be defined in Smart similar to the definition of a Markov chain. For example, the "bounded queue" Petri net is given as follows.

```
real lambda := 1.5;
real     mu := 2.0;

pn bqn(int B) := {
  place jobs;
  trans arrive, depart;
  firing(arrive: expo(lambda), depart: expo(mu));
  arcs(arrive : jobs, jobs : depart);
  inhibit(jobs : arrive : B);
  real probempty := prob_ss(tk(jobs)==0);
  real probfull  := prob_ss(tk(jobs)==B);
  real avgqueue  := avg_ss(tk(jobs));
};

compute(warning_file("/dev/null"));

print("B":-6, "Pr{empty}":-15, "Pr{full}":-15, "Avg":-15, "\n");

for (int b in {10..100..10}) {
  print(b:-6, bqn(b).probempty:-15);
  print(bqn(b).probfull:-15, bqn(b).avgqueue:-15, "\n");
}
```

The measures specified are, "steady state probability of no customers", "steady state probability of being full", and "steady state average number of customers". Smart will print a warning message, since the initial marking is empty; we therefore instruct Smart to send all warnings to "/dev/null", with the statement

```
compute(warning_file("/dev/null"));
```

We use a `for` loop to print measures for various values of B. The ":-6" in the `print` statements causes Smart to left-justify the argument with 6 spaces, similar to `printf("%-6d", b);` in C. When executed in Smart, this input file produces the following output.

```
B     Pr{empty}      Pr{full}       Avg
10    0.261015       0.0147018      2.51506
20    0.25059        0.000795072    2.9501
30    0.250031       4.46829e-05    2.99593
40    0.249998       2.52989e-06    2.99988
50    0.249977       1.62495e-07    3.00113
60    0.249914       2.801e-08      3.00448
70    0.249743       1.81306e-08    3.01408
```

```
80    0.249343       1.5569e-08     3.03809
90    0.248515       1.38327e-08    3.09159
100   0.246985       1.24615e-08    3.19908
```

Example 19.4

The Petri net definition in Smart for the machine shop model is:

```
// M is number of machines
// T is number of technicians
pn machines(int M, int T) := {
  place up, down;
  trans fail, repair;
  init(up:M);
  firing(fail:expo(tk(up)*lambda), repair:expo(min(tk(down), T)*mu);
  arcs(
    up : fail, fail : down,
    down : repair, repair : up
  );
  real avgup := avg_ss(tk(up));
};
```