CO4211/CO7211

Discrete Event Systems

Lecture 1:
Introduction and Motivation

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Organisational matters

Lecturer Michael Hoffmannn (**mh55@le.ac.uk**)

Duration 23rd January – 11th May

Normal timetable

Monday	09:00 - 11:00	FJ SW SR 3
Thursday	12:00 - 14:00	ATT LG02

Organisational matters

Formative Assessment

Three courseworks.

Summative Assessment

Written exam (100%).

Type of Material

This is a theoretical module.

No programming is involved.

Coursework require solution of exercises, proofs, etc.

Organisational matters

- Ask questions!
- If you don’t tell me that something is wrong I don’t know.
- Participate!

What is a surgery?

- Every week you will get a list of problems to solve.
- You should attempt to solve them at home before arriving to surgery.
- In surgery problems will be discussed.
- If you have no questions this means everything is clear and surgery will finish early.
- Questions are to be solved **before** arriving to surgery!

On to Motivation!

Most Complex Human Produced Artefacts ...

Complexity

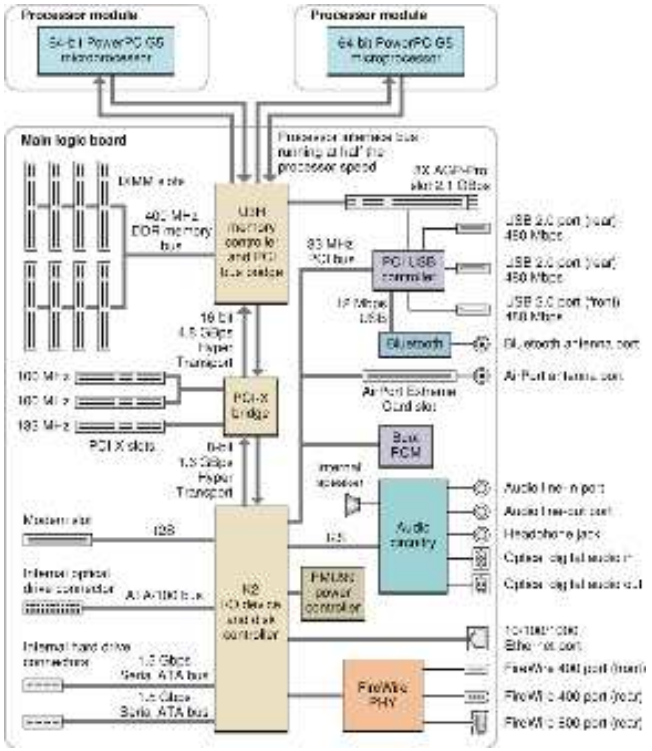
Computer hardware and software are amongst the most complex artefacts produced by humans.

- How do we handle complexity?
- Break to small parts.
- Apply abstraction.
- Construct models.

Computer hardware and software

Typically: Different *components:*

- processor;
- graphics unit;
- storage devices;
- main memory . . .



. . . communicating via *digital signals* . . .

- read certain block from disk;
- draw a line on the screen;
- interrupts . . .

Communication between different components is typically *event driven*.

Manufacturing Processes

Manufacturing Processes

- ordering of raw material;
- storage;
- production;
- quality control; . . .

Often event driven:

- *place an order* on low stock levels;
- *check quality* after production;
- *return unit to production* if it is faulty;
- *put unit into warehouse* if it is OK; . . .

So *events* occur throughout the production process and drive its progress.

Consumer electronics

Consumer electronics

Typical components are . . .

- sensors (temperature, water);
- users of the system;
- processors . . .

Different components interact by sending *signals* . . .

- temperature too low;
- start the machine;
- turn water on . . .

The *control* of the system decides on its actions depending on the signals.

Informal definition

We note the following characteristic properties of systems:

- A *discrete event system* consists of a number of *components* that interact to achieve the desired functionality.
- The *components* of a discrete system communicate by creating and responding to messages, called *events*
- The overall system embodies a notion of *state*, which is *discrete*. The *state transition mechanism* is *event driven*.

Examples of systems with discrete state spaces

- State of a machine: $\{$ waiting for coins, making coffee, idle, down, $\ldots\}$.
- Programme running on a computer: $\{$ waiting for input, processing, terminated, $\ldots\}$.
- Warehouse: $\{$ 0 items, 1 item, $\ldots\}$.

Formal vs informal

Informal definition. A *model* of a discrete event system is a formal description of selected aspects of the system consisting of the *data* that represents the chosen aspect and the *methods* that manipulate the data.

We will look at things more formally soon . . .

- every *particular notion of a model* has a *formal definition*.
- chosen aspect is guided by *abstraction:* “Make everything as simple as possible, but not simpler”.

This module

We will study four different *models* of DES, which are defined formally, along with their properties.

The emphasis is on the *modelling* and *analysis* of systems.

1. **State machines**: events occur sequentially.
2. **Petri nets**: events can occur concurrently.
3. **Markov chains**: events occur with a given probability;
4. **Queues**: events wait for resources to be available.

Typical questions

System properties Does the system behave according to its specification?

Building and testing the real system is expensive, but models allow us to:

- run *simulations* without actually building the real system;
- *prove* (mathematically, automated or otherwise) that the system has a desired property.

Typical questions (continued)

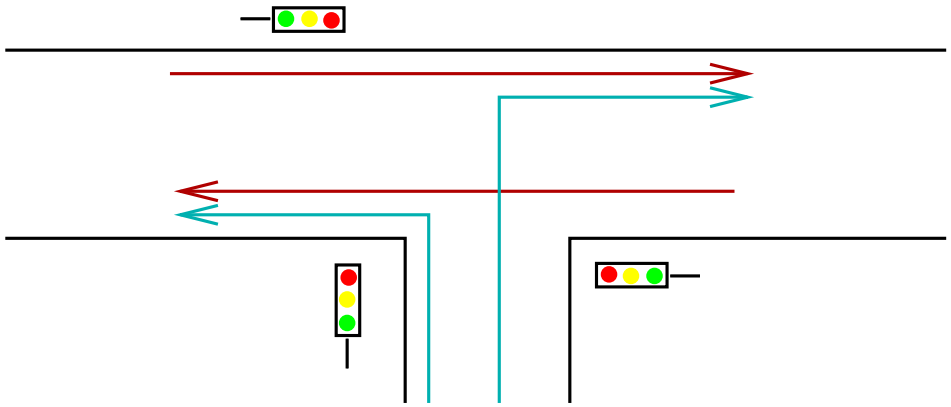
System safety is one of the most important properties.

- Can a user crash the system so that it enters an undesirable state?
- Is it possible that the system hangs, e.g. it doesn't respond to user interaction?

System performance. Will the system satisfy our minimal performance requirements?

- Will it be ready to serve requests at least 95% of the time?
- How many requests will typically have to queue to be dealt with later?

Example: Controlling traffic flow



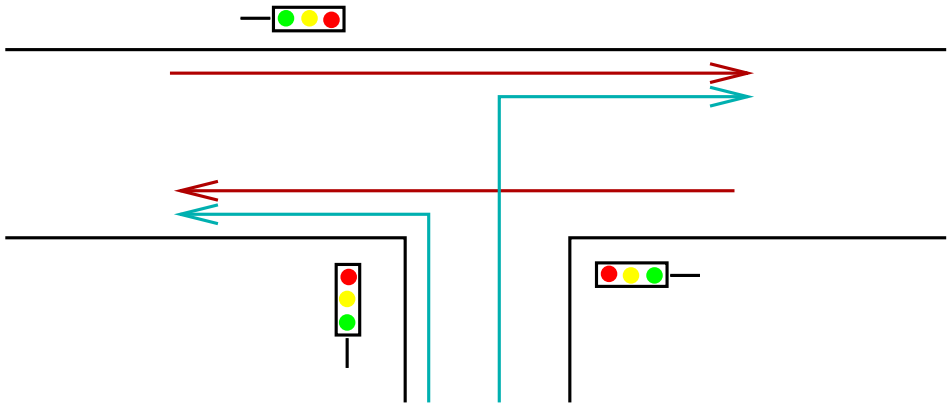
Modelling assumption. Vehicles only move in the directions indicated.

Goal of the system: safety and short queues.

The specification exhibits some typical features:

- *informal:* what is “safety” and when is a queue “short”?
- does not explicitly mention available *resources* and *information*.

Traffic flow: events



Available resources: In this case the *events* which the system can react to.

Modelling assumption. There are sensors that inform us:

- about the arrival of a car and the direction into which it intends to move;
- when a car has cleared the junction.

Traffic flow: events

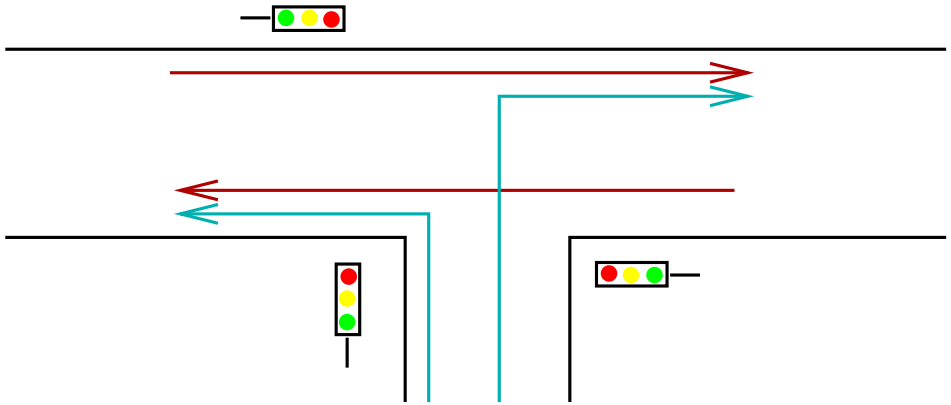
Notation for writing down the set of events:

- Let A_{se} indicate arrival from the south side of the junction, travelling eastwards.
- and similarly for the other directions; replace A by D for departure.

One possible **event set**:

$$\mathrm{E} = \{\mathrm{A}_{ew}, \mathrm{A}_{we}, \mathrm{A}_{se}, \mathrm{A}_{sw}, \mathrm{D}_{ew}, \mathrm{D}_{we}, \mathrm{D}_{se}, \mathrm{D}_{sw}\}.$$

Traffic flow: events



Is this a good starting point? Some possible shortcomings are:

- There is no event for “red” or “green”.
- We have no information about *when* an event occurs.

We can’t model everything!

The *model*, and therefore the choice of *events*, depends on the chosen *aspects* of the situation that we wish to scrutinise.

Traffic flow: states

State space of the model: number of waiting cars and state of the lights.

$$\begin{aligned} S &= \{(x_{se}, x_{sw}, x_{ew}, x_{we}, y_1, y_2) : \\ &\qquad x_{se}, x_{sw}, x_{ew}, x_{we} \geqslant 0,\ y_1, y_2 \in \{R, G\}\}; \end{aligned}$$

- y_1 is the light signal for vehicles on the east-west road;
- y_2 is the light signal for vehicles arriving from the south.

Many other choices are possible, depending on the chosen aspects!

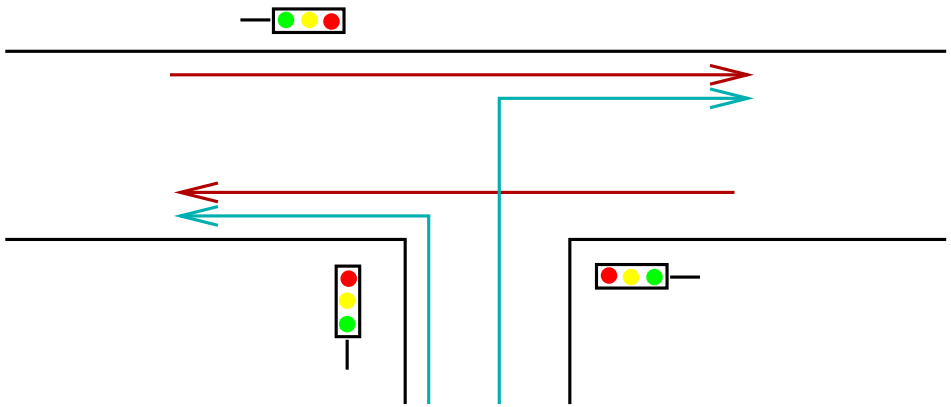
Traffic flow: requirements

Possible requirements

- The system should not allow cars from both directions to proceed simultaneously;
- everybody gets their turn (eventually? after at most five minutes?);
- what is the typical waiting time?

(Some of these questions cannot be answered unless we include time in the model.)

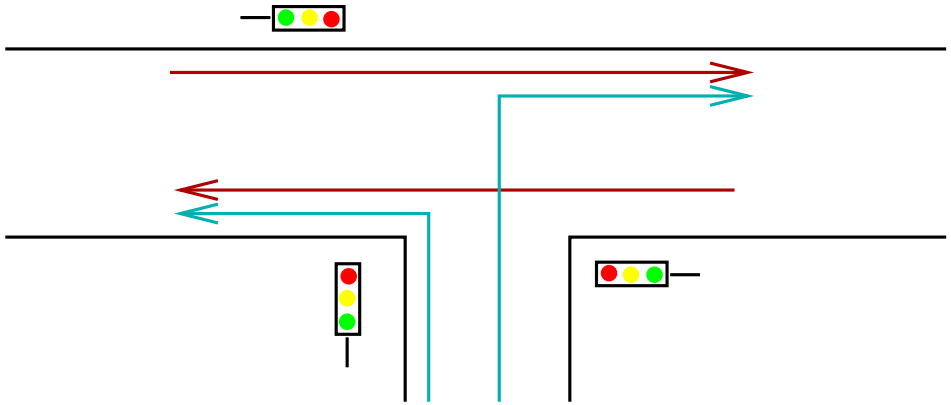
Traffic flow: possible strategies



Strategy 1. Extra safe: Allow east-west and west-east motion if there are no vehicles arriving from south. Allow south-east and south-west motion if there are no vehicles arriving from east or west.

Problem. Several cars together lead to all lights staying red forever and blocking the intersection. This is called *deadlock*.

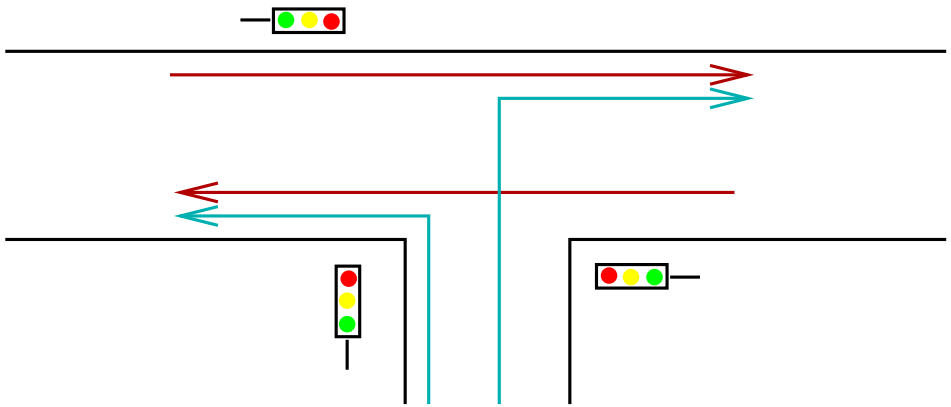
Traffic flow: possible strategies



Strategy 2. As long as there are cars waiting to travel either east-west or west-east, let them pass.

Problem. People arriving from the south have very long (possibly infinitely long) waits. This is called *starvation*.

Traffic flow: possible strategies



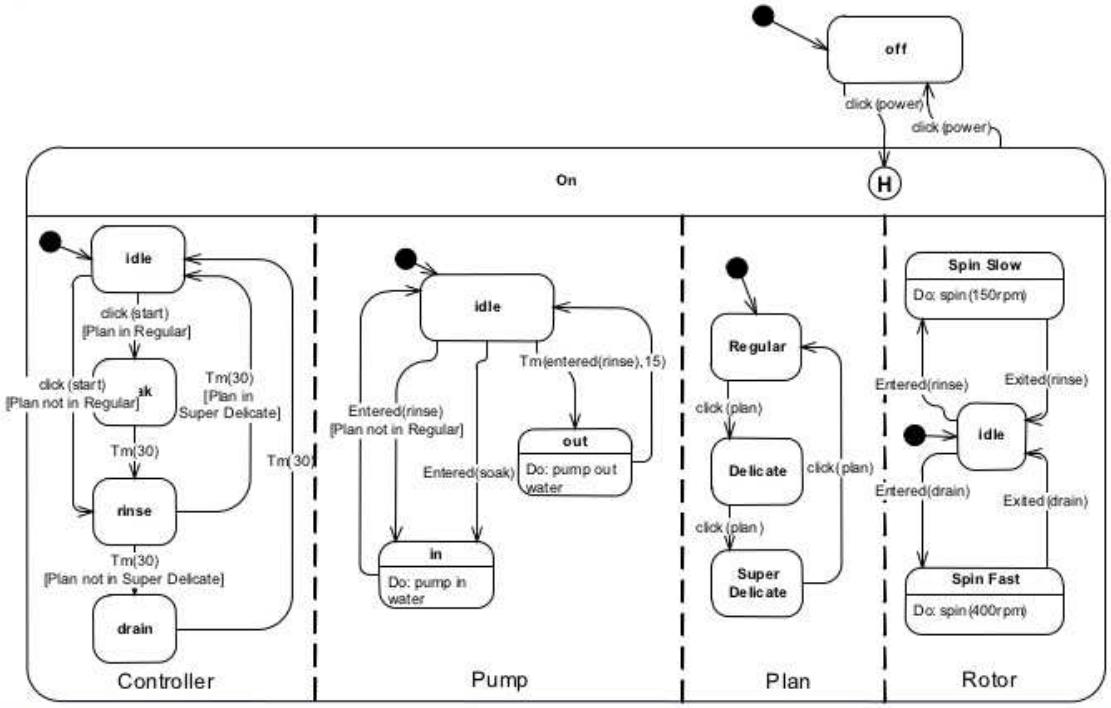
But what is a good strategy?

In due course, we will

- encounter possible ways of modelling different aspects of strategies;
- learn about the analysis of strategies expressed in different models.

Are you serious?

Interactions Between Parallel States



StateCharts Support

- Microsoft Internal Tool (USB3 drivers).
- MathWorks Simulink.
- Yakindu (open source).
- ...

Concluding note on abstraction.

"Del rigor en la ciencia", Jorge Luis Borges

En aquel Imperio, el Arte de la Cartografía logró tal Perfección que el Mapa de una sola Provincia ocupaba toda una Ciudad, y el Mapa del Imperio, toda una Provincia. Con el tiempo, estos Mapas Desmesurados no satisficieron y los Colegios de Cartógrafos levantaron un Mapa del Imperio, que tenía el Tamaño del Imperio y coincidía puntualmente con él. Menos Adictas al Estudio de la Cartografía, las Generaciones Siguientes entendieron que ese dilatado Mapa era Inútil y no sin Impiedad lo entregaron a las Inclemencias del Sol y los Inviernos. En los Desiertos del Oeste perduran despedazadas Ruinas del Mapa, habitadas por Animales y por Mendigos; en todo el País no hay otra reliquia de las Disciplinas Geográficas.
Suárez Miranda: Viajes de varones prudentes,
libro cuarto, cap. XLV, Lérida, 1658.

On Exactitude in Science **. . . In that Empire, the Art of Cartography attained such Perfection that the map of a single Province occupied the entirety of a City, and the map of the Empire, the entirety of a Province. In time, those Unconscionable Maps no longer satisfied, and the Cartographers Guilds struck a Map of the Empire whose size was that of the Empire, and which coincided point for point with it. The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast Map was Useless, and not without some Pitilessness was it, that they delivered it up to the Inclemencies of Sun and Winters. In the Deserts of the West, still today, there are Tattered Ruins of that Map, inhabited by Animals and Beggars; in all the Land there is no other Relic of the Disciplines of Geography.**

Suarez Miranda,Viajes de varones prudentes, Libro IV,Cap. XLV, Lerida, 1658

CO4211/CO7211

Discrete Event Systems

Lecture 2:
State Machines - Basics

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Example

Implementing a combination lock

Simplified model:

- lock has two buttons;
- you enter a four digit code;
- lock opens if the last two digits coincide.

Task: Implement the functionality.

Java implementation

Assume we have subclasses:

```
class Alarm extends Action { ... }
class Open  extends Action { ... }
```

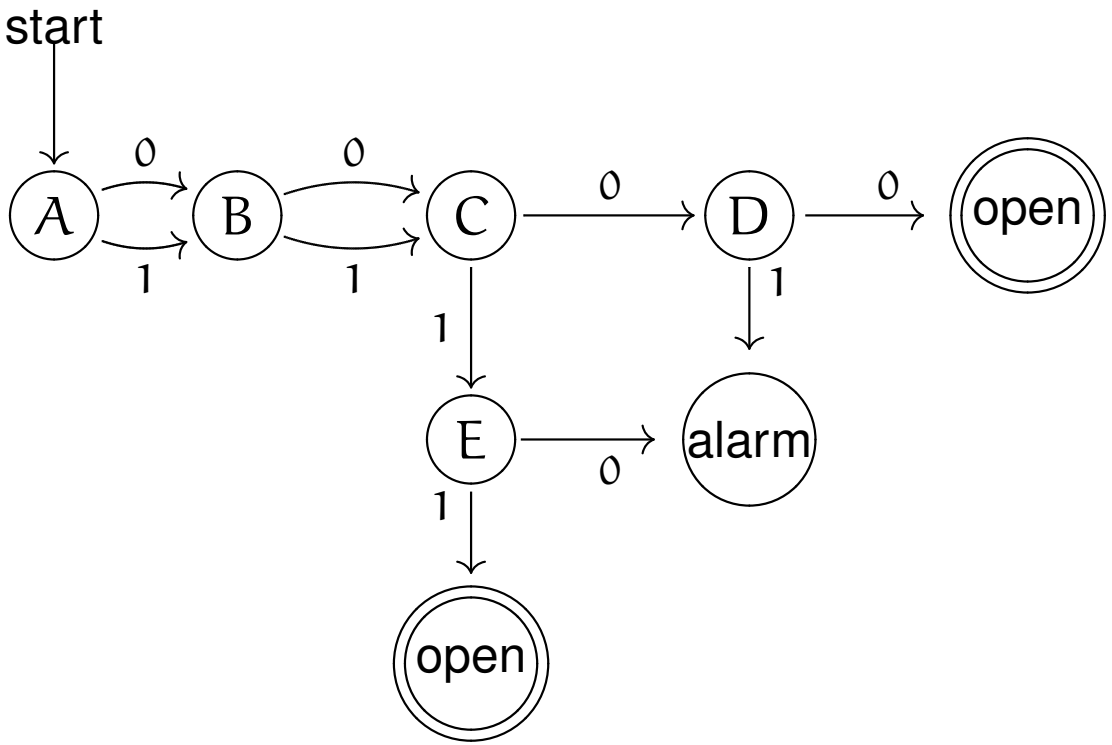
Java implementation of the combination lock:

```
Action open()
  int last = 0, current = 0;
  for (int count = 0; count < 4;
       count = count + 1) {
    last = current;
    getdigit (current);
  }
  if (last == current) return new Open()
  else return new Alarm();
}
```

Discussion of the Java implementation

- We don't need Java to design a combination lock (in fact, we don't even need a computer).
- We don't need integers (e.g. `int count`). Every integer introduces 2^{32} possibilities.
- We don't really need to do any arithmetic (`count = count + 1`).
- The control flow of Java obscures the simple essence of the lock's mechanism.

Combination lock – simplified



One could reasonably argue that both the nodes named “open” are really the same state.

Comparison with the Java implementation

The diagram is simpler . . .

- easier to draw the correct diagram than writing Java code;
- easier to understand the mechanism of the lock;
- just need two input tokens (0/1);

(In Java, one could use booleans)

. . . and much cleaner . . .

- no need for complicated Java syntax;
- smaller number of states; no states are redundant.

Alphabets and strings

We will now formulate the idea of “state machines”.

Combination lock example

- the inputs were binary digits;
- the lock processed a *sequence* of digits;
- there were “good” and “bad” sequences.

Alphabets and strings

Definitions

1. An *alphabet* (usually denoted by Σ) is a set of *symbols* (digits, characters, letters, ...).

2. A *string* (or a *word*) over an alphabet Σ is a finite sequence of symbols of Σ. The set of all words over Σ is denoted by Σ^*.

   - in particular, the *empty string*, denoted by ϵ, is a word (so that $\epsilon \in \Sigma^*$);
   - we denote the *length* of a string w (i.e. the number of symbols in w) by $|w|$; for example, $|\epsilon| = 0$ and $|\mathrm{rick}| = 4$.

Back to the combination lock

Questions

1. what is the alphabet Σ in the combination lock example?
2. what does the string ϵ signify in this example?
3. what are the “good” and the “bad” strings here?

Answers

Back to the combination lock

Questions

1. what is the alphabet Σ in the combination lock example?
2. what does the string ϵ signify in this example?
3. what are the "good" and the "bad" strings here?

Answers

1. $\Sigma = \{0, 1\}$;
2. ϵ signifies having pressed no button;
3. the "good" strings are those of length 4 with the same third and fourth digits; the remaining strings are "bad".

Finite automata: definition

A *finite automaton* (FA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where

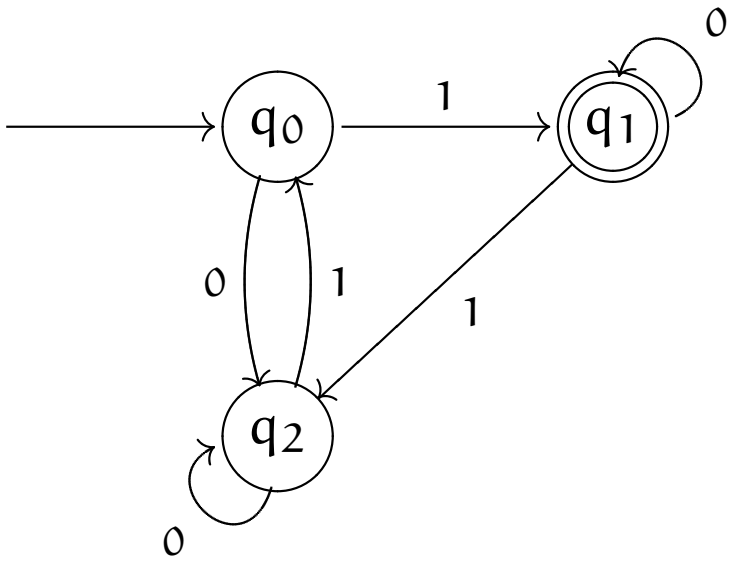
- Q is a finite set of *states*, the "state set" of the automaton;
- Σ is a finite set, the *alphabet* of the automaton;
- $\delta : Q \times \Sigma \to Q$ is the *transition function* of the automaton;
- $q_0 \in Q$ is the *initial state* or *start state*;
- $F \subseteq Q$ is the set of *accepting states* (or *final states*).

Remark. The notion of the actual input is *not* embodied in the definition of automaton.

Convenient notation. We write $p \xrightarrow{a} q$ if $\delta(p, a) = q$ (as in the graphical notation).

Finite automata - inputs

Automaton reading inputs:



- the source-less arrow denotes “start”;
- the input is read from left to right, taking appropriate transitions;
- a doubly-circled state denotes “OK”, “accept” or “good”.

Runs

Automaton **runs** on a string and reads it.

Definition. A run of A on string $w = a_0 a_1 \dots a_{n-1}$ is $r = q_0, q_1, \dots, q_n$ such that q_0 is the initial state and for every $0 \leq i < n$ we have $q_{i+1} = \delta(q_i, a_i)$.

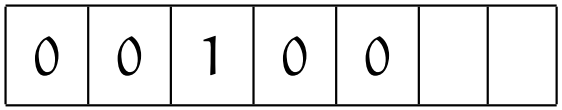
Informally, reading a string corresponds to a *path* in the graph of the automaton.

Definition. A run $r = q_0, q_1, \dots, q_n$ is accepting if $q_n \in F$.

Definition. A string w is accepted if the run of A on w is accepting.

Acceptance of strings

Input "tape": read one character at a time.



An automaton A *accepts* a string w if the path starting at q_0 which is labelled by w ends in an accepting state. We can express this algorithmically as follows.

1. put the automaton into state q_0;
2. if the input is empty, then go to step 5;
3. change state according to the first symbol of the input string;
4. chop the first symbol off the input string, and go to step 2;
5. if the current state is accepting, then accept; otherwise, reject.

Languages

A “language” is just a collection of strings. One can think of it as the set of “good words”.

Definition. A *language* over an alphabet Σ is a subset $L \subseteq \Sigma^*$ of the set of words over Σ.

Examples. Suppose that $\Sigma = \{0, 1\}$. The following are all languages over Σ:

- the set of all words $s \in \Sigma^*$ that have an even number of 1's;
- the set of all words $s \in \Sigma^*$ that contain the word 11011;
- the set of all words $s \in \Sigma^*$ that do not contain the word 11011.

Regular languages

Every finite automaton defines a language:

Definition. The *language accepted by a finite automaton* A, denoted by $L(A)$, is the set of all strings $w \in \Sigma^*$ such that A accepts w.

Does every language define an automaton?

Definition. A language $L \subseteq \Sigma^*$ is *regular*, if there exists an automaton A such that $L = L(A)$.

We will see later that not all languages are regular.

From languages to automata

Task. Given a language $\mathrm{L} \subseteq \Sigma^*$ (a set of "good" behaviours of a system or "correct" runs of a protocol), construct an automaton A that accepts L.

The "you are the automaton" method

- you will see the input string symbol by symbol;
- you don't know when it will end;
- you always need to be ready to answer "yes" or "no", i.e. to tell whether the string seen so far belongs to the language;
- you can change your strategy after reading each input symbol;
- keep it simple!

The “even” automaton

Question. If $\Sigma = \{0, 1\}$ and

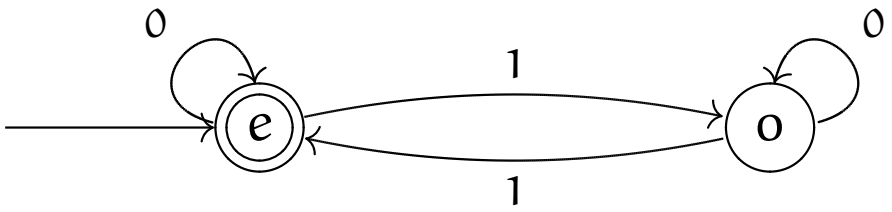
$$L = \{w \in \Sigma^* : \text{the number of } 1\text{'s in } w \text{ is even}\},$$

is there an automaton that accepts L?

Note. 0 is an even number (as $0 = 0 \times 2$), and so any string with no 1's should be accepted.

The “even” automaton

The automaton A below accepts the strings containing an even number of 1’s:



Idea:

state e is for “even”, state o is for “odd”; the automaton changes state whenever it reads a 1.

The “even” automaton

Formal definition

Let $A = (Q, \Sigma, \delta, q_0, F)$ where:

$$Q = \{e, o\}, \quad \Sigma = \{0, 1\},$$
$$q_0 = e, \qquad F = \{e\},$$

and $\delta : Q \times \Sigma \to Q$ is defined by

$$\delta(e, 0) = e, \qquad \delta(o, 0) = o,$$
$$\delta(e, 1) = o, \qquad \delta(o, 1) = e.$$

CO4211/CO7211

Discrete Event Systems

Lecture 3:
Regular Languages

Michael Hoffmann

University of Leicester

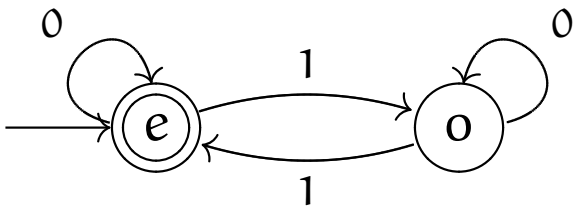
January 2018

Original slides: Rick Thomas and Nir Piterman

Finite automata and languages

Compare two notions:

- *finite automata*, such as . . .



- *languages* $\mathrm{L} \subseteq \Sigma^*$, such as . . .

$$\mathrm{L} = \{w \in \Sigma^* : s \text{ has an even number of 1's}\}.$$

Finite automata and languages

Questions

1. does every language define an automaton?
2. if not, what are the defining properties of those languages (*regular languages*) that do?

Why is this interesting?

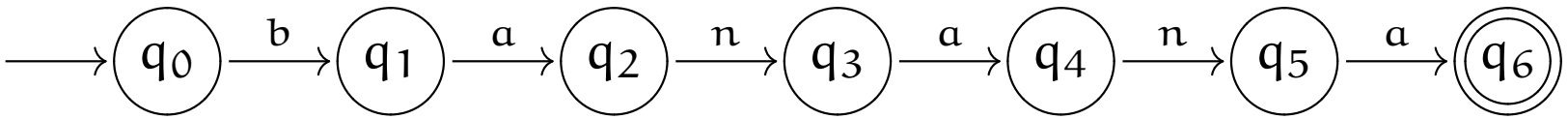
- Answering these questions will prevent us trying to construct automata for languages that are not regular!
- We will understand the modeling limits of the formalism and know when we need something else/stronger.

The easiest case

Let's start slowly, and look at *finite languages*.

Example. Is the language $L = \{banana\}$ regular (over $\Sigma = \{b, a, n\}$)?

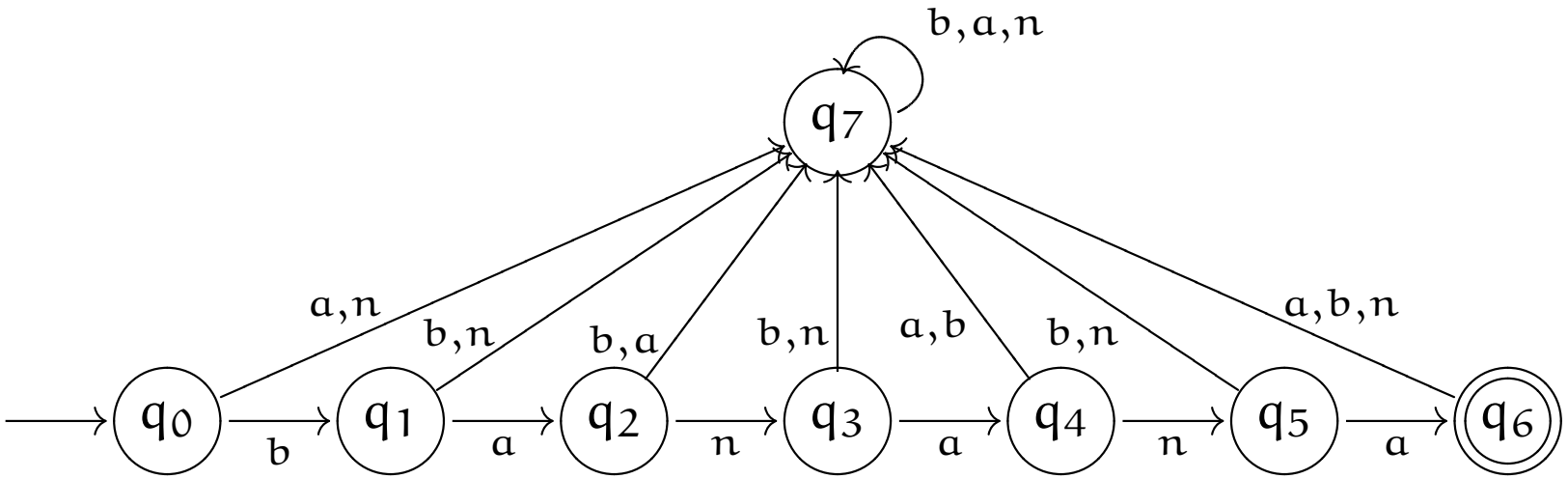
Answer. Of course it is, here is an automaton:



Hold on. There's a problem. What happens if A reads a while in state q_0? Can we fix this?

The banana automaton, repaired

Idea. Just add a new “fail state” that absorbs the “wrong” letters.



Constructing automata in this way we see that all languages containing just one word are regular.

How about finite languages in general?

Idea.

Given automata A accepting L and A' accepting L', can we run them "in parallel" and accept a string if either A or A' accepts that string?

Questions.

1. What are the states of the resulting "parallel automaton"?
2. How is the transition function defined in the new automaton?
3. When is a state in the new automaton initial or accepting?

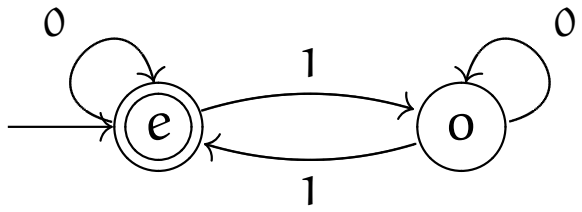
Automata in parallel

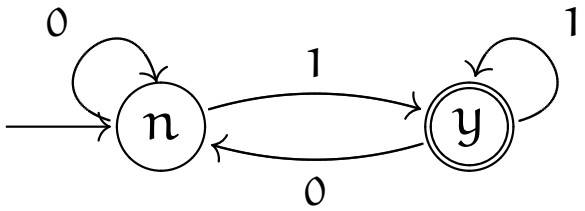
Answers.

1. States: We need to keep track of both the state of A and the state of A'; so the states of the new machine should be pairs (q, q') where q is a state of A and q' is a state of A';
2. Trans: When reading an input symbol, we have to account for both the action of A and the action of A', giving a new pair of states: $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $q' \xrightarrow{a} p'$;
3. Init/acc: The state (q, q') is accepting if either q or q' is accepting, and it is initial if both q and q' are initial.

Example

Let's run the following automata in parallel.

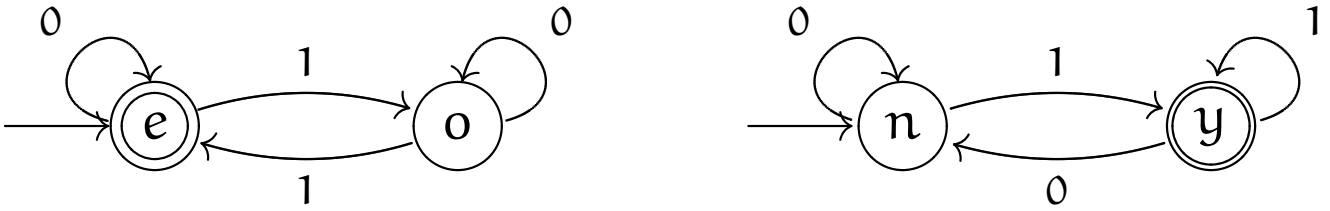




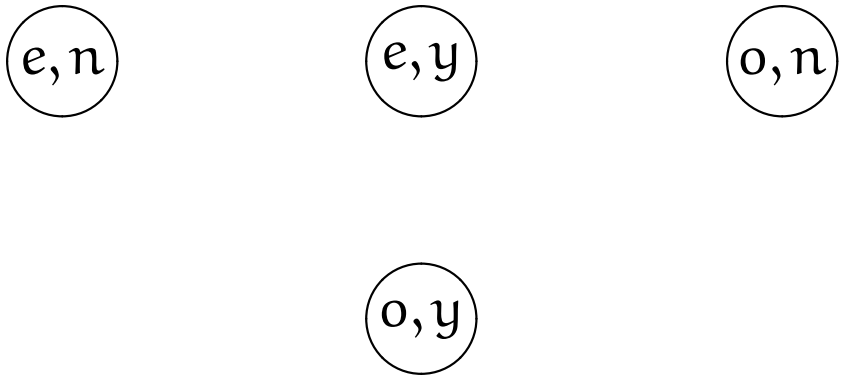
The language of the bottom automaton is $\{w \in \{0,1\}^* : w \text{ ends with a } 1\}$.

Example

Starting from the states ...

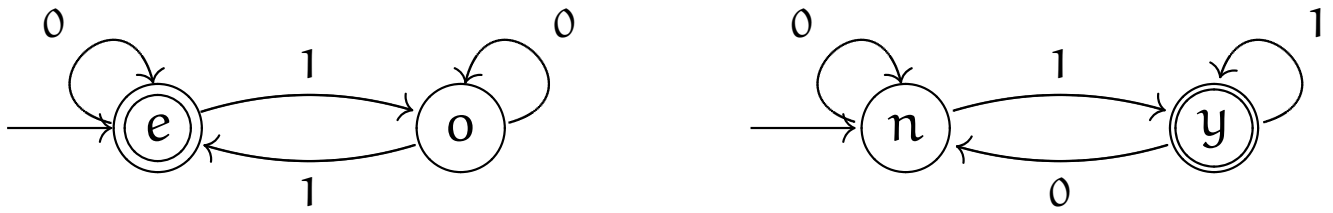


Step 1. The states of the resulting automaton are all pairs (x, y) with $x \in \{e, o\}$ and $y \in \{n, y\}$:

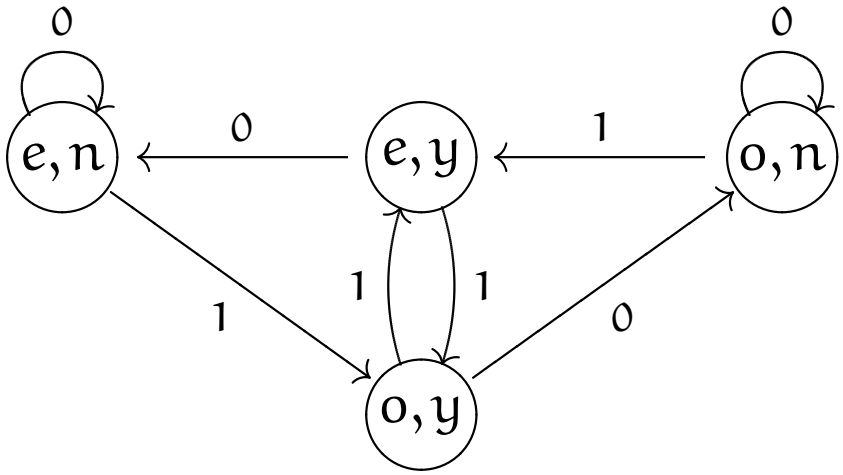


Example

Let's add transitions . . .

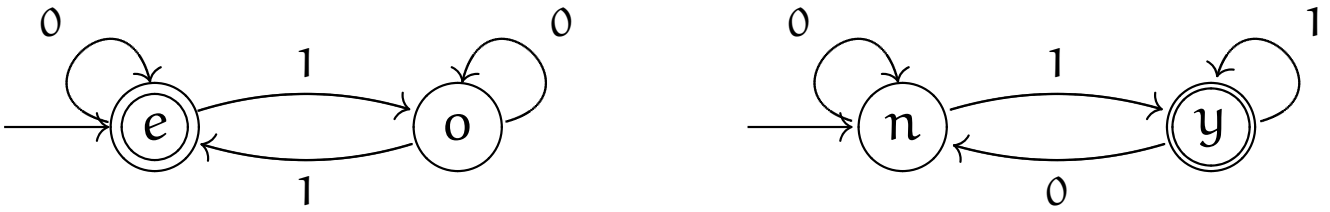


Step 2. We have $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $q' \xrightarrow{a} p'$.

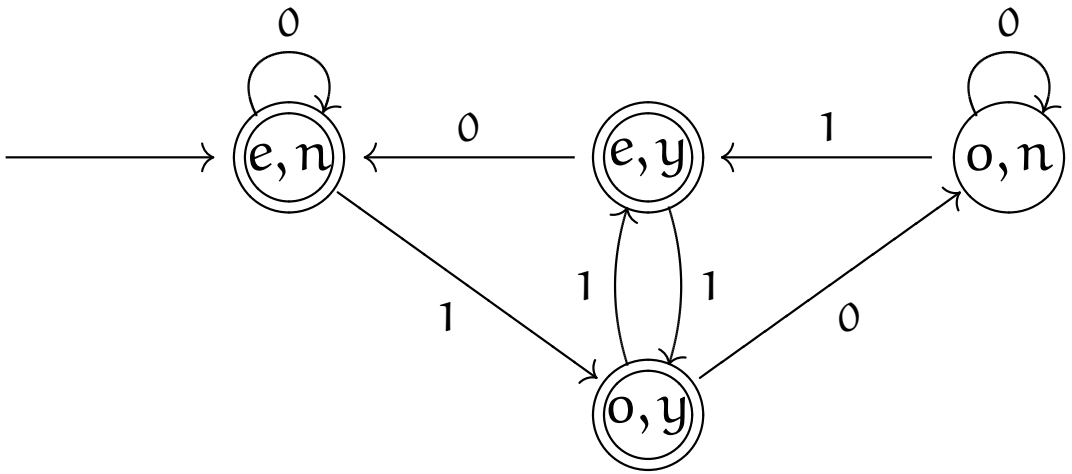


Example

Let's add accepting and initial states ...



Step 3. A state (q, q') is accepting, if q is accepting (for the first automaton) or q' is accepting (for the second), and initial if both q and q' are initial.



Union of regular languages

Theorem. If L and L' are regular languages over the alphabet Σ, then so is $L \cup L'$.

Informal idea. Given an automaton A for L and an automaton A' for L', construct the union automaton as we did above.

This is not really precise: an automaton is 5-tuple, not a drawing!

More formal idea. Suppose that the automaton $A = (Q, \Sigma, \delta, q_0, F)$ accepts L and that the automaton $A' = (Q', \Sigma, \delta', q_0', F')$ accepts L'. Then the automaton $A^{\cup} = (Q^{\cup}, \Sigma, \delta^{\cup}, q_0^{\cup}, F^{\cup})$ accepts $L \cup L'$ where:

- $Q^{\cup} = Q \times Q'$; $\quad q_0^{\cup} = (q_0, q_0')$;
- $F^{\cup} = \{(q, q') \in Q \times Q' : q \in F \text{ or } q' \in F'\}$;
- $\delta^{\cup}(q, q')(x) = (\delta(q, x), \delta'(q', x))$.

Finite versus infinite languages

Recall our question: which languages are regular? We now have . . .

- one element languages are regular;
- if two languages are regular, then so is their union.

Question. How about the empty language?

Answer. The empty language is accepted by every automaton with no accepting states.

So we have:

Corollary. All finite languages are regular.

Finite versus infinite languages

Question. Are all regular languages finite?

Answer. No. We have already seen some examples, e.g. the set of strings over $\Sigma = \{0, 1\}$ with an even number of 1's.

So we have to do some more work . . .

Properties of regular languages

Idea. Study constructions on automata:

- helps us to see which languages are regular;
- helps to mechanise constructions of automata.

We've already seen one example: the union of two regular languages is regular.

Question. How about complements? If L is regular, is $\Sigma^* - \mathrm{L}$ regular?

Equivalently: is there an automaton accepting precisely those words that $\mathcal{A}$ rejects?

Properties of regular languages

Answer. Yes there is; we just swap the accepting and the non-accepting states of A.

Theorem. If L is regular then $\Sigma^* - L$ is also regular.

Proof. Suppose $A = (Q, \Sigma, \delta, q_0, F)$ accepts L; then

$$A' = (Q, \Sigma, \delta, q_0, Q - F)$$

accepts $\Sigma^* - L$.

Intersection of regular languages

Idea. Given automata accepting L and L', can we run them in parallel and accept a word if *both* automata accept that word?

(This looks similar to computing the union.)

Construction of the intersection automaton of two automata A and A':

1. the *states* of the intersection automaton are the pairs (q, q') where q is a state of A and q' is a state of A';
2. the *transition function* is given by $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $q' \xrightarrow{a} p'$;
3. a state (p, q) is *accepting*, if *both* p and q are accepting (this is the difference between intersection and union!);
4. the *initial state* is the pair of initial states.

Intersection of regular languages

Theorem. If L and L' are regular languages over the alphabet Σ, then so is the language $L \cap L'$.

Formal construction.

Suppose that the automaton $A = (Q, \Sigma, \delta, q_0, F)$ accepts the language L and that the automaton $A' = (Q', \Sigma, \delta', q_0', F')$ accepts the language L'.

The automaton

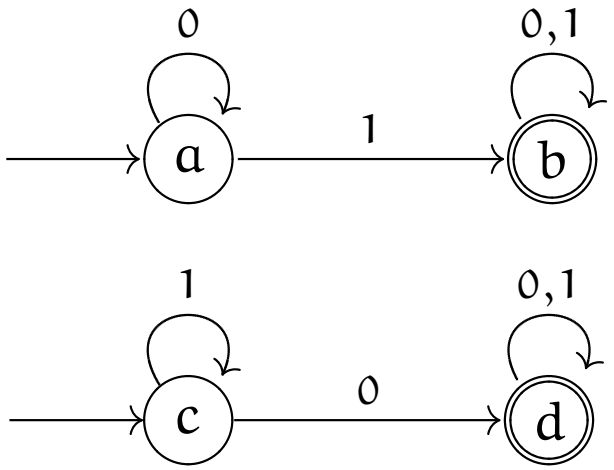
$$A^{\cap} = (Q^{\cap}, \Sigma, \delta^{\cap}, q_0^{\cap}, F^{\cap})$$

accepts the language $L \cap L'$ where:

- $Q^{\cap} = Q \times Q'; \quad q_0^{\cap} = (q_0, q_0');$
- $F^{\cap} = \{(q, q') \in Q \times Q' : q \in F \textit{ and } q' \in F'\};$
- $\delta^{\cap}(q, q')(x) = (\delta(q, x), \delta(q', x)).$

Example: intersection automaton

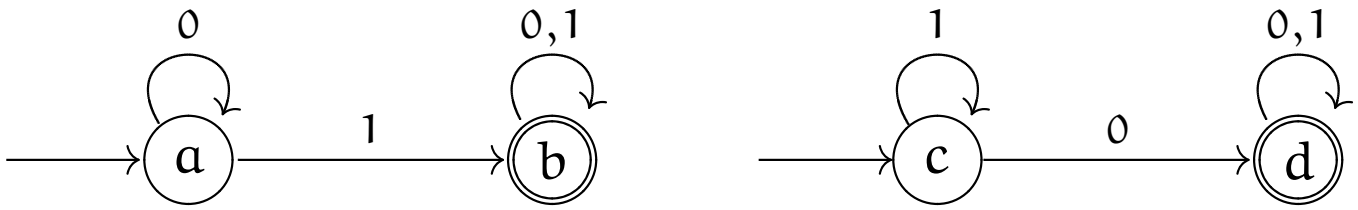
Let's compute the intersection automaton of the following automata:



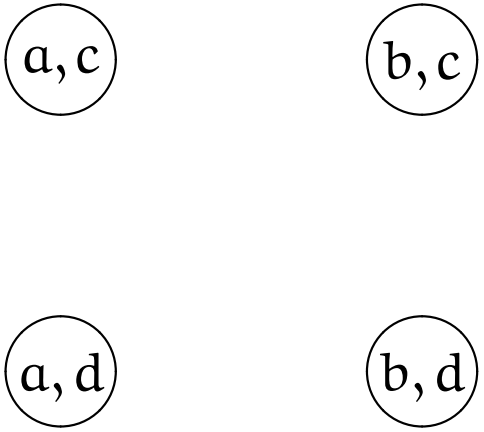
The first automaton accepts the strings containing at least one 1; the second automaton accepts strings containing at least one 0.

Example: intersection automaton

Starting from the states . . .

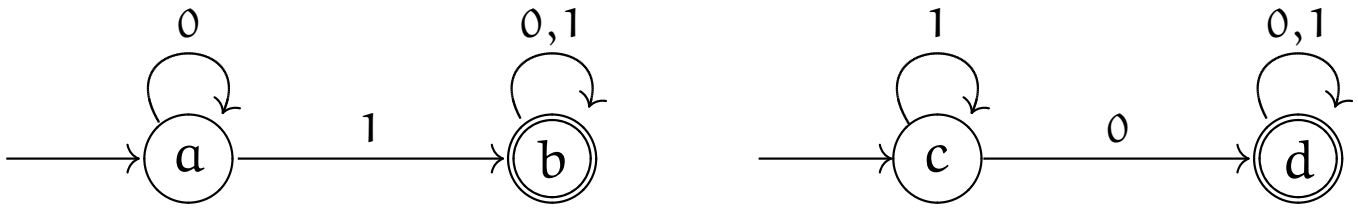


Step 1. The states are all pairs (x, y) with $x \in \{a, b\}$ and $y \in \{c, d\}$:

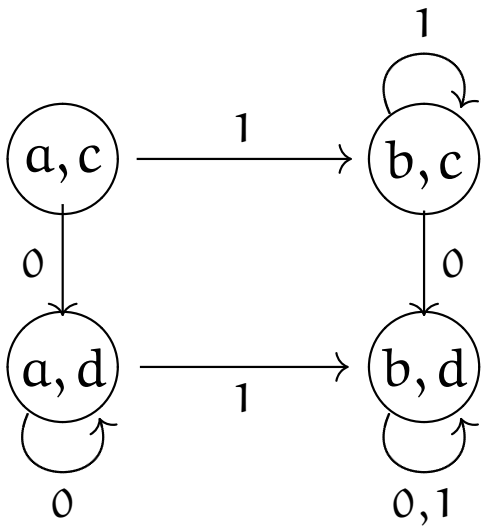


Example: intersection automaton

Let's add transitions.

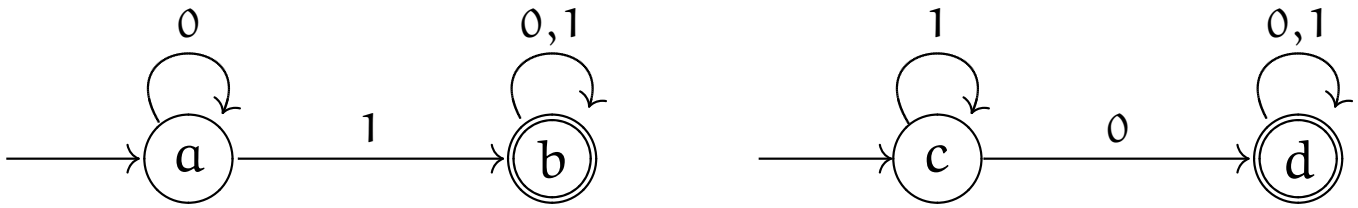


Step 2. We have $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p'$ and $q' \xrightarrow{a} p'$.

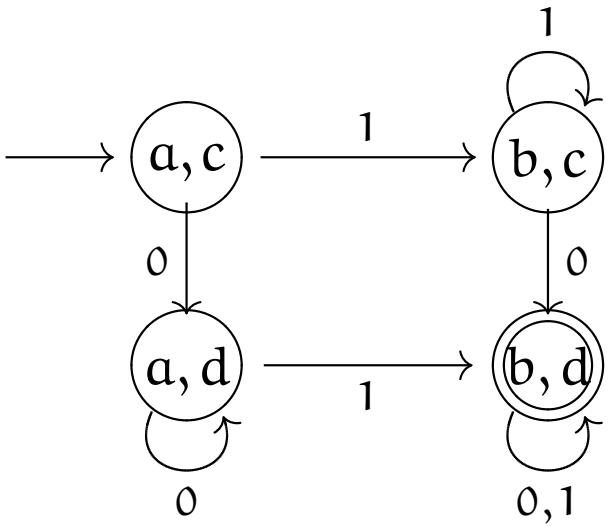


Example: intersection automaton

Let's add initial and accepting states.



Step 3. Pair (q, q') is accepting if both q and q' are. Similarly initial.



CO4211/CO7211

Discrete Event Systems

Lecture 4:
Nondeterministic Automata

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Properties of regular languages.

Idea. Study constructions on automata:

- helps us to see which languages are regular;
- helps us to mechanise constructions of the corresponding automata.

We've already seen some examples:

- the union of regular languages is regular;
- the intersection of regular languages is regular;
- the complement of a regular language is regular.

Moreover, the methods allow us to combine automata together in a constructive manner.

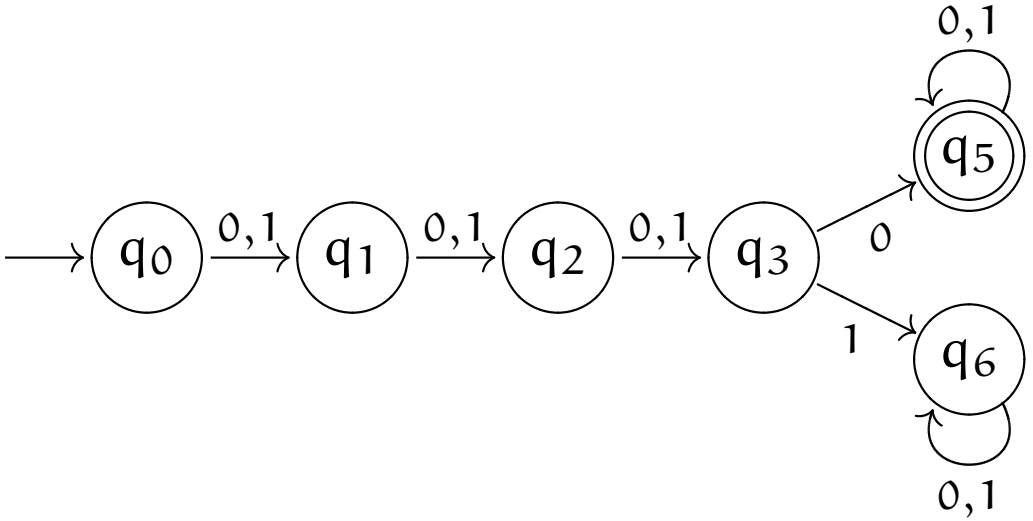
Reversing strings

Definition. If $L \subseteq \Sigma^*$ is a language, then $L^r = \{w^r : w \in L\}$ is the collection of all the *reversed strings* w^r formed from the strings w in L, where $(a_0 a_1 \dots a_{n-1})^r = a_{n-1} a_{n-2} \dots a_0$ (and $\epsilon^r = \epsilon$).

Question. If an automaton A accepts L, can we build an automaton A^r that accepts L^r?

Reversing strings

Example. Let $\Sigma = \{0, 1\}$ and consider:

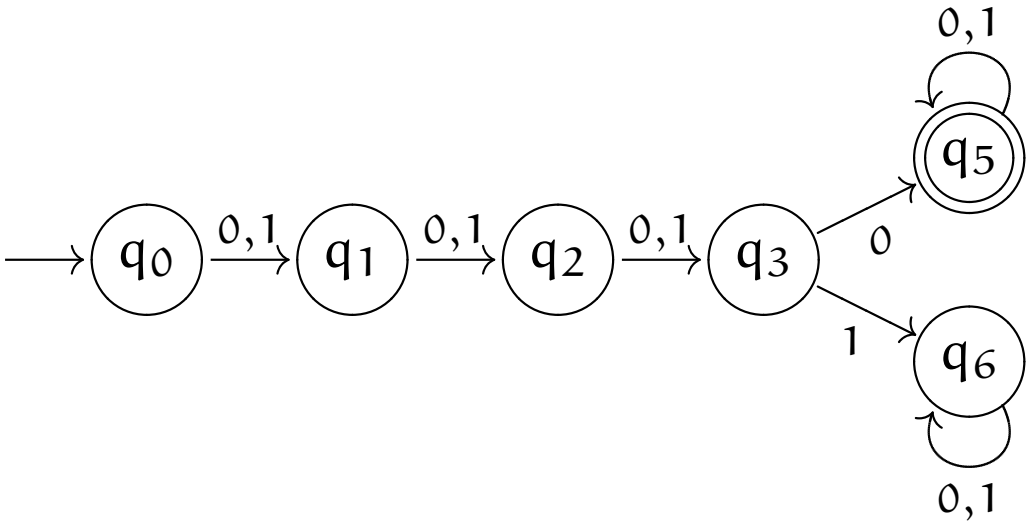


The language accepted by this automaton A is

$$L = L(A) = \{w \in \Sigma^* : \text{the } 4\text{-th bit of } w \text{ from the left is } 0\}.$$

Reversing strings

Example. Let $\Sigma = \{0, 1\}$ and consider:



The reversal K of $L = L(A)$ is the language

$$K = L^r = \{w \in \Sigma^* : \text{the } 4\text{-th bit of } w \text{ from the right is } 0\}.$$

Question. Can we construct a finite automaton accepting K?

Reversing strings

Question. Can we construct a finite automaton accepting K?

Problem. Constructing an automaton for L was straightforward; we know when we have reached the 4-th bit from the left (we start at the beginning of the string and count).

Constructing an automaton accepting K is not so obvious; how do we know when we have reached the 4-th bit from the right (since we don't know when the string is going to end)?

Reversing languages: a weird idea?

Idea.

- given an automaton A accepting L, we want an automaton A^r accepting L^r;
- instead of going from left to right, we go from right to left;
- in A^r, we start where A finishes ...
- ... and we finish in the starting position of A.

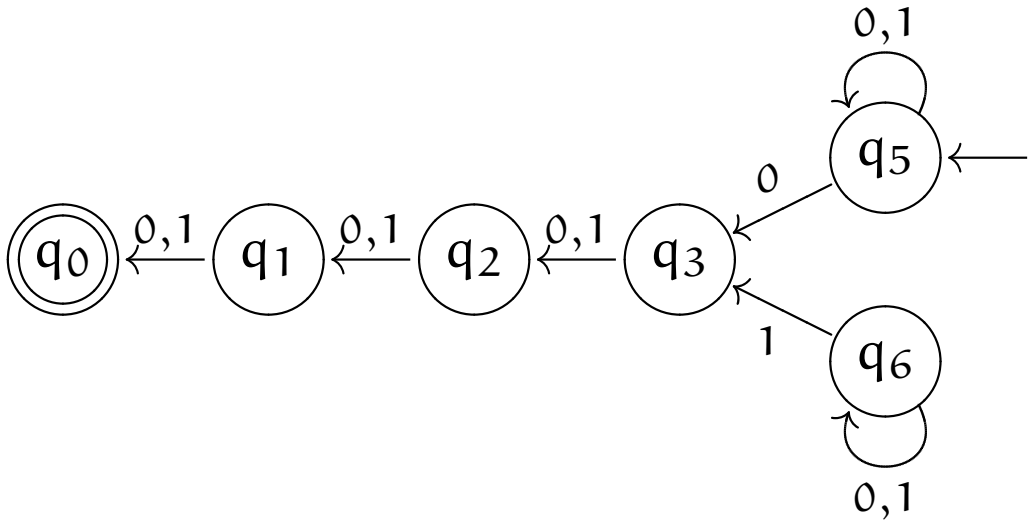
That means that we should ...

- reverse the directions of the arrows;
- swap the start state and the accept states.

Can we really do this?

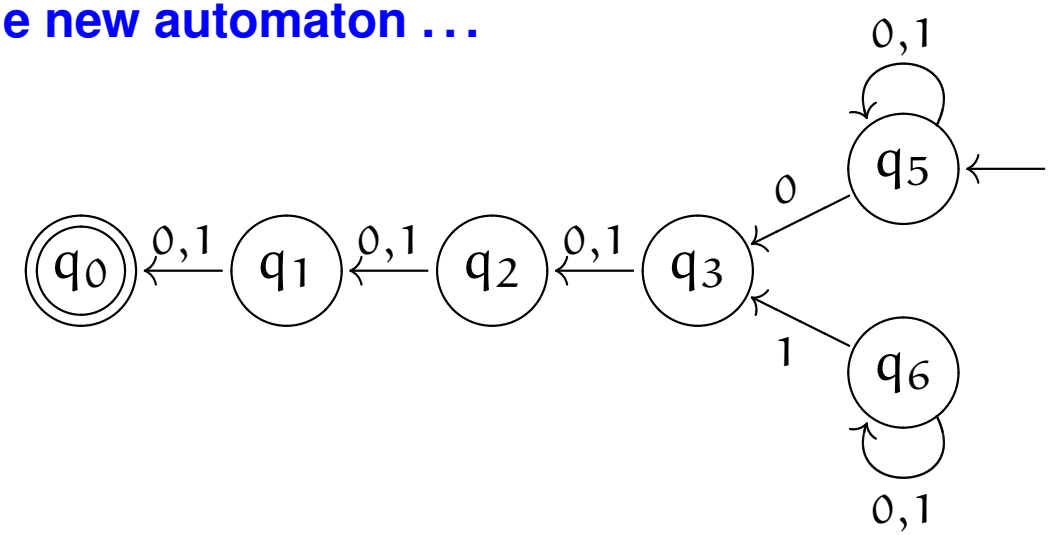
Reversing languages: a weird idea?

What we have suggested would transform our automaton into the automaton



Discussion of this weird idea

Looking at the new automaton . . .



1. there's an unreachable state (q_6); that's not nice, but it is allowed;
2. the old start state q_0 has no outgoing edges; this could be fixed;
3. however, the state q_5 now has *two* outgoing edges labelled 0!

The last issue (point 3) is a real issue we have not allowed this so far. This is known as *nondeterminism*.

How do we cope with nondeterminism?

Running the automaton: keep more than one state active!

- every state now has *a set of possible successors* (which could be empty) as opposed to just a single successor as before;
- each input string now describes *a set of paths* as opposed to a single path as before;
- we accept a string w if *there exists* an w-labelled path from the start state to an accepting state.

Automata of this type (where we allow the possibility of choice) are called *nondeterministic*.

Nondeterministic finite automata

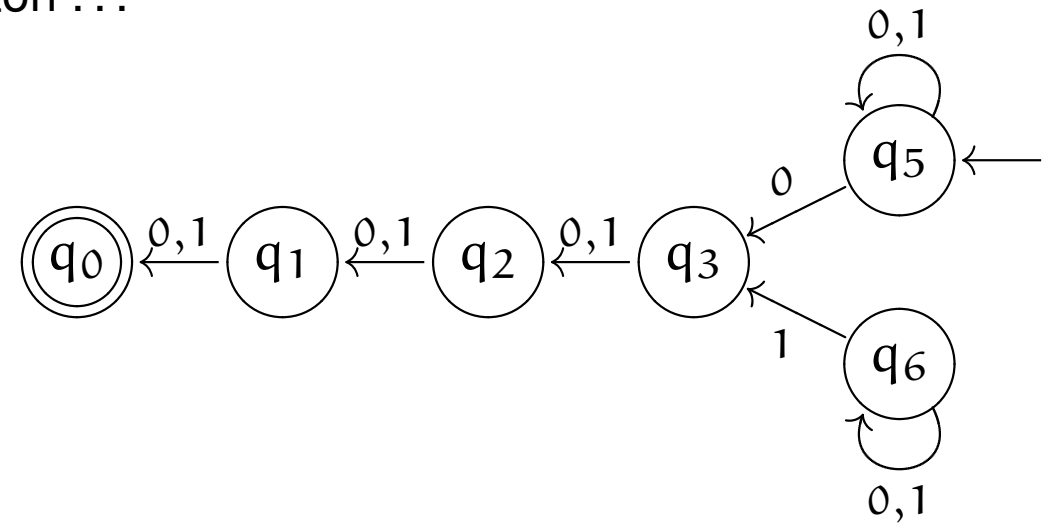
Notation. $\mathcal{P}(Q)$ is the set of all subsets of Q.

Definition. A *nondeterministic finite automaton* (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function;
- $q_0 \in Q$ is the start state;
- $F \subseteq Q$ is the set of accepting states.

Transition function in a nondeterministic automaton

In a nondeterministic finite automaton, the transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ yields a set of states, not just a single state. For example, in the automaton . . .

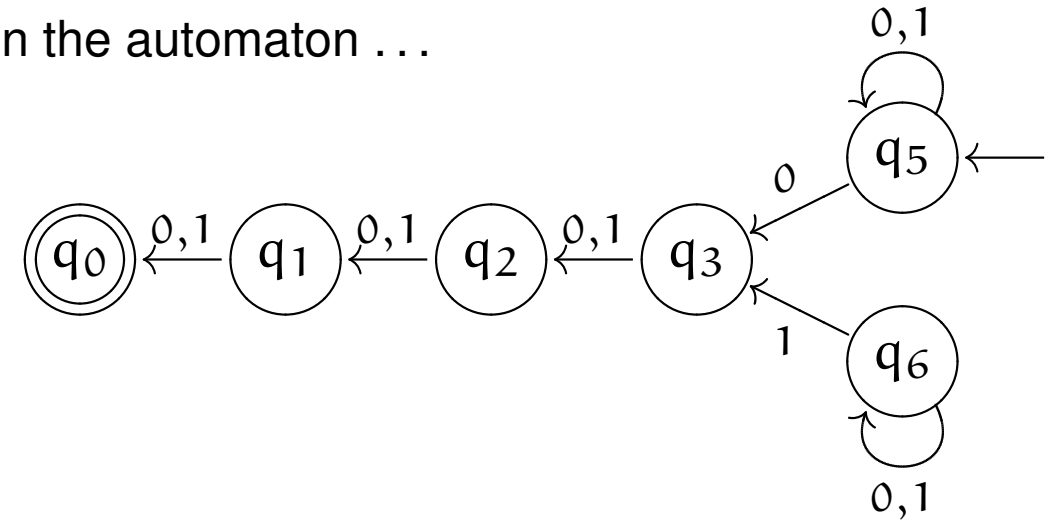


. . . we have $\delta(q_5, 0) = \{q_3, q_5\}$, $\delta(q_3, 0) = \{q_2\}$ and $\delta(q_0, 0) = \emptyset$.

Accepting strings in a nondeterministic automaton

The automaton A *accepts* the string w if there exists an w-labelled path from the start state to an accepting state.

For example, in the automaton . . .



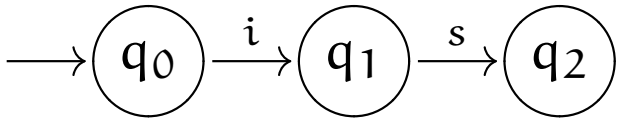
. . . we have that 00110111 is accepted as:

$$q_5 \xrightarrow{0} q_5 \xrightarrow{0} q_5 \xrightarrow{1} q_5 \xrightarrow{1} q_5 \xrightarrow{0} q_3 \xrightarrow{1} q_2 \xrightarrow{1} q_1 \xrightarrow{1} q_0$$

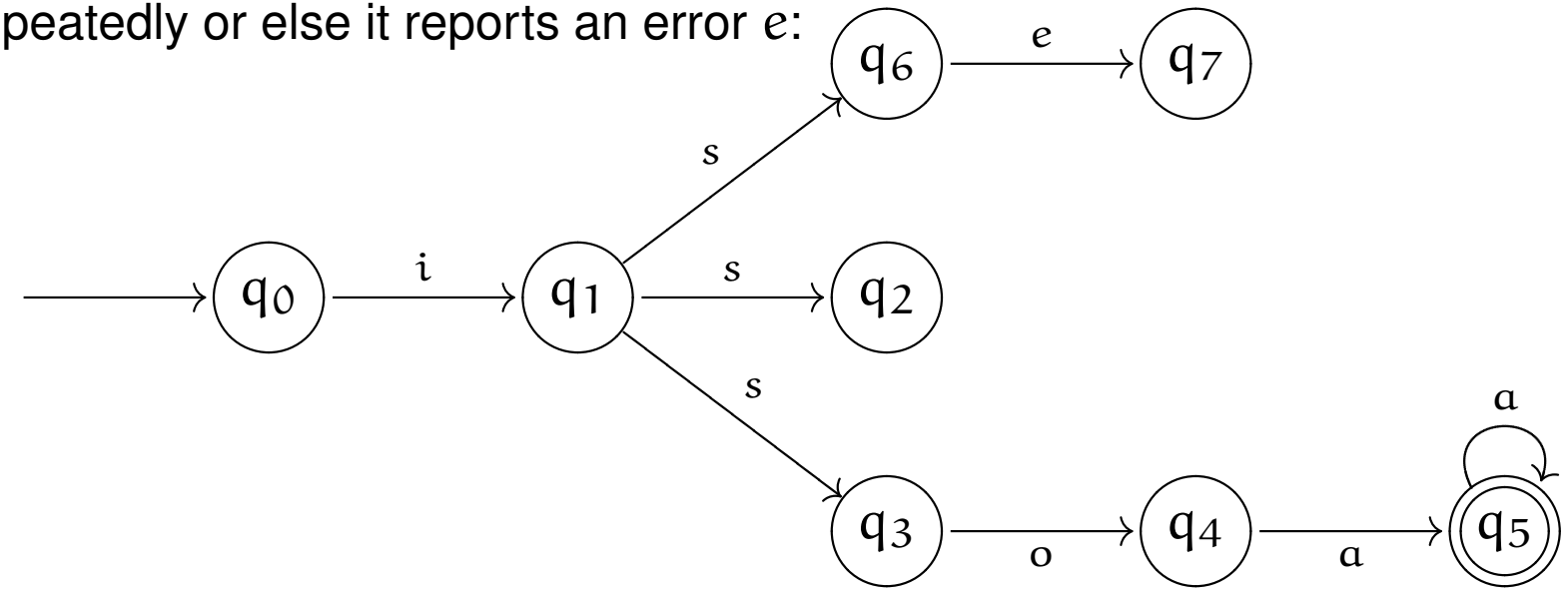
is an 00110111-labeled path from the start state to an accepting state.

NFAs: motivation

Example. After switching on, a machine performs an initialisation action i and a self-test s ...



... and then it either reports o (for "OK") and embarks on an action a repeatedly or else it reports an error e:



Bottom line: NFAs can be much more flexible for modelling!

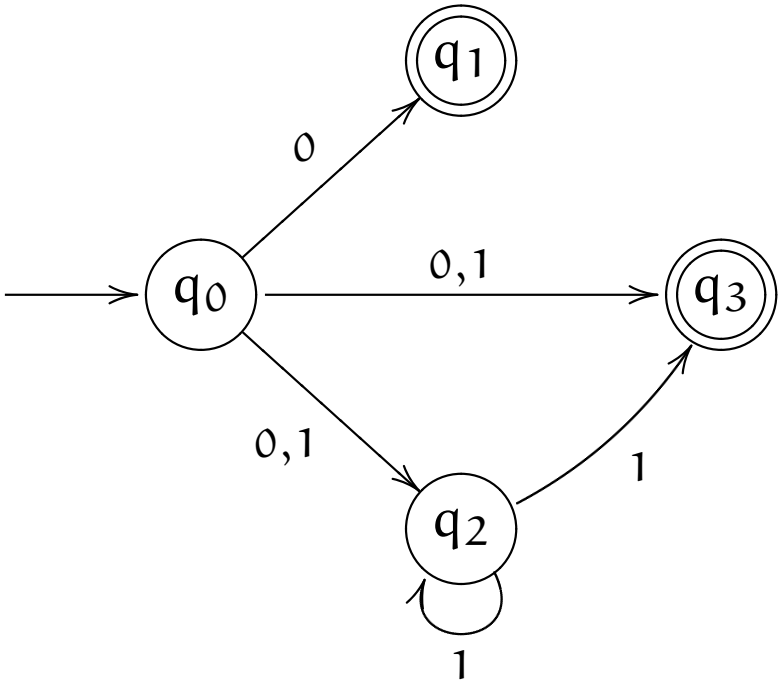
Nondeterministic Automata Acceptance

Definition. A run of a nondeterministic automaton A on a word $w = a_0 a_1 \cdots a_{n-1}$ is $r = q_0, q_1 \ldots, q_n$ such that for every $i \geq 0$ we have $q_{i+1} \in \delta(q_i, a_i)$.

Definition. A run $r = q_0, q_1, \ldots, q_n$ is accepting if $q_n \in F$.

Definition. A string w is accepted by A if there is an accepting run of A on w.

NFA example



$$F = \{q_1, q_3\},$$

and

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

is defined by:

$$\delta(q_0, 0) = \{q_1, q_3\}$$

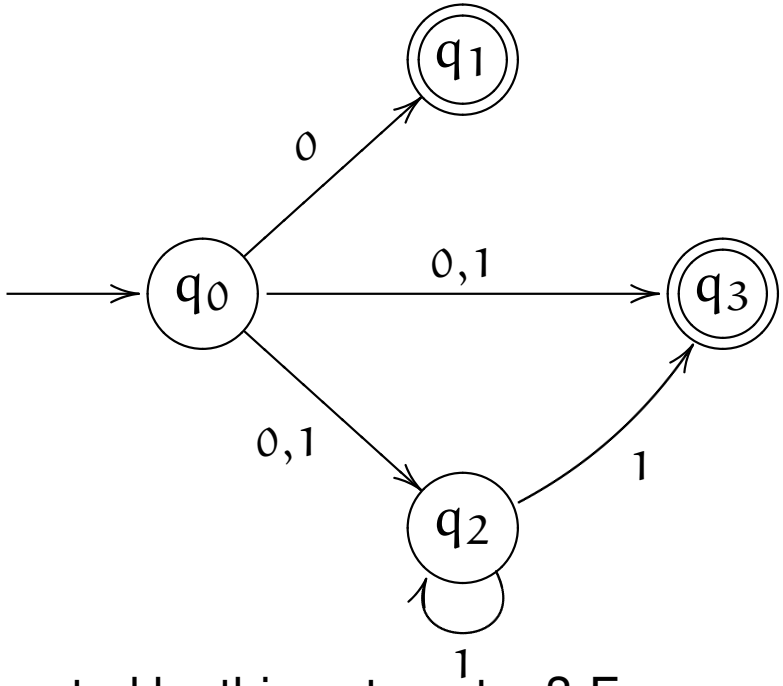
$$\delta(q_0, 1) = \{q_2, q_3\}$$

$$\delta(q_1, 0) = \emptyset \quad \ldots$$

Formally, $A = (Q, \Sigma, \delta, q_0, F)$ where:

$$Q = \{q_0, q_1, q_2, q_3\}, \ \Sigma = \{0, 1\},$$

NFA examples



Which strings are accepted by this automaton? For example, 1111 is accepted with the run q_0, q_2, q_2, q_2, q_3.

The language accepted by this automaton is

$$L = \{0,1\} \cup \{a1^k : k \geqslant 1 \text{ and } a \in \{0,1\}\}$$

Comparing deterministic and nondeterministic automata

	Deterministic automata	Nondeterministic automata
transition function	returns exactly one value	returns a set of values
acceptance	the (unique) path leads to an accepting state	there is at least one path to an accepting state

Comparing deterministic and nondeterministic automata

Questions.

- Are nondeterministic automata worth studying? In particular, what constructions can we do with nondeterministic automata?
- How do nondeterministic automata relate to deterministic ones? Is it possible to *implement* nondeterminism?

CO4211/CO7211

Discrete Event Systems

Lecture 5:
More on nondeterministic automata

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Nondeterministic automata

Recall . . .

A *nondeterministic finite automaton* (NFA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function;
- $q_0 \in Q$ is the start state;
- $F \subseteq Q$ is the set of accepting states.

In a nondeterministic finite automaton, the transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ yields a set of states, not just a single state.

Nondeterministic automata

Recall . . .

A nondeterministic automaton A *accepts* a string $w = a_0 \cdots a_1 \in \Sigma^*$ if there exists a run $r = q_0, q_1, \ldots, q_n$ such that

- for every $0 \leq i < n$ $q_{i+1} \in \delta(q_i, a_i)$.
- $q_n \in F$.

Comparing deterministic and nondeterministic automata

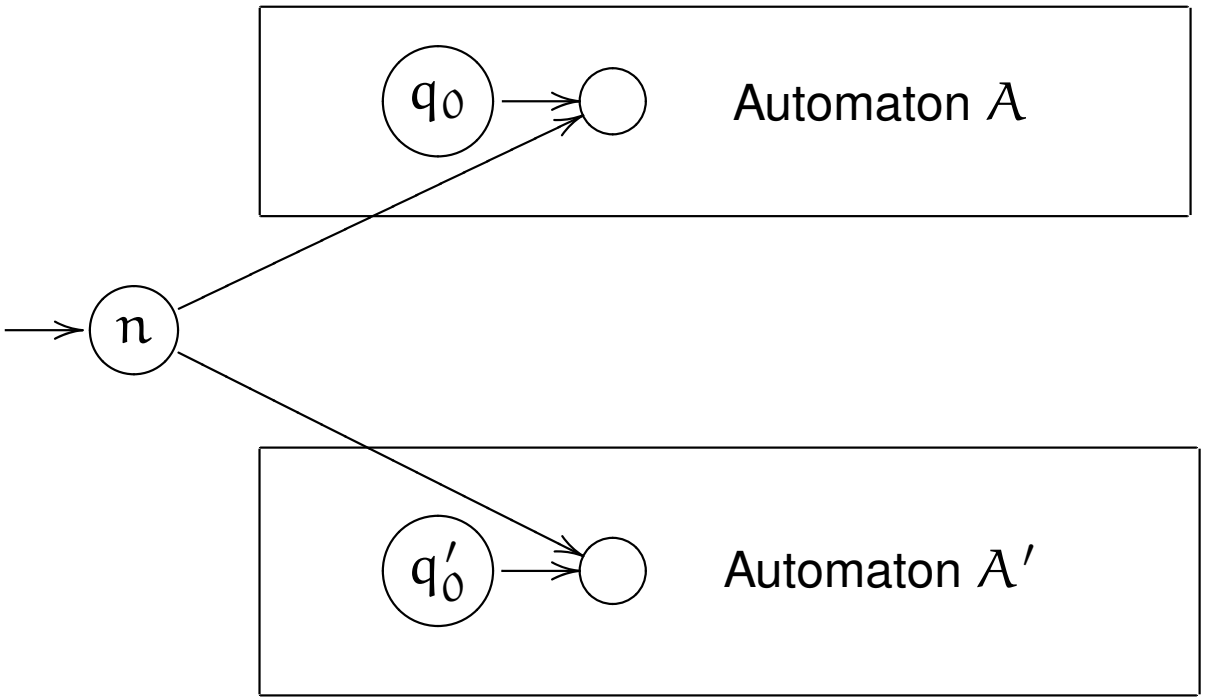
	Deterministic automata	Nondeterministic automata
transition function	returns exactly one value	returns a set of values
acceptance	the (unique) path leads to an accepting state	there is at least one path to an accepting state

Union of languages

Nondeterministic approach.

- Put the automata A for L and A' for L' next to one another;
- add a new start state (here called n) and add to it all the transitions of the original start states q_0 and q_0' of A and A';
- mark all accepting states of A and A' as accepting;
- if either q_0 or q_0' is accepting, mark the new start state as accepting.

Union of languages

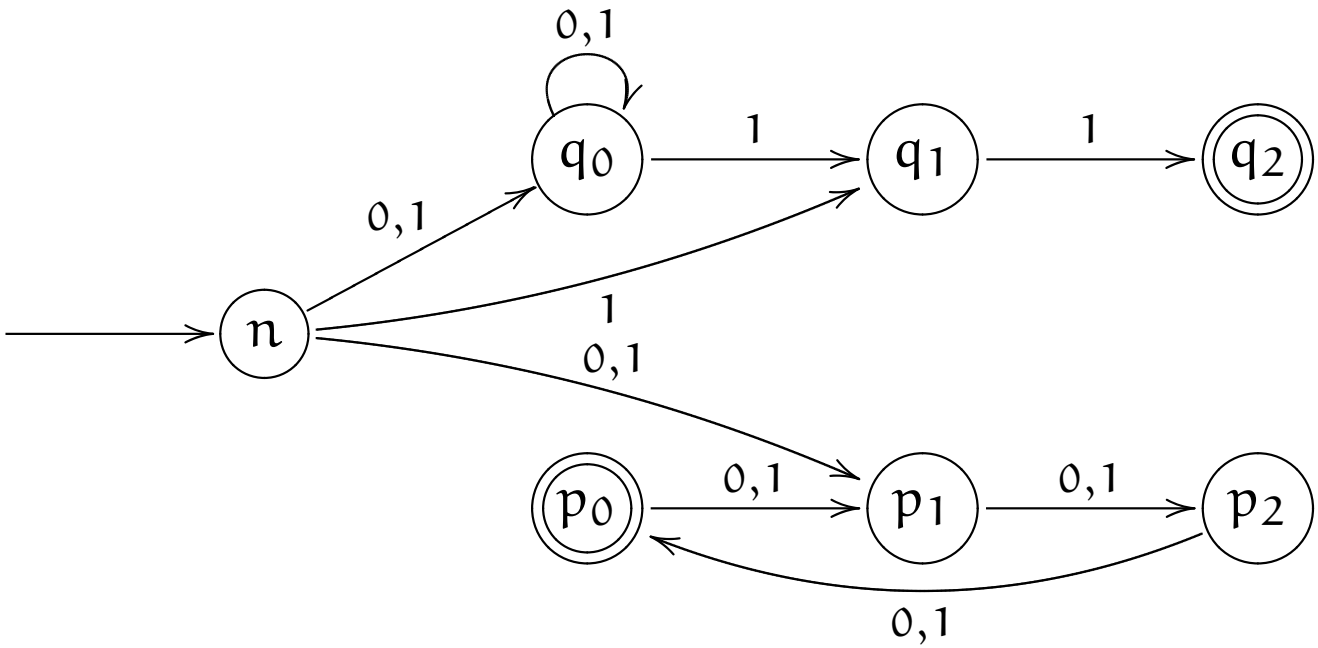


Union example

Let $\Sigma = \{0, 1\}$ and consider the language

$$\{w \in \Sigma^* : w \text{ ends with } 11\} \cup \{w \in \Sigma^* : |w| \% 3 = 0\}.$$

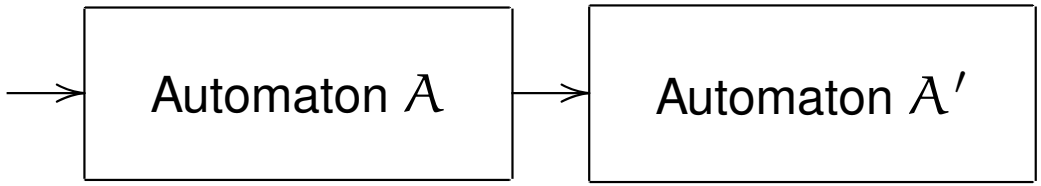
This is accepted by the automaton



Concatenation

Question. Suppose that automata $\mathcal{A}$ and $\mathcal{A}'$ each perform a certain task.

Is there an automaton that corresponds to *first* running the automaton $\mathcal{A}$ and *then* running the automaton $\mathcal{A}'$?



Remember. The behaviour of an automaton corresponds to the language it accepts, i.e. to a corresponding set of strings.

Concatenation of languages

Definition. If $w = a_1 \dots a_n$ and $v = b_1 \dots b_k \in \Sigma^*$, then

$$w \cdot v = a_1 \dots a_n b_1 \dots b_k$$

is the *concatenation* of w and v.

If L and L' are languages, then

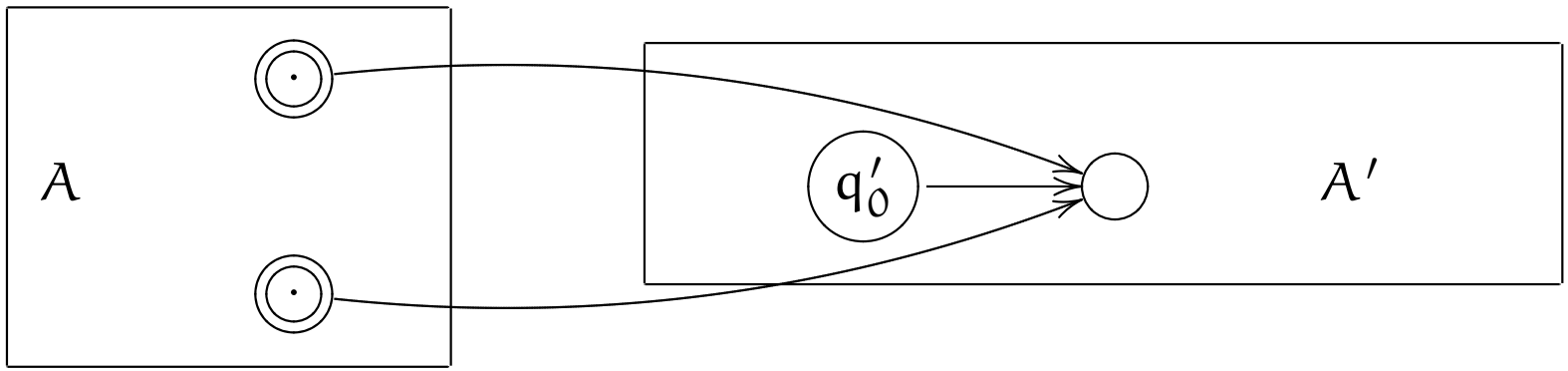
$$L \cdot L' = \{w \cdot w' : w \in L,\ w' \in L'\}.$$

Thus, given automata A and A', we are looking for an automaton that accepts $L(A) \cdot L(A')$.

Concatenation of languages

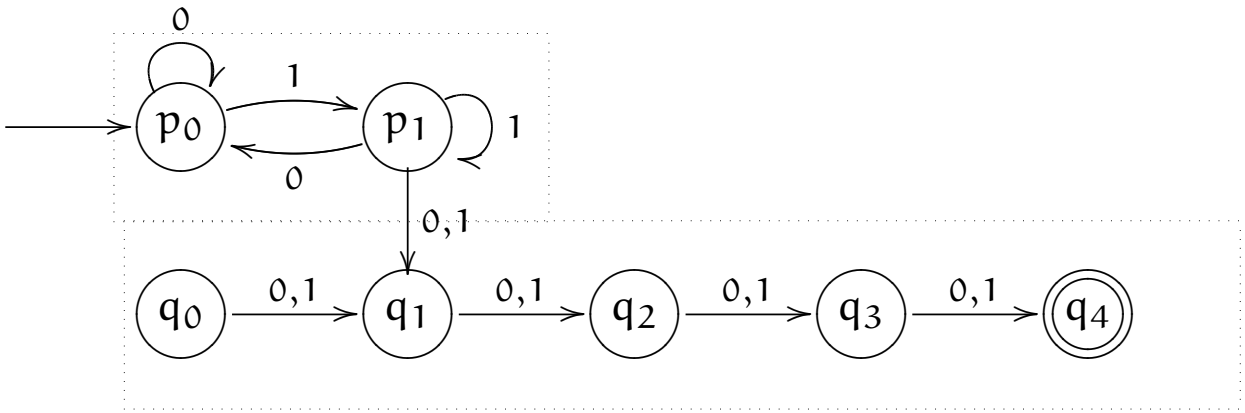
Idea.

- Put the automata A and A' next to one another;
- add the transitions of the initial state of A' to every accepting state of A;
- the start state of the new automaton is the start state of A;
- mark all accepting states of A' as accepting;
- if q_0' is accepting, mark all accepting states of A as accepting.



Concatenation example

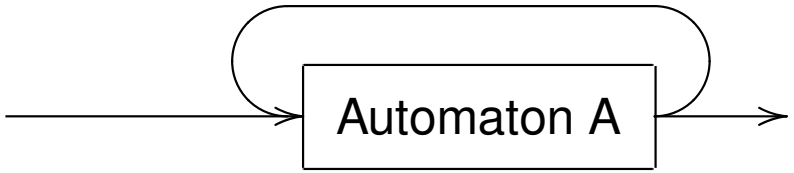
Let $L = \{w \in \Sigma^* : w \text{ ends with } 1\}$ and $L' = \{w \in \Sigma^* : w \text{ has length } 4\}$. Then $L \cdot L'$ is accepted by the automaton



- $w = 0010001 = w_1 \cdot w_2$ where $w_1 = 001 \in L$ and $w_2 = 0001 \in L'$, and so we have that $0010001 \in L \cdot L'$;
- $w = 11111$ can be split as $1 \cdot 1111$ and so $11111 \in L \cdot L'$;
- $w = 0010 \notin L \cdot L'$ as there is no appropriate splitting.

Repetition of actions

Question. Suppose that A performs a certain task. Can we construct an automaton that performs this task *repeatedly*?



Again: the action of an automaton is described by a language.

Definition. If $L \subseteq \Sigma^*$ is a language, then the language

$$L^* = \{w_1 \cdots w_n : n \geqslant 0, w_1, \ldots, w_n \in L\}$$

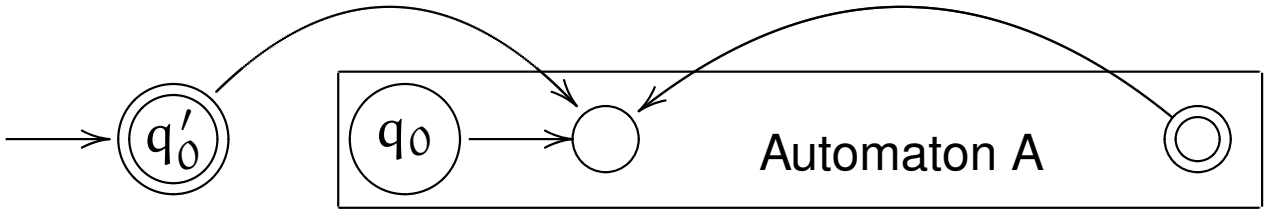
is called the *(Kleene) star* of L.

Kleene star

Given an automaton A that accepts L, we are looking for an automaton that accepts L^*.

Idea.

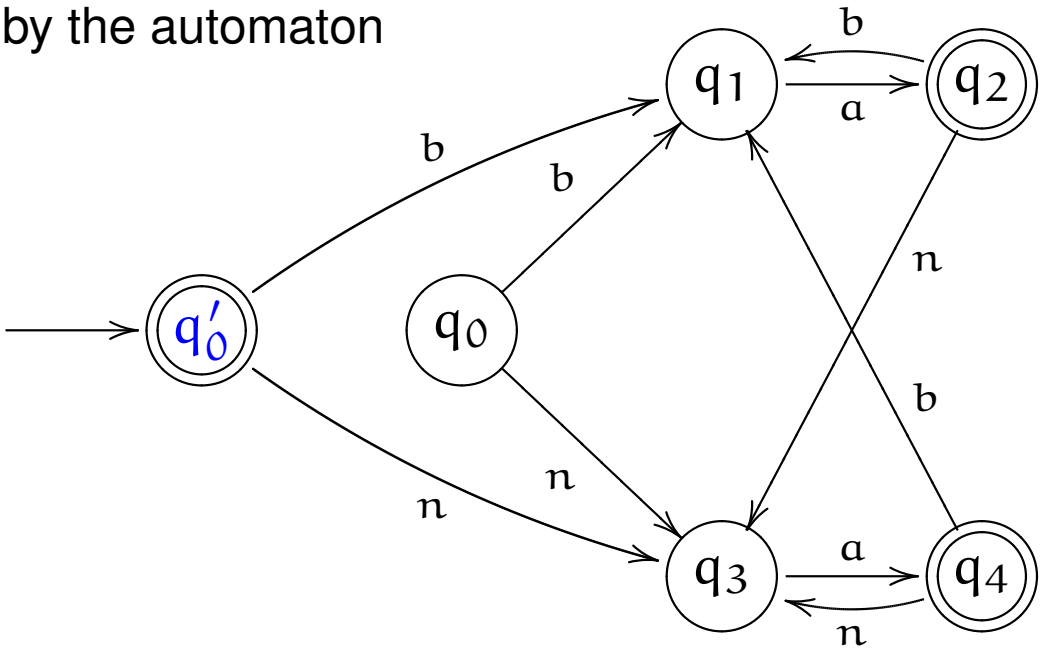
- Add to every accepting state all the transitions of the start state;
- add a new accepting initial state that has the same transitions as the original initial state.



Kleene star: example

Let $L = \{ba, na\}$. Then $L^* = \{w \in \Sigma^* : w$ is a sequence of ba's and na's $\}$. For example, $\epsilon \in L^*$, $ba \in L^*$, $baba \in L^*$ and $bana \in L^*$, but $bnaa \notin L^*$ and $baan \notin L^*$.

L^* is accepted by the automaton

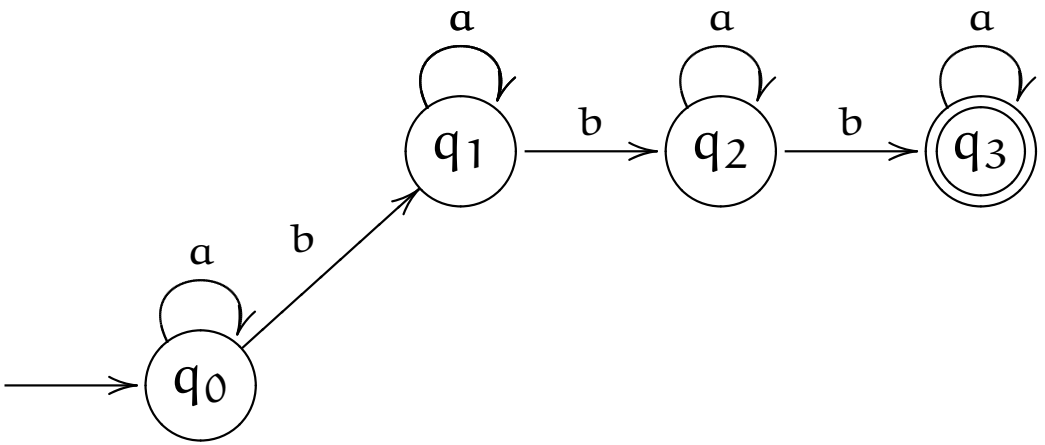


Example. Try the strings $banana$ and $nababa$.

Another example

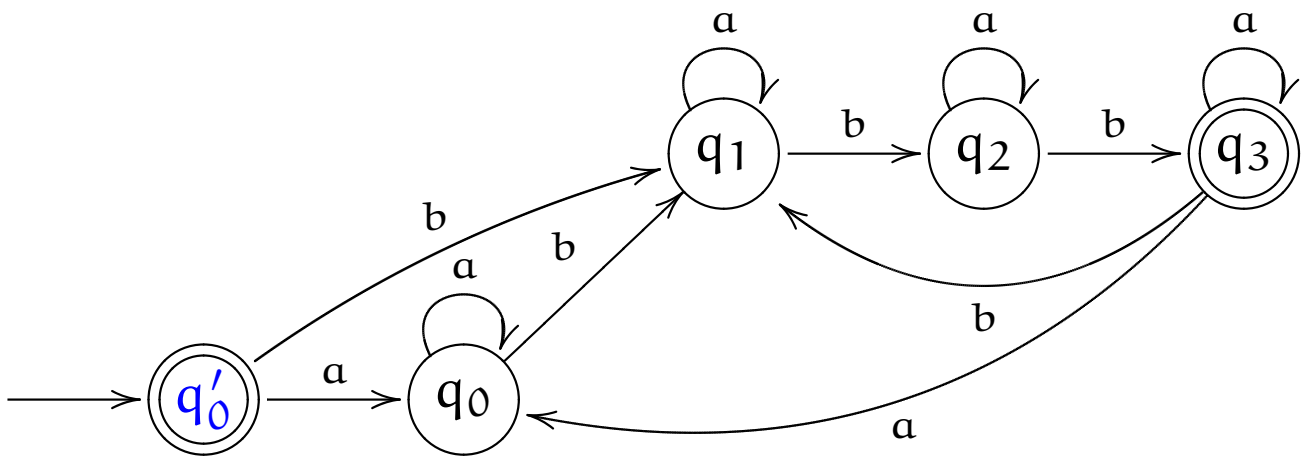
Consider the following automaton accepting the language

$$L = \{w : \sharp_b(w) = 3\}.$$



Another example (continued)

We can form an automaton accepting L^* as follows:



Here L^* is the language

$$\{\epsilon\} \cup \{w : \sharp_b(w)\%3 = 0 \text{ and } \sharp_b(w) > 0\}$$

Summary

Nondeterministic automata

- working with nondeterministic automata can be comparatively easy – some constructions are easier;
- on the face of it, we don't know to implement nondeterministic automata – we'll come to this issue later.

Theorem. If L and L' are languages accepted by nondeterministic automata, then so are the languages $\mathrm{L} \cup \mathrm{L}'$, $\mathrm{L} \cdot \mathrm{L}'$ and L^*.

Question. Do nondeterministic automata give more power?

CO4211/CO7211

Discrete Event Systems

Lecture 6:
From nondeterministic to deterministic automata

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Summary

Nondeterministic automata

- working with nondeterministic automata can be comparatively easy – some constructions are easier;
- however, on the face of it, we don't know how to implement nondeterministic automata.

Theorem. If L and L' are languages accepted by nondeterministic automata, then so are the languages $\mathrm{L} \cup \mathrm{L}'$, $\mathrm{L} \cdot \mathrm{L}'$ and L^*.

Question. Do nondeterministic automata give more power? Can we translate nondeterministic automata to deterministic automata?

Idea. We somehow need to keep track of multiple active states.

Removing Nondeterminism

Implementation and modelling.

Nondeterministic automata	**Deterministic automata**
+ modelling is generally easier;	- more constrained model;
+ more general?	- less general?
- impossible to implement?	+ easy to implement.

Note. Every deterministic automaton can be regarded as an example of a nondeterministic one.

Question. Can we get the best of both worlds? Can we convert nondeterministic automata to deterministic ones?

From NFAs to DFAs

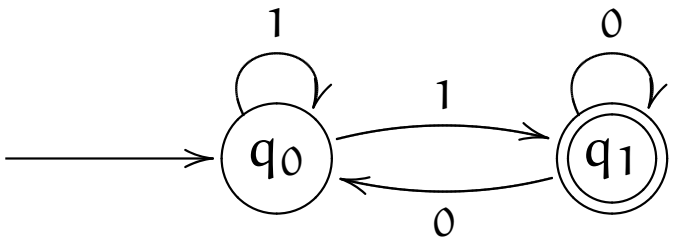
Converting NFAs to DFAs. We need to take care of nondeterministic transitions (transitions in an NFA give a *set* of successor states, not a single state).

Basic idea. Keep multiple states active.

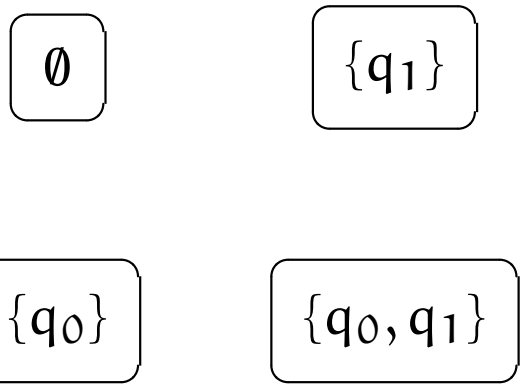
- NFAs have a *uniquely determined set* of active states at each step.
- We could view a set of states as a *single state* in a new automaton.
- The transitions then change the *set* of active states.

Powerset construction

Consider the NFA



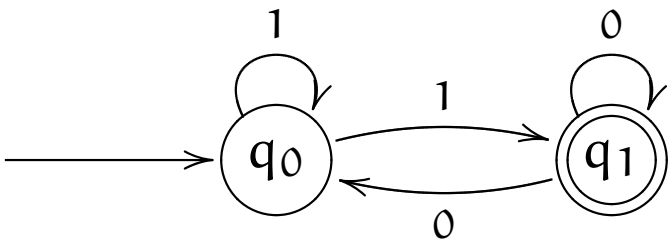
Step 1. A state of the new automaton is a *set* of states of the original one.

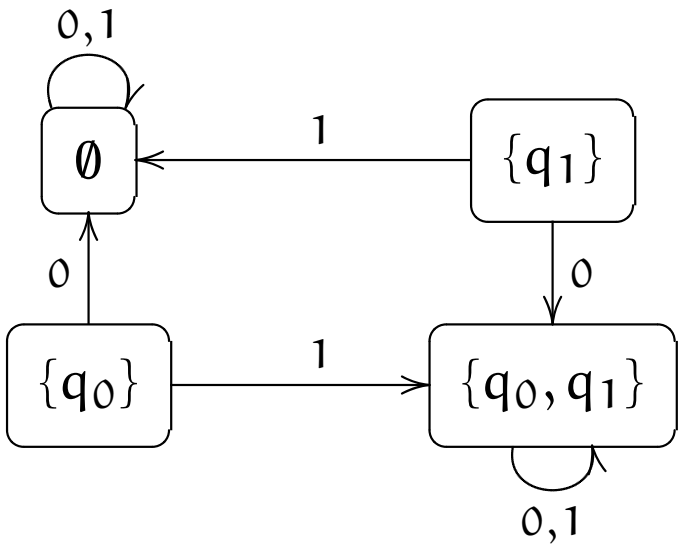


Note. Don't forget the empty set!

Powerset construction

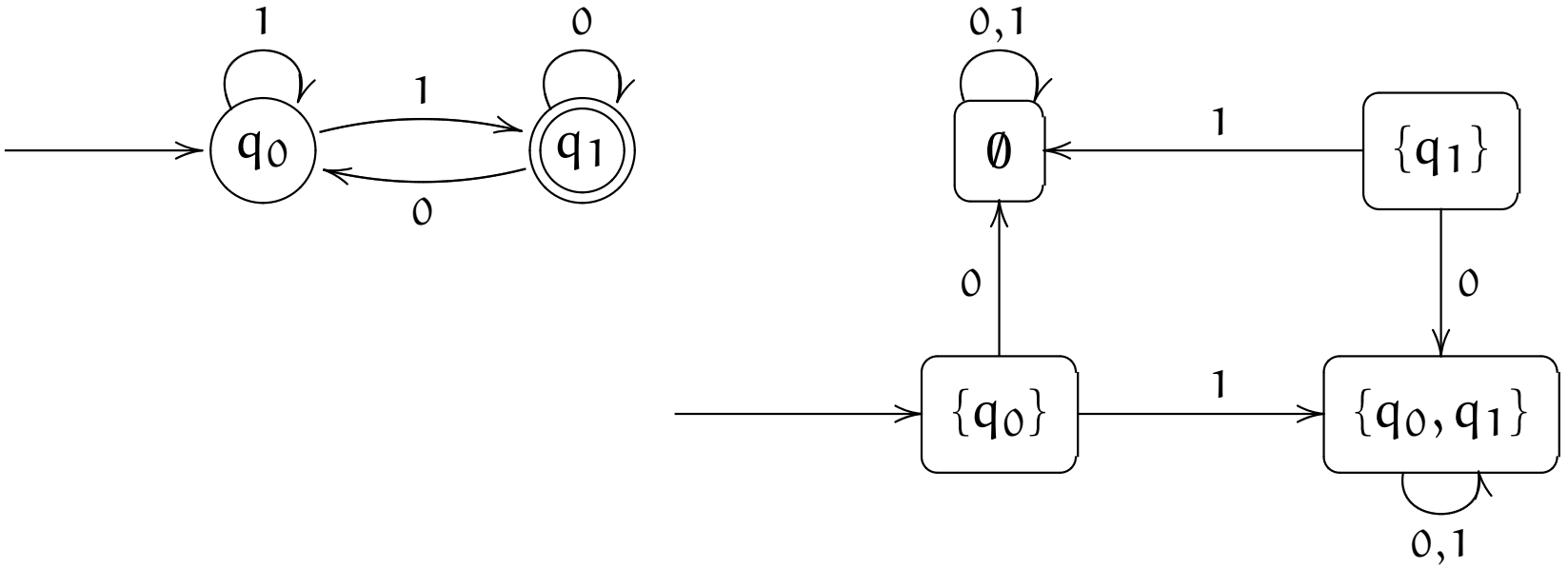
Step 2. The successor states of $P \in \mathcal{P}(Q)$ are the sets of states that can be reached from an *element* of P by reading the same letter.





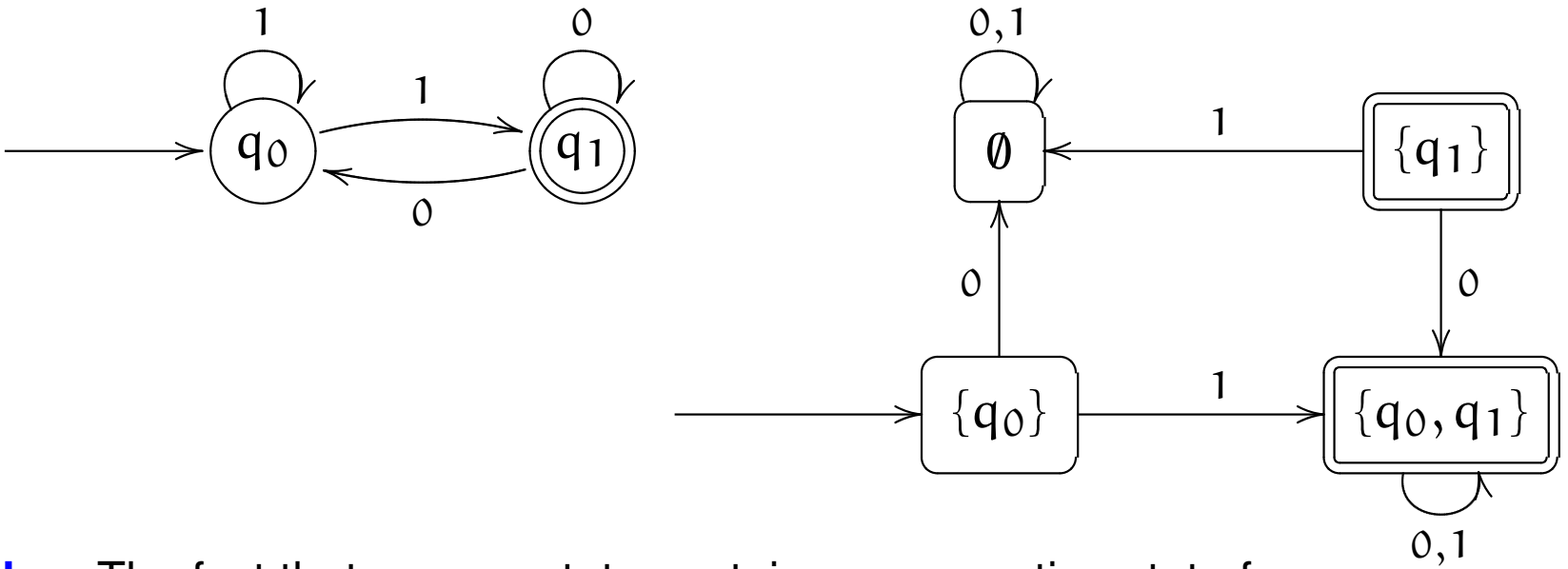
Powerset construction

Step 3. Use the *set* that contains the initial state:



Powerset construction

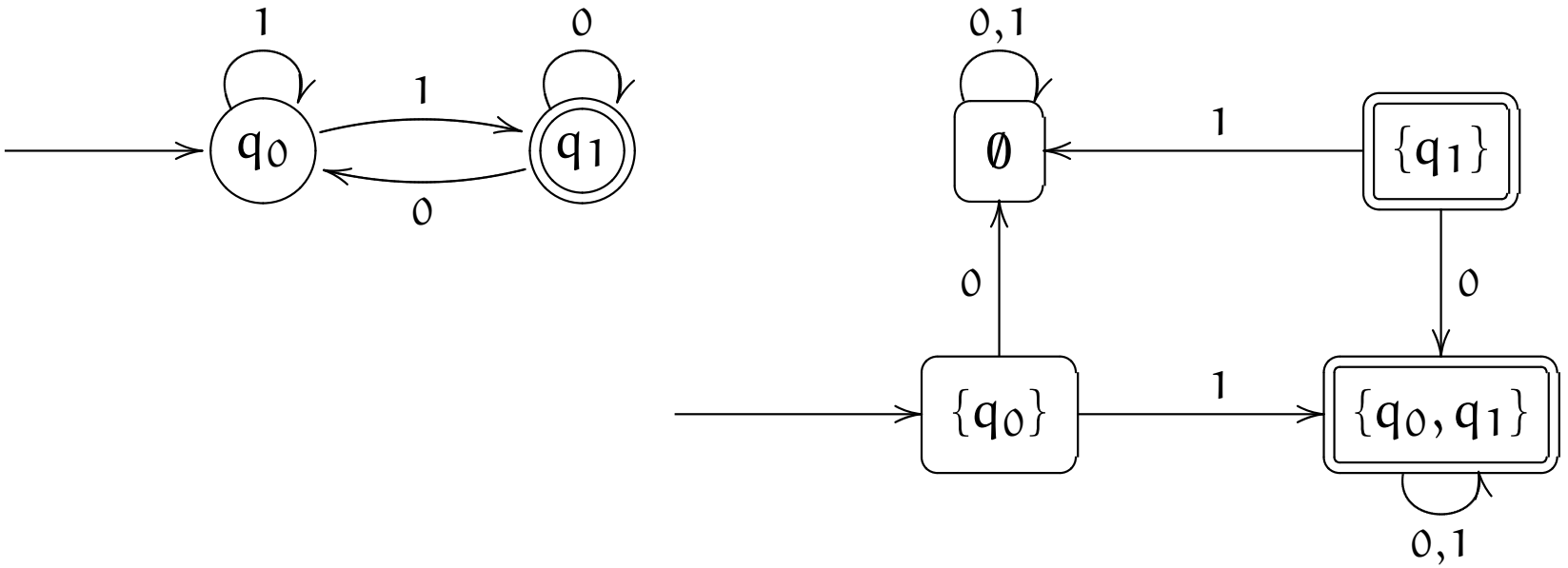
Step 4. A powerstate accepts if it contains an accepting state of the original machine. Here we get the powerstates $\{q_1\}$ and $\{q_0, q_1\}$ as these are the powerstates containing q_1:



Idea. The fact that a powerstate contains an accepting state from the original machine represents the possibility that we could have ended up in an accepting state.

Powerset construction

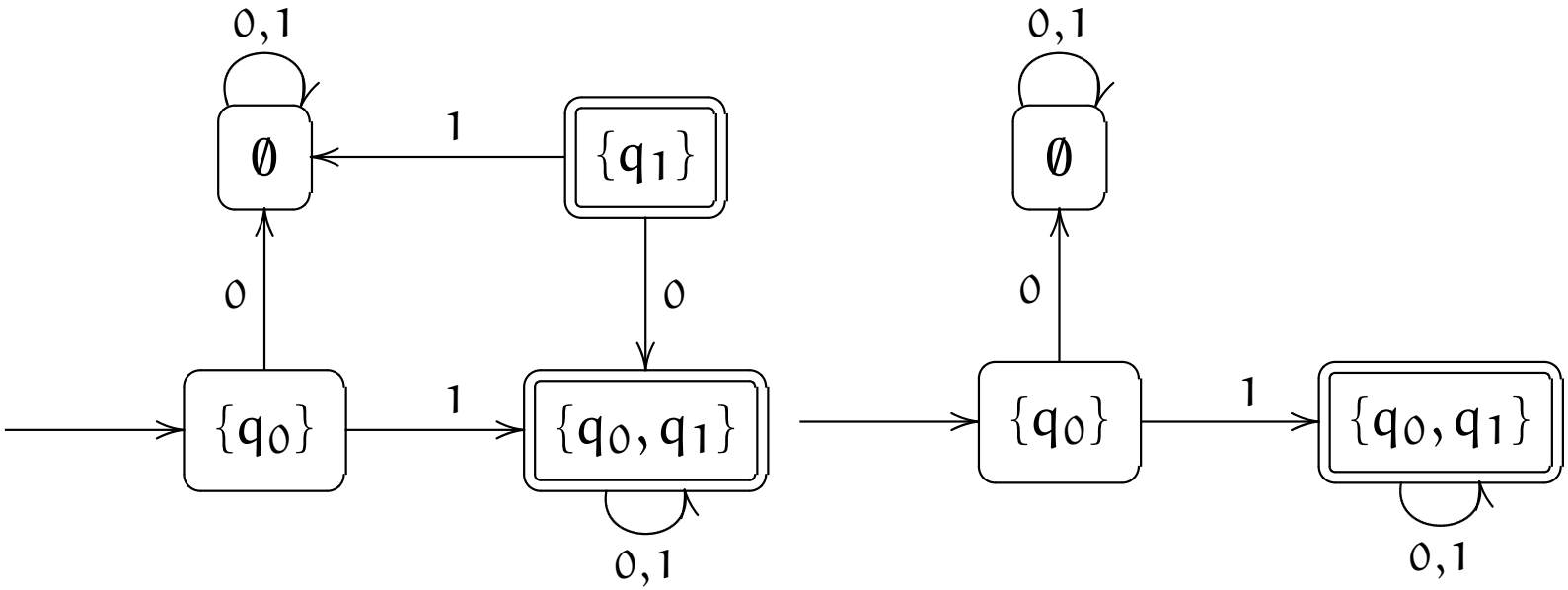
Step 5 (optional) It is possible that some of the powerstates in the new powerstate automaton are unreachable from the start powerstate; in our example, the powerstate $\{q_1\}$ is unreachable from the start powerstate $\{q_0\}$:



(Alternatively, we could have started with the start powerstate and generated powerstates until all transitions were defined.)

Powerset construction

We can simply remove such unreachable states:



(Starting from the start powerstate and generating powerstates would have yielded the same result.)

Powerset construction - formally

Given. An NFA $A = (Q, \Sigma, \delta, q_0, F)$.

The *powerset automaton*

$$A^{\mathcal{P}} = (Q^{\mathcal{P}}, \Sigma, \delta^{\mathcal{P}}, q_0^{\mathcal{P}}, F^{\mathcal{P}})$$

is given by:

- $Q^{\mathcal{P}} = \mathcal{P}(Q)$. This keeps multiple states active.
- $\delta^{\mathcal{P}} : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$.

  $(P, a) \mapsto \{q \in Q : p \xrightarrow{a} q \text{ for some } p \in P\}$.

Powerset construction (continued)

- $q_0^{\mathcal{P}} = \{q_0\}$.
- $F^{\mathcal{P}} = \{S \subseteq Q : S \cap F \neq \emptyset\}$.

  We accept if the powerstate contains an accepting state of A.

Translation NFA $\longrightarrow$ DFA

By examining this construction, we obtain:

Theorem. If A is an NFA then A and $A^{\mathcal{P}}$ accept the same language: $\mathrm{L}(A) = \mathrm{L}(A^{\mathcal{P}})$.

Corollary. If $\mathrm{L} = \mathrm{L}(A)$ for an NFA A then L is regular.

Consequences.

- *Nondeterministic* behaviour can be replaced by *deterministic* behaviour;
- nondeterministic automata cannot accept more languages than deterministic automata;
- we now have the best of both worlds: use the easy to use nondeterministic models and finally determinize them!

Deterministic vs nondeterministic

Best of both worlds!

Deterministic automata:

- easy to implement.

Nondeterministic automata:

- more general model (at first appearance);
- many constructions are simple.

Crucial relationship. For every nondeterministic automaton we can construct an equivalent deterministic automaton.

Moral of the story. We can work with nondeterministic automata, and convert to deterministic ones when looking for an implementation.

CO4211/CO7211

Discrete Event Systems

Lecture 7:
Synchronous composition of automata

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Building systems from components

Recall the first lecture. A *system* consists of several *components* that co-operate to give the desired behaviour.

Ways of putting components together:

- choice (leads to the union of languages);
- sequential composition (leads to the concatenation of languages);
- iteration (leads to the Kleene star of a language).

A missing construction:

- run systems in parallel with synchronization (a generalization of intersection).

Example

Queueing incoming jobs:

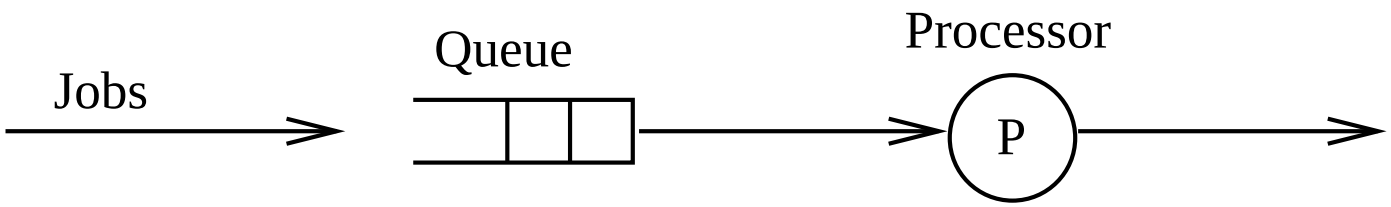
- a processor P continuously processes jobs;
- arriving jobs are stored in a queue of capacity 2;
- the queue dispatches the jobs to the processor;
- if the queue is full then the jobs are simply rejected.

Analysis of the system:

- two components: *Queue* and *Processor*;
- each component can be represented by a simple automaton;
- the two components need to work together to achieve the desired goal.

Can we put the components together to get a model of the whole system?

Job dispatch



Actions of the queue:

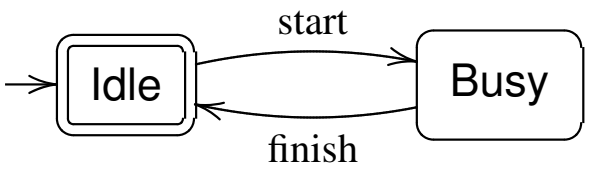
- accepts jobs if the queue is not full;
- dispatches jobs to the processor.

Actions of the processor:

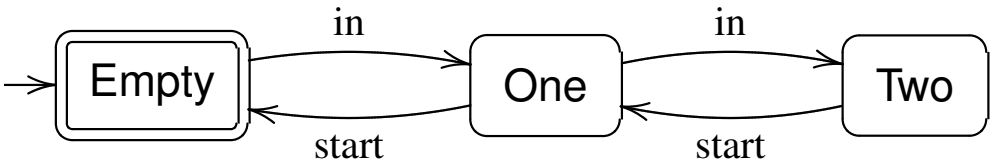
- accepts jobs from the queue;
- processes an accepted job.

Models of the two components

Processor



Queue



Models of the two components

For simplicity, we have omitted the error states from the two automata.

Actions

- *in*: job arrives in the queue;
- *start*: processing job begins;
- *finish*: processing job ends.

Co-operation. The processor needs to start on a job *at the same time* as the queue starts the job; the events are *synchronous*. Other events are independent (or *asynchronous*).

Making components work together

Idea. To make the two components co-operate:

- we run them in parallel (as with the intersection construction);
- we make sure that the "start" events are *processed at the same time* by the two components;
- all the other events are processed *independently*.

More precisely . . .

- *Processor* and *Queue* work over different alphabets: $\Sigma_{\mathrm{P}} = \{\mathrm{start}, \mathrm{finish}\}$ and $\Sigma_{\mathrm{Q}} = \{\mathrm{in}, \mathrm{start}\}$.
- for events in the intersection of the two alphabets, we demand *simultaneous execution*;
- for all other events, we allow *independent execution*.

Putting things in parallel

Given two automata $A = (Q, \Sigma, \delta, q_0, F)$ and $A' = (Q', \Sigma', \delta', q_0', F')$, we want to run A and A' "in parallel".

- the state set $Q^{\|}$ is $Q \times Q'$.

Forcing synchronous transitions:

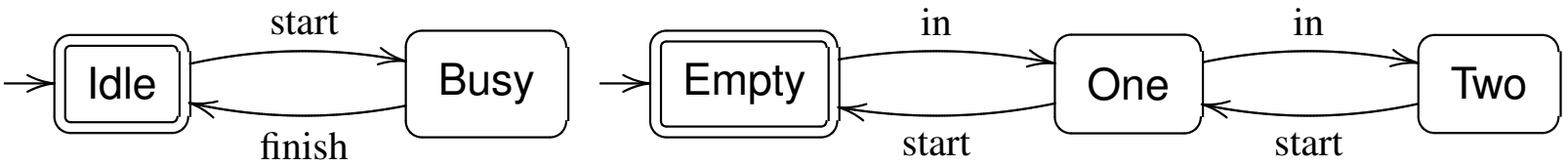
- if $a \in \Sigma \cap \Sigma'$ then $(q, q') \xrightarrow{a} (p, p')$ if and only if $q \xrightarrow{a} p$ and $q' \xrightarrow{a} p'$.

Implementing independent transitions:

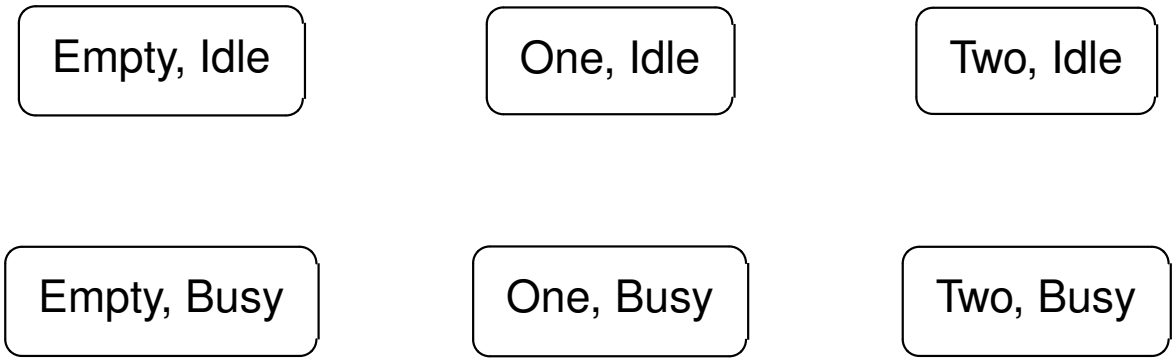
- if $a \in \Sigma - \Sigma'$ then $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $p' = q'$;
- if $a \in \Sigma' - \Sigma$ then $(q, q') \xrightarrow{a} (p, p')$ if $q' \xrightarrow{a} p'$ and $p = q$.

The *start state* is $q_0^{\|} = (q_0, q_0')$ and the *accepting states* are $F^{\|} = F \times F'$.

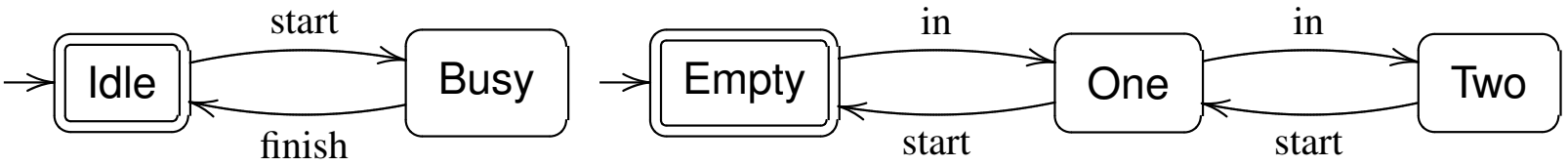
Back to the example



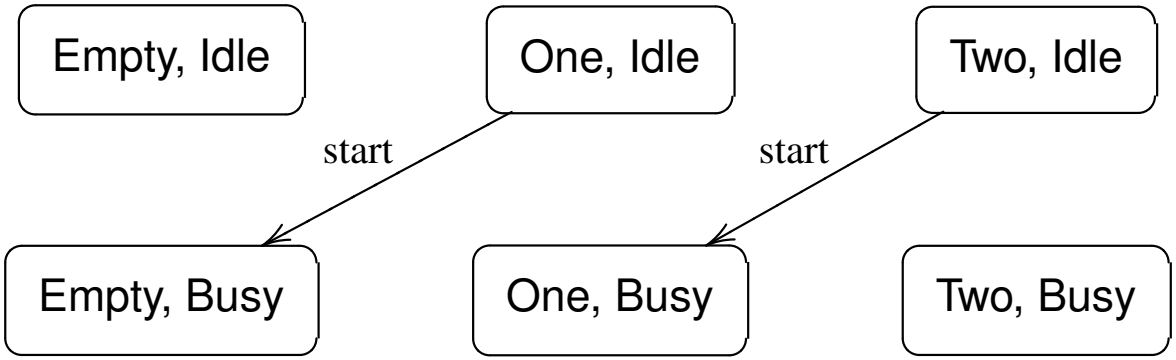
Step 1: The state set consists of all pairs of states.



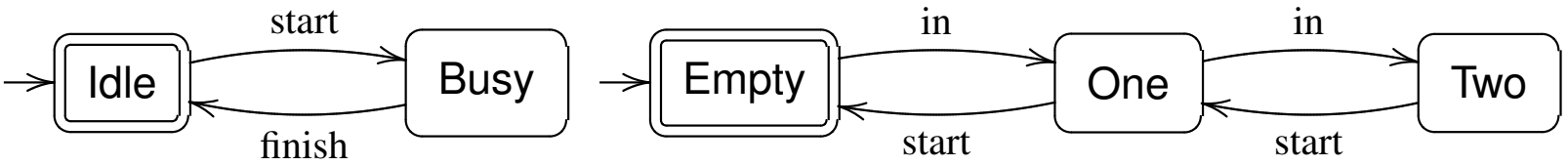
Back to the example



Step 2a: For $a \in \Sigma \cap \Sigma'$ we have $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $q' \xrightarrow{a} p'$. (In this example, $a =$ "start".)

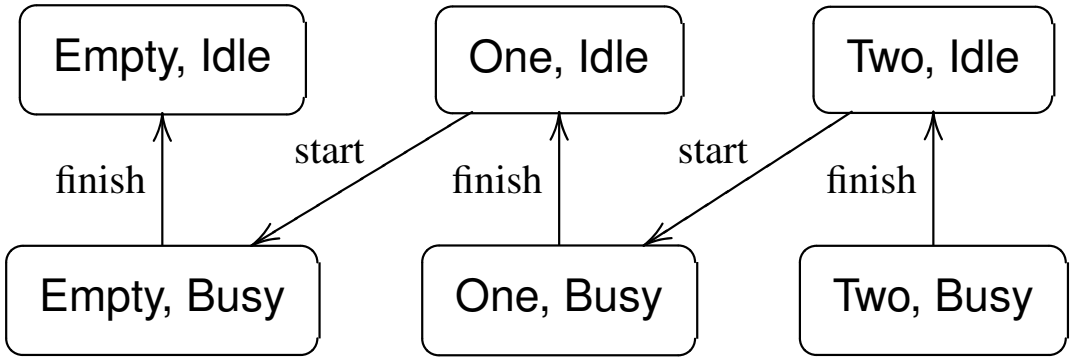


Back to the example

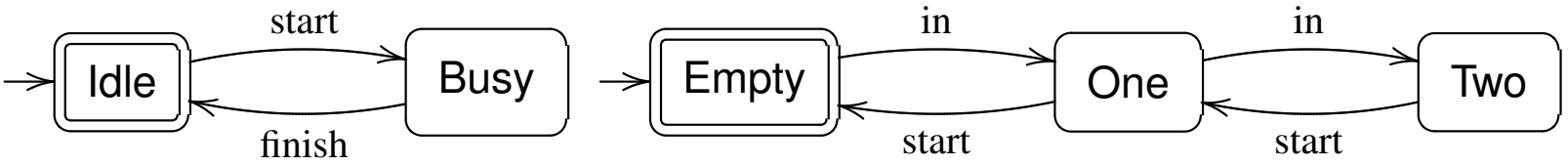


Step 2b: For $a \in \Sigma - \Sigma'$ we have $(q, q') \xrightarrow{a} (p, p')$ if $q \xrightarrow{a} p$ and $q' = p'$.

In the example, $a =$ "finish".

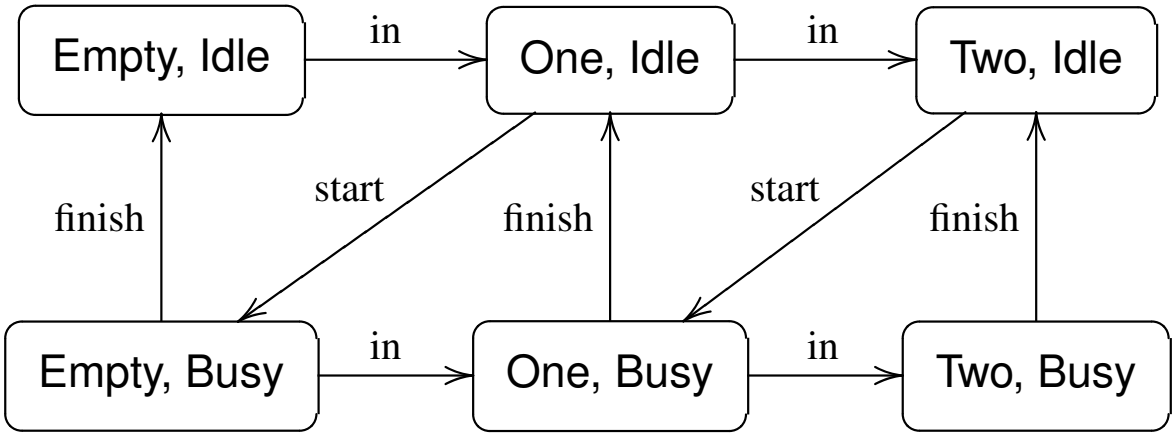


Back to the example

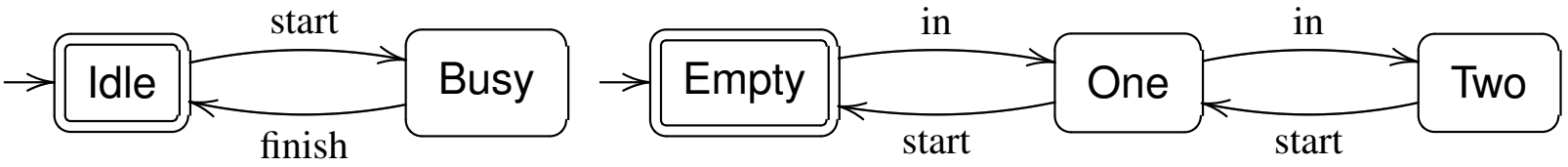


Step 2c: For $a \in \Sigma' - \Sigma$ we have $(q, q') \xrightarrow{a} (p, p')$ if $q = p$ and $q' \xrightarrow{a} p'$.

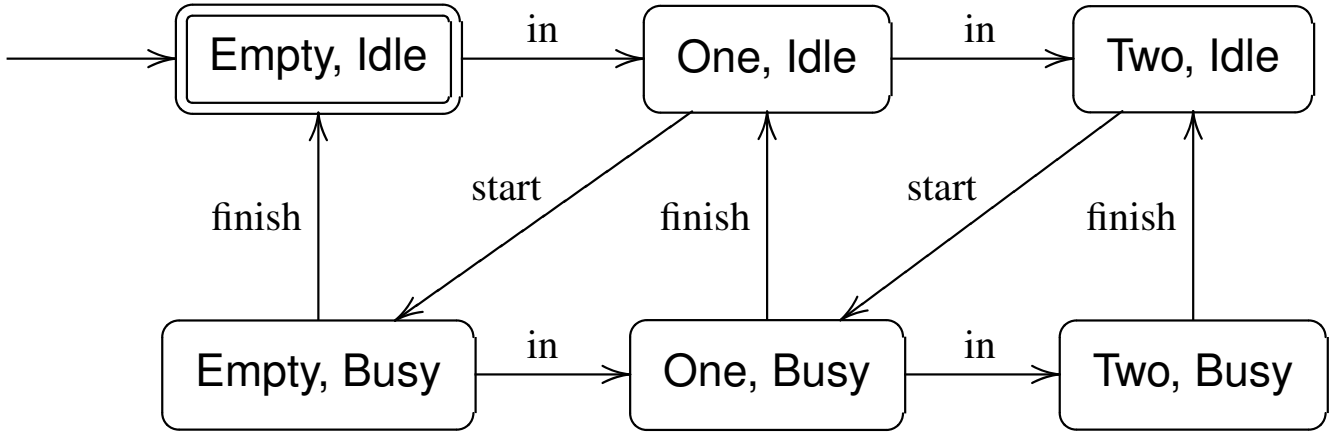
In the example, $a =$ "in".



Back to the example



Step 3: The start state is (q_0, q_0') and the set of accepting states is $F \times F' = \{(q_0, q_0')\}$.



Formal construction

Suppose we are given $A = (Q, \Sigma, \delta, q_0, F)$ and $A' = (Q', \Sigma', \delta', q_0', F')$.

The *synchronous composition* $A \parallel A'$ of A and A' is the automaton $A^{\parallel} = (Q^{\parallel}, \Sigma^{\parallel}, \delta^{\parallel}, q_0^{\parallel}, F^{\parallel})$ where:

- $Q^{\parallel} = Q \times Q'$;
- $\Sigma^{\parallel} = \Sigma \cup \Sigma'$;
- $q_0^{\parallel} = (q_0, q_0')$;
- $F^{\parallel} = F \times F'$,

and $\delta^{\parallel} : Q^{\parallel} \times \Sigma^{\parallel} \to \mathcal{P}(Q^{\parallel})$ is defined by:

$$\delta^{\parallel}((q,q'),a) = \begin{cases} \{(p,p') : p \in \delta(q,a), p' \in \delta(q',a)\} & a \in \Sigma \cap \Sigma' \\ \{(p,q') : p \in \delta(q,a)\} & a \in \Sigma - \Sigma' \\ \{(q,p') : p' \in \delta'(q',a) & a \in \Sigma' - \Sigma \end{cases}$$

Notes

Some observations:

- if $\Sigma = \Sigma'$, then synchronous composition is the same as intersection (which we discussed in Lecture 3);
- the automaton $\mathcal{A} \parallel \mathcal{B}$ is (essentially) the same as $\mathcal{B} \parallel \mathcal{A}$ (the only difference is in the labelling of the states);
- synchronous composition is sometimes called *parallel composition* (hence the symbol $\parallel$).

Language of $A \parallel A'$

Which strings are accepted by $A \parallel A'$?

- the two automata run concurrently (as with the intersection of two automata);
- however, one can have A-transitions without A' changing state, and vice-versa;
- the transitions taken only by A contribute a letter $a \in \Sigma - \Sigma'$ (and similarly for A').

Idea. Suppose that $s \in L(A \parallel A')$:

- deleting all the letters of $\Sigma' - \Sigma$ from s gives a string of $L(A)$;
- deleting all the letters of $\Sigma - \Sigma'$ from s gives a string of $L(A')$.

Projection functions

Definition. If Σ and Σ' are alphabets, we define the *projection functions*

$$P_\Sigma : (\Sigma \cup \Sigma')^* \to \Sigma^* \text{ and } P_{\Sigma'} : (\Sigma \cup \Sigma')^* \to \Sigma'^*$$

by $P_\Delta(s) = s$ with all the occurrences of letters not in Δ removed (for $\Delta = \Sigma$ and $\Delta = \Sigma'$).

Theorem. Suppose that A and A' are nondeterministic automata over the alphabets Σ and Σ' respectively. Then

$$s \in L(A \parallel A') \text{ if and only if } P_\Sigma(s) \in L(A) \text{ and } P_{\Sigma'}(s) \in L(A').$$

Thus

$$L(A \parallel A') = \{w \in (\Sigma \cup \Sigma')^* : P_\Sigma(w) \in L(A) \text{ and } P_{\Sigma'}(w) \in L(A')\}.$$

CO4211/CO7211

Discrete Event Systems

Lecture 8:
Regular expressions

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Searching in a text file

A neat application of finite automata: fast searching in text files:

- search a text file for the name “John Doe”;
- find occurrences with lower case initials: “john doe”, “John doe” and “john Doe”;
- allow a middle name: “John Gilbert Doe” or “John Raymond Doe” for example.

Searching in a text file

Idea: use an automaton!

- express all the possible variations of the name as a language;
- construct an automaton for this language and then run it on the file.

This is the fastest possible way!

- a *deterministic automaton* just requires *one action* per character;
- searching can therefore be done in *linear time* (time proportional to the file size).

Building languages step by step

Which languages can be expressed using (deterministic) automata?

- all 1-element languages (we've seen that).

Important observation. If L and L' are regular, then:

- $\mathrm{L} \cup \mathrm{L}'$ is regular (union);
- $\mathrm{L} \cdot \mathrm{L}'$ is regular (concatenation);
- L^* is regular (Kleene star).

General recipe:

- construct a non-deterministic automaton;
- convert this automaton to a deterministic one.

Idea. We can put these constructions together in a modular fashion!

Building up regular languages

Idea:

- we define a concise notation for regular languages: *regular expressions* (over an alphabet Σ);
- each regular expression r defines a language $L(r) \subseteq \Sigma^*$.

Starting point:

- if $a \in \Sigma \cup \{\epsilon\}$ then a is a regular expression and $L(a) = \{a\}$;
- $\emptyset$ is a regular expression with $L(\emptyset) = \emptyset$.

Building up regular languages

Constructions. If r_1 and r_2 are regular expressions then:

- $r_1 \cdot r_2$ is a regular expression and $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$;
- $r_1 | r_2$ is a regular expression and $L(r_1 | r_2) = L(r_1) \cup L(r_2)$;
- r_1^* is a regular expression and $L(r_1^*) = L(r_1)^*$.

Note. More constructions are possible (such as intersection and complement).

Convention. We use brackets in regular expressions to specify how the expression is built up, for example $(r_1 \cdot r_2) | r_3$.

The "$\cdot$" is often omitted: $(r_1 r_2) | r_3$.

Example: Find "John Doe"

Construct a regular expression to find "John Doe".

- Alphabet: all the characters of the Latin alphabet (upper and lower case) plus ␣ (representing a space), i.e.

$$\Sigma = \{\mathrm{A}, \ldots, \mathrm{Z}, \mathrm{a}, \ldots z, ␣\}.$$

Step 1. Regular expression for the first name:

- the first letter is either j or J $\rightsquigarrow$ regular expression $\mathrm{j}\,|\,\mathrm{J}$ ...
- ... followed by "ohn" $\rightsquigarrow$ regular expression $(\mathrm{j}|\mathrm{J})\mathrm{ohn}$.

Finding “John Doe”, continued

Step 2. Regular expression for the second name:

- D or d followed by “$\mathrm{o}e$” $\rightsquigarrow$ regular expression $(\mathrm{D}\,|\,\mathrm{d})\mathrm{o}e$.

Step 3. First name and last name, every capitalisation:

- regular expression

$$((\mathrm{j}|\mathrm{J})\mathrm{o}\hbar n)\llcorner(\mathrm{d}|\mathrm{D})\mathrm{o}e.$$

Finding “John Doe”, continued

What about an unknown middle name?

- every sequence of letters $\mathrm{a}, \ldots, z, \mathrm{A}, \ldots, \mathrm{Z}$ (no spaces!);
- the regular expression $\alpha = \mathrm{a}|\mathrm{b}|\ldots|z|\mathrm{A}|\mathrm{B}|\ldots|\mathrm{Z}$ stands for “some character”;
- a non-empty sequence of arbitrary characters $\rightsquigarrow$ the regular expression $\alpha\alpha^*$.

Step 4. Putting things together. Either first name + last name or first name + unknown middle name + last name:

$$(\underbrace{(\mathrm{j}|\mathrm{J})\mathrm{ohn}}_{\text{first}}\,\underbrace{(\mathrm{d}|\mathrm{D})\mathrm{oe}}_{\text{last}}) \mid (\underbrace{(\mathrm{j}|\mathrm{J})\mathrm{ohn}}_{\text{first}}\, \underbrace{\alpha\alpha^*}_{\text{middle}}\, \underbrace{(\mathrm{d}|\mathrm{D})\mathrm{oe}}_{\text{last}})$$

From regular expressions to deterministic automata

Top-down construction of an automaton.

Given a regular expression r, we construct an automaton $A(r)$ that accepts $L(r)$:

1. if $r = a$ for $a \in \Sigma$ or $r = \emptyset$, construct an automaton that accepts $L(r)$.
   Otherwise:
   - if $r = r_1 \cdot r_2$, build $A(r_1)$ and $A(r_2)$ and then return an automaton for $L(r_1) \cdot L(r_2)$;
   - if $r = r_1 | r_2$, build $A(r_1)$ and $A(r_2)$ and then return an automaton for $L(r_1) \cup L(r_2)$;
   - if $r = r_1^*$, build $A(r_1)$ and then return an automaton for $L(r_1)^*$.
2. Make the resulting automaton deterministic.

From regular expressions to deterministic automata

Why does this work?

- Taking the regular expression apart produces smaller expressions;
- eventually, we are left with the base cases $r = a$ (with $a \in \Sigma$) and/or $r = \emptyset$;
- all the constructions can be carried out building up the automata.

From regular expressions to automata

Theorem. For every regular expression r we can construct a (deterministic) automaton A such that $\mathrm{L}(\mathrm{A}) = \mathrm{L}(\mathrm{r})$.

Discussion.

- Description of the *system behaviour* as a *regular expression* gives an *automaton*.
- We just write down the expression – the rest is automatic!
- But does this only apply in special cases?

How about the converse? Can we always capture the language $\mathrm{L}(\mathrm{A})$ of an automaton A as $\mathrm{L}(\mathrm{r})$ for some regular expression r?

Regular expressions are equivalent to automata

Theorem. (*Stated without proof*) For every non-deterministic finite automaton A there is a regular expression r with $\mathrm{L}(\mathrm{r}) = \mathrm{L}(\mathrm{A})$.

Good news . . .

- for every system whose behaviour we can describe by a regular expression, we can construct an automaton for that system;
- the behaviour of systems modelled by finite automata can be described by regular expressions.

But . . .

- if the behaviour of a system cannot be described by a regular expression, then that system cannot be modelled by a finite automaton.

Regular expressions: some examples

Let $\Sigma = \{a, b\}$.

1. $L = \{w \in \Sigma^* : w$ has even length$\}$;
2. $L = \{w \in \Sigma^* :$ every a in w is followed by a $b\}$;
3. $L = \{w \in \Sigma^* : w$ contains an even number of a's$\}$;
4. $L = \{w \in \Sigma^* : w$ does not contain $ba\}$.

Regular expressions:

1. $r = ((a|b)(a|b))^*$ or $r = (aa|ab|ba|bb)^*$;
2. $r = b^*|(b^*abb^*)^*$ or $r = (b|ab)^*$;
3. $r = b^*|(b^*ab^*ab^*)^*$ or $r = (b|ab^*a)^*$;
4. $r = a^*b^*$.

Making life easier

Some more constructions:

- if r is a regular expression then so is $r?$ and $L(r?) = \{\epsilon\} \cup L(r)$;
- if r is a regular expression then so is $r\{n\}$ (where $n \geqslant 0$) and $L(r\{n\}) = \underbrace{L(r) \cdots L(r)}_{n \text{ times}}$;
- if r is a regular expression then so is $r+$ and $L(r+) = L(r) \cdot L(r^*)$.

Note. All of these new constructions can be expressed in terms of the starting points and the original constructions "$\_ \cdot \_$", "$\_ \mid \_$" and "$\_^*$"; so we still have methods for building up the automata.

Making life easier

Finding "John Doe":

- an easier regular expression: $\underbrace{(\mathrm{j}|\mathrm{J})\mathrm{ohn}}_{\text{first}}\quad\underbrace{((\sqcup\alpha+)?)}_{\text{middle?}}\quad\underbrace{\sqcup(\mathrm{d}|\mathrm{D})\mathrm{oe}}_{\text{last}}$;
- regular expressions of this general form are implemented in the UNIX-command `grep`.

CO4211/CO7211

Discrete Event Systems

Lecture 9:
Optimising automata

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

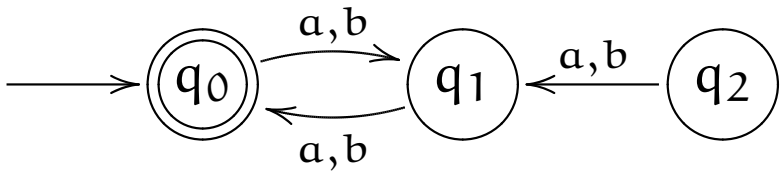
Optimising automata

Question. Given an automaton A, can it be simplified?

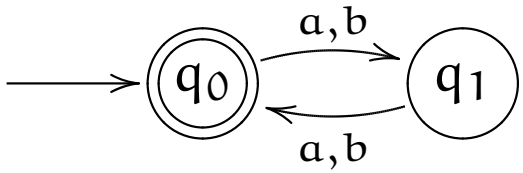
Question. What do we mean by "simplified" here?

Answer. We want to minimize the number of states.

Example.



Obviously, state q_2 can be removed in this example without affecting the language accepted by the automaton:



Unreachable states.

In general. A state is called *unreachable* if there is no path to it from the start state.

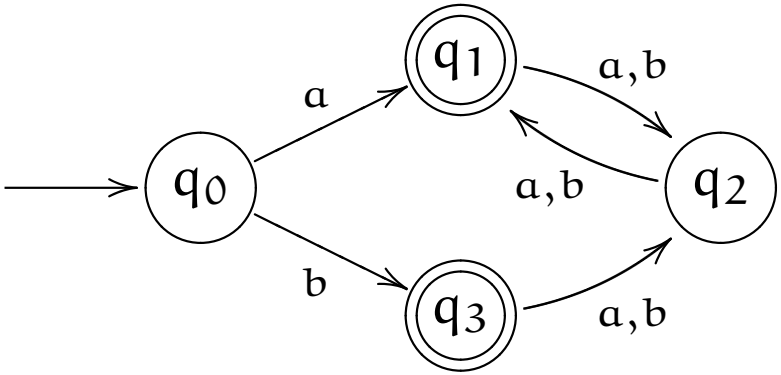
Observation. It is straightforward to determine the unreachable states in an automaton A with inputs Σ. We start at the start state q_0 and systematically see which states we can reach by following transitions. Once we have exhausted all the states we can reach (i.e. once we have generated a set S of states containing q_0 such that $\delta(p, a) \in S$ for all $p \in S$ and all $a \in \Sigma$), then we have found all the reachable states in A . The remaining states are unreachable and can be deleted.

Theorem. If A' is the automaton that results from removing all unreachable states from A (and every transitions involving them), then $L(A) = L(A')$.

Optimising automata

Another example.

Consider the automaton $\mathcal{A}$ shown below:



Observation:

- the states q_1 and q_3 exhibit the same behaviour: a word is accepted starting from q_1 if and only if it is accepted starting from q_3;
- so, if we "merge" the states q_1 and q_3 into a single state, we will get an automaton that accepts the same language as $\mathcal{A}$.

Minimization of deterministic automata

Goal. Collapse all "equivalent" states to a single state. This will give a smaller automaton accepting the same language.

Question. When are two states equivalent? This may not be immediately obvious.

Potentially easier question. When are two states *not* equivalent?

Observation. Two states p and q are not equivalent if there is a string w such that one of the two states $\delta(\mathrm{p}, w)$ and $\delta(\mathrm{q}, w)$ is an accept state and the other is not.

Minimization

Two states p and q are not equivalent if there is a string w such that one of $\delta(p,w)$ and $\delta(q,w)$ is an accept state and the other is not.

1. if $w = \epsilon$ then one of the states $\delta(p,\epsilon) = p$ and $\delta(q,\epsilon) = q$ is an accept state and the other is not;
2. if $w \neq \epsilon$, then $w = av$ for some $a \in \Sigma$ and some $v \in \Sigma^*$. Let

$$p' = \delta(p,a) \text{ and } q' = \delta(q,a).$$

Then:

$$\delta(p,w) = \delta(p,av) = \delta(p',v);\ \ \delta(q,w) = \delta(q,av) = \delta(q',v).$$

Given that one of $\delta(p,w)$ and $\delta(q,w)$ is an accept state and the other is not, we must have that one of $\delta(p',v)$ and $\delta(q',v)$ is an accept state and the other is not.

Minimization

Developing an algorithm. Two states p and q are not equivalent if either:

- one of the states p and q is an accept state and the other is not;
- there exists $a \in \Sigma$ such that the states

$$p' = \delta(p, a) \text{ and } q' = \delta(q, a)$$

  are not equivalent.

Idea. Mark all the pairs of states that are not equivalent; then the unmarked pairs must be equivalent.

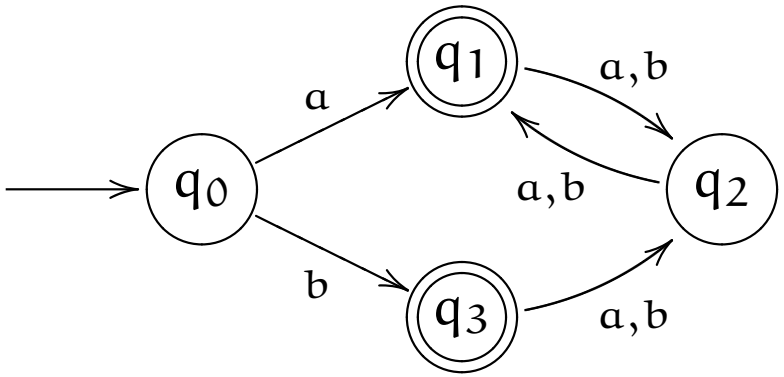
Minimization

The algorithm.

- *Step 1:* remove all unreachable states;
- *Step 2:* if $q \in F$ is an accept state and $q' \notin F$ is not an accept state, then q and q' are *not* equivalent;
- *Step 3:* at each stage, for each pair of equivalent states q and p, if $q \xrightarrow{a} q'$ and $p \xrightarrow{a} p'$ (for some $a \in \Sigma$), and if q' and p' are not equivalent, then q and p are *not* equivalent;
- *Step 4:* repeat Step 3 until no new pairs of inequivalent states are found;
- *Step 5:* merge each pair of equivalent states into a single state.

Finding non-equivalent states

Lining it up in a table.



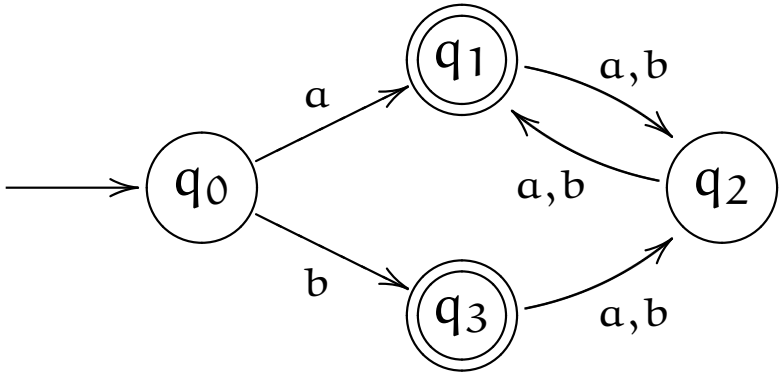
	q_0	q_1	q_2	q_3
q_0	–	✓		✓
q_1		–	✓	
q_2			–	✓
q_3				–

We mark all the pairs of states which we know are not equivalent to begin with, i.e. all the pairs (q, p) where q is accepting and p is not accepting (or vice-versa).

Finding non-equivalent states

Second round: We look for more non-equivalent pairs

of states using transitions.



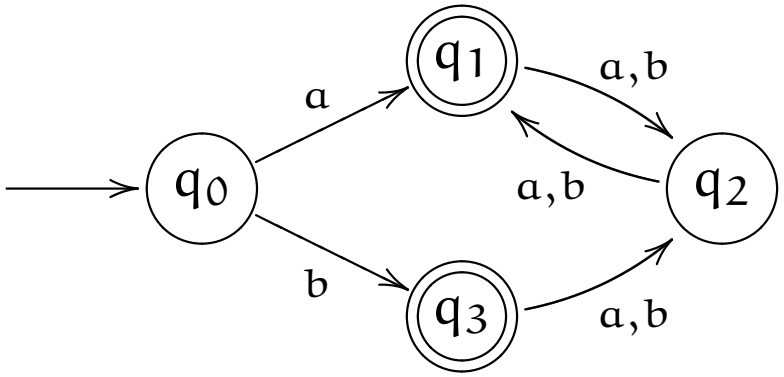
	q_0	q_1	q_2	q_3
q_0	–	✓		✓
q_1		–	✓	
q_2			–	✓
q_3				–

Here we are marking pairs (p, q) that lead to non-equivalent states *via the same symbol*: we mark (p, q) if $p \xrightarrow{x} p'$ and $q \xrightarrow{x} q'$ (for some x), where the pair (p', q') is already marked.

Finding non-equivalent states

Second round: We look for more non-equivalent pairs

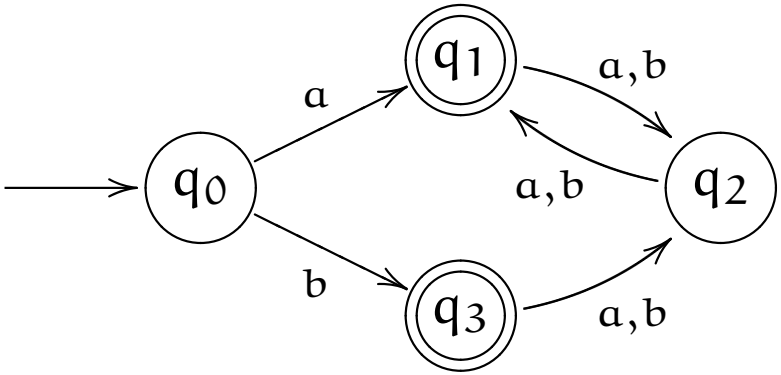
of states using transitions.



	q_0	q_1	q_2	q_3
q_0	–	✓		✓
q_1		–	✓	
q_2			–	✓
q_3				–

In our example (above) there are no more states to mark! Now all the *marked* pairs of states are *non-equivalent* and all the *non-marked* pairs are *equivalent*. (In general, this takes more than two rounds.)

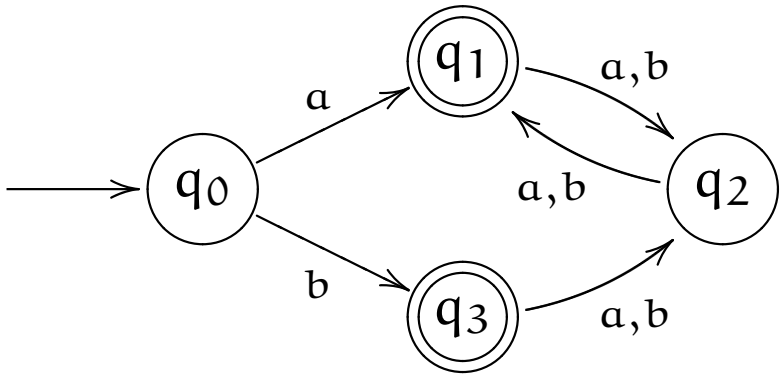
The minimal automaton



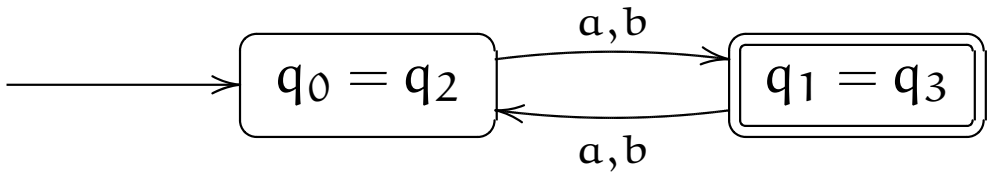
Construct the minimal automaton: we collapse the *unmarked* pairs $(q_0, q_2), (q_1, q_3)$:

- if the start state is merged with another state, this gives the new start state;
- if two accepting states are merged, then the resulting state is accepting;
- if two non-accepting states are merged, then the resulting state is not accepting.

The minimal automaton



The resulting automaton (shown below) is the *minimal automaton* constructed from the original automaton.



Minimization algorithm

We restate our algorithm for minimizing an automaton in terms of the table used in the above example.

Given: a deterministic automaton $A = (Q, \Sigma, \delta, q_0, F)$:

1. remove all the unreachable states from A;
2. construct a table with all the pairs (p, q) of distinct states;
3. mark all the pairs (p, q) where $p \in F$ and $q \notin F$ (or vice-versa);

Minimization algorithm (continued)

4. repeat
   - for all unmarked pairs (p, q) and all $a \in \Sigma$, check whether $p \xrightarrow{a} p'$ and $q \xrightarrow{a} q'$ for *marked* (p', q');
   - if "yes" then mark (p, q);

   until no more states are being marked;
5. collapse all the pairs of states that are *not* marked;
6. this gives the *minimal automaton*.

Notes:

- we only need to consider (p, q) *or* (q, p);
- pairs (p, p) will *never* be marked.

Summary

In fact, the automaton we obtain in this way really is minimal in the following sense:

Theorem. For each regular language L the minimal automaton $\mathcal{A}$ accepting L is uniquely defined (up to the labelling of the states).

As a consequence we have:

Corollary. Given two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ accepting the same language, the minimization algorithm given above will yield the same results (up to the labelling of the states) when applied to $\mathcal{A}_1$ and to $\mathcal{A}_2$.

Given this, we have an algorithm to determine whether or not two finite automata $\mathcal{A}_1$ and $\mathcal{A}_2$ accept the same language (we minimize the two automata and see if the results are equal).

CO4211/CO7211

Discrete Event Systems

Lecture 10:
From automata to Petri nets

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Which languages are regular?

So far we have seen:

- some languages which can be accepted (or event systems that can be modelled) by finite automata;
- these languages correspond to regular expressions;
- various constructions for combining automata.

Question. How do we know whether some language can be recognised by an automaton?

Which languages are regular?

Why is this interesting? We don't want to waste time looking for an automaton that doesn't exist.

Possible answer. If the language is not regular, then we cannot represent it by a regular expression.

This doesn't really help. We can't try *all* regular expressions!

A non-regular language

Let $\Sigma = \{a, b\}$. Can we construct an automaton for the language $L = \{a^n b^n : n \geqslant 0\}$?

Problem. This seems to require *counting*:

- we have to *remember* the number of a's and/or b's ...
- ... and we only have a fixed memory (a finite number of states);
- so we cannot count up to an arbitrarily large number.

It turns out that the language $L = \{a^n b^n : n \geqslant 0\}$ *is not regular*.

Understanding regular languages: we must also understand their *limitations.*

Another non-regular language

Let $\Sigma = \{a, b\}$ and K be the language

$$\{w \in \Sigma^* : \sharp_a(w) = \sharp_b(w)\}.$$

How do we show that K is not regular?

Let J be the regular language

$$\{a^m b^n : m, n \geqslant 0\}.$$

If K were regular then $J \cap K$ would be regular. But $J \cap K$ is the language

$$\{a^n b^n : n \geqslant 0\}$$

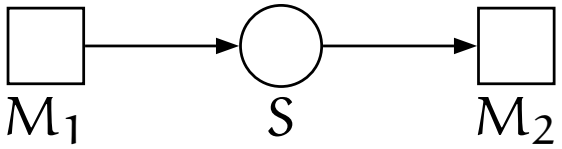
which we commented above is not regular.

So K is not regular.

A manufacturing example

Consider the following situation:

- We have two machines M_1 and M_2.
- M_1 produces an item which it puts into a storage area S.
- M_2 removes an item from S which it wraps and passes on for dispatch.

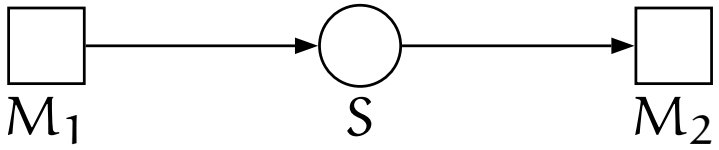


Let us just concentrate on M_1 putting items into S and M_2 retrieving items from S.

Let a denote an action of M_1 (i.e. M_1 placing an item in S) and let b denote an action of M_2 (i.e. M_2 retrieving an item from S).

A manufacturing example

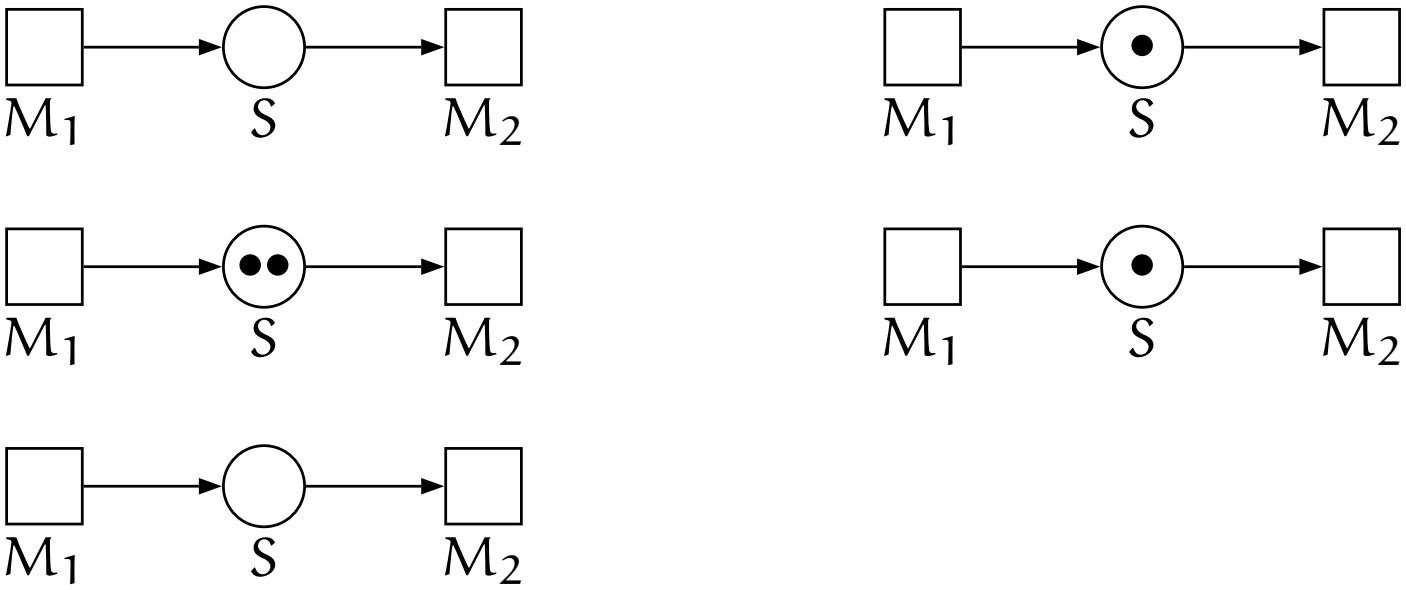
Let $\Sigma = \{a, b\}$. Which elements of Σ^* represent valid sequences of actions of the system?



Let us assume that there are no items in S to begin with and that we want to end up with no items in S at the end.

A manufacturing example

The string $aabb$ is valid: we have the following sequence in the system:



The string $aababb$ is also valid but the strings aba and $abba$ are invalid.

A manufacturing example

We see that, for a string w to be valid, we must have:

(i) the number $\sharp_{\mathrm{a}}(w)$ of occurrences of a in w equals the number of occurrences $\sharp_{\mathrm{b}}(w)$ of b in w;

(ii) at each stage the number of occurrences of b to date does not exceed the number of occurrences of a to date.

The first condition (i) says that every item which M_1 has put into S must be cleared by M_2.

The second condition (ii) says that M_2 can only clear an item from S if M_1 has deposited one there which has not yet been cleared by M_2.

A manufacturing example

(i) the number $\sharp_{\mathrm{a}}(w)$ of occurrences of a in w equals the number of occurrences $\sharp_{\mathrm{b}}(w)$ of occurrences of b in w;

(ii) at each stage the number of occurrences of b to date does not exceed the number of occurrences of a to date.

We could rephrase condition (ii) as follows:

(ii) If v is a prefix of w then $\sharp_{\mathrm{b}}(v) \leqslant \sharp_{\mathrm{a}}(v)$.

(We say that v is a *prefix* of w if $w = vu$ for some string u.)

We see that the language of valid strings is:

$$\{w \in \Sigma^* : \sharp_{\mathrm{a}}(w) = \sharp_{\mathrm{b}}(w) \text{ and if } v \text{ is a prefix of } w \text{ then } \sharp_{\mathrm{b}}(v) \leqslant \sharp_{\mathrm{a}}(v)\}.$$

A manufacturing example

Let L be the language

$$\{w \in \Sigma^* : \sharp_a(w) = \sharp_b(w) \text{ and if } v \text{ is a prefix of } w \text{ then } \sharp_b(v) \leqslant \sharp_a(v)\}.$$

This language is also not regular.

Again, let J be the regular language

$$\{a^m b^n : m, n \geqslant 0\}.$$

If L were regular then $J \cap L$ would be regular. But $J \cap L$ is the language

$$\{a^n b^n : n \geqslant 0\}$$

which we saw above is not regular.

So L is not regular.

Non-Regular Languages

In a regular language, if there are "enough" words of the form xy^iz then **all** words of the form xy^iz must be in the language.

This is called *the pumping lemma* and will not be covered. It is a strong tool for proving that languages are not regular.

We now turn to study a model that captures also non-regular languages.

Events and conditions

We need a more general model to capture the last manufacturing example.

Basic idea. *Events* depend on, and change, *conditions*.

Example. Processing production requests.

Events:

- order arrives;
- start producing item;
- finish producing item;
- deliver item.

Conditions:

- workshop is waiting;
- order is waiting;
- workshop is busy;
- order is processed.

Events and conditions

Events can *depend on* conditions being met; for example:

- *start processing* should only happen when the conditions *workshop waiting* and *order waiting* are true.

Events can *change* conditions; for example:

- after *start processing*, the condition *workshop busy* is true.

Graphical formalism

Graphical notation.

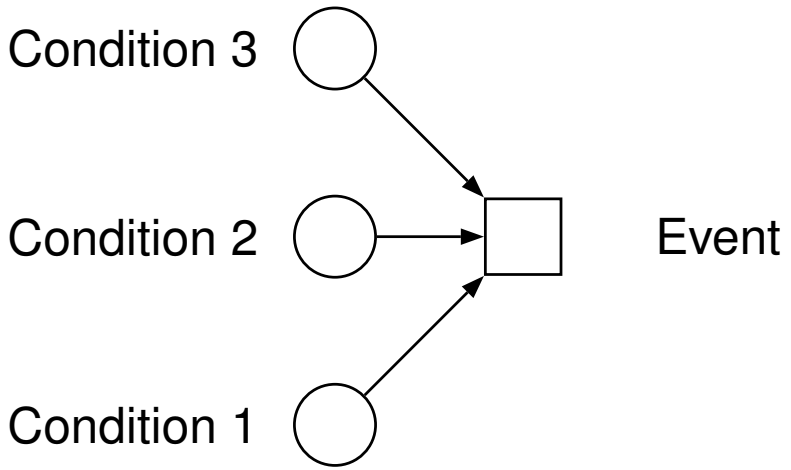
- Event: □

  (Transition)

- Condition: ◯

  (Place)

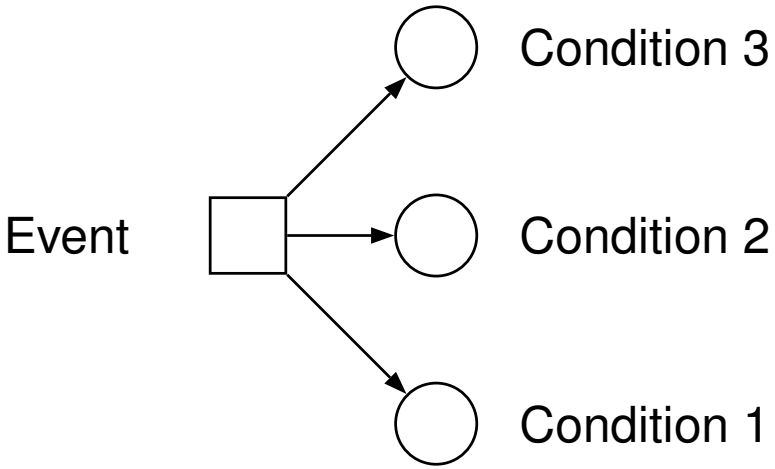
Event depending on conditions.



(*Preconditions* of the event.)

Graphical formalism

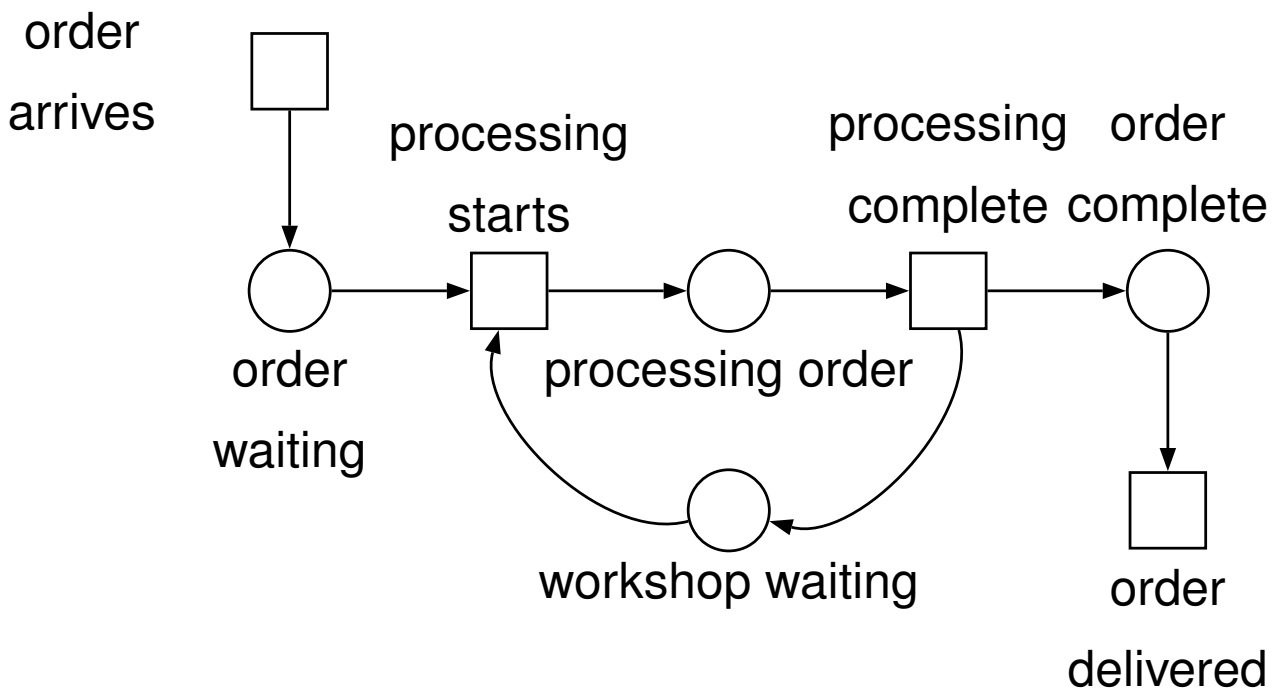
Event changing conditions.



(*Postconditions* of the event.)

These are components of a *Petri net*.

Putting things together



- The event (□) *processing starts* depends on conditions (◯) *order waiting* and *workshop waiting.*
- The event (□) *processing complete* makes conditions *order complete* and *workshop waiting* true.

CO4211/CO7211

Discrete Event Systems

Lecture 11:
Petri nets - the basics

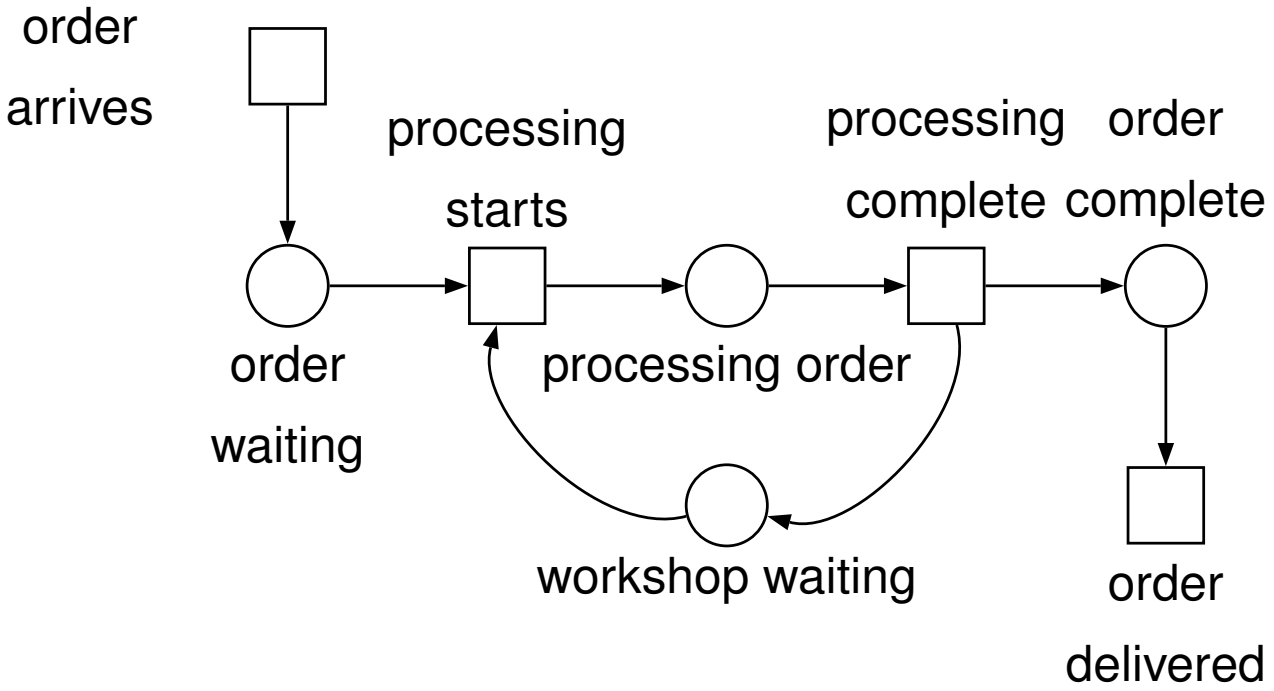
Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Putting things together



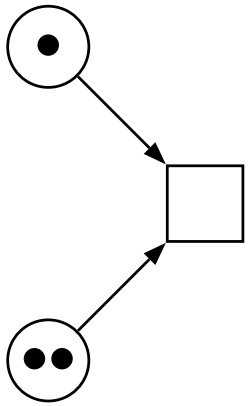
- The event (□) *processing starts* depends on conditions (◯) *order waiting* and *workshop waiting.*
- The event (□) *processing complete* makes conditions *order complete* and *workshop waiting* true.

Dynamics of the formalism

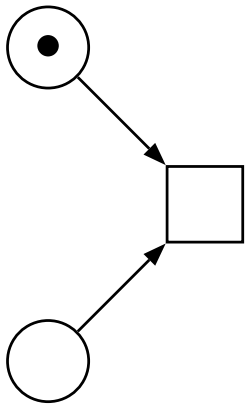
Transitions of Petri net graphs.

- Make some conditions true by putting *tokens* into the circles.
- An event is *enabled* if all its preconditions contain at least one token.

Enabled event.



Disabled event.



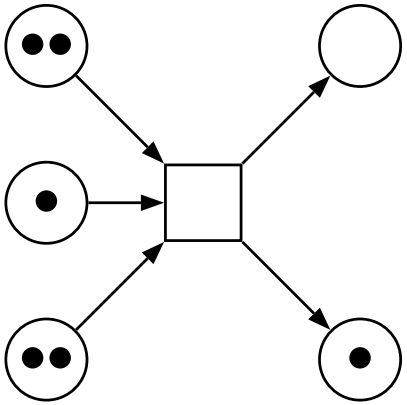
Enabled events can be executed in the next step – tokens are removed from the preconditions when this happens.

Firing a transition

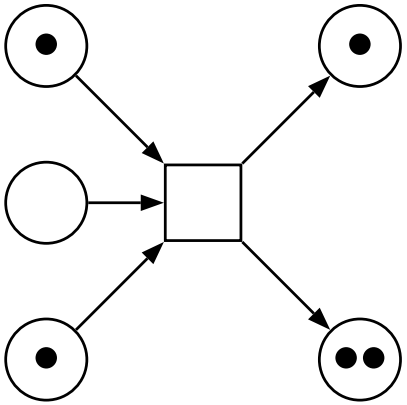
Events change the system state: this is called *firing*.

- Remove one token from each precondition of the event.
- Put one token into each postcondition of the event.

Before firing.



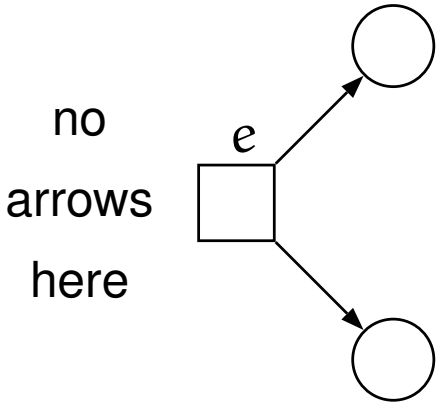
After firing.



Note. In general, there can be *more than one* token in a place.

Special cases

Events without preconditions.

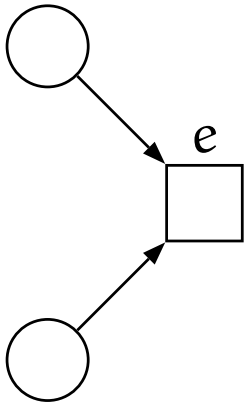


Event e is always enabled, i.e. it can happen at any time.

Events of this type are called *sources*.

Special cases

Events without postconditions.



no arrows here

Event e does not change the conditions, i.e. it has no consequences.

Events of this type are called *sinks*.

Tokens as resources

Example.

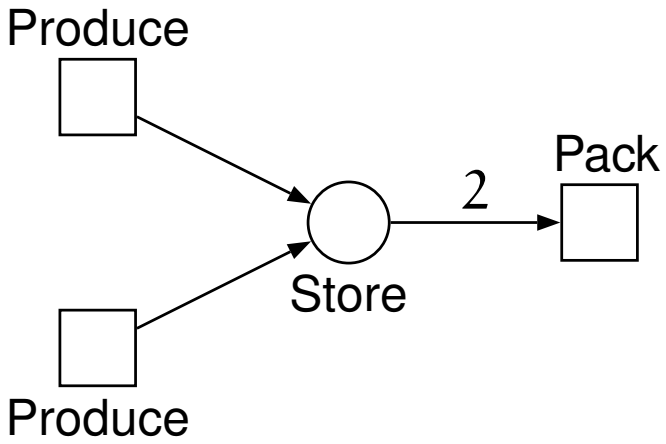
- Two machines M_1 and M_2 continuously produce cans and put them onto a conveyor.
- Once two cans are produced, they are packaged together by a machine P.
- There is no requirement that that the cans which are packaged together come from different machines.

Think of *tokens* as *resources*.

- One token here represents one can that is being produced.
- How can we model the fact that P needs *two* cans?

Tokens as resources: preconditions

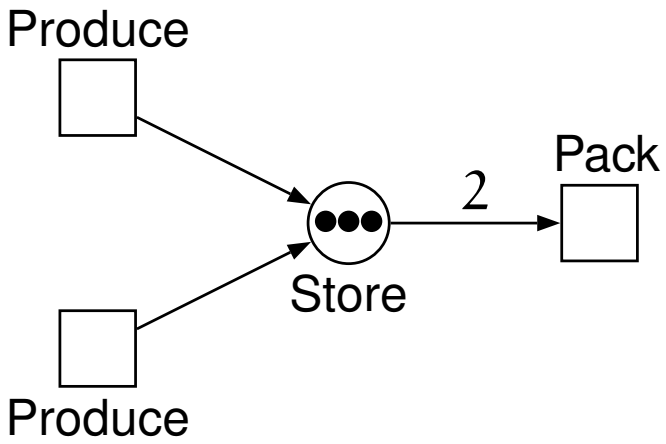
Weighted edges determine the preconditions of events.



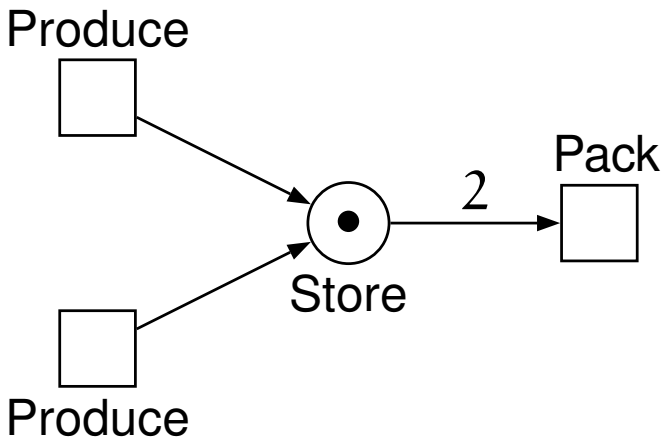
- Pack is enabled if and only if there are at least 2 tokens in Store.
- After Pack fires, both the tokens are removed.
- We place a number above the arrow to signify this.

Tokens as resources: preconditions

Before firing Pack.



After firing Pack.



Tokens as resources: postcondition

Example. Unpacking packs of two items.

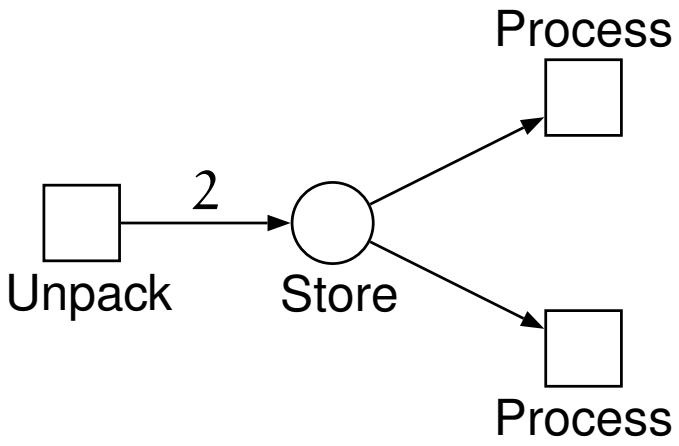
- A machine U receives packages consisting of two items and unpacks them.
- The items are put on a conveyor and then processed by two machines M_1 and M_2 processing one item at a time.
- There is no requirement that the contents are processed by any particular machine.

"Tokens as resources" idea.

- Each unpacking event produces two tokens.
- These tokens can be consumed by M_1 and M_2 independently.

Tokens as resources: postcondition

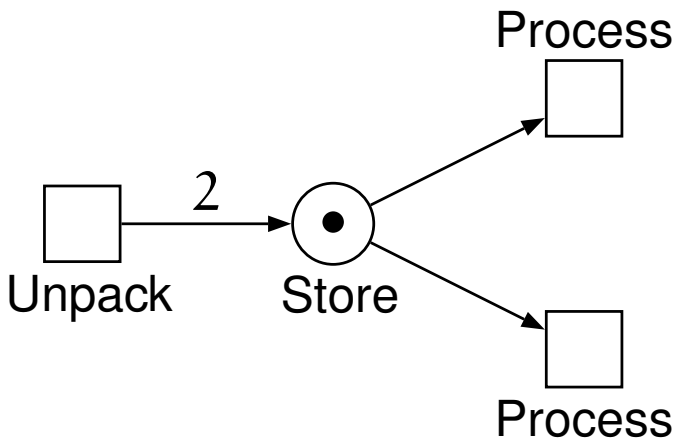
Weighted edges affect the postconditions of events.



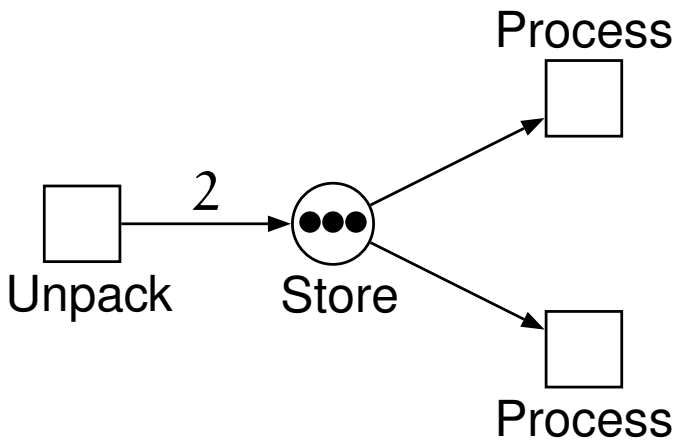
- Firing of Unpack creates two tokens.
- Again, we put a weight above the arrow to denote this.

Tokens as resources: postcondition

Before firing Unpack.



After firing Unpack.



Petri nets: basic definition

A *Petri net* is a quadruple $N = (P, T, F, \omega)$ where:

- P is a finite set of *places* (the *conditions*);
- T is a finite set of *transitions* (the *events*);
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* of the net;
- $\omega : F \to \mathbb{N}$ is the *weight function*.

For convenience, we have the convention that $\omega(x, y) = 0$ if $(x, y) \notin F$.

Petri nets: basic definition

The flow relation: typical elements.

- $(\mathrm{p}, \mathrm{t}) \in \mathrm{P} \times \mathrm{T}$: this connects place p with transition t;
- $(\mathrm{t}, \mathrm{p}) \in \mathrm{T} \times \mathrm{P}$: this connects transition t with place p.

The weight function.

- Weights of arcs of the form (p, t) determine the preconditions (e.g. “Packing”).
- Weights of arcs of the form (t, p) determine the postconditions (e.g. “Unpacking”).

Presets and postsets

Suppose that $N = (P, T, F, \omega)$ is a Petri net.

Preset of $x \in P \cup T$: the elements connected to x by an *incoming arrow*.

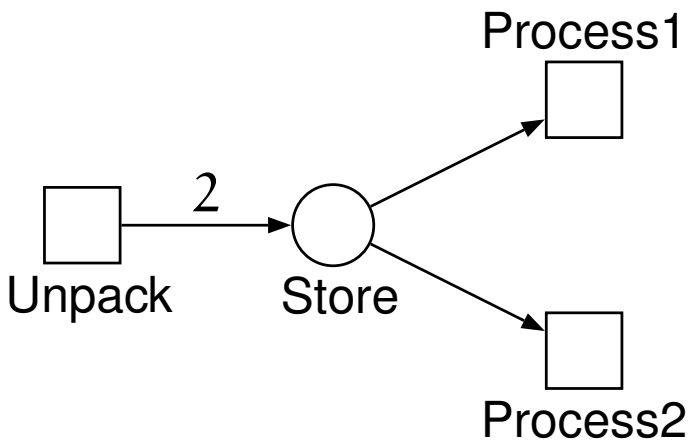
$$\bullet x = \{y \in P \cup T : (y, x) \in F\}.$$

Postset of $x \in P \cup T$: the elements connected to x by an *outgoing arrow*.

$$x\bullet = \{y \in P \cup T : (x, y) \in F\}.$$

Presets and postsets

Example.



$$\bullet\text{Store} = \{\text{Unpack}\}$$

$$\text{Store}\bullet = \{\text{Process1}, \text{Process2}\}$$

$$\bullet\text{Process1} = \{\text{Store}\}$$

$$\text{Process1}\bullet = \emptyset$$

Of course, in a full example, there would probably be other places and transitions associated with those shown above.

Dynamics of a Petri net

Current state:

- true conditions at a given time;
- each place can have more than one token.

Definition. Suppose that $N = (P, T, F, \omega)$ is a Petri net. A *marking* of N is a function $\mu : P \to \mathbb{N}$.

A transition $t \in T$ is *enabled* under a marking μ, if $\mu(p) \geqslant \omega(p, t)$ for all $p \in {\bullet}t$.

Idea.

- *Marking*: this gives the number of tokens at each place.
- *Enabled*: there are sufficient tokens at all the incoming places.

Firing relation

Marking.

This represents the current state of the system.

Execution of a transition.

- This can happen only if the transition is enabled, i.e. only if there are sufficient tokens in its preset.
- Remove the appropriate tokens from the preset.
- Put the appropriate tokens in the postset.

Firing relation

Definition.

Let M be the set of all markings for a net $N = (P, T, F, \omega)$. The *next state function* $\delta : M \times T \to M$ is defined as follows:

- $\delta(\mu, t)$ is defined if t is enabled under μ.
- If $\delta(\mu, t)$ is defined, then $\delta(\mu, t)$ is the marking μ' defined by

$$\text{Forall } p \in P \,.\, \mu'(p) = \mu(p) - \omega(p,t) + \omega(t,p).$$

We write $\mu \xrightarrow{t} \mu'$.

Remember. Our convention is that, if $(x, y) \notin F$, then $\omega(x, y) = 0$.

Comparison with automata

Petri net dynamics.

Sequence of transitions.

$$\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \mu_2 \cdots \xrightarrow{t_n} \mu_n$$

Comparision with automata.

- The Petri net is a more flexible model – it expresses causality nicely.
- *Behaviours* are *sequences of transitions* – so we have languages here too!

CO4211/CO7211

Discrete Event Systems

Lecture 12:
Examples of Petri nets

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Petri nets: basic definitions

A *Petri net* is a quadruple $N = (P, T, F, \omega)$ where:

- P is a finite set of *places* (the *conditions*);
- T is a finite set of *transitions* (the *events*);
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* of the net;
- $\omega : F \to \mathbb{N}$ is the *weight function*.

For convenience, we let $\omega(x, y) = 0$ if $(x, y) \notin F$.

Definition. Suppose that $N = (P, T, F, \omega)$ is a Petri net. A *marking* of N is a function $\mu : P \to \mathbb{N}$.

A transition $t \in T$ is *enabled* under a marking μ if $\mu(p) \geqslant \omega(p, t)$ for all $p \in {\bullet}t$.

Firing relation

Definition. Let M be the set of all markings for a net $N = (P, T, F, \omega)$. The *next state function* $\delta : M \times T \to M$ is defined as follows:

- $\delta(\mu, t)$ is defined if t is enabled under μ.
- If $\delta(\mu, t)$ is defined, then $\delta(\mu, t)$ is the marking μ' defined by

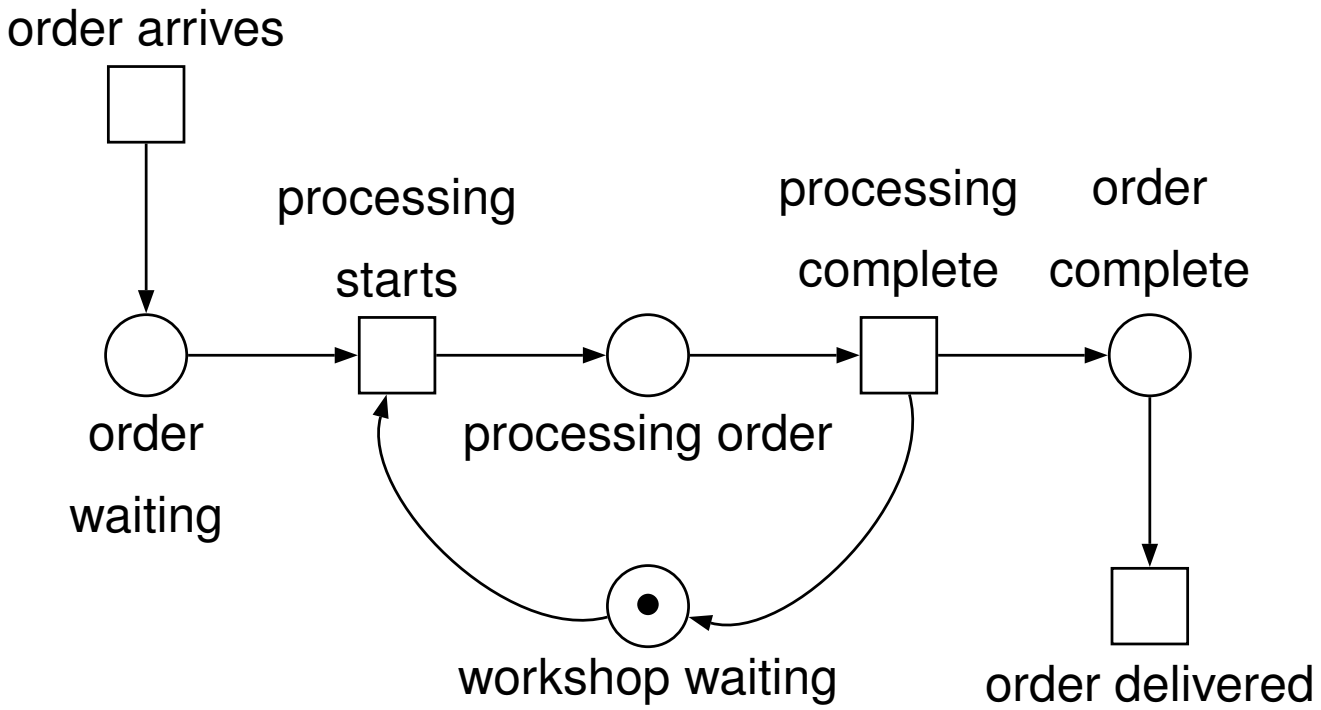
$$\text{Forall } p \in P \,.\, \mu'(p) = \mu(p) - \omega(p,t) + \omega(t,p).$$

  We write $\mu \xrightarrow{t} \mu'$.

Remember. Our convention is that, if $(x, y) \notin F$, then $\omega(x, y) = 0$.

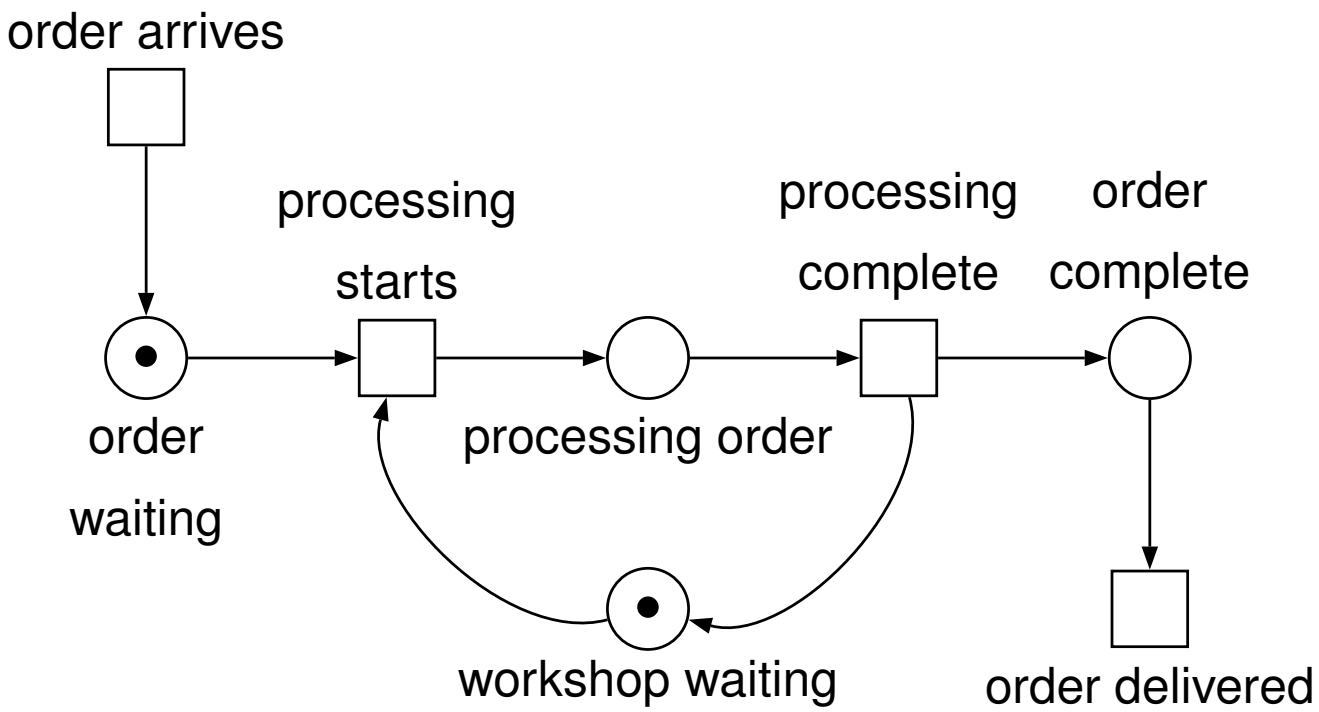
An example

Consider the following example from Lecture 11. We start with the initial marking of the net shown below (note that there is a token in “workshop waiting”):



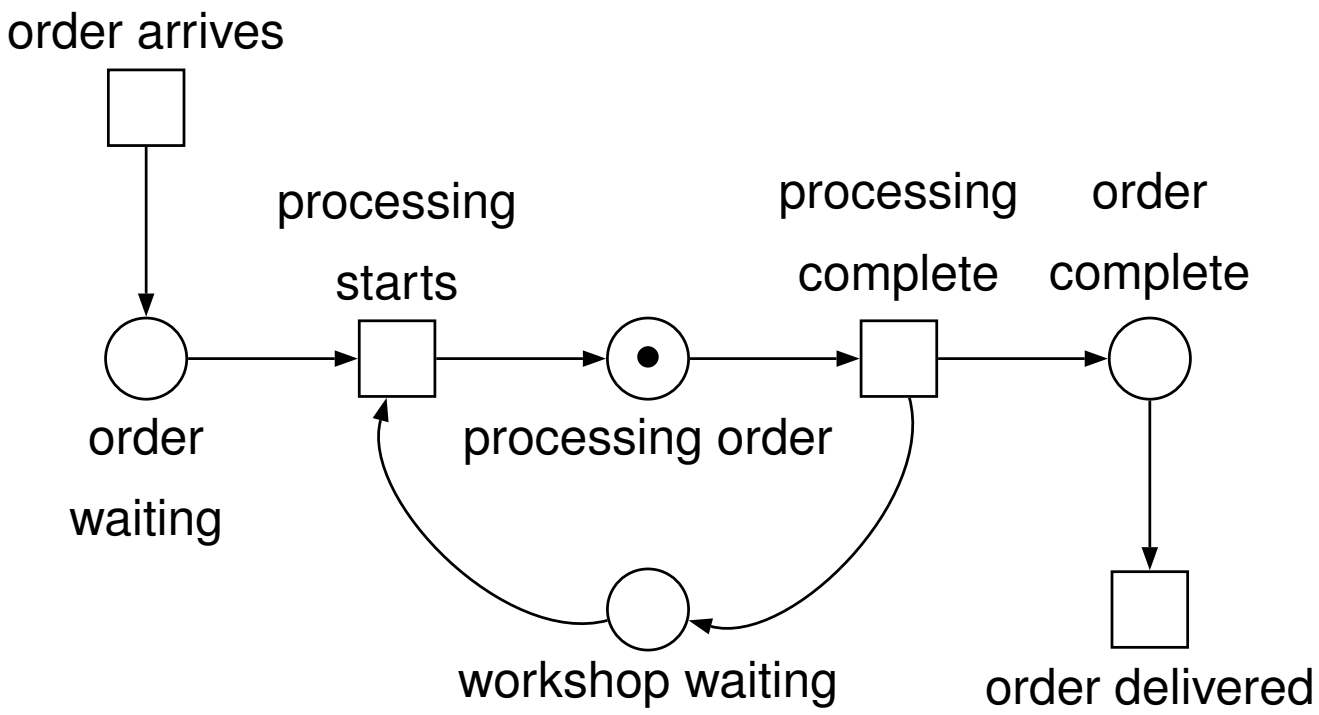
An example

If the event “order arrives” fires, a token appears in the place “order waiting”:



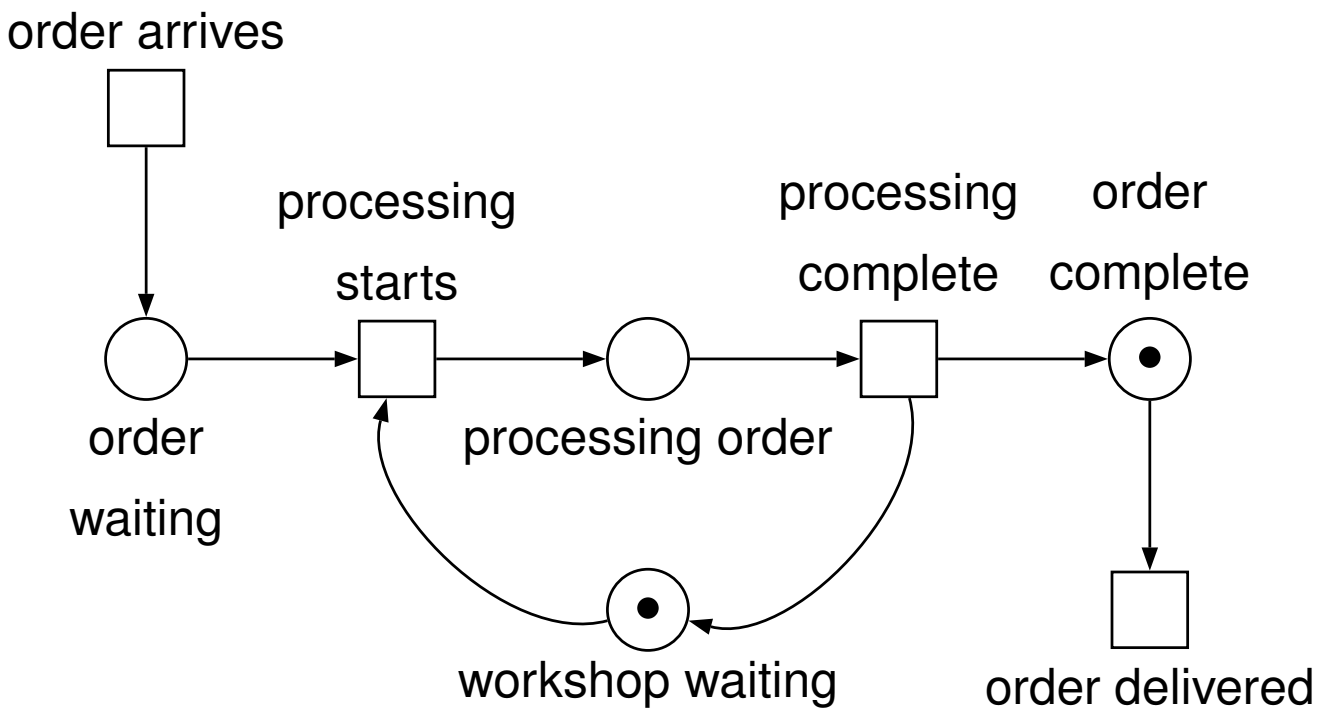
An example

The event “processing starts” can now fire; the tokens disappear from “order waiting” and “workshop waiting” and a token appears in “processing order”:



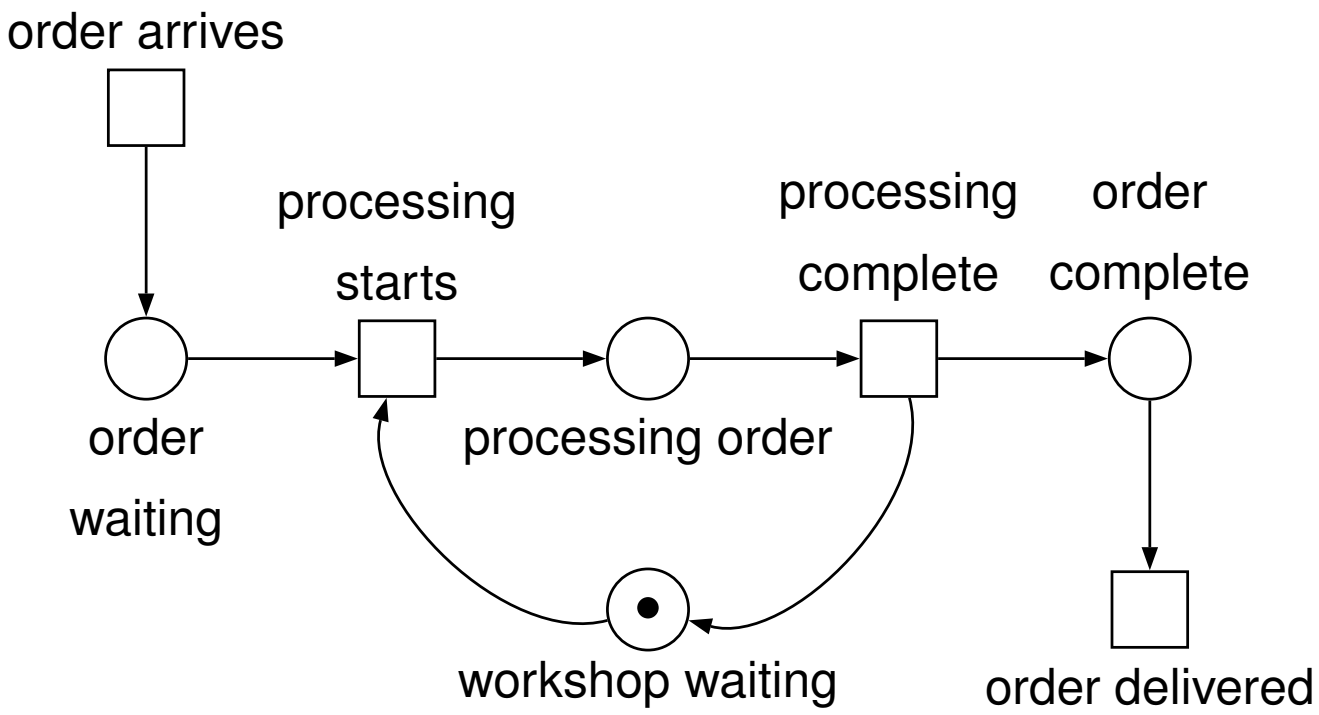
An example

The event “processing complete” can now fire; the token disappears from “processing order” and tokens appear in “workshop waiting” and “order complete”:



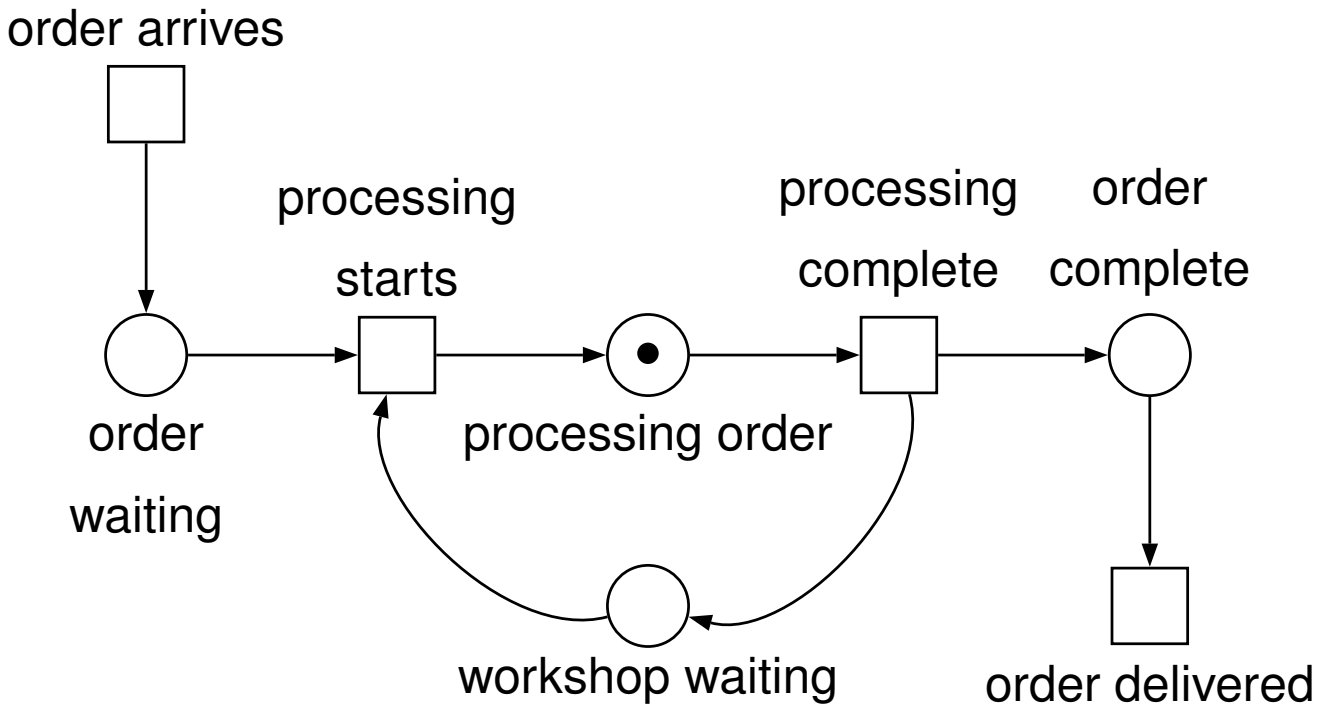
An example

Now the event “order delivered” can fire; the token disappears from “order complete” and we return to the initial marking:



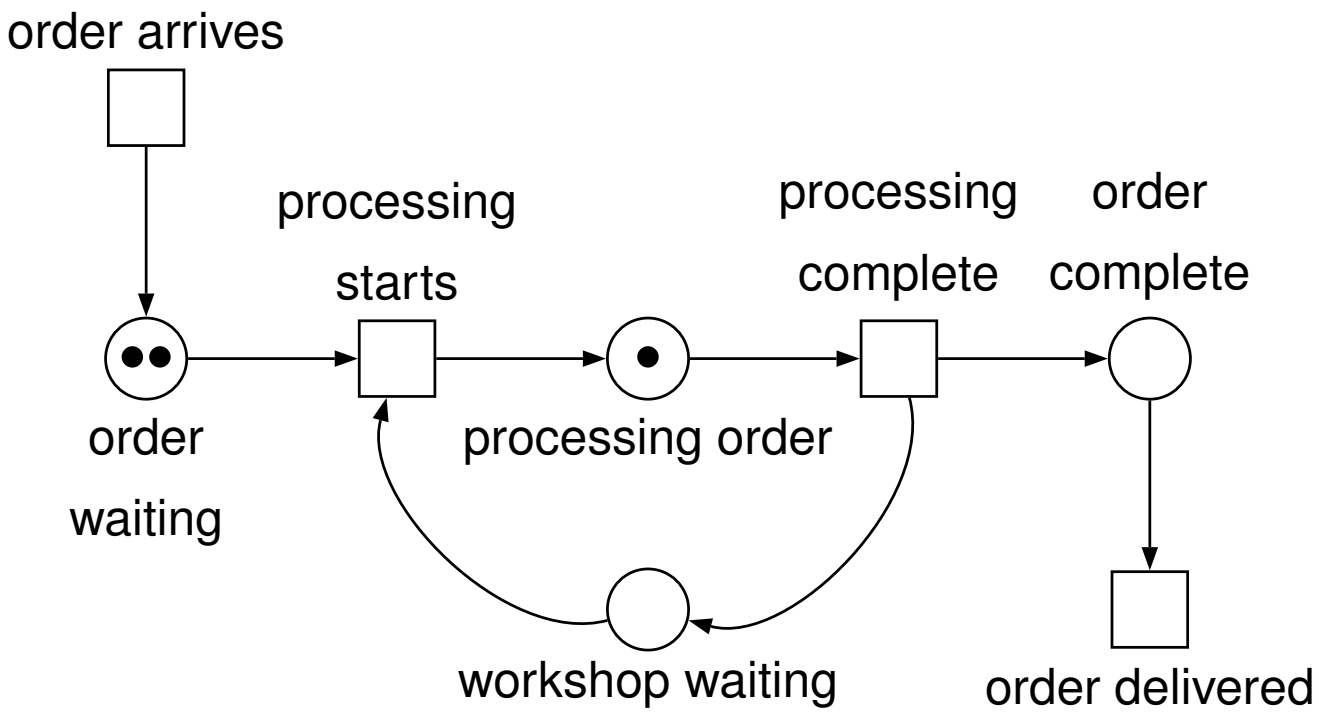
An example

Note that these are not the only possible markings we can reach from the initial one. For example, if we have reached the situation shown below (where the workshop is currently busy) . . .



An example

. . . then more orders can arrive whilst the workshop is busy. For example, if the event “order arrives” fires twice, then we reach the marking shown below:



Typical Petri net problems

Mutual exclusion.

- two processes share a common resource;
- under no circumstances should they access that resource simultaneously.

Some examples:

- modifications in databases;
- use of printers;
- use of shared memory;
- access to hard disks.

Bank tellers

Bank tellers.

- Resource: a shared variable `bal` – the "balance".
- both tellers process requests independently using *local variables*.

Teller 1 – deposits.

1. authenticate
2. $b_1 := bal$
3. $b_1 := b_1 + 100$
4. $bal := b_1$
5. continue

Teller 2 – withdraws.

1. authenticate
2. $b_2 := bal$
3. $b_2 := b_2 - 100$
4. $bal := b_2$
5. continue

Sample run

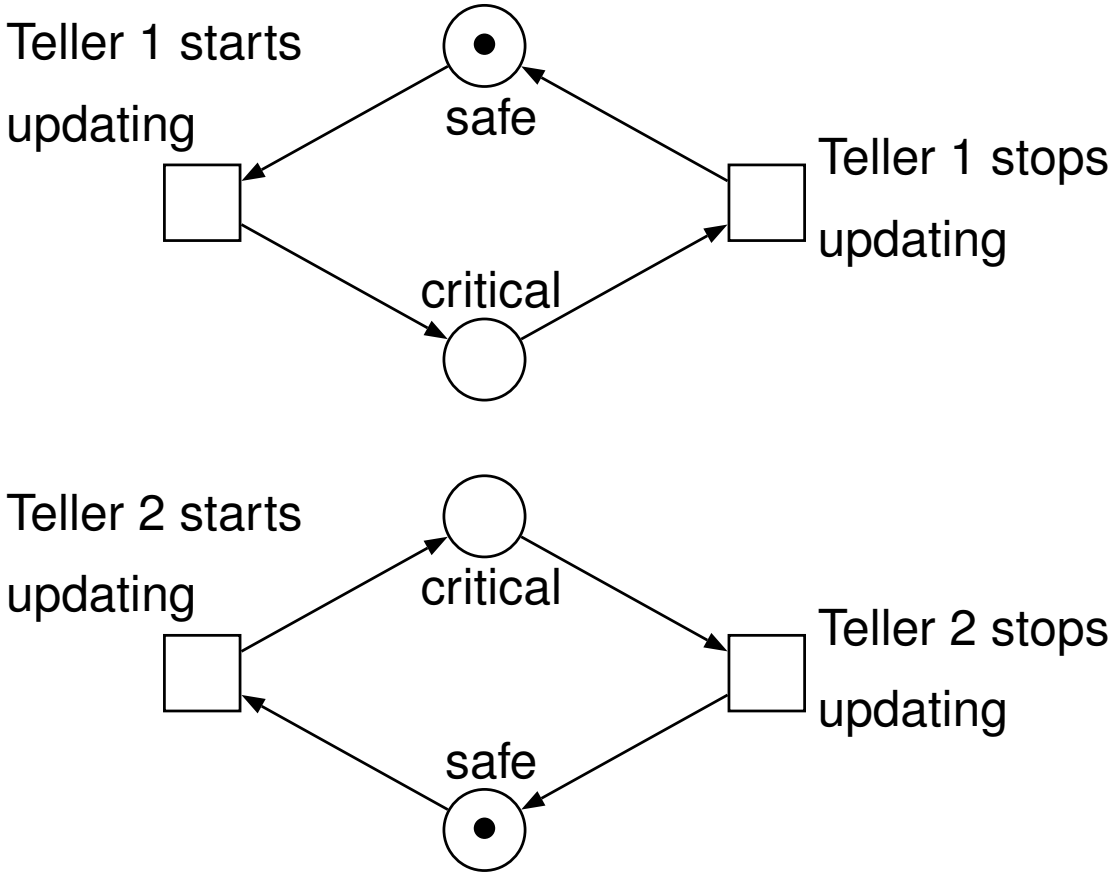
A possible execution of the system starting with $bal = 500$:

Teller 1	Teller 2	Balance
$b_1 := bal$		$bal = 500$
$b_1 := b_1 + 100$		$bal = 500$
	$b_2 := bal$	$bal = 500$
	$b_2 := b_2 - 100$	$bal = 500$
$bal := b_1$		$bal = 600$
	$bal := b_2$	$bal = 400$

So that's £100 deposit lost!

Petri net solution

Without synchronisation:

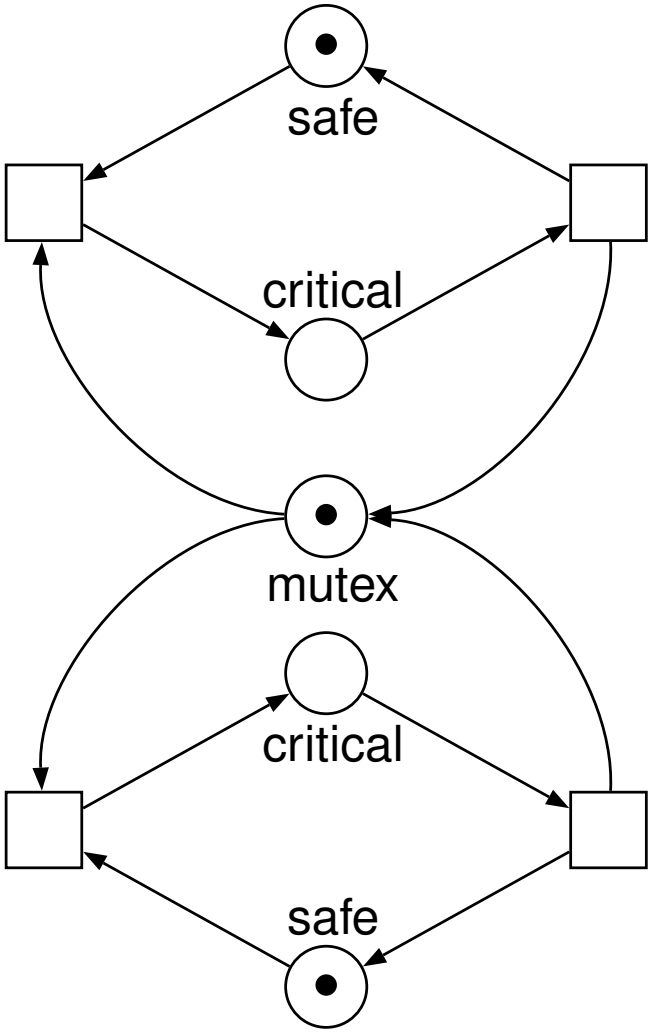


Petri net solution

Idea:

- each process has to request permission before entering the critical section;
- this can be modelled by requiring one *additional* token before entering the critical section . . .
- . . . which should be put back (for the other process) afterwards.

With synchronisation . . .



- mutex controls access to the critical sections;
- each process consumes the mutex-token before it enters the critical section, thus blocking the other process . . .
- . . . and puts it back once it's done.

Producer / consumer problem

Another typical problem.

- A process P continuously produces objects, and puts them into a buffer.
- A process C continuously consumes objects that it takes out of the buffer.

Examples.

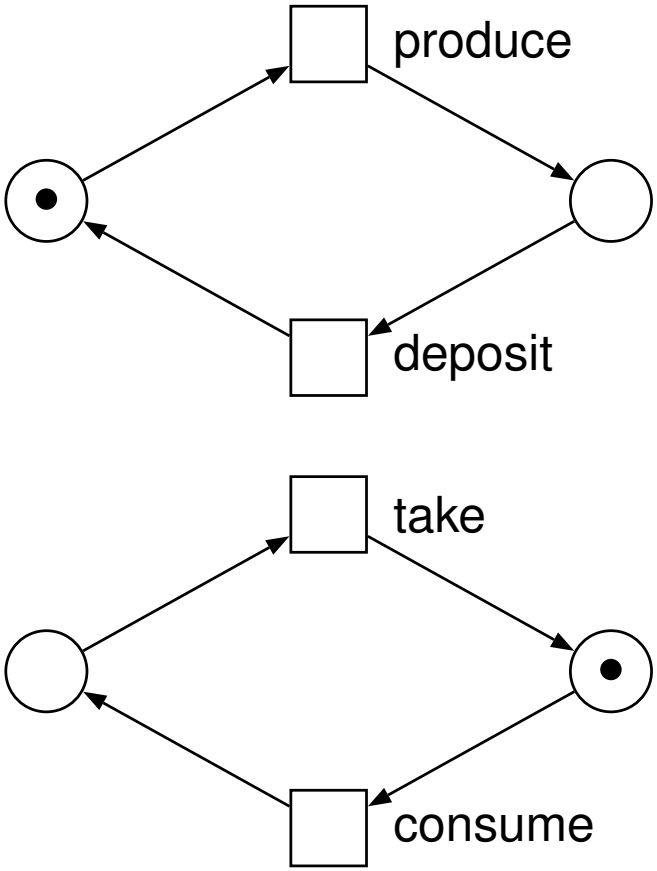
- Data transmission protocols.
- Pipelining in arithmetical/logical units.

Two goals.

- The consumer should not try to take an object if the buffer is empty.
- The producer should not try to deposit an object if the buffer is full.

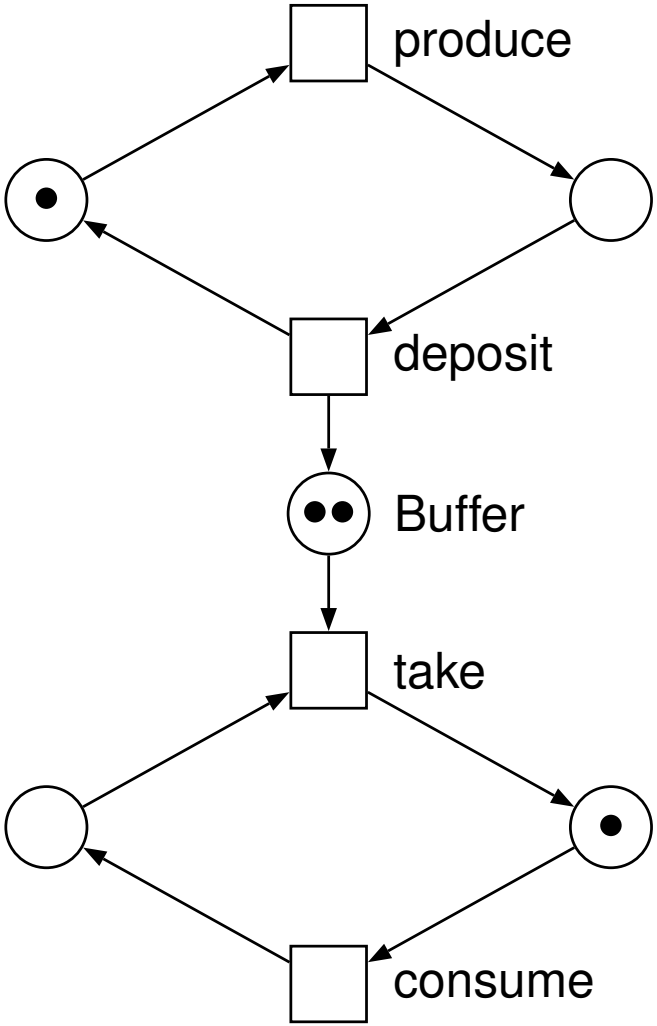
Petri net solution

Without synchronisation . . .



- It should be recorded that an item has been produced.
- Only in this case should the consumer be allowed to proceed.

With synchronisation . . .



- After producing an item, the producer puts a token into the buffer.
- Before consuming an item, the consumer needs to take a token out of the buffer.

Finite capacity buffer

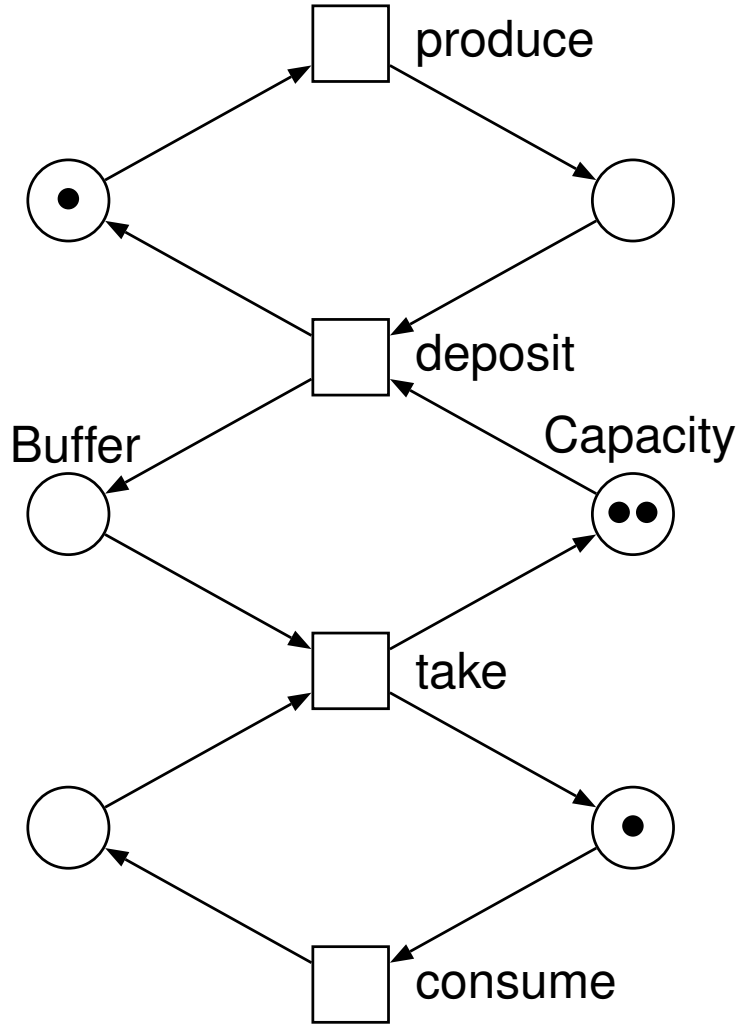
Question. How can we ensure that the buffer has a *finite capacity*, say $\mathfrak{n}$ elements?

- Prohibit the producer from depositing a token if a certain capacity is exceeded ...
- ... *or:* the producer has to request permission before it can deposit a token.

Idea. Introduce a new place (Capacity) that records the available space in the buffer. Initially, there are $\mathfrak{n}$ tokens in Capacity. Whenever the producer wants to deposit, it requests one token. Whenever the consumer removes an object, it puts one token back.

Invariant. Number of tokens in Buffer $+$ number of tokens in Capacity = $\mathfrak{n}$.

Synchronisation + finite capacity



Idea.

- The producer can only deposit an item if there's still space – controlled by Capacity.
- If the consumer takes out an item, then the capacity is increased by one.

CO4211/CO7211

Discrete Event Systems

Lecture 13:
Languages of Petri nets

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Language of a Petri net

What is the *behaviour* of a Petri net?

Idea. As was the case for automata, the behaviour is the set of all the *legal sequences of transitions*.

For Petri nets, in addition to what we have seen so far, we will need:

- a notion of *initial state*;
- a notion of *final* or *accepting states*;
- a labelling function that gives *names* to *transitions*.

Language of a Petri net

Definition. A *labelled Petri net* is a tuple $N = (P, T, F, \omega, \Sigma, l, \mu_0, A)$ where

- (P, T, F, ω) is a Petri net;
- Σ is an *alphabet*;
- $l : T \to \Sigma$ is a *labelling function*;
- μ_0 is the *initial marking*;
- A is a set of *accepting markings*;

This allows us to view the *behaviour* of a Petri net as a *set of strings* over Σ.

Language of a Petri net

Definition. Given a labelled Petri net $N = (P, T, F, \omega, \Sigma, l, \mu_0, A)$, a string $w = a_1 a_2 \ldots a_k \in \Sigma^*$ is *accepted* by N if there exists a sequence μ_0, μ_1, $\ldots$, μ_k of markings (starting at the initial marking μ_0) and transitions

$$\mu_0 \xrightarrow{t_1} \mu_1 \xrightarrow{t_2} \mu_2 \xrightarrow{t_3} \ldots\ldots \xrightarrow{t_k} \mu_k$$

such that:

- $l(t_i) = a_i$ for all i with $1 \leqslant i \leqslant n$;
- $\mu_k \in A$.

The *language* $L(N)$ of a labelled Petri net N is then the set of all strings accepted by N.

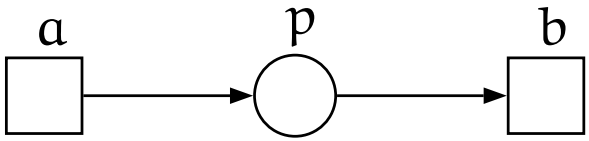
Recall: we write $\mu \xrightarrow{t} \mu'$ if μ and μ' are markings with the transition t enabled under μ and if the firing of t takes us from μ to μ'.

Language of a Petri net

Idea. A string is *accepted* if there is a *path* from the *initial marking* to an *accepting marking* using transitions whose labels (read in sequence) are the symbols of the string.

Question. How does this compare to automata?

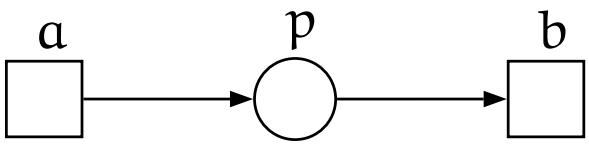
Consider the labelled Petri net N ...



... with $\Sigma = \{\mathrm{a}, \mathrm{b}\}$. The initial marking μ_0 is defined by $\mu_0(\mathrm{p}) = 0$ and the set A of accepting markings is defined to be $\{\mu_0\}$. Here we have a place p with two transitions labelled by a and b.

Question. What is the language of this labelled net?

Example



Initial marking μ_0 defined by $\mu_0(p) = 0$. Accepting markings $A = \{\mu_0\}$.

What is the language of this labelled net?

Answer. In this case $L(N) = \{w \in \Sigma^* : w$ satisfies (1) and (2)$\}$ where:

(1) the number of a's in w is the same as the number of b's in w;

(2) every prefix of w has no more b's than a's.

Condition (2) corresponds to the fact that we cannot execute b unless we have executed a previously.

Example

In our example,

$$L(N) = \{w \in \Sigma^* : w \text{ satisfies (1) and (2)}\}$$

where:

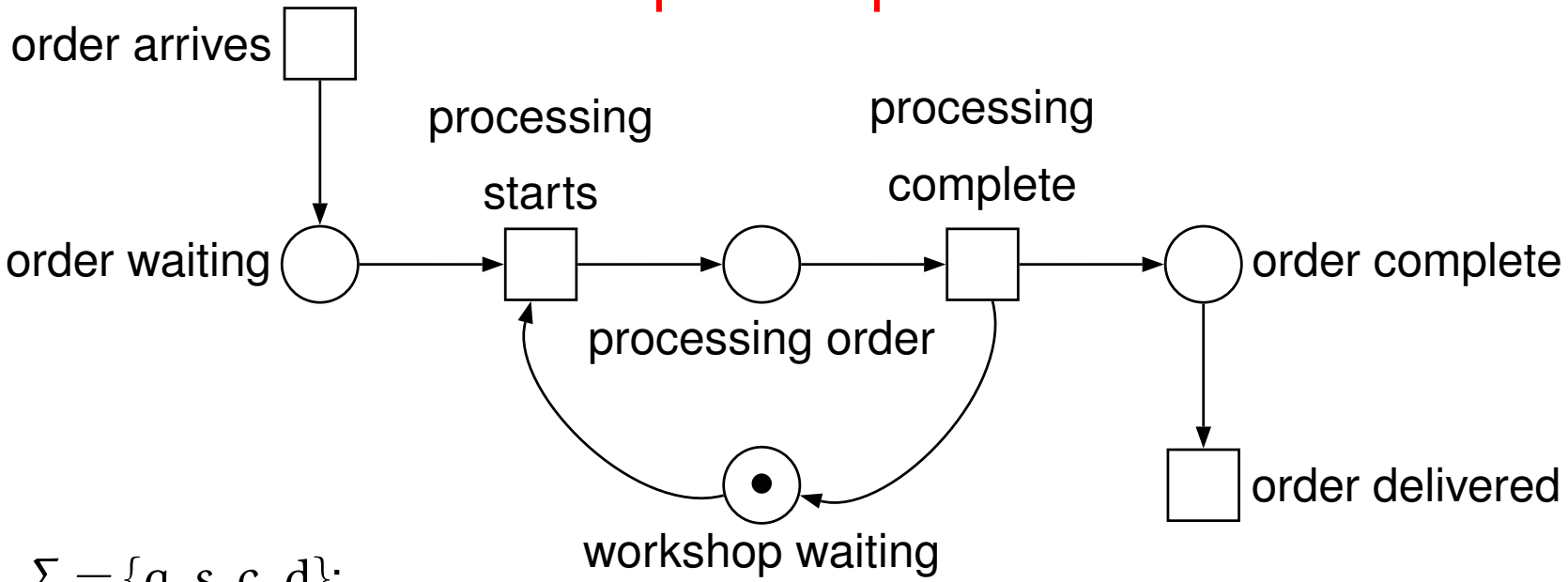
(1) the number of a's in w is the same as the number of b's in w;

(2) every prefix of w has no more b's than a's.

This is essentially the language of the manufacturing example we discussed in Lecture 10; we saw there that this language is not regular.

So Petri nets can accept languages that automata cannot accept!

Workshop example revisited



$\Sigma = \{a, s, c, d\}$:

- a: order arrives;
- s: processing starts;
- c: processing complete;
- d: order delivered.

$A = \{\mu_0\}$ where μ_0 puts one token into the place "workshop waiting" (and no tokens in the other places) as shown above.

Language of the workshop example

Observations:

- at the end, all the completed orders must have left the system;
- there cannot be more delivered orders than completed orders *in all intermediate points*;
- there cannot be more completed orders than started orders *in all intermediate points*;
- there cannot be more started orders than arrived orders *in all intermediate points*.

Language of the workshop example

Language. As usual, we write $\sharp_x(w)$ for the number of occurrences of the symbol x in the string w.

If w is a string accepted in the workshop example, then:

- w needs to satisfy $\sharp_a(w) = \sharp_s(w) = \sharp_c(w) = \sharp_d(w)$;
- every *prefix* v of w needs to satisfy $\sharp_c(v) \geqslant \sharp_d(v)$ – the second condition above ...
- ... and $\sharp_s(v) \geqslant \sharp_c(v)$ – the third condition ...
- ... and $\sharp_a(v) \geqslant \sharp_s(v)$ – the last condition.

Note that "in all intermediate points" is expressed by considering *prefixes* of the accepted string.

Language of the workshop example

Another condition. There is at most one order currently being processed at a given time. So, if w is an accepted string and v is a prefix of w, then we must have that

$$\sharp_s(v) \leqslant \sharp_c(v) + 1.$$

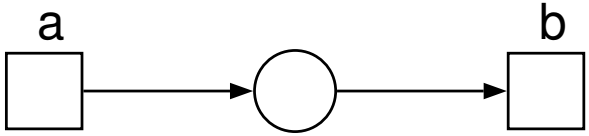
Putting all this together yields that the language of this labelled Petri net is:

$$\{w \in \Sigma^* : \quad \sharp_a(w) = \sharp_s(w) = \sharp_c(w) = \sharp_d(w),\ \sharp_c(v) \geqslant \sharp_d(v),\ \sharp_a(v) \geqslant \sharp_s(v),$$
$$\text{and } \sharp_c(v) + 1 \geqslant \sharp_s(v) \geqslant \sharp_c(v) \text{ for every prefix } v \text{ of } w\},$$

where $\Sigma = \{a, s, c, d\}$.

Automata versus Petri nets

We considered the labelled Petri net



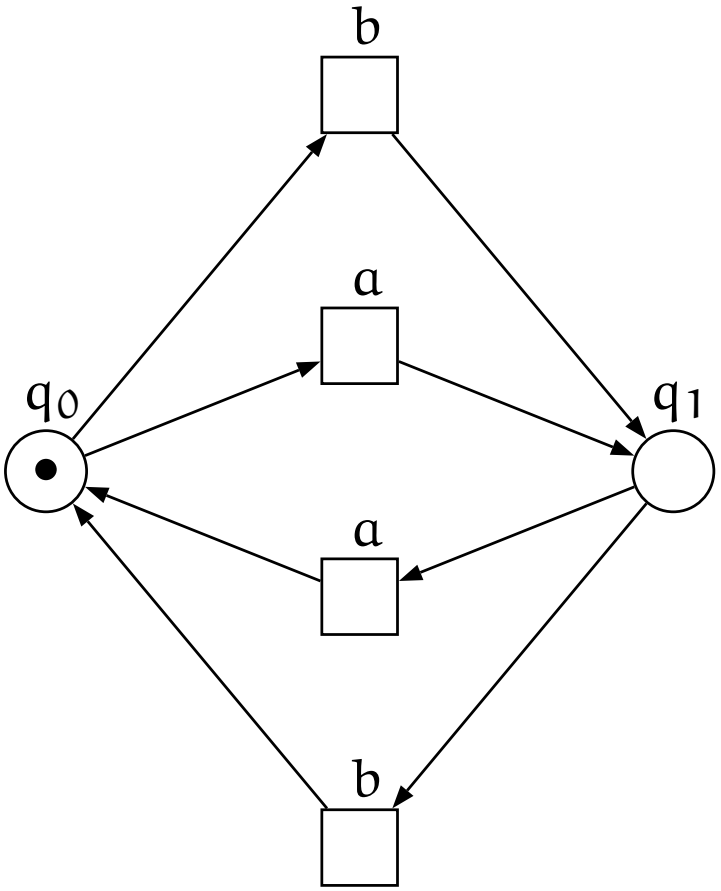
with initial marking μ_0 defined by $\mu_0(p) = 0$ and accepting markings $F = \{\mu_0\}$ and saw that it accepted the language

$$L(N) = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} w \text{ has the same number of } a\text{'s and } b\text{'s,} \\ \text{every prefix of } w \text{ has no more } b\text{'s than } a\text{'s} \end{array} \right\}.$$

So Petri nets can accept language that automata cannot accept. However, we then have the natural question:

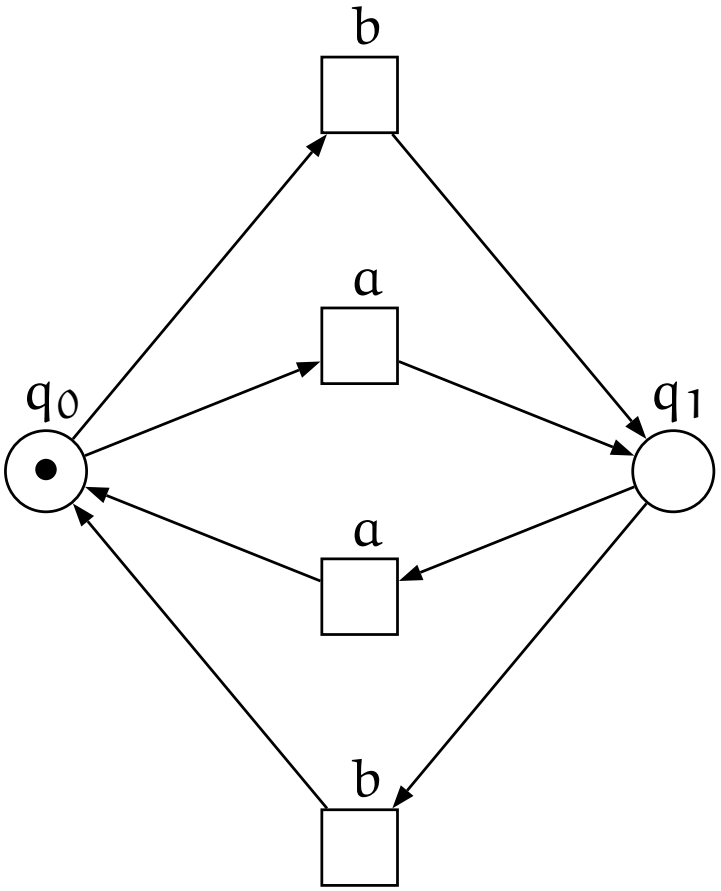
Question. Is every language accepted by an automaton also accepted by a Petri net?

Another example



- Initial marking: $\mu_0 = [1, 0]$ – i.e. one token in place q_0 and no tokens in place q_1 (as shown).
- Accepting markings: $A = \{\mu_0\}$.

Another example



What is the language of this labelled Petri net?

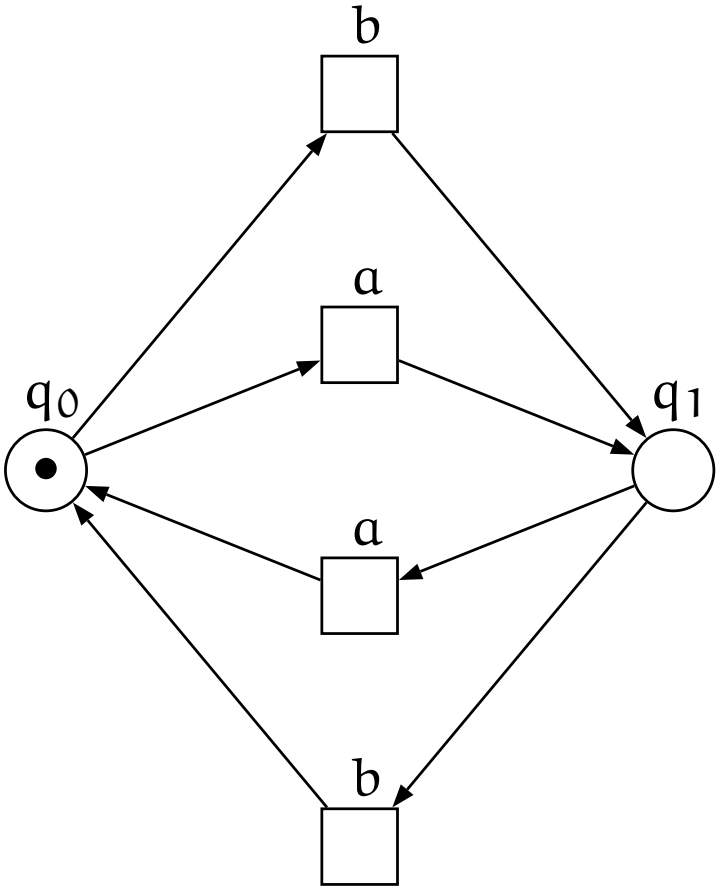
Answer.

$L = \{w \in \{a, b\}^* : |w| \text{ is even}\}$.

But we've seen this language – it's also accepted by an automaton!

Comparison to automata

Compare the Petri net . . .

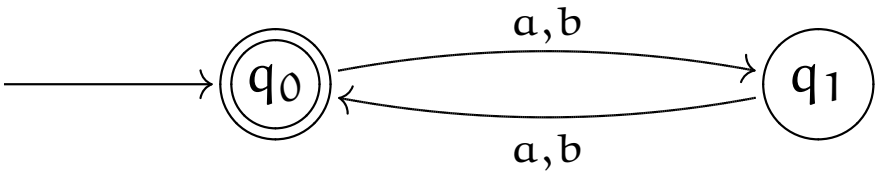


with

- $\mu_0 = [0, 1]$ (as shown),
- $F = \{\mu_0\}$, ...

Comparison to automata

. . . to the automaton . . . . . .



with

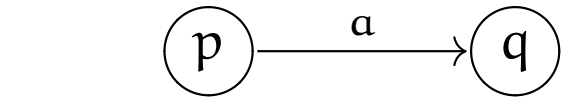
- start state q_0;
- accepting states $F = \{q_0\}$.

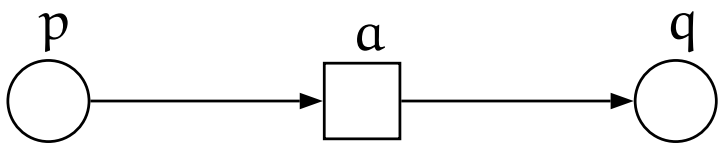
They accept the same *language* – i.e. they represent the same *behaviour*!

From automata to Petri nets

Given a (deterministic) finite automaton $A = (Q_A, \Sigma_A, \delta_A, q_0^A, F_A)$, we can construct a labelled Petri net N with $L(A) = L(N)$.

The *states* of the automaton become the *places* of the net and the *arrows* in the automaton become the *transitions* of the net (with the input on the arrow in the automaton becoming the label of the transition in the net):

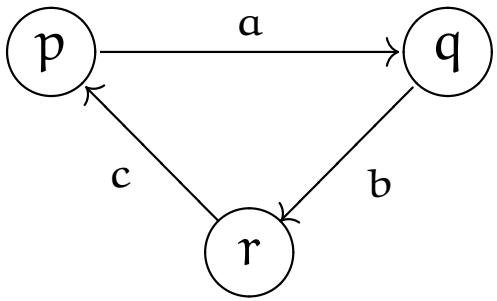


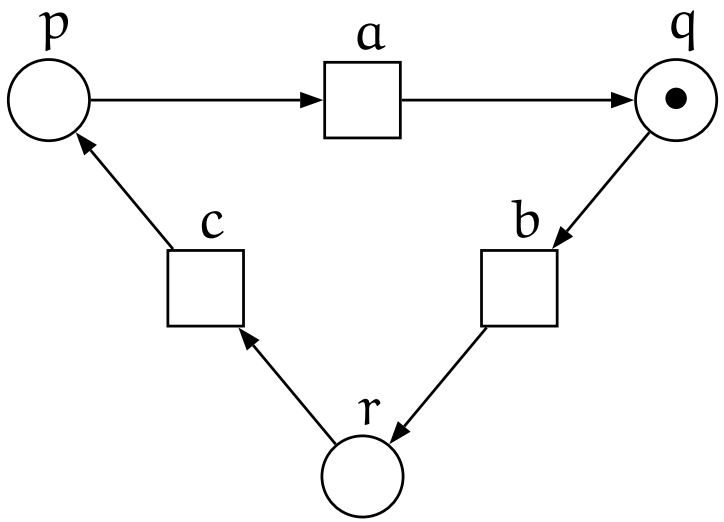


From automata to Petri nets

Each *transition* in the automaton moves *one token* in the net (the position of this token indicates the current state of the automaton):

Automaton - current state q.





Petri net - single token in place q.

General construction

Consider the labelled net $N = (P_N, T_N, F_N, \omega_N, \Sigma_N, l_N, \mu_0^N, A_N)$ where:

- $P_N = Q_A$; $\Sigma_N = \Sigma_A$;
- $T_N = Q_A \times \Sigma_A$ – there is one transition in N for each (state, symbol) pair in A;
- $F_N = \underbrace{\{(q,(q,a)) : q \in Q_A, a \in \Sigma_A\}}_{\textit{places} \to \textit{transitions}} \cup \underbrace{\{((q,a),q') : \delta_A(q,a) = q'\}}_{\textit{transitions} \to \textit{places}}$;
- $\omega_N(f) = 1$ for all $f \in F_N$;
- if $t = (q,a) \in T_N$ then $l_N(t) = a$;
- $\mu_0^N(q_0^A) = 1$ and $\mu_0^N(q) = 0$ for $q \neq q_0^A$;
- $A_N = \{\mu_q : q \in F_A\}$, where $\mu_q^N(q) = 1$ and $\mu_q^N(q') = 0$ for $q' \neq q$.

The result

A *special feature* of every Petri net constructed from an automaton in this way is that there is *exactly* one token present in the net at all times.

Theorem. For every finite automaton $\mathcal{A}$ there exists a Petri net N with $\mathrm{L}(\mathrm{N}) = \mathrm{L}(\mathcal{A})$.

As we have seen, the converse to this result is not true.

CO4211/CO7211

Discrete Event Systems

Lecture 14:
Safety in Petri nets

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Special Petri nets

Definition. A *Petri net with initial state* is a tuple $N_I = (P, T, F, \omega, \mu_0)$ where:

- $N = (P, T, F, \omega)$ is a Petri net, and
- μ_0 is an *initial marking* of N.

A place $p \in P$ of N_I is said to be k-*safe* if $\mu(p) \leqslant k$ for all markings μ that can be reached from the initial marking μ_0.

A Petri net with initial state is said to be k-*safe* if all its places are k-safe.

A Petri net with initial state is said to be *safe* if it is 1-safe.

Bounded Petri nets

A Petri net with initial state is said to be *bounded* if it is k-safe for some k.

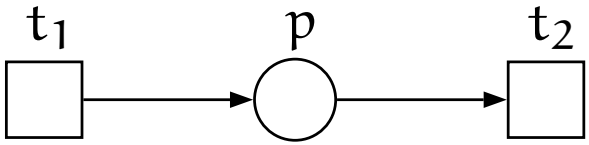
A Petri net with initial state that is not bounded is said to be *unbounded*.

Theorem. A Petri net with initial state is bounded if and only if the set of all reachable markings is finite.

Some possible considerations.

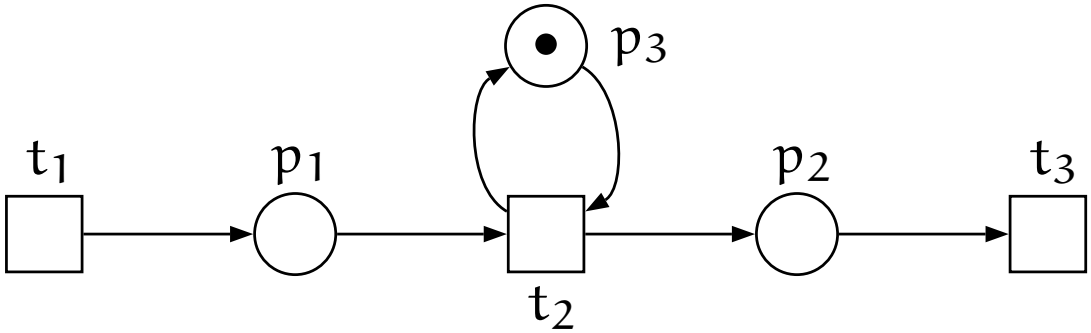
- A bounded Petri net can be implemented in *hardware* with *finite* resources.
- Places of safe Petri nets can be implemented by *flip-flops*.

Some unbounded Petri nets



Is this Petri net bounded or unbounded?

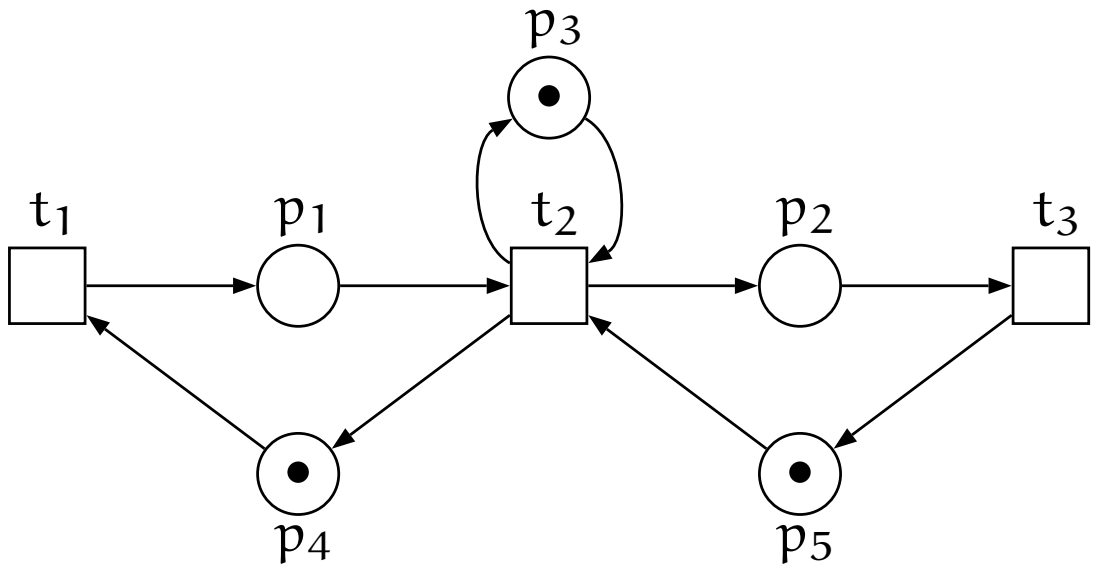
Tokens can be produced without limit at place p; so this Petri net is not k-safe for every k, and hence it is *unbounded*.



Is this Petri net k-safe (for some k) or unbounded?

Tokens can be produced without limit at places p_1 and p_2; so this Petri net is not k-safe for every k, and hence it is *unbounded*.

A safe Petri net

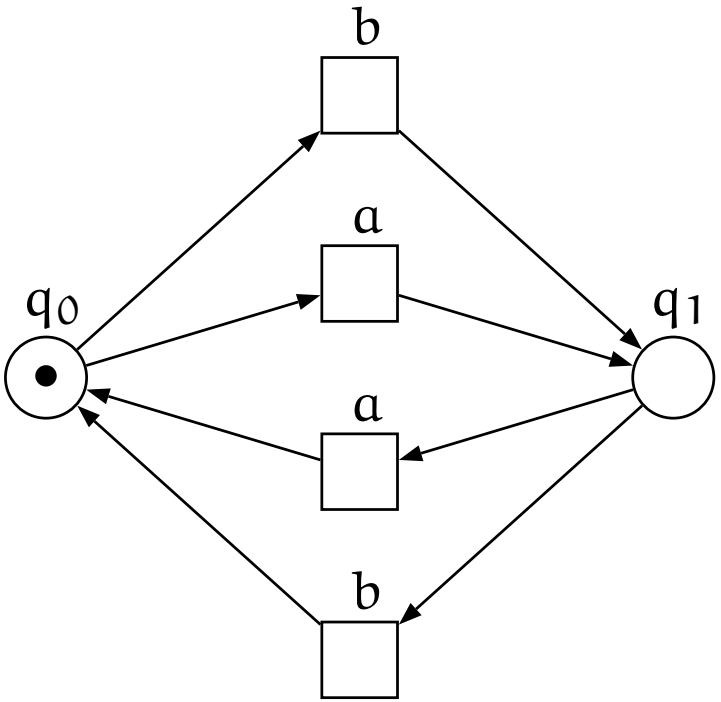


Is this Petri net bounded or unbounded?

In this net every token that is produced consumes a token: so this Petri net is *1-safe* (and hence *safe* and *bounded*). At each stage we have at most one token in every place.

Petri nets from automata

Special case: Consider Petri nets constructed from automata, such as:

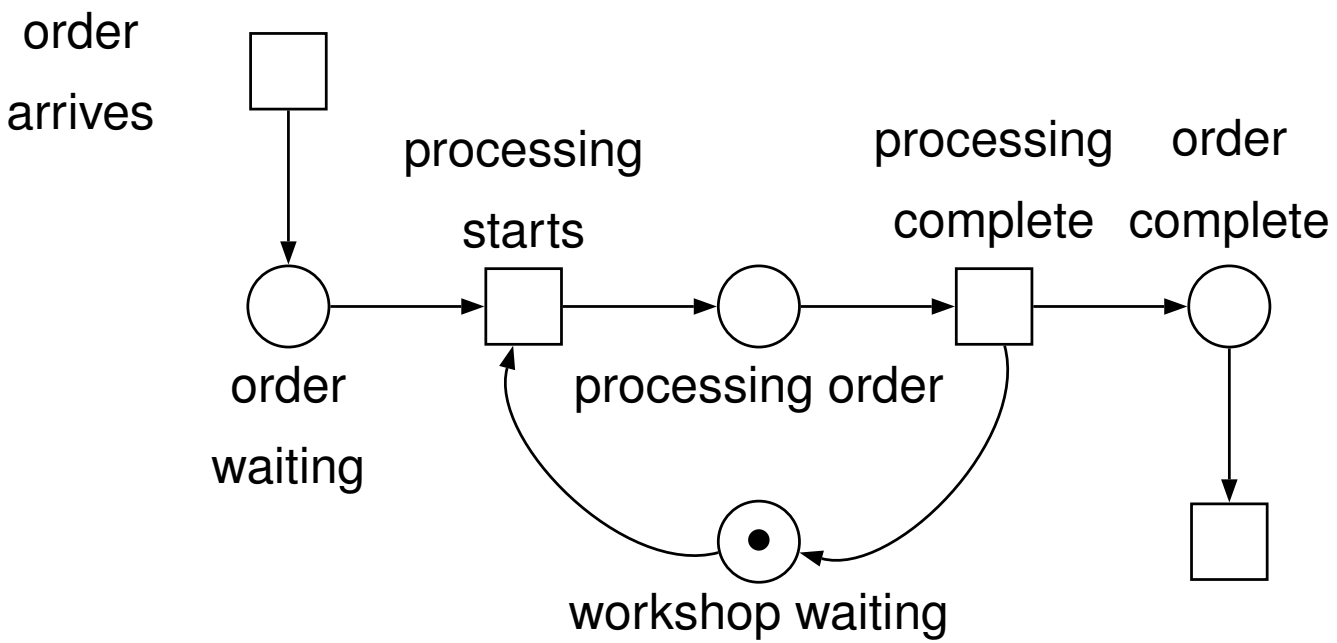


Observations.

- The initial marking contains one token only.
- Each transition moves this token.

So all Petri nets constructed from automata are *safe*; we always have precisely one token present in the net at every stage.

Workshop example revisited



This Petri net is not k-safe for every value of k; for example, we can produce arbitrarily many tokens in “order waiting”. So this Petri net is *unbounded*.

Dining philosophers

Dining philosophers (Dijkstra, 1971).

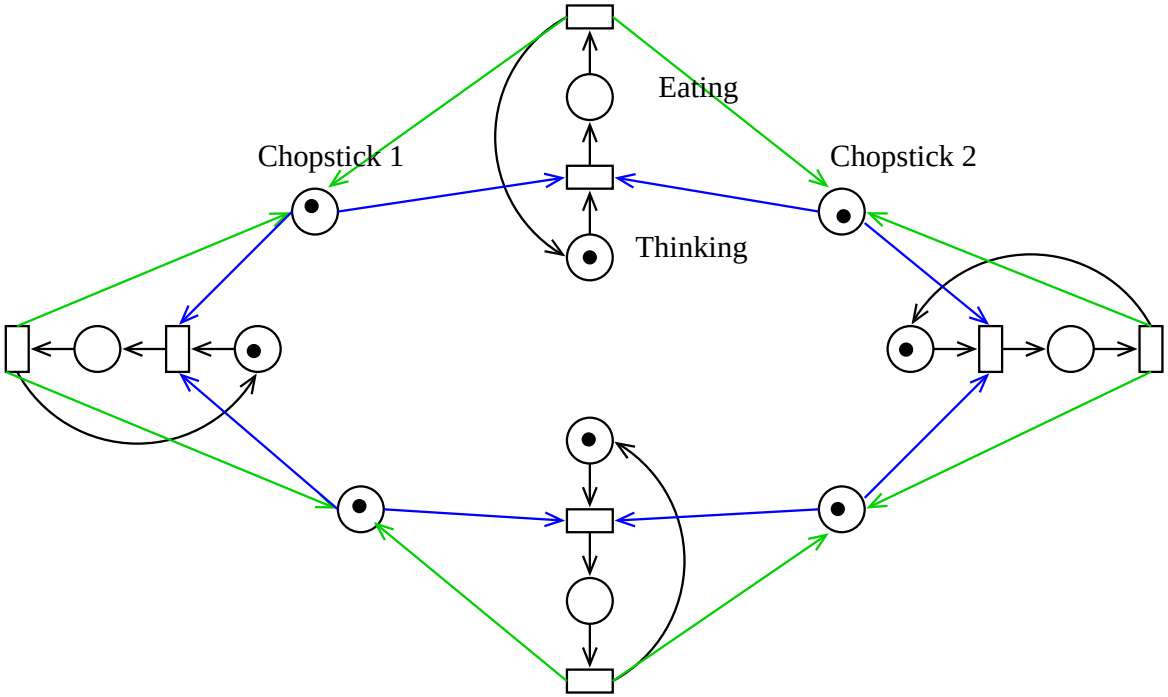
- Four philosophers sit around a table and alternate between eating and thinking;
- there is one chopstick between each pair of philosphers;
- to start eating, each philosopher needs *two* chopsticks.

Problem.

- if every philosopher picks up the chopstick to her left, then none of them can start eating (*deadlock*).

Goal. Introduce a synchronisation mechanism such that no deadlocks can occur.

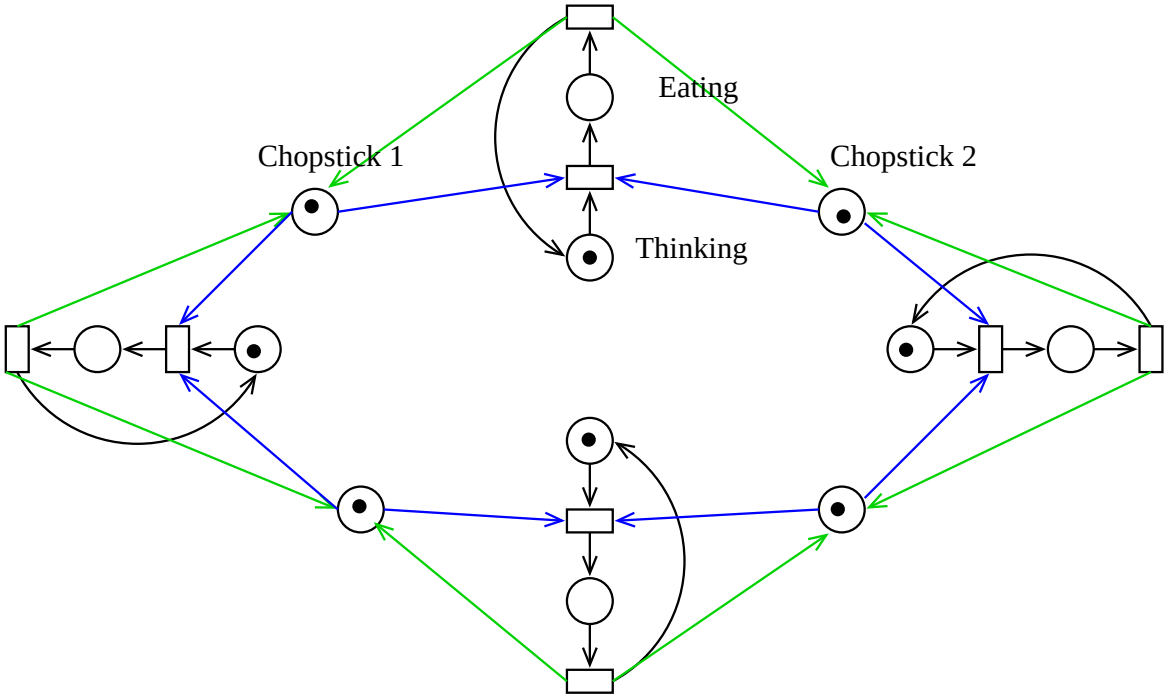
Petri net solution



- Green arrows: put *both* chopsticks back.
- Blue arrows: request *both* chopsticks.

Is this Petri net k-safe (for some k) or unbounded?

Dining philosophers - safety?



- Each philosopher is either eating or thinking – no double tokens.
- Each chopstick is either available or not – no double tokens.

So this Petri net is *safe*.

Reader/writer problem

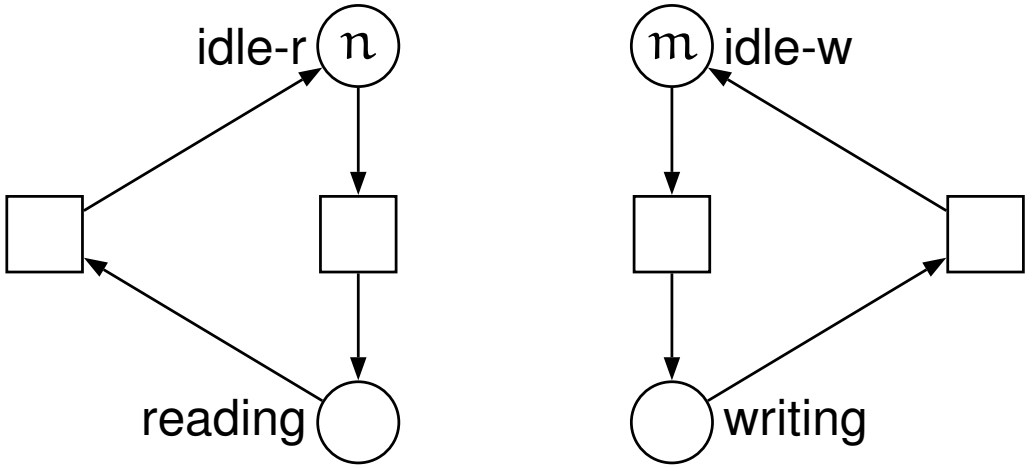
Description of the problem.

- n readers and m writers share a common resource (such as a memory location);
- at most k processes can access the resource at all times;
- at most one writer should be allowed access to the resource at all times;
- at no time should both readers and writers access the resource.

Petri net solution.

- Tokens as resources: n readers and m writers $\rightsquigarrow$ n (respectively m) tokens.
- Bound on the number of processes $\rightsquigarrow$ k “permission tokens”.

Without synchronisation

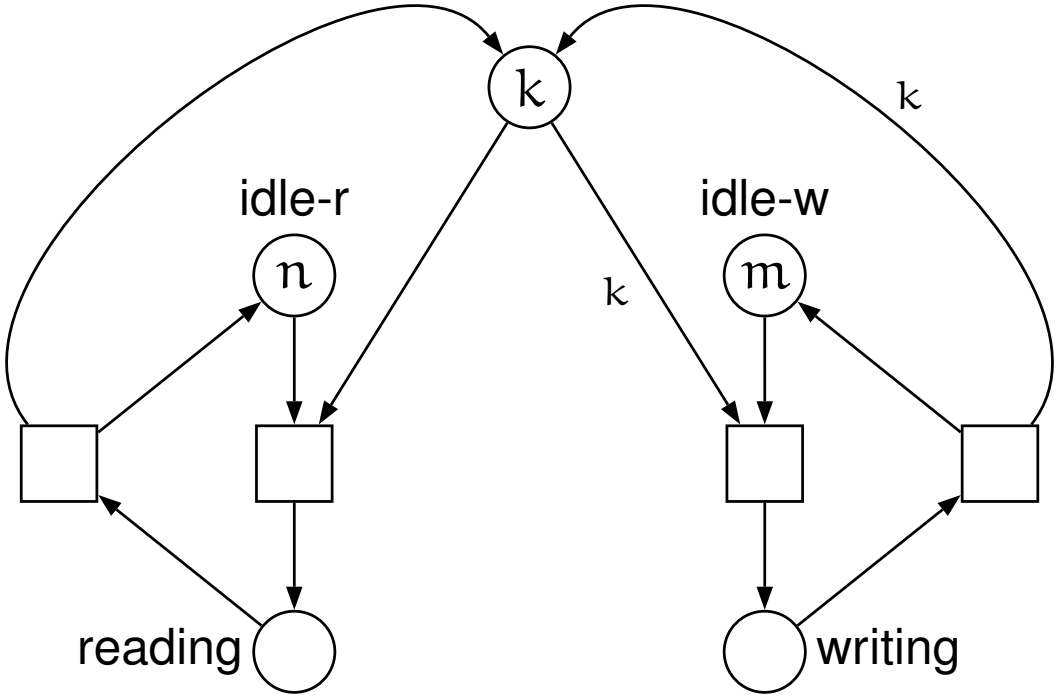


Notation. Number r in a place $\rightsquigarrow r$ tokens in that place.

Usual problem. In this Petri net, readers and writers can access the resource independently; we need to synchronise the two Petri nets.

Note. In addition, we have not yet modelled the fact that at most one writer should be allowed access at all times.

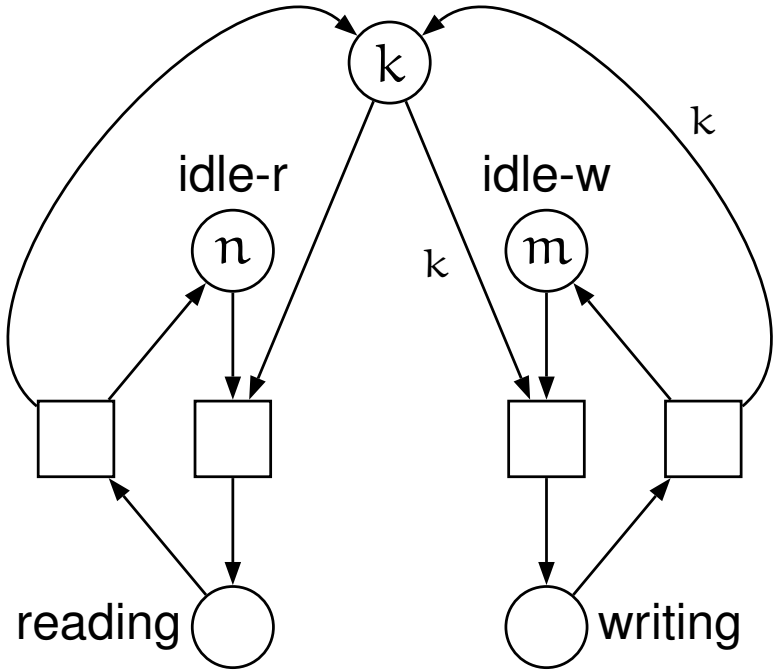
With synchronisation



Points to note:

- n readers and m writers $\rightsquigarrow$ n (respectively m) tokens;
- at most k processes active $\rightsquigarrow$ k tokens in the synchronisation place;
- *each* writer requires *all* the “permission tokens”.

Reader/writer problem - safety



- at most k active readers $\Longrightarrow$ at most k tokens in reading;
- n readers in total $\Longrightarrow$ at most n tokens in idle-r;
- at most one active writer $\Longrightarrow$ at most one token in writing;
- m writers in total $\Longrightarrow$ at most m tokens in idle-w.

So this Petri net is $\max\{m, n, k\}$-safe.

CO4211/CO7211

Discrete Event Systems

Lecture 15:
Algorithms in Petri nets

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Is it bounded?

General question. How can we decide whether a Petri is bounded?

- We could run it though a simulator and see what happens. That might not catch *all* the possibilities.
- We could check it manually. We might make mistakes (and the problem could be too large to make this feasible).

Is it bounded?

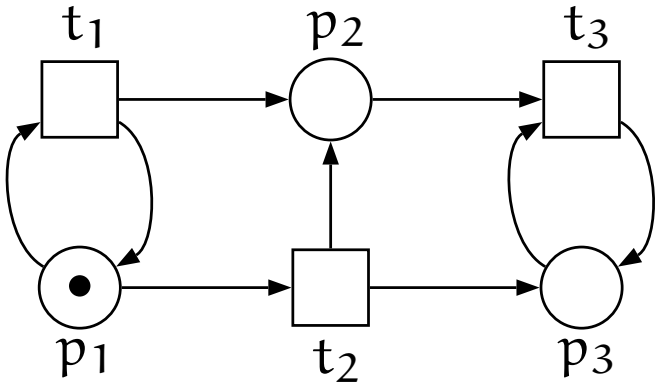
Idea. Construct the *reachability tree*.

- *root note*: the initial marking;
- *subnodes of a node* μ: all the markings μ' for which we have $\mu \xrightarrow{t} \mu'$ for some transition t.

If we have the tree . . .

- . . . it will have all the possible markings as nodes – so it seems that we could check if the net is bounded . . .
- . . . but the tree could be infinite!

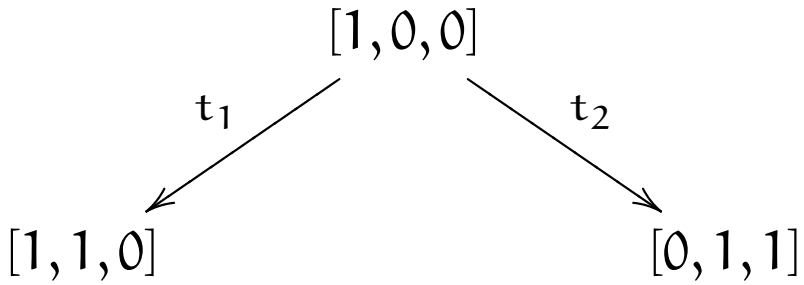
Reachability tree



Initial marking. $\mu_0 = [1,0,0]$

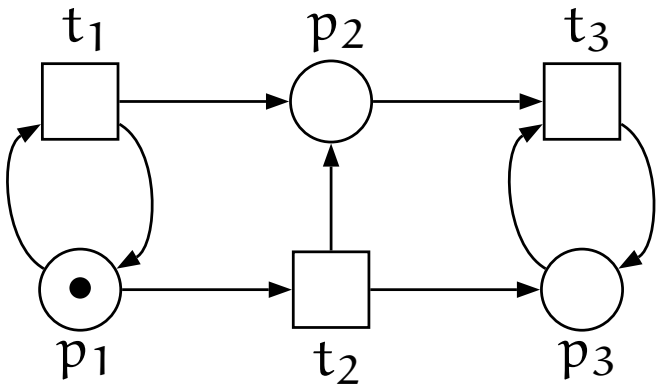
(as shown).

First layer of the tree.

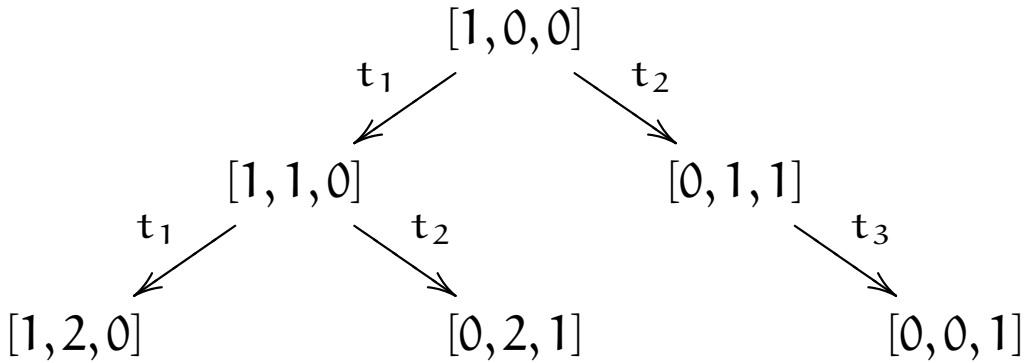


Note that transition t_3 is not enabled at the start.

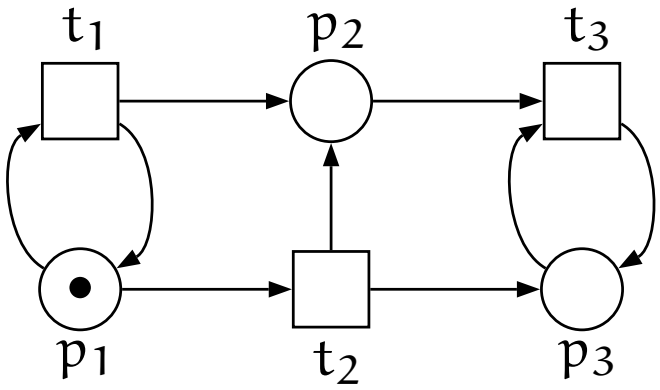
Reachability tree



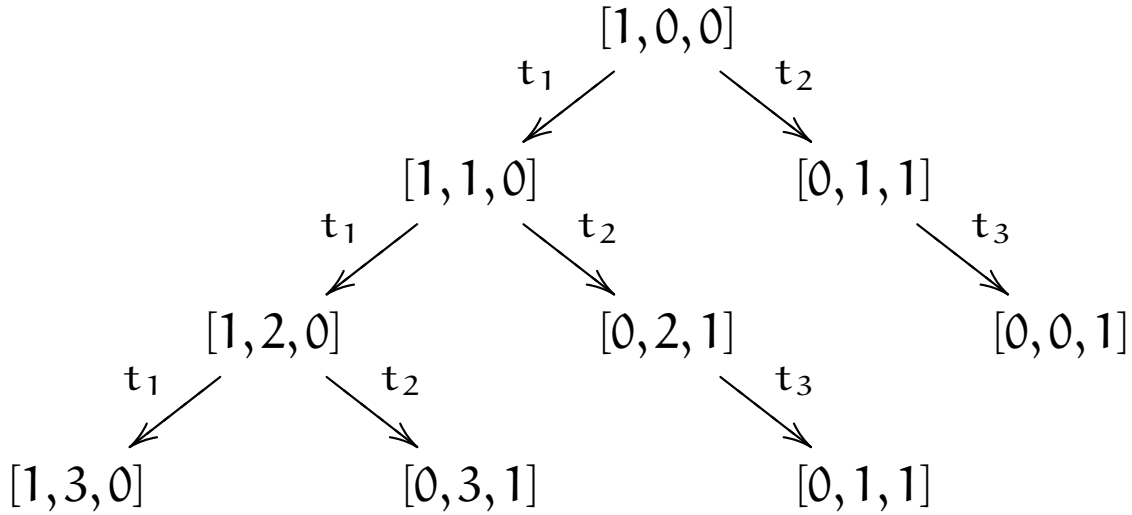
Second layer.



Reachability tree



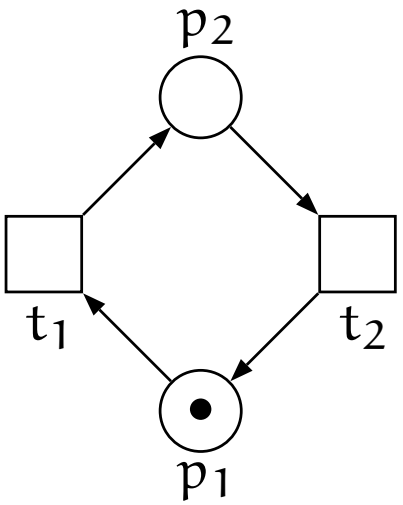
Third layer.



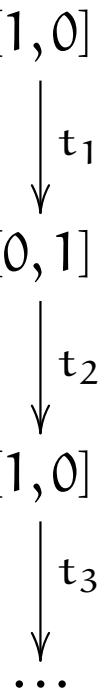
Generating the reachability tree

Problem 1. The tree can be infinite and contain repeated markings.

Petri net:



Reachability tree:

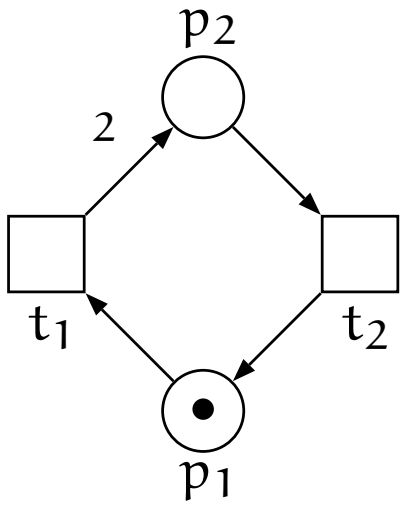


Here there are *repeated markings*.

Generating the reachability tree

Problem 2. The number of possible tokens in a place might be unbounded.

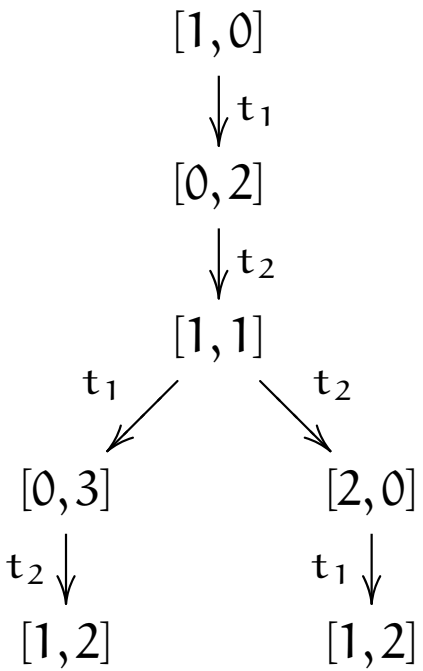
Petri net:



different reachable markings.

Reachability tree:

Here there are *infinitely many*


[1,0]
t1
[0,2]
t2
[1,1]
t1
t2
[0,3]
[2,0]
t2
t1
[1,2]
[1,2]

Making the tree finite

Question. How can we construct a *finite* reachability tree?

- Don't pursue transitions from duplicate nodes – this doesn't really add new information.
- Take care of arbitrarily large numbers of tokens in a place – that's more difficult!

Example. Consider the transition sequence

$$[1,0] \xrightarrow{t_1} [0,2] \xrightarrow{t_2} [1,1].$$

- We have produced an extra token in place 2.
- Repetition of this sequence produces an arbitrarily large number of tokens in place 2.

Arbitrarily many tokens

Idea: We introduce a new notation for "arbitrarily many tokens": $*$.

Definition. If μ, μ' are markings, then we write $\mu < \mu'$ if μ' has some extra tokens as compared to μ, i.e. if:

- for all $p \in P$ we have $\mu'(p) \geqslant \mu(p)$;
- there exists $p \in P$ such that $\mu'(p) > \mu(p)$.

Arbitrarily many tokens

If $\mu < \mu'$ we then define an *extended marking*

$$\mu \oplus \mu' : P \to \mathbb{N} \cup \{*\}$$

by

$$\mu \oplus \mu'(p) = \begin{cases} \mu(p) & \mu(p) = \mu'(p) \\ * & \mu'(p) > \mu(p) \end{cases}$$

The symbol $*$ indicates the possibility of an arbitrarily large number of tokens in that place.

For example, $[1,0,0] \oplus [1,0,1] = [1,0,*]$ and $[1,1,1] \oplus [2,1,3] = [*,1,*]$.

Arbitrarily many tokens

We can extend $\oplus$ to extended markings in the natural way (where we take $* > k$ for every natural number k). For example, $[1, 0, *] + [1, 1, *] = [1, *, *]$.

Idea. If we have a transition sequence

$$\mu_k \longrightarrow \mu_{k+1} \longrightarrow \ldots \longrightarrow \mu_{k+n}$$

with $\mu_k < \mu_{k+n}$ in the reachability tree, then we replace μ_{k+n} by $\mu_k \oplus \mu_{k+n}$.

Theorem. A Petri net with initial state is unbounded if and only if there exists a reachable marking μ and a sequence of transitions σ such that $\mu \xrightarrow{\sigma} \mu'$ for some marking μ' with $\mu' > \mu$.

Transitions between extended markings

Definition. Suppose that $N = (P, T, F, \omega)$ is a Petri net and that μ, μ' are extended markings for P. We define

$$\mu \xrightarrow{t} \mu' \iff \mu'(p) = \mu(p) - w(p,t) + w(t,p)$$

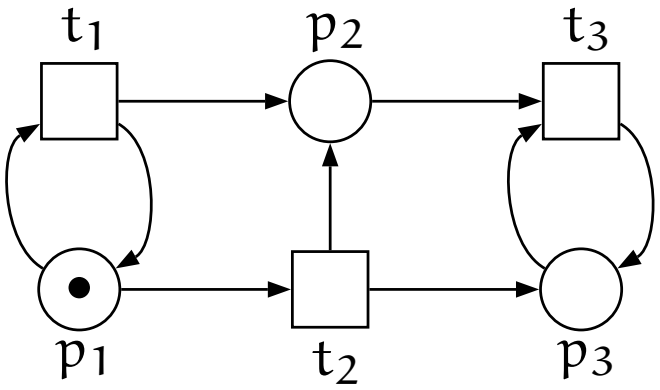
for all $p \in P$ (which is as before), but where (in addition) we have

$$* + a = *, \qquad * - a = *,$$

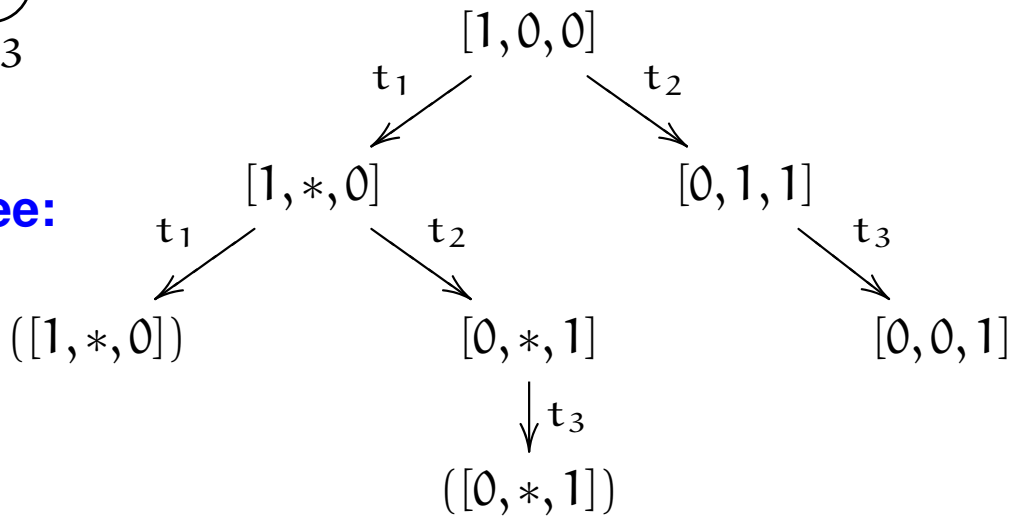
for all $a \in \mathbb{N} \cup \{*\}$.

We can now apply this to extended markings in the reachability tree.

Example



Reachability tree:



Note. The markings in brackets represent those that have appeared in the tree previously (i.e. they are duplicated markings).

Reachability tree algorithm

Algorithm for constructing the reachability tree for $N = (P, T, F, \omega)$ *and* μ_0*.*

1. (initialise) Frontier $:= \{\mu_0\}$ and we start the tree with μ_0 as the root;
2. pick an element $\mu \in$ Frontier;
3. for each (t, μ') with $\mu \xrightarrow{t} \mu'$ do
   (a) for each marking ν on the path from μ_0 to μ' in the tree with $\nu < \mu'$ let $\mu' := \nu \oplus \mu'$;
   (b) add the edge $\mu \xrightarrow{t} \mu'$ to the tree;
   (c) if μ' is already in the tree put it in brackets; else Frontier $:=$ Frontier $\cup \{\mu'\}$;
4. Frontier $:=$ Frontier $- \{\mu\}$;
5. if Frontier $= \emptyset$ then stop; otherwise continue with step 2.

This creates a *finite* reachability tree which is k*-safe* if all the markings μ in the tree satisfy $\mu(p) \neq *$ and $\mu(p) \leqslant k$ for all $p \in P$.

Decidable properties of Petri nets

The following properties of Petri nets are decidable:

- *liveness* (every transition can always occur again at some point in the future no matter what marking we have currently reached);
- *deadlock-freedom* (in each reachable marking at least one transition is enabled);
- *persistence* (if t_1 and t_2 are distinct transitions and if t_1 and t_2 are both enabled at some reachable marking μ, then firing t_1 cannot disable t_2).

Undecidable properties of Petri nets

The following properties of Petri nets are undecidable:

- *regularity* (is the language of a labelled Petri net regular?);
- *language equivalence* (do two Petri nets accept the same language?).

Note. We saw that the language equivalence problem for finite automata *was* decidable: just minimize the two automata and see if the resulting automata are the same (up to the labelling of the states).

CO4211/CO7211

Discrete Event Systems

Lecture 16:
Probability

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Why probability?

So far, system behaviour has been uniquely specified:

- we were looking at possible sequences of events, but not at their *probabilities*.

Example. Job scheduling:

- a queue dispatches jobs to two processors;
- if the queue is full, the jobs are rejected.

Question. Does it happen often that jobs are rejected?

Alternative formulation. What is the *probability* of a job being rejected?

Examples

Some natural questions:

- how long does it take to process a certain task?
- when will the next job arrive?
- what is the success rate of a packet transmission?

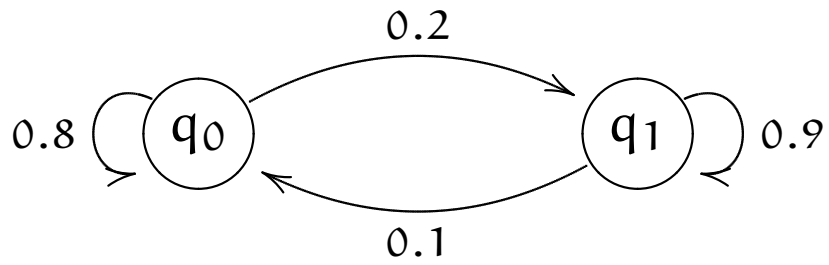
Quantitative information. So far, we could only say *yes* or *no*. However, we could not ask questions such as:

- What is the *likelihood* of a system failure?
- How many jobs are there in the queue *on average*?
- What is the *probability* that the system enters into a dangerous state?

Underlying model

Idea. A collection of states with probabilities attached to changes of state.

Example.



Intended reading. In going from one time step to the next, we change state

- from q_0 to q_0 with probabability 0.8;
- from q_1 to q_1 with probabability 0.9;
- from q_0 to q_1 with probabability 0.2;
- from q_1 to q_0 with probability 0.1.

What does this all mean?

Probability basics

Before looking at models, let us introduce some facts on probability.

Let $[0,1]$ denote the set $\{x \in \mathbb{R} : 0 \leqslant x \leqslant 1\}$.

Probability spaces. This is where "things happen".

A (discrete) *probability space* is a pair (Ω, μ) where:

- Ω is a countable set of *elementary events*;
- $\mu : \Omega \to [0,1]$ assigns a *probability* to each event;
- $\sum_{\omega \in \Omega} \mu(\omega) = 1$.

In our examples, the set Ω will generally be finite. In addition:

- an *event* A is a subset $A \subseteq \Omega$ and $\mu(A)$ is defined to be $\sum_{\omega \in A} \mu(a)$;
- events A and B are said to be *independent* if $\mu(A \cap B) = \mu(A) \cdot \mu(B)$.

Example

Rolling a die:

- elementary events: the set of possible outcomes, i.e.

$$\Omega = \{1,2,3,4,5,6\};$$

- in this case all the elementary events are equally likely: we have that

$$\mu(\omega) = \frac{1}{6} \text{ for all } \omega \in \Omega.$$

Some events:

- $E = \{2,4,6\}$: "we get an even number" and $\mu(E) = 3 \times 1/6 = 1/2$;
- $O = \{1,3,5\}$: "we get an odd number" and $\mu(O) = 1/2$;
- $D = \{3,6\}$: "the number is divisible by 3" and $\mu(D) = 2 \times 1/6 = 1/3$.

Independent events

Independent events. Which events have a bearing on one another?

- E and O *are not* independent:

$$\mu(\mathrm{E} \cap \mathrm{O}) = 0 \neq \frac{1}{6} \times \frac{1}{6} = \mu(\mathrm{E}) \times \mu(\mathrm{O}).$$

- E and D *are* independent:

$$\begin{aligned} \mu(\mathrm{E} \cap \mathrm{D}) &= \mu(\{6\}) \\ &= \frac{1}{6} \\ &= \frac{1}{2} \times \frac{1}{3} \\ &= \mu(\mathrm{E}) \times \mu(\mathrm{D}). \end{aligned}$$

The concept of independent events will be of importance later.

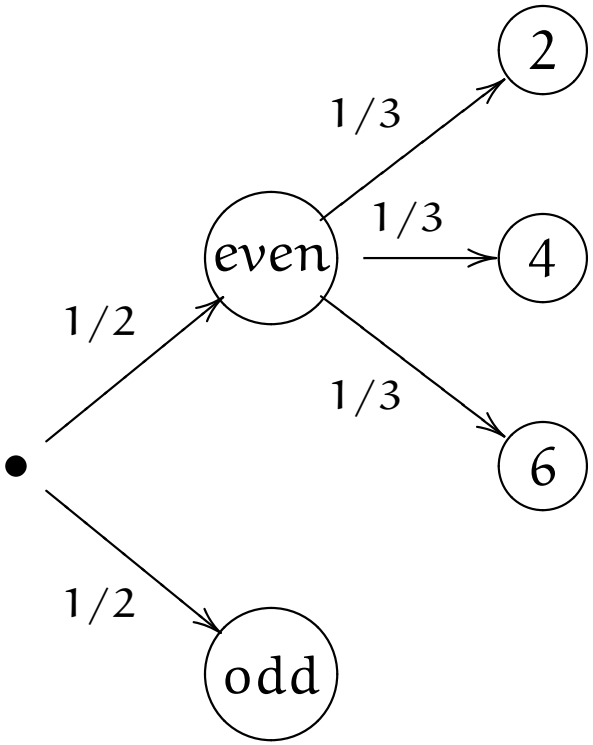
Independent events

Idea. Two events $\mathbb{E}$ and $\mathbb{F}$ are independent if, knowing that one has happened, we don't know any more about the likelihood of the other.

Die example. Suppose we *know* that the number is even. What is the chance of it being divisible by 3?

- there is now a smaller set of possible outcomes, $\{2,4,6\}$, all of which are equally likely;
- the chance of picking a number divisible by 3 *out of the smaller set* $\{2,4,6\}$ is $1/3$;
- the chance of picking a number divisible by 3 *out of the whole set* $\{1,2,3,4,5,6\}$ was $2/6 = 1/3$.

Independent events, as a tree



So:

$$\mu(\text{even and divisible by } 3) = \mu(\{6\}) = \frac{1}{6}$$

as we already knew!

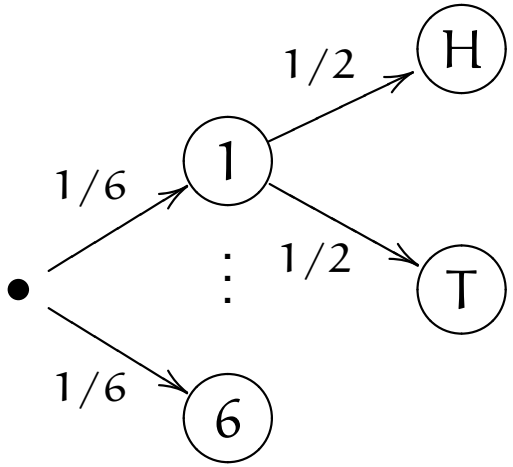
Combining events

Example. First roll a die and then toss a coin.

- $D = (\Omega_D, \mu_D)$ where $\Omega_D = \{1, 2, \ldots, 6\}$ and $\mu_D(\omega) = 1/6$ for each $\omega \in \Omega_D$;
- $C = (\Omega_C, \mu_C)$ where $\Omega_C = \{H, T\}$ and $\mu_C(\omega) = 1/2$ for each $\omega \in \Omega_C$.

Outcomes of the *combined* events:

- one number and one side of the coin: $\Omega_D \times \Omega_C$;
- the chance of getting $(x, y) \in \Omega_D \times \Omega_C$ is $\mu((x, y)) = \mu_D(x) \cdot \mu_C(y)$.



Transmission of data packets

Setup. In a lossy network a packet is received with probability p (where $0 \leqslant p \leqslant 1$).

Elementary events.

$$\Omega = \{\omega_1, \omega_2, \ldots\}$$

where

- ω_i signifies a successful transmission on the i-th attempt.

Successful transmission on the i-th attempt means that we have failed on attempts $1, 2, \ldots, i-1$, but then succeeded on attempt i.

Transmission of data packets

Definition of the probability function.

- successful on first attempt:
  $\mu(\omega_1) = p;$
- successful on second attempt:
  $\mu(\omega_2) = (1-p) \cdot p;$
  - fail on first attempt: $1-p;$
  - success on second attempt:
    $p;$
- successful on third attempt:
  $(1-p)^2 \cdot p;$
  - fail on first attempt: $1-p;$
  - fail on second attempt:
    $(1-p);$
  - success on third attempt: $p;$
- .....

General case.

$$\begin{aligned} \mu(\omega_i) &= (1-p)^{i-1} \cdot p \qquad \text{and} \\ \sum_{i \geqslant 1} \mu(\omega_i) &= \sum_{i \geqslant 1} (1-p)^{i-1} \cdot p = 1. \end{aligned}$$

Basic laws of probability

Suppose that (Ω, μ) is a probability space and that A and B are events.

Then the following hold:

1. $\mu(\emptyset) = 0$ and $\mu(\Omega) = 1$.
2. $\mu(\Omega - A) = 1 - \mu(A)$.
3. $\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B)$; hence
4. $\mu(A \cup B) \leqslant \mu(A) + \mu(B)$.

Remember. An *elementary event* is an element ω in Ω but an *event* is a subset A of Ω.

Conditional probability

Idea. We want to express the probability $\mu(A|B)$ of an event A happening knowing that an event B has already happened:

$$\mu(A|B) = \frac{\mu(A \cap B)}{\mu(B)}.$$

Special case: A and B are independent:

$$\mu(A|B) = \frac{\mu(A \cap B)}{\mu(B)} = \frac{\mu(A)\mu(B)}{\mu(B)} = \mu(A).$$

So, for independent events, *knowing* that B has already happened has no impact on the likelihood of A happening.

Conditional probability

Example. Take a standard pack of playing cards: four suits (spades, hearts, diamonds, and clubs) with each suit having cards of values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (Jack), 12 (Queen) and 13 (King).

Draw a card c at random. Consider the following three events:

A: the card c is a spade;

B: the card c has an even number on it;

C: the card c is a Queen.

We have:

$$\mu(A) = \frac{1}{4}; \quad \mu(B) = \frac{24}{52} = \frac{6}{13}; \quad \mu(C) = \frac{4}{52} = \frac{1}{13}$$

Conditional probability

A and B are independent events:

$$\mu(A \cap B) = \frac{6}{52} = \frac{1}{4} \times \frac{6}{13} = \mu(A) \cdot \mu(B).$$

A and C are independent events:

$$\mu(A \cap C) = \frac{1}{52} = \frac{1}{4} \times \frac{1}{13} = \mu(A) \cdot \mu(C).$$

B and C are not independent events:

$$\mu(B \cap C) = \frac{4}{52} \neq \frac{6}{13} \times \frac{1}{13} = \mu(B) \cdot \mu(C).$$

Conditional probability

For the conditional probabilities $\mu(B|C)$ and $\mu(C|B)$, we have:

$$\mu(B|C) = \frac{\mu(B \cap C)}{\mu(C)} = \frac{\mu(C)}{\mu(C)} = 1;$$

$$\mu(C|B) = \frac{\mu(C \cap B)}{\mu(B)} = \frac{\mu(C)}{\mu(B)} = \frac{1/13}{6/13} = \frac{1}{6}.$$

Application of conditional probability

Typical application:

- we fix a time step (such as 1 second);
- we attach probabilities to the events happening at every step.

Modelling assumption. The system is memoryless, i.e. the probability of an event happening only depends on the events seen in the last timestep, and not on the past before that.

This is the *Markov property*; we will explain this more fully later.

Our models

Notion of model. We will need *conditional probabilities*, with expressions such as

$$\mu(E_{t+1}|F_t),$$

which gives the probability of an event E_{t+1} at time $t+1$ occurring given that the event occurring at the last time step t was F_t.

Note. We will only consider models where these conditional probabilities are independent of time, i.e. the same for all t.

Such systems are called *homogeneous*.

CO4211/CO7211

Discrete Event Systems

Lecture 17:
Random Variables

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Random variables

When we consider a probability space (Ω, μ), the elements of Ω are arbitrary objects. In some instances, it is convenient to be able to manipulate them as numbers.

Definition. Given a probability space (Ω, μ), a *random variable* on (Ω, μ) is a function $X : \Omega \to \mathbb{R}$.

There is a more general definition which allows a wider possibility of sets than $\mathbb{R}$, but that won't concern us here.

Example

Example. If we toss a coin, we have

$$\Omega = \{\mathrm{Head}, \mathrm{Tail}\}, \;\; \mu(\mathrm{Head}) = \mu(\mathrm{Tail}) = 1/2,$$

and we could then have $X : \mathrm{Head} \mapsto 1$, $\mathrm{Tail} \mapsto 0$.

Example. We have two components each of which is either "a" acceptable or "d" (defective); here we have

$$\Omega = \{(a,a),(a,d),(d,a),(d,d)\}$$

(and there would be some associated probabilities). The random variable X could deliver the number of acceptable components, so that:

$$X(a,a) = 2,\; X(a,d) = X(d,a) = 1,\; X(d,d) = 0.$$

Expected value

If we have a discrete probability space (Ω, μ) and a random variable $X : \Omega \to \mathbb{R}$, we define the *expected value* of X to be

$$\sum_{\omega \in \Omega} X(\omega)\mu(\omega).$$

Example. Let $\Omega = \{(a,a),(a,d),(d,a),(d,d)\}$ and X be as on the previous slide and let

$$\mu(a,a) = \frac{1}{2},\ \mu(a,d) = \frac{1}{6},\ \mu(d,a) = \frac{1}{4},\ \mu(d,d) = \frac{1}{12}.$$

Then

$$E(X) = (2 \times \frac{1}{2}) + (1 \times \frac{1}{6}) + (1 \times \frac{1}{4}) + (0 \times \frac{1}{12}) = \frac{17}{12}.$$

Packet transmission

Example. Routing network packets from a machine A to a machine B, where the success of the transmission depends on chance.

We might make an attempt to transmit a packet at times $t = 0, 1, 2, 3, \ldots$.

If so, then a natural model would be to have a random variable

$$X_t = \begin{cases} 1 & \text{the transmission at time } t \text{ was successful;} \\ 0 & \text{otherwise} \end{cases}$$

for each $t \in \mathbb{N}$.

Model of the system

In this sort of situation, in order to model the system, we would have one random variable for every time point.

System model. An indexed family of random variables $(X_t)_{t \geqslant 0}$.

We would then make some modelling assumptions, such as X_{t+1} only depending on X_t for each t.

In this case, the system could be described using *transition probabilities*; for example, we could have:

$$\mu(X_{t+1} = 0 | X_t = 0) = 0.8, \qquad \mu(X_{t+1} = 0 | X_t = 1) = 0.1,$$
$$\mu(X_{t+1} = 1 | X_t = 0) = 0.2, \qquad \mu(X_{t+1} = 1 | X_t = 1) = 0.9.$$

Random processes

A *random process* is a collection $\{X_i : i \in I\}$ of random variables X_i indexed by some set I.

In our case I will represent time. As we will be considering *discrete* time, we will take I to be $\mathbb{N}$. The element 0 of $\mathbb{N}$ represents the starting point of the process.

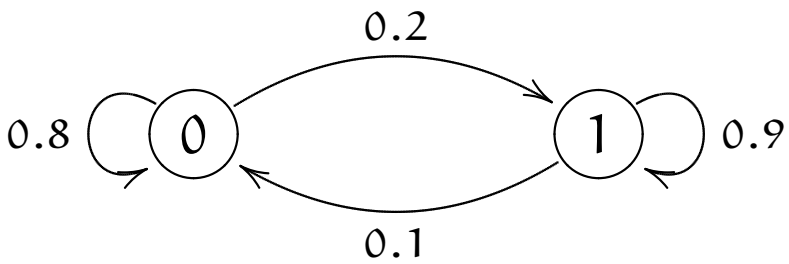
(If we were considering continuous time, I would probably be the set $\mathbb{R}^+$ of positive real numbers.)

Model of the system

Transition probabilities:

$$\mu(X_{t+1} = 0 | X_t = 0) = 0.8, \qquad \mu(X_{t+1} = 0 | X_t = 1) = 0.1,$$
$$\mu(X_{t+1} = 1 | X_t = 0) = 0.2, \qquad \mu(X_{t+1} = 1 | X_t = 1) = 0.9.$$

We can represent this as a *transition diagram*:



Model. We have a random process, i.e. one random variable X_t for every time point t. In this case, $X_t = 0$ or $X_t = 1$ for every $t \in \mathbb{N}$.

Markov chains

A random process is called a *Markov process* if the future at every point depends only on the current state of the system and not the past.

In our setting, this becomes:

$$\begin{aligned}\mu(X_{t+1} = q_{t+1} | X_t = q_t, X_{t-1} = q_{t-1}, \ldots, X_0 = q_0) \\ = \mu(X_{t+1} = q_{t+1} | X_t = q_t).\end{aligned}$$

Such a random process is called a *Markov chain*.

Probabilistic transition systems

Definition. A *probabilistic transition system* is a pair (Q, p) where:

- Q is a set of states, and
- $p : Q \times Q \to [0, 1]$ is a transition probability function where

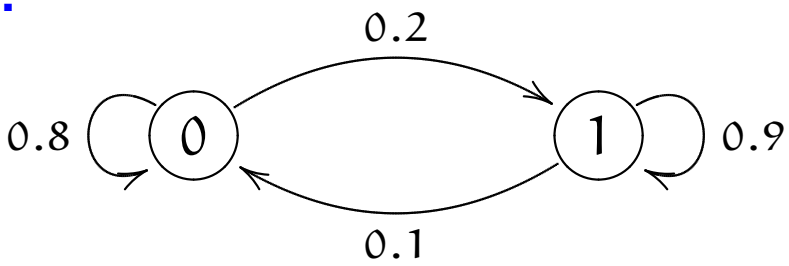
$$\sum_{q' \in Q} p(q, q') = 1 \text{ for all } q \in Q.$$

Idea:

- $p(q, q')$ is the *probability* that the system makes a transition from state q to q' in the next time step ...
- ... and these probabilities (for every fixed q) have to add up to one – the system *has* to make a step.

Alternative representation

Transition diagram:



Alternative representation:

$$M = \begin{pmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{pmatrix} = \begin{pmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{pmatrix}$$

where the (i,j)-entry p_{ij} represents $\mu(X_{t+1} = j | X_t = i)$.

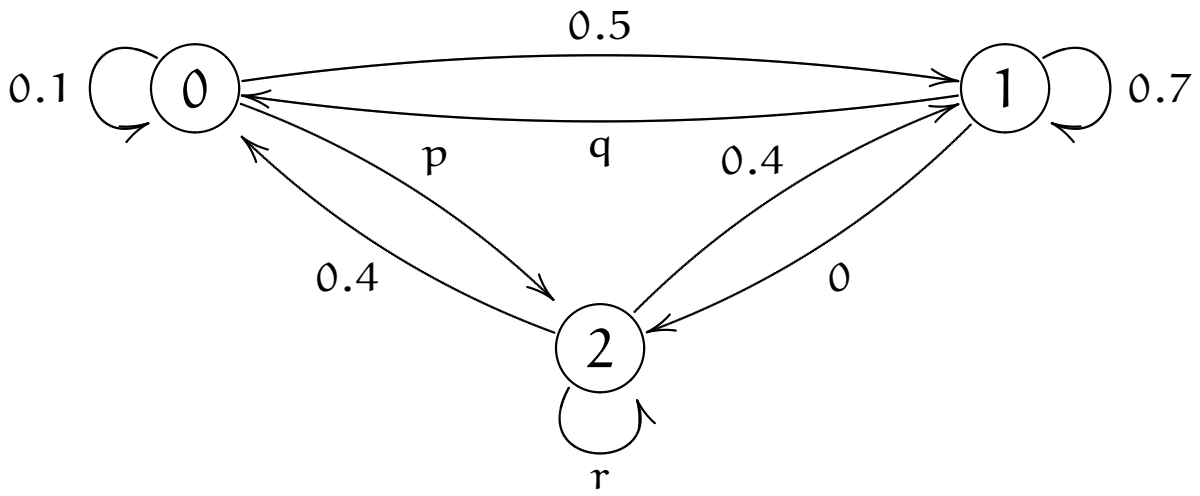
The sequence $(X_t)_{t \geqslant 0}$ of random variables is the Markov chain; it is *discrete* (we are taking a discrete model of time) and *homogeneous* (the probability $\mu(X_{t+1} = j | X_t = i)$ is independent of t).

Missing probabilities

Observation. The condition

$$\sum_{r \in Q} p(q,r) = 1 \qquad \text{for all } q \in Q$$

means that we can sometimes recover *missing probabilities*; for example, consider:

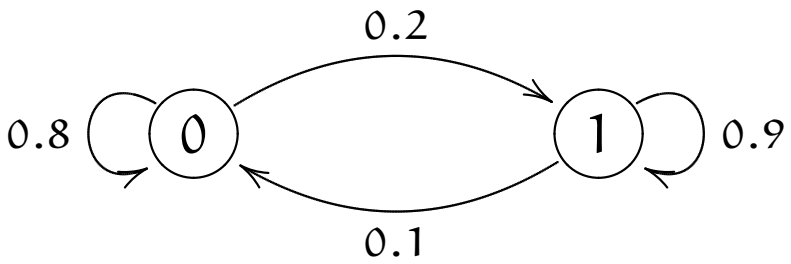


Missing probabilities: $p = 0.4$, $q = 0.3$, $r = 0.2$.

We will adopt the convention that all arcs labelled 0 are omitted.

Evolution of the system

Transition diagram:



Question. What is the probability that the t-th transmission is successful (represented here by the state 1)?

Evolution of the system

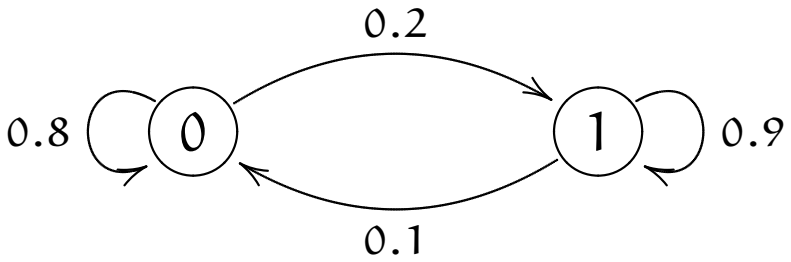
Question. What is the probability that the t-th transmission is successful?

Answer. We want $\mu(X_t = 1)$. Here is how it looks.

t	$\mu(X_t = 0)$	$\mu(X_t = 1)$	
0	*0*	*1*	*the given initial state*
1	0.1	0.9	
2	0.17	0.83	
3	0.219	0.781	
4	0.253	0.747	
	...	...	
1000	0.333	0.667	

Question. Could we have deduced this without a long calculation?

Transition probabilities

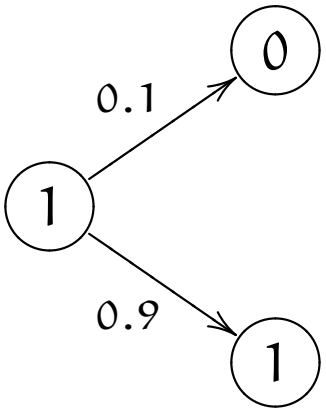


$$\begin{pmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{pmatrix}$$

We write $\pi_t = (p_0, p_1)$ if the system is in state i with probability p_i at time t.

Suppose that we definitely start in state 1, i.e. we have $\pi_0 = (0,1)$.

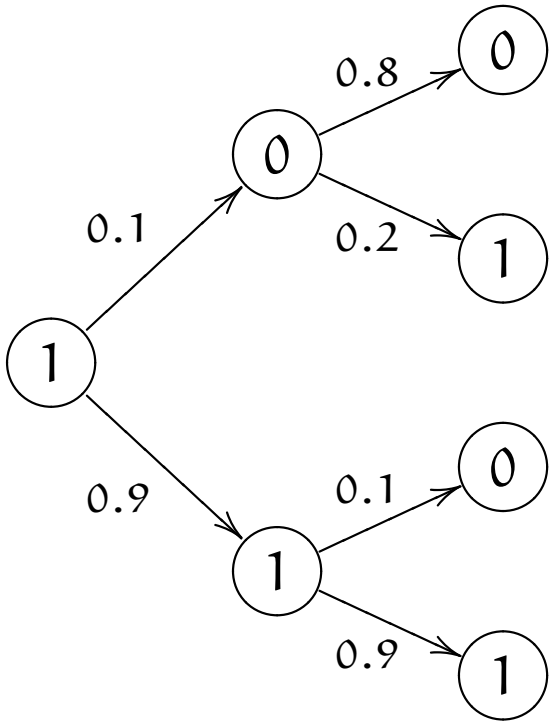
Next step:



$$\pi_1 = \pi_0 \cdot M = (0,1) \cdot \begin{pmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{pmatrix} = (0.1, 0.9)$$

Transition probabilities

After two time steps:



We get: $\pi_2 = (0.1, 0.9) \cdot \begin{pmatrix} 0.8 & 0.2 \\ 0.1 & 0.9 \end{pmatrix} = (0.17, 0.83).$

k-step transition probabilities

Given:

- a probabilistic transition system with n states, i.e. an $n \times n$ transition matrix M;
- an initial probability distribution $\pi_0 = (p_1, \ldots, p_n)$.

Wanted:

- the probability distribution after k time steps.

Note: One can study probabilistic transition systems with an infinite number of states, but we will concentrate on finite systems here.

k-step transition probabilities

General algorithm.

- We write $\pi_t = (p_1, \ldots, p_n)$ if the system is in state i with probability p_i at time t.
- π_0 is given.
- Put $\pi_{t+1} = \pi_t \cdot M$ and continue until we find π_k.

Formula.

$$\pi_{t+k} = \pi_{t+k-1} \cdot M = (\pi_{t+k-2} \cdot M) \cdot M = (\pi_{t+k-3} \cdot M) \cdot M \cdot M = \ldots$$

So we have that $\pi_{t+k} = \pi_t M^k$. In particular we see that

$$\pi_k = \pi_0 M^k.$$

Analyzing Markov chains

Given:

- a Markov chain with n states, i.e. an $n \times n$ transition matrix M;
- an initial probability distribution $\pi_0 = (p_1, \ldots, p_n)$.

The probability distribution after k steps is given by $\pi_k = \pi_0 \cdot M^k$.

Question. What is the long-term behaviour of the Markov chain?

What happens to the probability distribution π_k as k becomes very large?

CO4211/CO7211

Discrete Event Systems

Lecture 18:
Analyzing Markov chains

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Analyzing Markov chains

Given:

- a probabilistic transition system with n states, i.e. an $n \times n$ transition matrix M;
- an initial probability distribution $\pi_0 = (p_1, \ldots, p_n)$.

The probability distribution after k steps is given by $\pi_k = \pi_0 \cdot M^k$.

Question. What is the long-term behaviour of the system?

What happens to the probability distribution π_k as k becomes very large?

To answer this question we need to describe various properties of probabilistic transition systems.

Notation

Suppose that we have a probabilistic transition system with n states, i.e. an $n \times n$ transition matrix

$$M = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} \end{pmatrix}.$$

Notation

For every $k \geqslant 1$, let

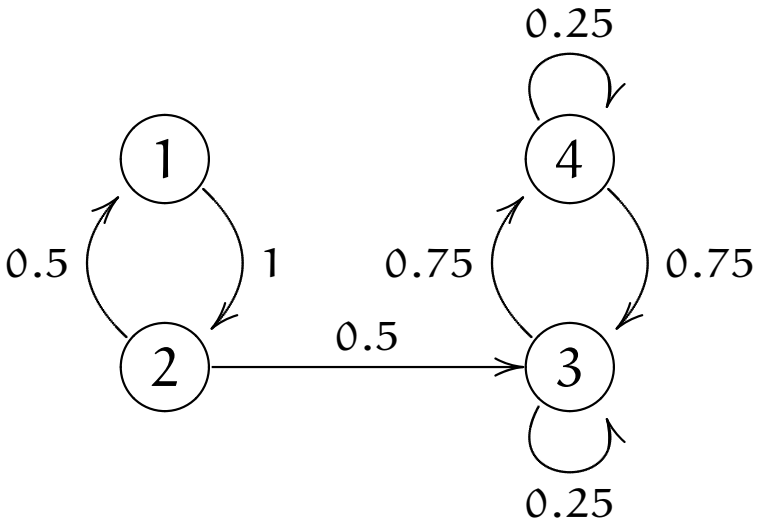
$$M^k = \begin{pmatrix} p_{1,1,k} & p_{1,2,k} & \cdots & p_{1,n,k} \\ p_{2,1,k} & p_{2,2,k} & \cdots & p_{2,n,k} \\ \vdots & \vdots & & \vdots \\ p_{n,1,k} & p_{n,2,k} & \cdots & p_{n,n,k} \end{pmatrix}.$$

Here $p_{i,j,k}$ is the probability that we move from state i to state j in k steps.

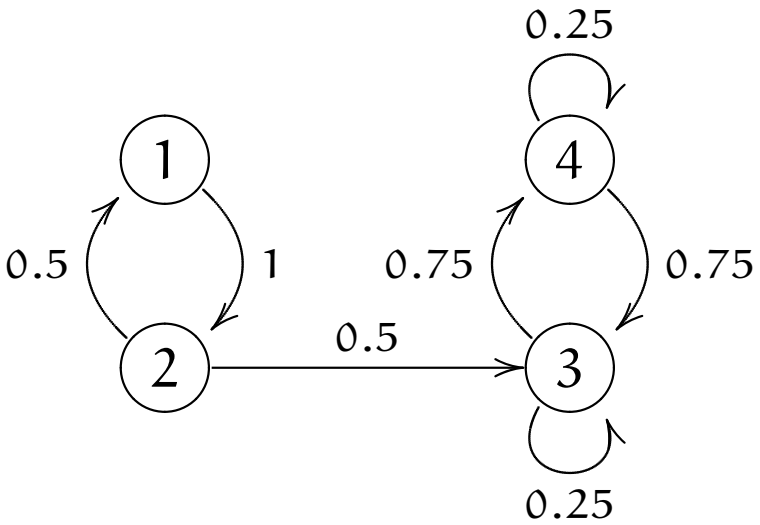
Reachability

Definition. We say that a state j is *reachable* from a state i if $p_{i,j,k} > 0$ for some $k \geqslant 1$.

In the state transition diagram, this is equivalent to finding a path from state i to state j (without using arcs with label 0 if such arcs are included in the diagram). For example, consider the transition diagram below:



Reachability



We see that:

state 1 is only reachable from itself and state 2;

state 2 is only reachable from itself and state 1;

state 3 is reachable from all four states;

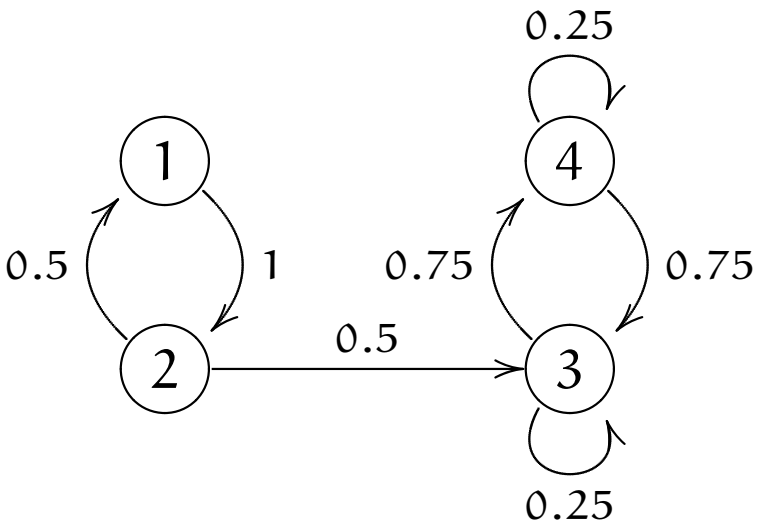
state 4 is reachable from all four states.

Closed sets

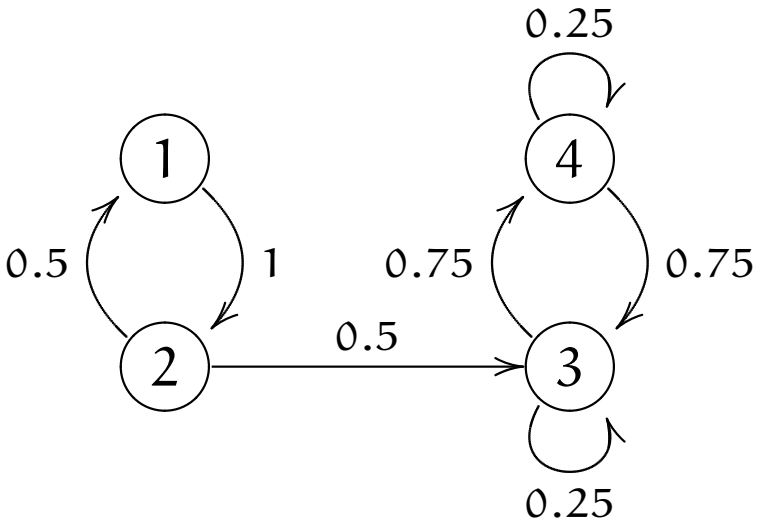
Definition. A subset S of the state space is said to be *closed* if $p_{i,j} = 0$ for all states $i \in S$ and $j \notin S$.

This means that there is no feasible transition from a state in S to a state outside S.

For example, consider the transition diagram below:



Closed sets



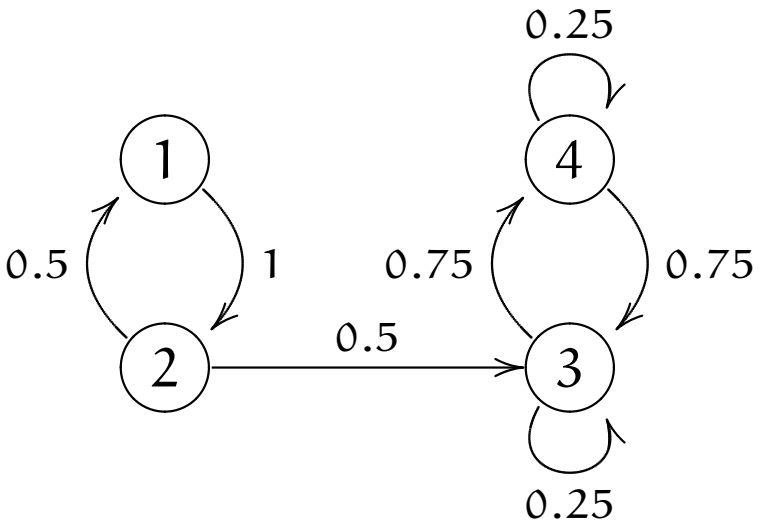
We see that the set $\{1,2,3,4\}$ of all the states is closed (this will always be the case). The set $\{3,4\}$ of states is also closed.

However, the set $S = \{2,3,4\}$ of states is not closed (as $p_{2,1} = 0.5 > 0$ with $2 \in S$ and $1 \notin S$).

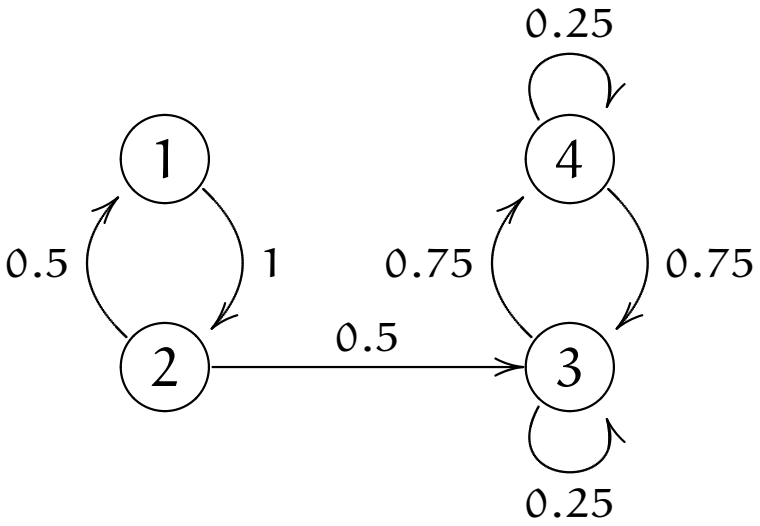
Irreducible sets

Definition. A closed set S of states is said to be *irreducible* if state j is reachable from state i for all states i and j in S.

For example, consider the transition diagram below:



Irreducible sets



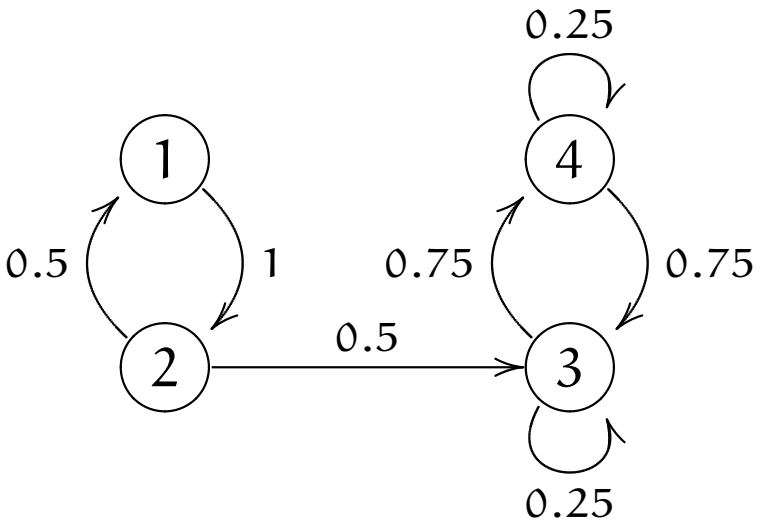
We see that the set $\{1,2,3,4\}$ of all states is not irreducible; we cannot reach state 2 from state 3 for example.

However, the set $\{3,4\}$ is irreducible; we can reach state 3 from state 4 and state 4 from state 3.

Irreducible systems

Definition. A probabilistic transition system is said to be *irreducible* if its state space is irreducible.

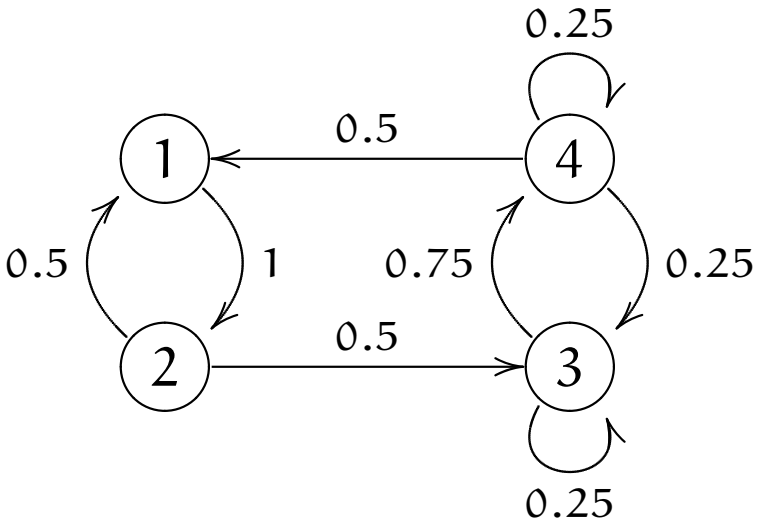
For example, consider the transition diagram below:



This system is not irreducible: as we saw above, the set $\{1,2,3,4\}$ of states is not irreducible.

Irreducible systems

For another example, consider the transition diagram below:



In this system, every state is reachable from every state; so the system is irreducible.

Recurrence

Suppose that a Markov chain is in a particular state i; will it return to state i?

If the answer is "definitely yes", then the state i is said to be "recurrent"; otherwise i is "transient".

More formally, we define the *hitting time* of a Markov chain as follows:

$$T_{i,j} = \min\{k > 0 : X_0 = i, X_k = j\}.$$

This is the *first* time that the chain is in state j after starting in state i.

If $i = j$, then $T_{i,i}$ is the first time that the chain returns to state i having started there; this is the *recurrence time* of state i.

Recurrence

Let

$$\theta_{i,k} = \mu(T_{i,i} = k),$$

i.e. $\theta_{i,k}$ is the probability that the recurrence time of state i is k. We then let

$$\theta_i = \sum_{k=1}^{\infty} \theta_{i,k},$$

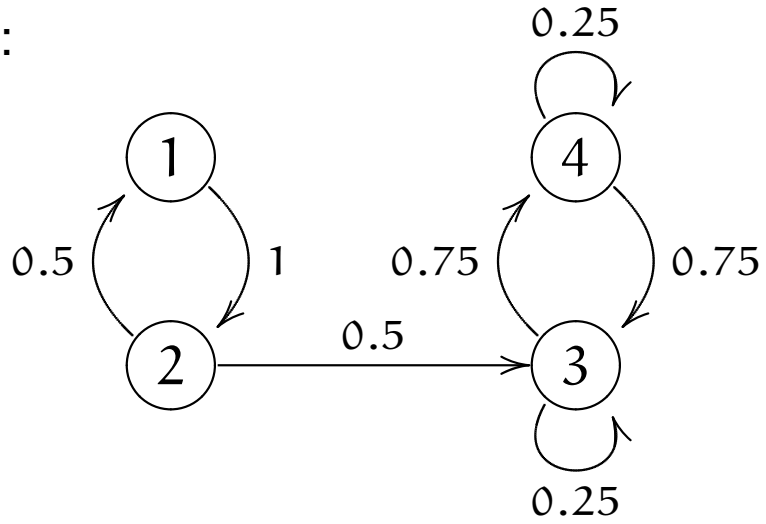
so that θ_i is the probability of returning to state i (at some stage) after having started there.

Definition. A state i is said to be *recurrent* if $\theta_i = 1$ and *transient* if $\theta_i < 1$.

A recurrent state is definitely visited again but a transient state may not be revisited (with probability $1 - \theta_i$).

Recurrence

Consider the system:



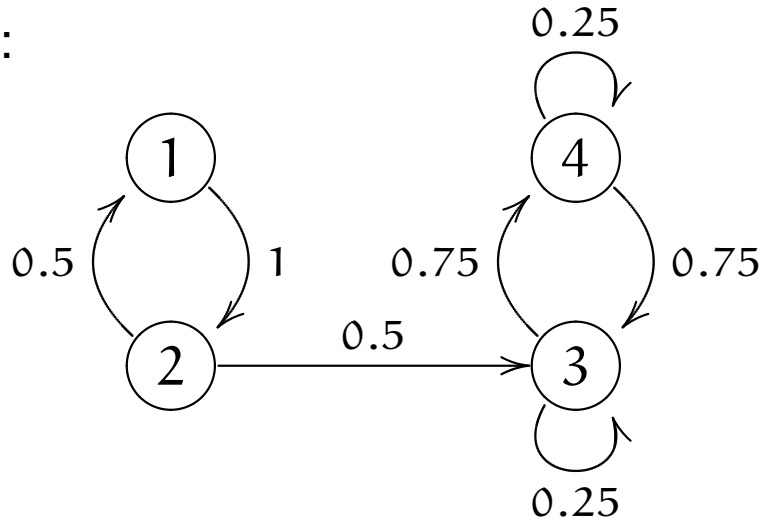
$$\theta_{1,1} = \mu(T_{1,1} = 1) = 0; \quad \theta_{1,2} = \mu(T_{1,1} = 2) = 1/2;$$

$$\theta_{1,k} = \mu(T_{1,1} = k) = 0 \quad \text{if } k \geqslant 3.$$

Here: $\theta_1 = \sum_{k=1}^{\infty} \theta_{1,k} = 1/2$, and state 1 is therefore transient.

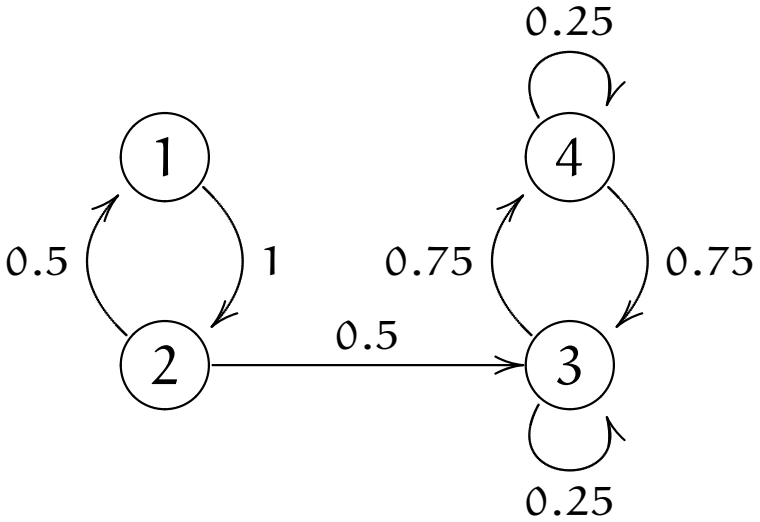
Recurrence

Consider the system:



$$
\begin{aligned}
\theta_{4,1} = \quad & \mu(T_{4,4} = 1) = \quad 1/4; \\
\theta_{4,2} = \quad & \mu(T_{4,4} = 2) = \quad 3/4 \times 3/4 = 9/16; \\
\theta_{4,3} = \quad & \mu(T_{4,4} = 3) = \quad 3/4 \times 1/4 \times 3/4 = 9/64; \\
\theta_{4,4} = \quad & \mu(T_{4,4} = 4) = \quad 3/4 \times (1/4)^2 \times 3/4 = 9/256; \\
\ldots \quad & \ldots \quad \ldots
\end{aligned}
$$

Recurrence



Here:

$$\begin{aligned}\theta_4 &= \sum_{k=1}^{\infty} \theta_{4,k} = \tfrac{1}{4} + \tfrac{9}{16} + \tfrac{9}{64} + \tfrac{9}{256} + \dots \\ &= \tfrac{1}{4} + \tfrac{9}{16}(1 + \tfrac{1}{4} + \tfrac{1}{16} + \tfrac{1}{64} + \dots) \\ &= \tfrac{1}{4} + \tfrac{9}{16}\left(\tfrac{1}{1-\frac{1}{4}}\right) = \tfrac{1}{4} + \tfrac{3}{4} = 1,\end{aligned}$$

and state 4 is therefore recurrent.

Recurrence

Let i be a recurrent state. We let M_i be the *mean recurrence time* of state i:

$$M_i = \sum_{k=1}^{\infty} k \times \theta_{i,k}.$$

Given that $\theta_{i,k}$ is the probability that the first time we revisit state i is after k steps, we see that M_i is the expected time before we return to state i.

If M_i is finite then we say that the state i is *positive recurrent*; otherwise we say that i is *null recurrent*.

We are interested in systems with a finite number of states; we will see that this distinction (between null recurrent and positive recurrent) is not so important in such a situation.

Recurrence

We have divided our states into three types:

- *positive recurrent*: definitely revisited with finite expected recurrence time;
- *null recurrent*: definitely revisited but with infinite expected recurrence time;
- *transient*: may not be revisited.

Theorem. If S is a finite (closed) irreducible set of states then every state in S is positive recurrent.

Hence we have:

Corollary. If we have a finite irreducible Markov chain then every state is positive recurrent.

Periodic states

In some Markov chains we can only revisit a state after a number of steps which is a multiple of some integer $d \geqslant 2$; such states are called "periodic".

More formally, a state i is said to be *periodic* if the greatest common divisor of the set

$$\{k > 0 : p_{i,i,k} > 0\}$$

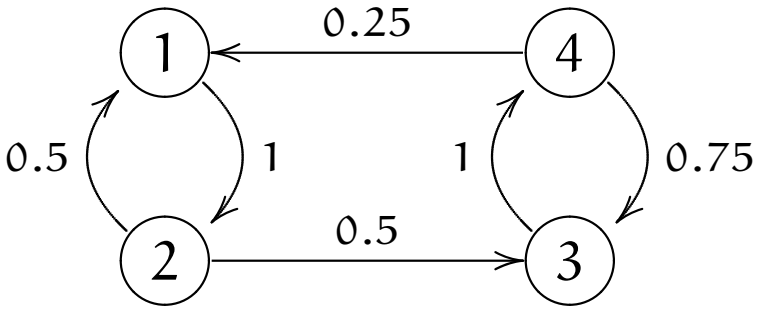
is greater than one and *aperiodic* otherwise.

(Remember that $p_{i,i,k}$ is the probability that we move from state i to state i in k steps.)

As a result, if we start in state i and if k is not a multiple of d, then we cannot be back in state i after k steps.

Periodic states

Consider the system:



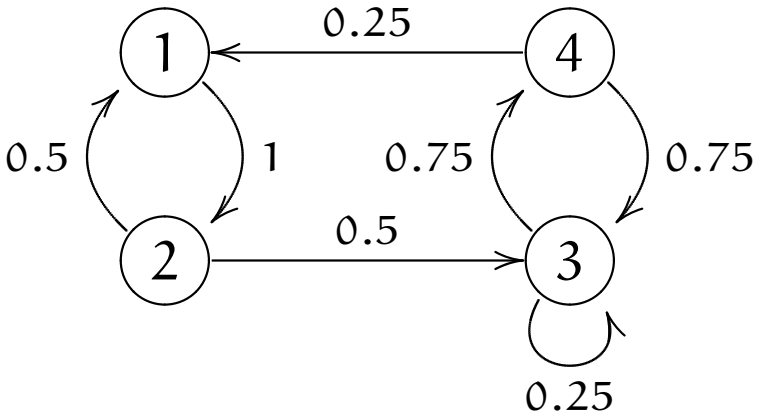
The system is irreducible (we can reach every state from every other state).

If we leave state 1 we can only return after $2, 4, 6, 8, 10, \ldots$ steps; so state 1 is periodic.

In fact, all the states in this system are periodic.

Periodic states

Consider the system:



The system is irreducible (we can reach every state from every other state).

If we leave state 1 we can only return after $2, 4, 5, 6, 7, \ldots$ steps; so state 1 is aperiodic.

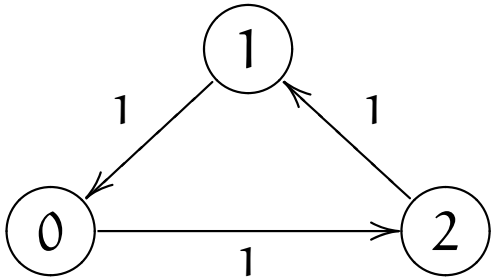
In fact, all the states in this system are aperiodic.

Periodic states

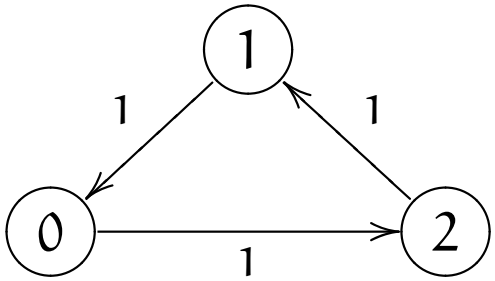
A very important fact here is the following:

Theorem. If a probabilistic transition system is irreducible then either all its states are periodic or all its states are aperiodic.

If the states are periodic then we would not expect to get a stationary behaviour. For example, consider the system



Periodic states



If the initial probability distribution is $(1,0,0)$, then the successive probability distributions will be

$$(0,1,0),(0,0,1),(1,0,0),(0,1,0),\ldots\ldots;$$

the system clearly cannot settle down to a stationary distribution.

CO4211/CO7211

Discrete Event Systems

Lecture 19:
Stationary Behaviour

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Remember . . .

A state j is *reachable* from a state i if $p_{i,j,k} > 0$ for some $k \geqslant 1$.

A subset S of the state space is *closed* if $p_{i,j} = 0$ for every states $i \in S$ and $j \notin S$.

A closed set S of states is *irreducible* if state j is reachable from state i for all states i and j in S.

A probabilistic transition system is *irreducible* if its state space is irreducible.

A *recurrent* state is one that will definitely be visited again whilst a *transient* state may not be revisited

Remember . . .

If the mean recurrence time is finite then the state is positive recurrent; otherwise it is null recurrent.

Theorem. If S is a finite (closed) irreducible set of states then every state in S is positive recurrent.

Corollary. If we have a finite irreducible Markov chain then every state is positive recurrent.

A state is periodic if we can only revisit it after a number of steps which is a multiple of some integer $\mathrm{d} \geqslant 2$; otherwise it is aperiodic.

Theorem. If a probabilistic transition system is irreducible then either all its states are periodic or all its states are aperiodic.

Restating our question

The question we asked was the following:

Given:

- a probabilistic transition system with n states, i.e. an $n \times n$ transition matrix M;
- an initial probability distribution $\pi_0 = (x_1, \ldots, x_n)$.

The probability that the system is in state s after k steps is given by $\pi_k = \pi_0 \cdot M^k$.

Question. What is the long-term behaviour of the system?

What happens to the distribution π_k as k gets very large?

Stationary behaviour

We could ask the following:

Question. When does the behaviour of a Markov chain settle down to a stable pattern?

Given that the probability distribution π_k after k steps is given by $\pi_k = \pi_0 \cdot M^k$, we are asking if there is a probability distribution π such that $\pi_k \to \pi$ as $k \to \infty$.

If such a probability distribution $\pi = (p_1, p_2, \ldots, p_n)$ exists, then we should have:

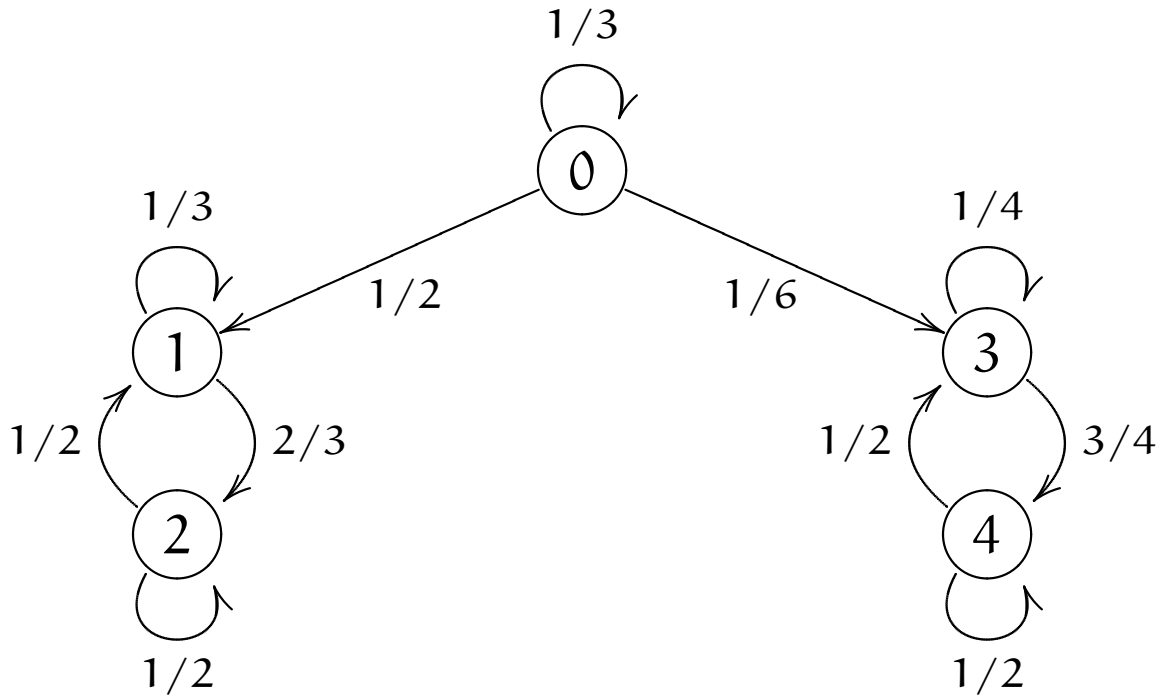
$$\pi = \pi \cdot M; \quad p_i \geqslant 0 \text{ for all } i; \quad \sum_{i=1}^{n} p_i = 1.$$

We call such a distribution π (if it exists) a *stationary probability distribution*.

Reducible chains

If the Markov chain is *reducible* (i.e. not irreducible), then it eventually enters some irreducible closed subset of the set of states and stays there.

There is the question of the probability of entering any particular such irreducible subset; for example, consider the system below:



Reducible chains

In this example, state 0 is transient. There are two irreducible sets of states, $\{1,2\}$ and $\{3,4\}$.

The probability that we end up in $\{1,2\}$ is

$$\frac{1}{2}\left(1+\frac{1}{3}+\frac{1}{9}+\cdots\right) = \frac{1}{2}\left(\frac{1}{1-\frac{1}{3}}\right) = \frac{3}{4},$$

and the probability that we end up in $\{3,4\}$ is

$$\frac{1}{6}\left(1+\frac{1}{3}+\frac{1}{9}+\cdots\right) = \frac{1}{6}\left(\frac{1}{1-\frac{1}{3}}\right) = \frac{1}{4}.$$

Periodic systems

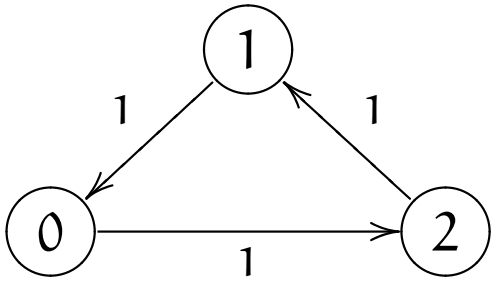
Question. When does the behaviour of a Markov chain settle down to a stable pattern?

We will concentrate on irreducible systems.

If the irreducible system is periodic, then no such limiting distribution π exists. We have a number $d > 1$ such that we can only revisit states after k steps if k is a multiple of d.

Periodic systems

For example, if we have the system



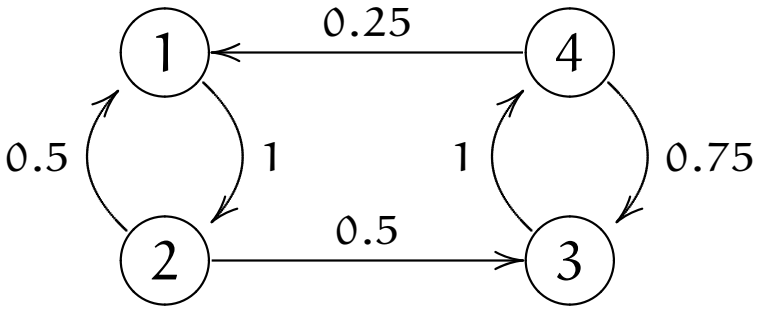
and the initial probability distribution is $(1,0,0)$, then (as we saw before) the successive probability distributions will be

$$(0,1,0),(0,0,1),(1,0,0),(0,1,0),\ldots\ldots;$$

the system clearly cannot settle down to a stationary distribution.

Periodic systems

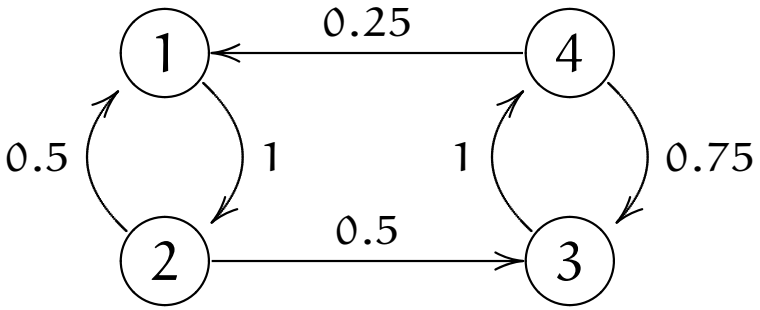
Let's look at another example:



Here the transition matrix is

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 1/4 & 0 & 3/4 & 0 \end{pmatrix}$$

Periodic systems



If the initial probability distribution is $(1,0,0,0)$ then successive probability distributions are:

$$(0,1,0,0) \qquad (\tfrac{1}{2},0,\tfrac{1}{2},0) \qquad (0,\tfrac{1}{2},0,\tfrac{1}{2})$$
$$(\tfrac{3}{8},0,\tfrac{5}{8},0) \qquad (0,\tfrac{3}{8},0,\tfrac{5}{8}) \qquad (\tfrac{11}{32},0,\tfrac{21}{32},0)$$
$$(0,\tfrac{11}{32},0,\tfrac{21}{32}) \qquad (\tfrac{43}{128},0,\tfrac{85}{128},0) \qquad \ldots\ldots$$

Stationary behaviour

The situation for finite irreducible aperiodic systems is summed up as follows:

Theorem. Consider a finite irreducible probabilistic transition system.

1. The limit $\pi = \lim_{k\to\infty} \pi_k$ always exists and is independent of the initial probability vector π_0.
2. There is a unique stationary probability distribution $\pi = (p_1, p_2, \ldots, p_n)$ such that

$$\pi = \lim_{k\to\infty} \pi_k \quad \text{and} \quad p_i > 0 \text{ for all } i.$$

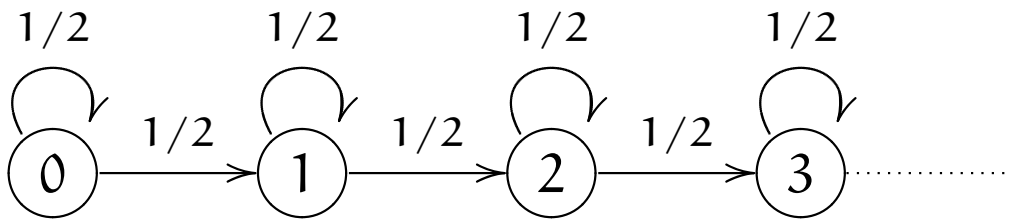
We may determine π by solving the equations

$$\pi = \pi \cdot M \quad \text{and} \quad \sum_{i=1}^{n} p_i = 1.$$

Infinite systems

Note. The fact that every state is positive recurrent in a finite irreducible probabilistic transition system is crucial here.

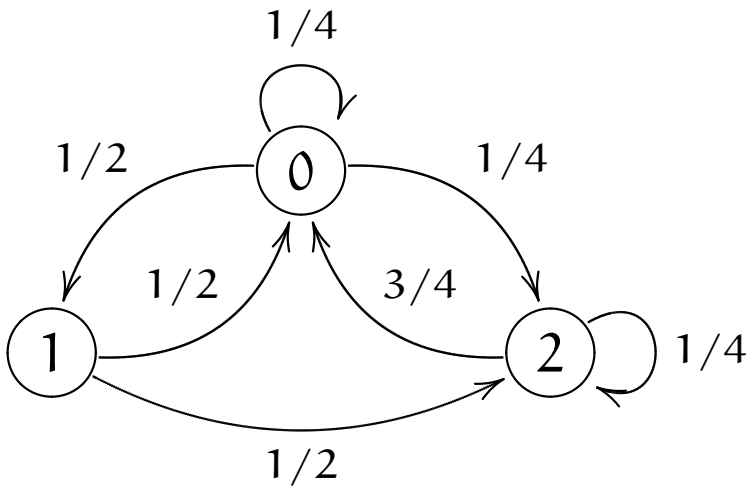
The theorem we have just given is not valid for infinite systems. Such a system can consist of transient or null recurrent states; for example, we could have the following system where all the states are transient:



In a case such as this we have that $\pi = \lim_{k\to\infty} \pi_k$ is the distribution $(0, 0, 0, \ldots)$ and no stationary probability distribution exists.

Example

Consider the system below:



This is irreducible and so should have a stationary probability distribution.

Example

The transition matrix for this system is:

$$M = \begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & 1/2 \\ 3/4 & 0 & 1/4 \end{pmatrix}.$$

We want to solve the equations

$$(x,y,z)M = (x,y,z), \quad x+y+z = 1.$$

in other words

$$\begin{aligned} x/4+y/2+3z/4 &= x & \qquad x/2 &= y \\ x/4+y/2+z/4 &= z & \qquad x+y+z &= 1 \end{aligned}$$

Example

$$\begin{aligned} x/4+y/2+3z/4 &= x & \qquad x/2 &= y \\ x/4+y/2+z/4 &= z & \qquad x+y+z &= 1 \end{aligned}$$

From the second equation we have that $x = 2y$. Putting that into the third equation yields that

$$y/2+y/2+z/4 = z,$$

which gives that $y = 3z/4$ and so $x = 2y = 3z/2$. Now using $x+y+z = 1$ we get that $13z/4 = 1$, so that

$$x = 6/13,\ y = 3/13,\ z = 4/13.$$

Hence the unique stationary probability distribution here is

$$(6/13, 3/13, 4/13).$$

CO4211/CO7211

Discrete Event Systems

Lecture 20:
Introduction to Queuing Theory

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Queueing theory

In the last part of the course we will introduce some topics in the theory of queues and analyze some simple examples. We will have the following general situation:

- there are items that arrive at a service centre to obtain service;
- there are servers in the service centre that can provide service;
- if an item arrives and no server is available then the item joins a queue waiting for service;
- when a server is available an item is processed by that server;
- when a server has finished processing an item the item leaves the service centre.

Examples

There are many examples of queuing systems, such as:

- check-in counters at airports;
- printers processing jobs submitted to them;
- traffic control;
- manufacturing systems;
- allocation of beds in the health service;
- operating systems;
- . . . . . .

Notation

In general, we use the following notation to describe queuing systems (the *Kendal notation*):

$$\mathrm{A}/\mathrm{B}/\mathrm{m}/\mathrm{N}-\mathrm{S}$$

Here:

- A represents the distribution of times between successive arrivals;
- B represents the distribution of service times;
- m is the number of servers available;
- N is the maximum size of the queue (this is taken to be unbounded if no number is specified here);
- S is the service discipline; this is taken to be FIFO (first-in-first-out) if nothing is specified here (we will only consider this case).

Assumptions

We will make some assumptions about our queues:

- the arrival and service rates are not affected by the number of items already in the system.
- the inter-arrival and service times are independent (for example, a long arrival time does not make it more likely that the next one will be short).

Such a system is said to be *Markovian*; the probability of what happens next does not depend on what happened previously.

We will look at $\mathrm{M}/\mathrm{M}/\mathrm{m}/\mathrm{N}$ systems where "M" stands for "Markovian" (i.e. we will consider systems where both the arrival rates and service rates are Markovian).

These assumptions are good for modelling certain situations (but not others).

Arrival and service rates

Two critical values for our system are the following:

- the average arrival rate λ;
- the average service rate μ.

Given these values,

- the average inter-arrival time is $\frac{1}{\lambda}$;
- the average inter-service time is $\frac{1}{\mu}$.

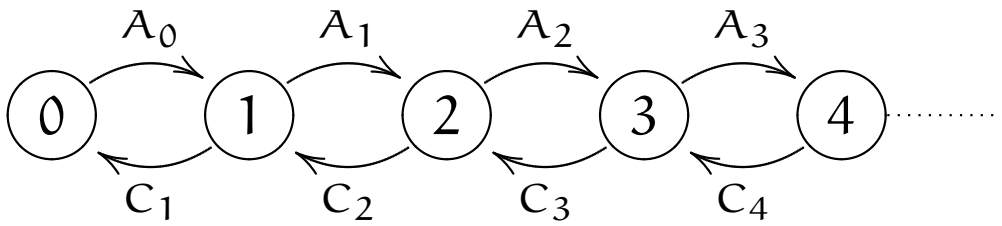
We will first consider $M/M/1$ queues.

We define ρ to be $\frac{\lambda}{\mu}$ - we call this the *utilization rate* (we'll see why it is called this later).

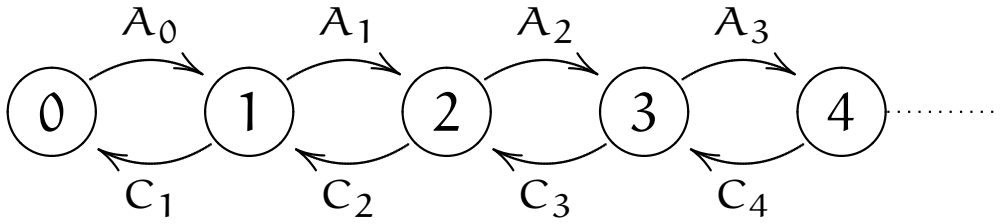
Events in $M/M/1$ queues

Some events. Our state space is $\{0, 1, 2, 3, \ldots\}$, where n represents the number of items in the system (including the item being served if there is one). If we are in state n, then there are two events that can occur:

- A_n: a new item arrives and joins the system.
- C_n: an item is served.



Steady state in $M/M/1$ queues



Let P_n denote the probability that there are n items in the system.

Event A_n occurs with rate λP_n for $n \geqslant 0$.

Event C_n occurs with rate μP_n for $n \geqslant 1$.

In a steady-state, the event A_n occurs as often as event C_{n+1} for each n.

So we have:

$$\lambda P_n = \mu P_{n+1} \text{ for } n \geqslant 0.$$

Steady state in $M/M/1$ queues

$$\lambda P_n = \mu P_{n+1} \text{ for } n \geqslant 0.$$

This gives that:

$$P_{n+1} = \frac{\lambda}{\mu} P_n = \rho P_n \text{ for } n \geqslant 0.$$

So we have:

$$P_1 = \rho P_0,$$

$$P_2 = \rho P_1 = \rho^2 P_0,$$

$$P_3 = \rho P_2 = \rho^3 P_0,$$

$$\ldots\ldots$$

Steady state in $M/M/1$ queues

$$P_1 = \rho P_0, \;\; P_2 = \rho P_1 = \rho^2 P_0, \;\; P_3 = \rho P_2 = \rho^3 P_0, \;\; \ldots\ldots$$

Now we know that $\sum_{n=0}^{\infty} P_n = 1$, and so

$$\sum_{n=0}^{\infty} \rho^n P_0 = 1.$$

It is clear that we must have $0 < \rho < 1$ here (i.e. we must have that $0 < \lambda < \mu$). Since

$$\sum_{n=0}^{\infty} \rho^n = \frac{1}{1-\rho},$$

we have that $P_0 = 1 - \rho$. So the probability that there is at least one item in the system is ρ; hence the term “utilization rate”.

Steady state in $M/M/1$ queues

We have:

$$P_1 = \rho P_0, \;\; P_2 = \rho P_1 = \rho^2 P_0, \;\; P_3 = \rho P_2 = \rho^3 P_0, \;\; \ldots\ldots$$

and $P_0 = 1 - \rho$. So we have:

$$P_1 = \rho(1-\rho),$$
$$P_2 = \rho^2(1-\rho),$$
$$P_3 = \rho^3(1-\rho),$$
$$\ldots\ldots$$

In general, we have that

$$P_n = \rho^n(1-\rho).$$

Expected number of items

Let L denote the expected number of items in the system, i.e.

$$L = \sum_{n=0}^{\infty} n P_n.$$

For the $M/M/1$ queue, we see that:

$$\begin{aligned} L &= \sum_{n=0}^{\infty} n\rho^n(1-\rho) \\ &= \rho(1-\rho)\sum_{n=1}^{\infty} n\rho^{n-1}. \end{aligned}$$

Expected number of items

Now we know that:

$$\frac{1}{1-\rho} = 1+\rho+\rho^2+\rho^3+\ldots,$$

and so

$$\begin{aligned}
\frac{1}{(1-\rho)^2} &= (1+\rho+\rho^2+\rho^3+\ldots)(1+\rho+\rho^2+\rho^3+\ldots) \\
&= 1+2\rho+3\rho^2+4\rho^3+\ldots \\
&= \sum_{n=1}^{\infty} n\rho^{n-1}.
\end{aligned}$$

Expected number of items

So we have that the expected number of items in the system is:

$$\begin{aligned} L &= \rho(1-\rho)\sum_{n=1}^{\infty} n\rho^{n-1} \\ &= \rho(1-\rho)\frac{1}{(1-\rho)^2} \\ &= \frac{\rho}{1-\rho} \\ &= \frac{\lambda/\mu}{1-\lambda/\mu} \\ &= \frac{\lambda}{\mu-\lambda}. \end{aligned}$$

Remember that $0 < \lambda < \mu$ here, so that $\mu - \lambda > 0$.

Little's law

We now state the following result:

Little's law. The average number of items in the system is equal to the arrival rate multiplied by the average response time.

We should stress that Little's law is true in general, not just for Markovian systems.

As stated above, by the "system" here, we mean the queue together with the items being served.

Turnaround time

We can now derive some further properties of the $M/M/1$ queue.

Let T denote the average turnaround time of an item in the system (i.e. the average time from the item's arrival in the system to the time when it has been served).

By Little's law, we have that $L = \lambda T$ and so

$$T = \frac{1}{\lambda} L = \frac{1}{\lambda} \left(\frac{\lambda}{\mu - \lambda} \right) = \frac{1}{\mu - \lambda}.$$

Other measures

By contrast, let W denote the average time an item waits in the queue (i.e. the average time from the item's arrival to the time when it starts being served); then:

$$W = T - \frac{1}{\mu} = \frac{1}{\mu - \lambda} - \frac{1}{\mu} = \frac{\mu - (\mu - \lambda)}{\mu(\mu - \lambda)} = \frac{\lambda}{\mu(\mu - \lambda)}.$$

If we let L_q denote the average number of items in the queue, then:

$$L_q = \lambda W = \lambda \frac{\lambda}{\mu(\mu - \lambda)} = \frac{\lambda^2}{\mu(\mu - \lambda)}.$$

CO4211/CO7211

Discrete Event Systems

Lecture 21:
Queues with a Maximum Capacity

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Notation

We are using the *Kendal notation* to represent the types of queueing system. In our case, we will have:

$$\mathrm{M}/\mathrm{M}/\mathrm{m}/\mathrm{N}$$

Here "M" represents "Markovian"; when we consider the distribution of times between successive arrivals and the distribution of service times, the probability of what happens next does not depend on what happened previously.

The number m represents the number of servers available and the number N (if present) reflects the capacity of the queue.

In the last lecture we considered the $\mathrm{M}/\mathrm{M}/1$ queue.

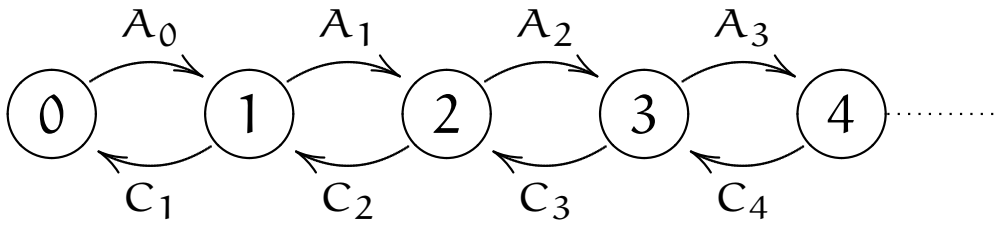
Arrival and service rates

Two critical values for our systems were:

- the average arrival rate λ;
- the average service rate μ;

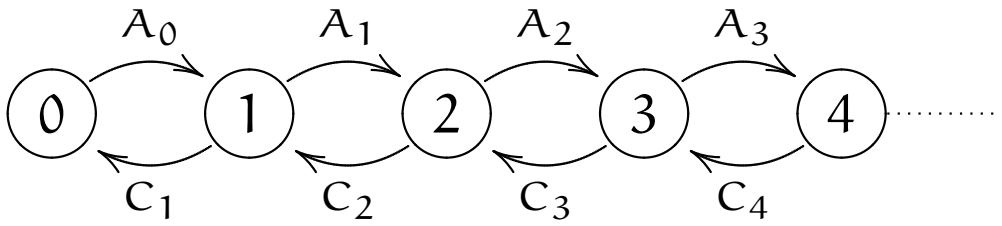
We then defined the utilization rate to be $\rho = \frac{\lambda}{\mu}$.

The state space was $\{0, 1, 2, 3, \ldots\}$, where state n denotes that there are n items in the system.



- A_n: a new item arrives and joins the system.
- C_n: an item is served in the system.

Events in $M/M/1$ queues



Let P_n denote the probability that there are n items in the system.

Event A_n occurs with rate λP_n for $n \geqslant 0$.

Event C_n occurs with rate μP_n for $n \geqslant 1$.

In a "steady-state", the event A_n occurs as often as event C_{n+1} for each n.

This gives that:

$$P_{n+1} = \frac{\lambda}{\mu} P_n = \rho P_n \text{ for } n \geqslant 0.$$

We solved this to get:

$$P_n = \rho^n (1 - \rho) \text{ for } n \geqslant 0.$$

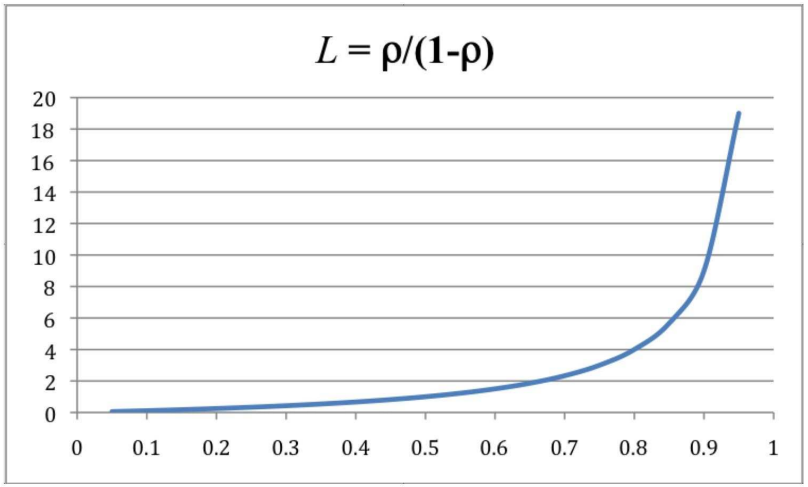
Expected number of items

Let L denote the expected number of items $\sum_{n=0}^{\infty} nP_n$ in the system.

For the $M/M/1$ queue, we saw that

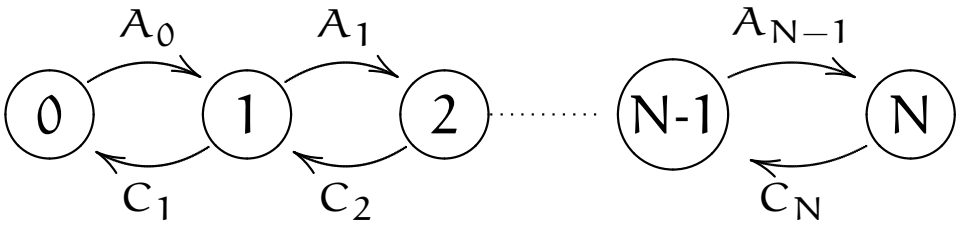
$$L = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda}.$$

We indicate how L grows with respect to ρ below.



M/M/1/N queues

Suppose now we add the condition that our system has a maximum capacity of N. Our state space now looks like

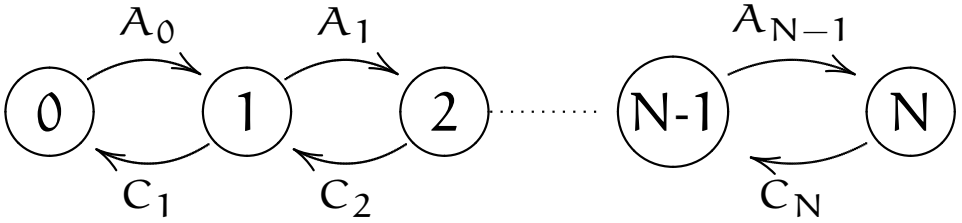


This time our state space is $\{0, 1, 2, \ldots N\}$ as we can have a maximum of N items in the system.

As before the events are:

- A_n: a new item arrives and joins the system.
- C_n: an item is served.

Steady state in $M/M/1/N$ queues



Note that a new item cannot arrive once we have N items in the system.

As with $M/M/1$ queues, the event A_n occurs with rate λP_n and the event C_n occurs with rate μP_n. In a steady-state, the event A_n occurs as often as event C_{n+1} for each n. So we again have

$$\lambda P_n = \mu P_{n+1} \text{ for } n \geqslant 0.$$

This gives (as before) that

$$P_{n+1} = \frac{\lambda}{\mu} P_n = \rho P_n \text{ for } n \geqslant 0.$$

Steady state in $M/M/1/N$ queues

So we have:

$$P_1 = \rho P_0,\ P_2 = \rho P_1 = \rho^2 P_0,\ P_3 = \rho P_2 = \rho^3 P_0,\ \ldots, P_N = \rho P_{N-1} = \rho^N P_0.$$

Now we know that $\sum_{n=0}^{N} P_n = 1$, and so

$$\sum_{n=0}^{N} \rho^n P_0 = 1.$$

Since

$$\sum_{n=0}^{N} \rho^n = \frac{1-\rho^{N+1}}{1-\rho},$$

we have that

$$P_0 = \frac{1-\rho}{1-\rho^{N+1}}.$$

Steady state in $M/M/1/N$ queues

We have that:

$$P_0 = \frac{1-\rho}{1-\rho^{N+1}},$$

and then that

$$P_n = \rho P_{n-1} = \ldots = \rho^n P_0 = \rho^n \frac{1-\rho}{1-\rho^{N+1}}$$

for $0 \leqslant n \leqslant N$.

Note that we do not need to have $0 < \rho < 1$ here; the formula above is valid for $\rho > 1$ as well.

Note that $\rho = 1$ is a special case and the analysis for that possibility has to be handled separately.

Expected number of items

As before, let L denote the expected number of items in the system, i.e.

$$L = \sum_{n=0}^{\infty} nP_n.$$

For the $M/M/1/N$ queue, we see that:

$$L = \sum_{n=0}^{N} n\rho^n \frac{1-\rho}{1-\rho^{N+1}} = \frac{1-\rho}{1-\rho^{N+1}} \sum_{n=0}^{N} n\rho^n.$$

Let X denote the quantity $\sum_{n=0}^{N} n\rho^n$.

Expected number of items

$$
\begin{aligned}
X - \rho X &= \sum_{n=0}^{N} n\rho^n - \sum_{n=0}^{N} n\rho^{n+1} \\
&= \sum_{n=1}^{N} n\rho^n - \sum_{n=1}^{N+1} (n-1)\rho^n \\
&= \left(\sum_{n=1}^{N} \rho^n \right) - N\rho^{N+1} \\
&= \rho(1 + \rho + \rho^2 + \ldots \rho^{N-1}) - N\rho^{N+1} \\
&= \rho \left(\frac{1-\rho^N}{1-\rho} \right) - N\rho^{N+1}
\end{aligned}
$$

So

$$
X = \rho \left(\frac{1-\rho^N}{(1-\rho)^2} \right) - \frac{N\rho^{N+1}}{1-\rho}.
$$

Expected number of items

For the $M/M/1/N$ queue, we have that:

$$L = \frac{1-\rho}{1-\rho^{N+1}} X$$

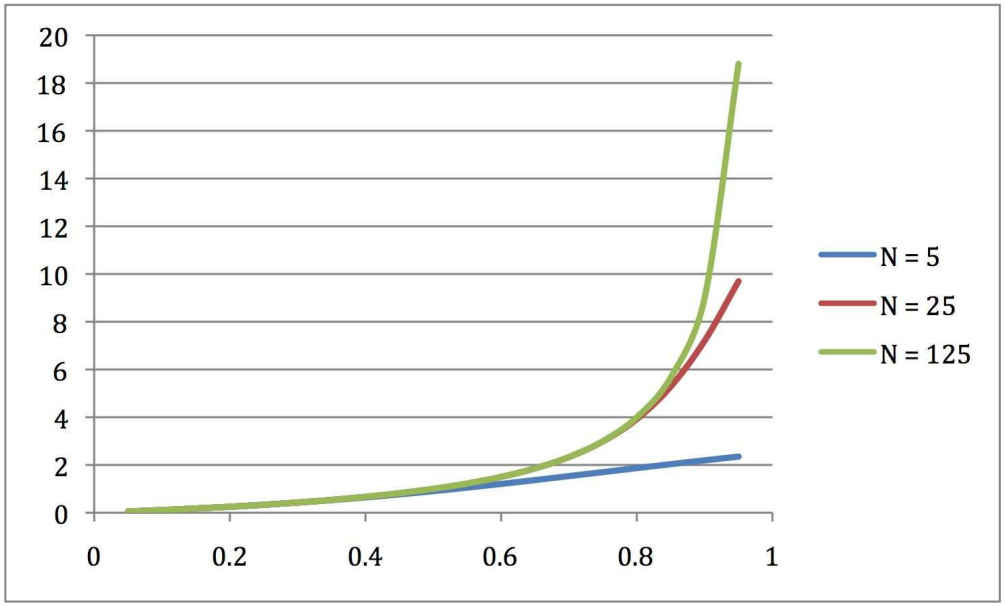
$$X = \sum_{n=0}^{N} n\rho^n = \rho\left(\frac{1-\rho^N}{(1-\rho)^2}\right) - \frac{N\rho^{N+1}}{1-\rho}$$

This gives that

$$\begin{aligned} L &= \frac{1-\rho}{1-\rho^{N+1}} X \\ &= \frac{1-\rho}{1-\rho^{N+1}} \left[\rho\left(\frac{1-\rho^N}{(1-\rho)^2}\right) - \frac{N\rho^{N+1}}{1-\rho}\right] \\ &= \frac{\rho}{1-\rho^{N+1}} \left[\frac{1-\rho^N}{1-\rho} - N\rho^N\right] \end{aligned}$$

Expected number of items

We show below how L grows with respect to ρ for $\mathrm{N} = 5$, $\mathrm{N} = 25$ and $\mathrm{N} = 125$.



Of course, this time, L is bounded above by N.

Blocking probability

Question. If we have an $M/M/1/N$ queue, what is the probability that an item arriving in the system is rejected?

Answer. The probability that an item is rejected is P_N, the probability that there are N items in the system. As above, this is given by:

$$P_N = \rho^N \frac{1-\rho}{1-\rho^{N+1}}.$$

Example

Suppose we have a machine with space for incoming parts. The parts arrive at a rate of one part per minute. We want to ensure that no more than 10% of the parts are rejected. What constraints are there on the system?

Given that

$$P_N = \rho^N \frac{1-\rho}{1-\rho^{N+1}},$$

we must choose ρ and N such that

$$\rho^N \frac{1-\rho}{1-\rho^{N+1}} < 0.1$$

Blocking probability

In our example, we need to choose ρ and N such that

$$P_N = \rho^N \frac{1-\rho}{1-\rho^{N+1}} < 0.1$$

Given that $\lambda = 1$ here, we have that $\rho = \lambda/\mu = 1/\mu$. So the parameters we can vary are μ and N.

Let us take the following scenario:

- Queuing space costs £800 per unit.
- We can buy a machine processing 0.5 parts per minute for £1000.
- We can buy a machine processing 1.2 parts per minute for £3000.
- We can buy a machine processing 2 parts per minute for £5000.

Which option do you choose?

Blocking probability

If $\mu = 0.5$ then $\rho = 2$ and our condition

$$P_N = \rho^N \frac{1-\rho}{1-\rho^{N+1}} < 0.1$$

gives that

$$\frac{2^N}{2^{N+1}-1} < 0.1$$

This gives $2^N < 0.1 \times 2^{N+1} - 0.1$, so that

$$1 < 0.2 - 0.1 \times 2^{-N},$$

which clearly cannot happen. So this option does not yield a solution at all.

Blocking probability

Let us consider the next case. Here $\mu = 1.2 = 6/5$ so that $\rho = 5/6$. Our condition becomes

$$P_N = \left(\frac{5}{6}\right)^N \frac{1 - \frac{5}{6}}{1 - \left(\frac{5}{6}\right)^{N+1}} < 0.1$$

which gives that

$$\frac{5^N}{6^{N+1} - 5^{N+1}} < 0.1$$

Testing this condition for successive values of N yields that $N = 6$ (for $N = 5$ the rejection rate is 10.07% and for $N = 6$ it is 7.74%).

The total cost here is £3000 for the machine and $6 \times$ £800 $=$ £4800 for the queuing space, i.e. a total of £7800.

Blocking probability

Let us turn to the last possibility where $\mu = 2$. Here $\rho = 1/2$ and our condition

$$P_N = \rho^N \frac{1-\rho}{1-\rho^{N+1}} < 0.1$$

gives that

$$\frac{1}{2^N}\left(\frac{\frac{1}{2}}{1-\frac{1}{2^{N+1}}}\right) < 0.1$$

and so $\frac{1}{2^{N+1}-1} < 0.1$. We find here that the smallest solution is $N = 3$ (for $N = 2$ the rejection rate is 14.29% and for $N = 3$ it is 6.67%).

Blocking probability

If we take $\mu = 2$ and $N = 3$ then total cost is £5000 for the machine and $3 \times$ £$800 =$ £2400 for the queuing space, i.e. a total of £7400. So this is the cheapest option.

Note that the possibility $\mu = 1.2$, $N = 5$ gave a rejection rate of 10.07% which is only just over 10% and so might be acceptable; if it were, then this would be the cheapest option (£3000 for the machine and $5 \times$ £$800 =$ £4000 for the queuing space, i.e. a total of £7000).

Little's law

Little's law. The average number of items in the system is equal to the arrival rate multiplied by the average response time.

For $M/M/1$ systems we used Little's law to deduce that, if T denotes the average turnaround time of an item in the system, then

$$T = \frac{1}{\lambda} L = \frac{1}{\mu - \lambda}.$$

For $M/M/1/N$ systems we have to be a little careful. The appropriate arrival rate to use here is $(1 - P_N)\lambda$, as opposed to λ, as items can only join the system if there are fewer than N items present. We can then calculate T for $M/M/1/N$ systems by

$$T = \frac{1}{(1 - P_N)\lambda} L.$$

CO4211/CO7211

Discrete Event Systems

Lecture 22:
Queues with Multiple Servers

Michael Hoffmann

University of Leicester

January 2018

Original slides: Rick Thomas and Nir Piterman

Notation

We are using the *Kendal notation* to represent the types of queueing system. In our case, we will have:

$$\mathrm{M}/\mathrm{M}/\mathrm{m}/\mathrm{N}$$

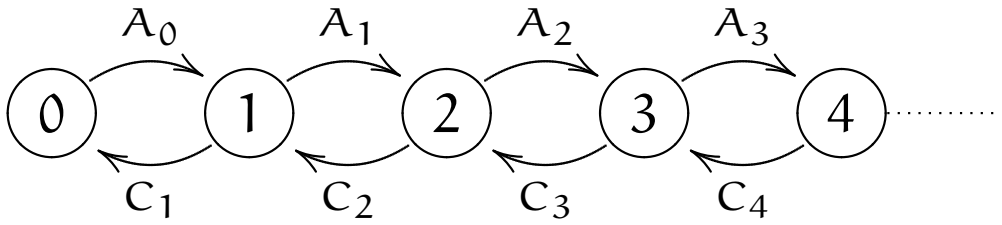
Here "M" represents "Markovian"; when we consider the distribution of times between successive arrivals and the distribution of service times, the probability of what happens next does not depend on what happened previously.

The number m represents the number of servers available and the number N (if present) reflects the capacity of the queue.

In the previous lectures we considered the $\mathrm{M}/\mathrm{M}/1$ and $\mathrm{M}/\mathrm{M}/1/\mathrm{N}$ queues. We now consider $\mathrm{M}/\mathrm{M}/\mathrm{m}$ queues. We will assume in what follows that $\mathrm{m} \geqslant 2$.

$M/M/m$ queues

As before, our state space is $\{0, 1, 2, 3, \ldots\}$, where state n denotes that there are n items in the system.



We again have the events:

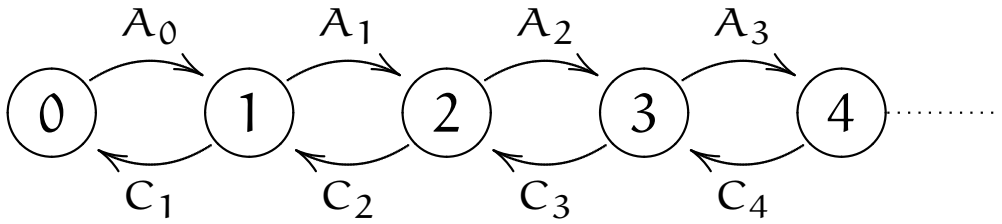
A_n: a new item arrives. C_n: an item is served.

We also have the values:

the average arrival rate λ; the average service rate μ.

Note that μ is the average service rate of each server.

$M/M/m$ queues



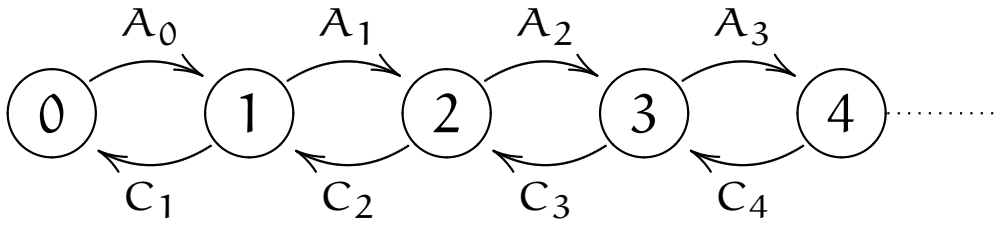
As before, let P_n denote the probability that there are n items in the system.

We again have that the event A_n occurs with rate λP_n for $n \geqslant 0$. However, the rate at which C_n occurs is a bit more complicated.

If $1 \leqslant n < m$ then the number of items in the system is less than the number of servers available; so the event C_n occurs with rate $n\mu P_n$ for $1 \leqslant n < m$.

If $n \geqslant m$ then the number of items in the system is at least the number of servers available; so the event C_n occurs with rate $m\mu P_n$ for $n \geqslant m$.

Steady state in $M/M/m$ queues



The event A_n occurs with rate λP_n for $n \geqslant 0$. The event C_n occurs with rate

$$\begin{cases} n\mu & \text{if } 1 \leqslant n < m; \\ m\mu & \text{if } n \geqslant m. \end{cases}$$

In a "steady-state", the event A_n occurs as often as event C_{n+1} for each n.

Steady state in $M/M/m$ queues

This gives that:

$$P_1 = \frac{\lambda}{\mu} P_0,$$
$$P_2 = \frac{\lambda}{2\mu} P_1 = \frac{\lambda}{\mu}\frac{\lambda}{2\mu} P_0,$$
$$P_3 = \frac{\lambda}{3\mu} P_2 = \frac{\lambda}{\mu}\frac{\lambda}{2\mu}\frac{\lambda}{3\mu} P_0,$$
$$\ldots\ldots$$
$$P_{m-1} = \frac{\lambda}{(m-1)\mu} P_{m-2} = \frac{\lambda}{\mu}\frac{\lambda}{2\mu}\cdots\frac{\lambda}{(m-1)\mu} P_0.$$

Thereafter (i.e. for $n \geqslant m$) we have that

$$P_n = \frac{\lambda}{m\mu} P_{n-1}.$$

Steady state in $M/M/m$ queues

Let σ denote $\lambda/m\mu$; we divide the arrival rate λ by $m\mu$ which is the maximum possible rate at which service can occur. Then we have:

$$P_1 = m\sigma P_0,$$
$$P_2 = \frac{(m\sigma)^2}{2!} P_0,$$
$$P_3 = \frac{(m\sigma)^3}{3!} P_0,$$
$$\ldots\ldots$$
$$P_{m-1} = \frac{(m\sigma)^{m-1}}{(m-1)!} P_0,$$

Steady state in $M/M/m$ queues

$$P_m = \sigma P_{m-1} = \sigma \frac{(m\sigma)^{m-1}}{(m-1)!} P_0 = \sigma^m \frac{m^m}{m!} P_0,$$
$$P_{m+1} = \sigma P_m = \sigma^{m+1} \frac{m^m}{m!} P_0,$$
$$P_{m+2} = \sigma P_{m+1} = \sigma^{m+2} \frac{m^m}{m!} P_0,$$
$$\ldots\ldots$$

In general we have that

$$P_n = \begin{cases} \frac{(m\sigma)^n}{n!} P_0 & \text{if } 1 \leqslant n \leqslant m-1; \\ \sigma^n \frac{m^m}{m!} P_0 & \text{if } n \geqslant m. \end{cases}$$

Steady state in $M/M/m$ queues

As always, we have that $\sum_{n=0}^{\infty} P_n = 1$. This gives that

$$\left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \sum_{n=m}^{\infty} \sigma^n \frac{m^m}{m!}\right] P_0 = 1.$$

We see that

$$\left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^m}{m!} \sum_{n=0}^{\infty} \sigma^n\right] P_0 = 1,$$

and so

$$\left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^m}{m!} \frac{1}{1-\sigma}\right] P_0 = 1,$$

and then

$$P_0 = \frac{1}{\left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^m}{m!} \frac{1}{1-\sigma}\right]}.$$

Note that we are assuming we have $0 < \sigma < 1$ here.

Example

As an example, let us consider the case $m = 2$. Here $\sigma = \lambda/2\mu$ and we have

$$P_1 = 2\sigma P_0,\ P_2 = \sigma P_1,\ P_3 = \sigma P_2,\ \ldots\ldots$$

and so $P_n = 2\sigma^n P_0$ for all $n \geqslant 1$. Then, given that

$$\sum_{n=0}^{\infty} P_n = 1,$$

we have that

$$P_0(1 + 2\sigma + 2\sigma^2 + 2\sigma^3 + \ldots) = 1,$$

and so

$$P_0\left[-1 + 2(1 + \sigma + \sigma^2 + \ldots)\right] = 1,$$

i.e.

$$P_0\left[-1 + 2\frac{1}{1-\sigma}\right] = P_0\left[\frac{1+\sigma}{1-\sigma}\right] = 1.$$

Example

We saw that, if $m = 2$, then we have

$$P_0 \left[\frac{1+\sigma}{1-\sigma} \right] = 1.$$

and then

$$P_0 = \frac{1-\sigma}{1+\sigma} = \frac{2\mu-\lambda}{2\mu+\lambda}.$$

Then:

$$P_n = 2\sigma^n P_0 = \frac{2\sigma^n(1-\sigma)}{1+\sigma}$$

for $n \geqslant 1$.

Busy servers in $M/M/m$ queues

Let B denote the expected number of servers that are busy in an $M/M/m$ queue. We have that

$$B = \sum_{n=0}^{m-1} nP_n + mP_{\geqslant m}$$

where $P_{\geqslant m}$ is the probability that we are in state n for some $n \geqslant m$. Then

$$\begin{aligned} P_{\geqslant m} = \sum_{n=m}^{\infty} P_n &= \sum_{n=m}^{\infty} \sigma^n \frac{m^m}{m!} P_0 \\ &= \frac{m^m}{m!} \sigma^m P_0 \sum_{n=0}^{\infty} \sigma^n \\ &= \frac{m^m}{m!} \frac{\sigma^m}{1-\sigma} P_0 \end{aligned}$$

Busy servers in $M/M/m$ queues

We have that

$$
\begin{aligned}
B &= \sum_{n=0}^{m-1} n P_n + m \frac{m^m}{m!} \frac{\sigma^m}{1-\sigma} P_0 \\
&= \sum_{n=0}^{m-1} n \frac{(m\sigma)^n}{n!} P_0 + m \frac{(m\sigma)^m}{m!} \frac{1}{1-\sigma} P_0 \\
&= 0 + m\sigma P_0 + \sum_{n=2}^{m-1} n \frac{(m\sigma)^n}{n!} P_0 + m \frac{(m\sigma)^m}{m!} \frac{1}{1-\sigma} P_0 \\
&= m\sigma \left[1 + \sum_{n=2}^{m-1} \frac{(m\sigma)^{n-1}}{(n-1)!} + \frac{(m\sigma)^{m-1}}{(m-1)!} \frac{1}{1-\sigma} \right] P_0 \\
&= m\sigma \left[1 + \sum_{n=1}^{m-2} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^{m-1}}{(m-1)!} \frac{1}{1-\sigma} \right] P_0 \\
&= m\sigma \left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} - \frac{(m\sigma)^{m-1}}{(m-1)!} + \frac{(m\sigma)^{m-1}}{(m-1)!} \frac{1}{1-\sigma} \right] P_0 \\
&= m\sigma \left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^{m-1}}{(m-1)!} \frac{1-(1-\sigma)}{1-\sigma} \right] P_0 \\
&= m\sigma \left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^{m-1}}{(m-1)!} \frac{\sigma}{1-\sigma} \right] P_0 \\
&= m\sigma \left[1 + \sum_{n=1}^{m-1} \frac{(m\sigma)^n}{n!} + \frac{(m\sigma)^m}{m!} \frac{1}{1-\sigma} \right] P_0 \\
&= m\sigma.
\end{aligned}
$$

Busy servers in $M/M/m$ queues

Remembering that, as $0 < \sigma = \lambda/m\mu < 1$, we have that

$$0 < \frac{\lambda}{\mu} < m.$$

So

$$0 < B = m\sigma = \frac{\lambda}{\mu} < m.$$

Expected length in $M/M/m$ queues

Let L denote the expected number $\sum_{n=0}^{\infty} nP_n$ of items in the system. For the $M/M/1$ queue, we saw that

$$L = \frac{\rho}{1-\rho} = \frac{\rho}{(1-\rho)^2} P_0.$$

For $M/M/m$ queues (with $m \geqslant 2$) the situation is rather more complicated and we get

$$L = m\sigma + \frac{(m\sigma)^m}{m!} \frac{\sigma}{(1-\sigma)^2} P_0.$$

We have omitted the steps needed to derive this formula.

Remembering that $0 < \sigma = \lambda/m\mu < 1$, we see that $L \to 0$ as $\sigma \to 0$ and $L \to \infty$ as $\sigma \to 1$.

Example

As an example, let us consider the case $m = 2$. Here $\sigma = \lambda/2\mu$ and we showed above that

$$P_n = 2\sigma^n P_0 = \frac{2\sigma^n(1-\sigma)}{1+\sigma}$$

for $n \geqslant 1$. If we use

$$L = \sum_{n=0}^{\infty} nP_n.$$

here, we get that

$$\begin{array}{rcll} L & = & 0 + \sum_{n=1}^{\infty} nP_n & = & \sum_{n=1}^{\infty} \frac{2n\sigma^n(1-\sigma)}{1+\sigma} \\ & = & \frac{2\sigma(1-\sigma)}{1+\sigma} \sum_{n=1}^{\infty} n\sigma^{n-1} & = & \frac{2\sigma(1-\sigma)}{1+\sigma} \frac{1}{(1-\sigma)^2} \\ & = & \frac{2\sigma}{(1-\sigma)(1+\sigma)} & = & \frac{4\lambda\mu}{(2\mu-\lambda)(2\mu+\lambda)}. \end{array}$$

Example

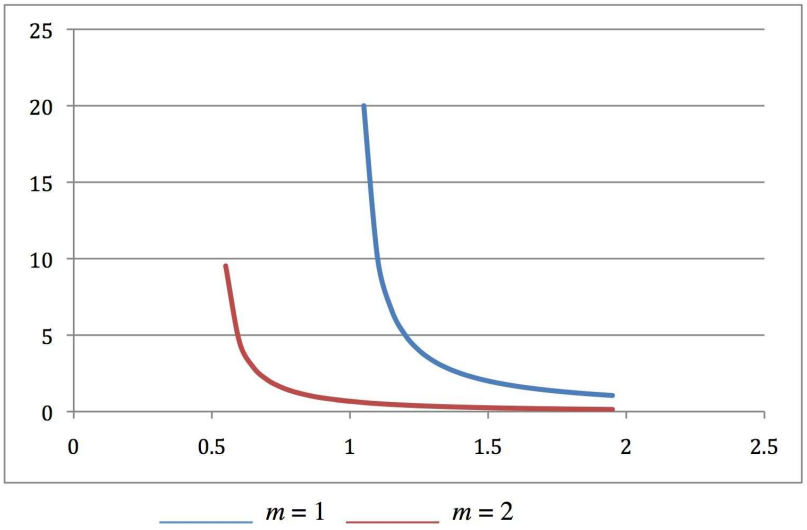
For $m = 1$ we had

$$L = \frac{\rho}{1-\rho} = \frac{\lambda}{\mu-\lambda},$$

and for $m = 2$ we had

$$L = \frac{4\lambda\mu}{(2\mu-\lambda)(2\mu+\lambda)}.$$

We show below a comparison of these two expected lengths (taking λ to be 1 in both cases):



Little's law

Little's law. The average number of items in the system is equal to the arrival rate multiplied by the average response time.

For $M/M/1$ systems we used Little's law to deduce that, if T denotes the average turnaround time of an item in the system, then

$$T = \frac{1}{\lambda} L = \frac{1}{\mu - \lambda}.$$

Little's law

For $M/M/m$ queues we get

$$\begin{aligned} T &= \frac{1}{\lambda} L \\ &= \frac{1}{\lambda}\left[m\sigma + \frac{(m\sigma)^m}{m!}\frac{\sigma}{(1-\sigma)^2}P_0\right] \\ &= \frac{1}{\mu} + \frac{1}{\mu}\frac{(m\sigma)^m}{m!}\frac{1}{m(1-\sigma)^2}P_0. \end{aligned}$$

CO4211/CO7211

Discrete Event Systems

Lecture 23:
Markov Decision Processes

Michael Hoffmann

University of Leicester

January 2018

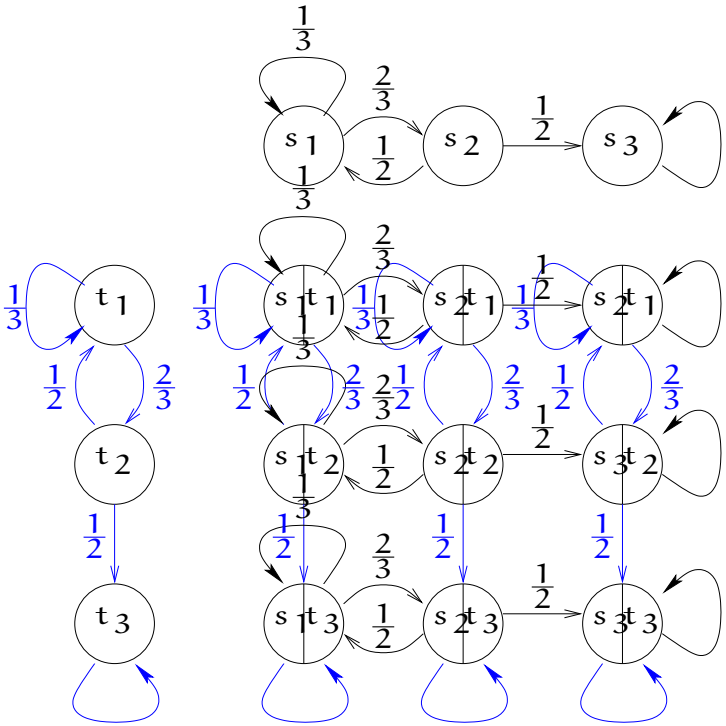
Motivation

- Include within Markov chains the option of choice / input.
- Combines nondeterminism and probability.
- Used in optimization problems, dynamic programming, and artificial intelligence.
- Arizes naturally with asynchronous composition.

Asynchronous Composition of Markov Chains

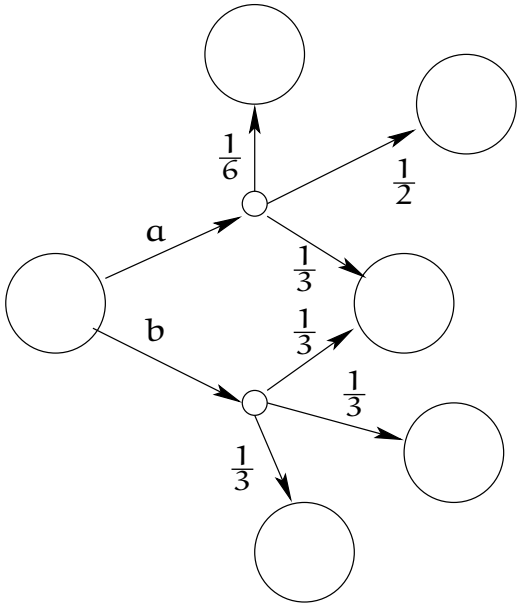
Running two Markov chains in parallel.

- A schedular chooses which process to advance.
  - Blue transitions – scheduler chooses system on left.
  - Black transitions – scheduler chooses system on the top.
- Questions about worst/best possible schedule can be asked.

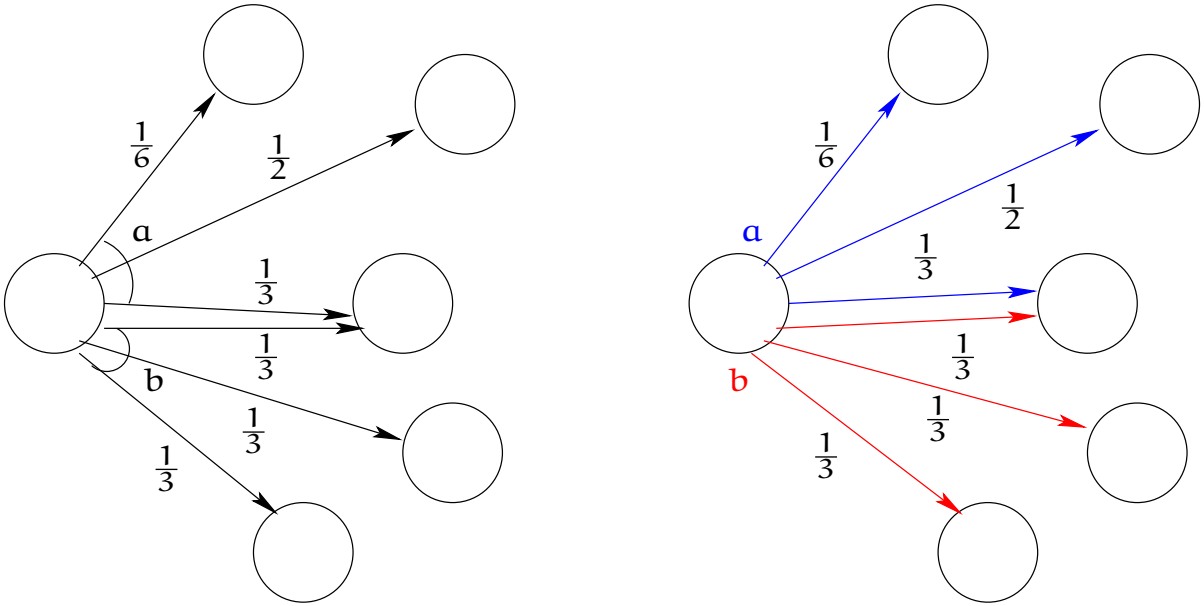


Intuition

- Reintroduce the alphabet.
- Each state and alphabet letter correspond to a probabilistic transition.
- The choice of the alphabet letter dictates the probability of next states.



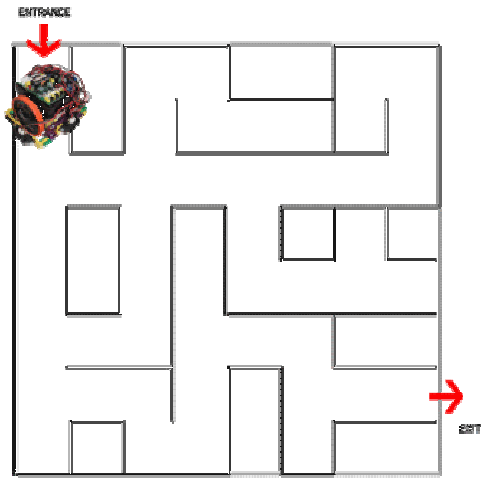
Other Possible Representations



Example

A robot navigates a maze.

- It needs to get to the exit.
- It may need to avoid a certain location.
- It can move left, right, front, and back.
- Actions may have stochastic effects.



We want to come up with a sequence of actions that will lead it to the exit.

Does this correspond to finding a word over alphabet $\{\mathrm{l}, \mathrm{r}, \mathrm{f}, \mathrm{b}\}$ such that the probability of reaching the exit is maximal?

Markov Decision Process

Definition. A *Markov Decision Process* is a triplet (Q, Σ, p) where:

- Q is a set of states,
- Σ is a finite alphabet, and
- $p : Q \times \Sigma \times Q \to [0, 1]$ is a *partial* transition probability function where for every $q \in Q$ and $\sigma \in \Sigma$ such that for some q' we have $p(q, \sigma, q')$ is defined:

$$\sum_{q' \in Q} p(q, \sigma, q') = 1$$

  For every state there is a letter σ such that $p(q, \sigma, q')$ is defined.

If for some $\sigma \in \Sigma$ we have $p(q, \sigma, q')$ is defined then σ is *possible* from q.

Markov Decision Process (cont.)

Idea:

- $p(q, \sigma, q')$ is the *probability* that after choice σ the system transitions from q to q',
- for some states some actions are not defined,
- if an action is defined then the probabilities sum up to one – the system *has* to make a step, and
- some action is always defined!

Notice: A Markov chain is an MDP with alphabet of size 1!

What can we do with this?

- The combination of nondeterminism with probability creates a problem.
- We can no longer analyze such systems.
- We remove nondeterminism by considering policies.
- The result is a Markov chain, which can be analyzed.
- Analysis of MDPs corresponds to finding best/worst policies to achieve a certain goal.

Policy / Strategy / Word

- A policy associates with a state and a time a distribution over the alphabet.
- The combination of the MDP with a policy creates a (potentially infinite) Markov chain.
- The Markov chain can be analyzed.
- In most cases, deterministic choices that depend only on the state are sufficient. These are called *stationary*.

What about a word? What kind of a policy does a word defined? What does it depend on?

Policy Definition

Definition. For an MDP (Q, Σ, p) a *policy* is a function

$$f : Q^+ \to D(\Sigma), \text{ where:}$$

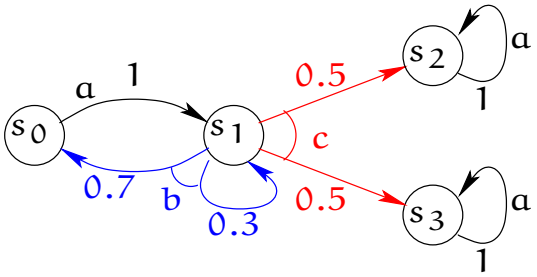
- $D(\Sigma) = \{d : \Sigma \to [0,1] \mid \sum_{\sigma \in \Sigma} d(\sigma) = 1\}$ is the set of distributions over Σ.
- For every $\nu \cdot q$ and $\sigma \in \Sigma$ we have $f(\nu \cdot q)(\sigma) > 0$ implies σ is possible from q.

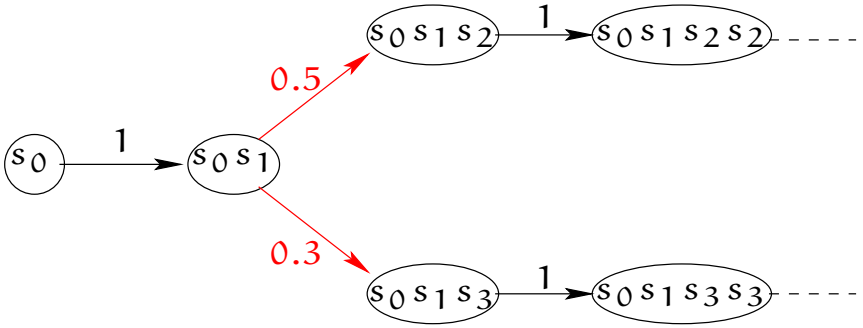
A policy is *deterministic* if for every $\nu \cdot q$ there is a letter σ such that $f(\nu \cdot q)(\sigma) = 1$.

A policy is *memoryless* if for every $\nu \cdot q$ and $\nu' \cdot q$ we have $f(\nu \cdot q) = f(\nu' \cdot q)$.

A deterministic and memoryless policy is called *stationary*.

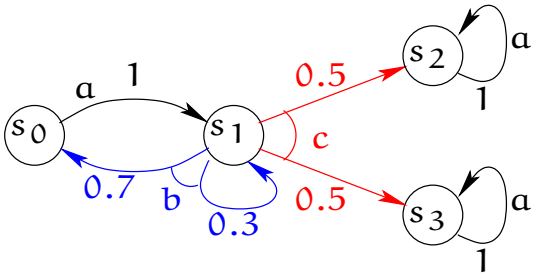
Policy Example

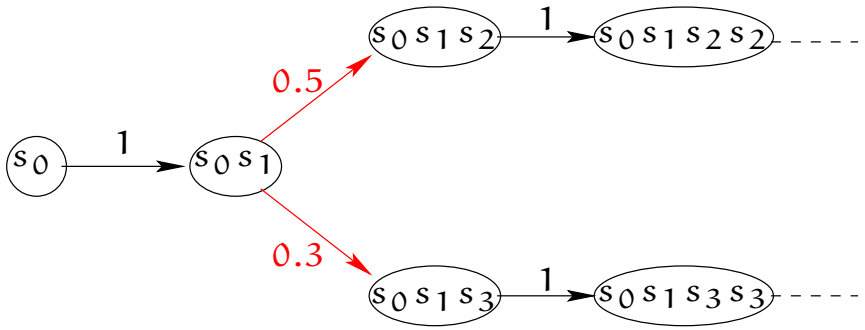


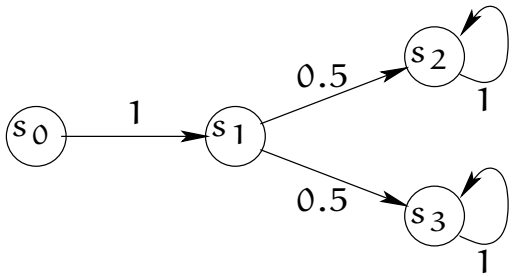


Pick c.

Policy Example

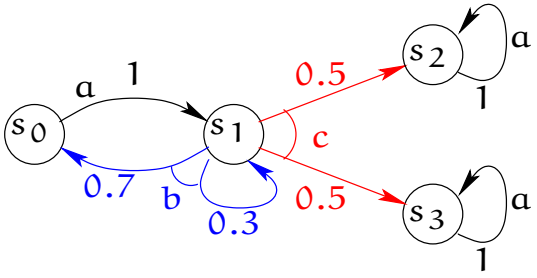


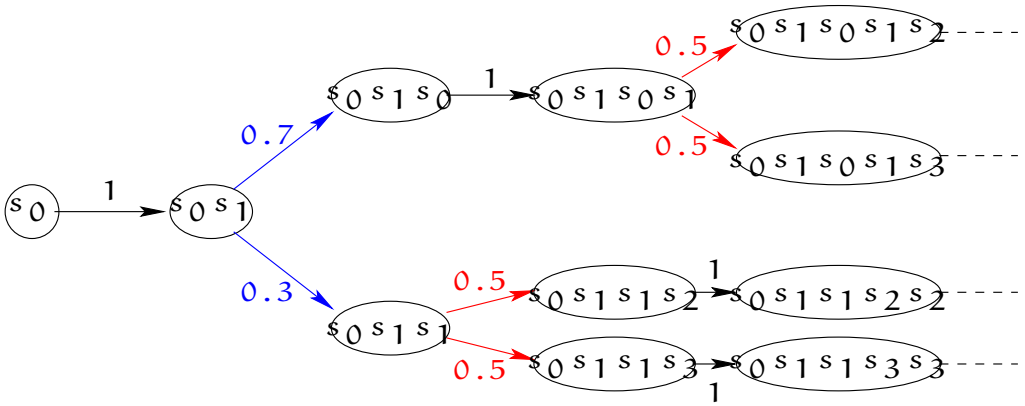




Pick c.

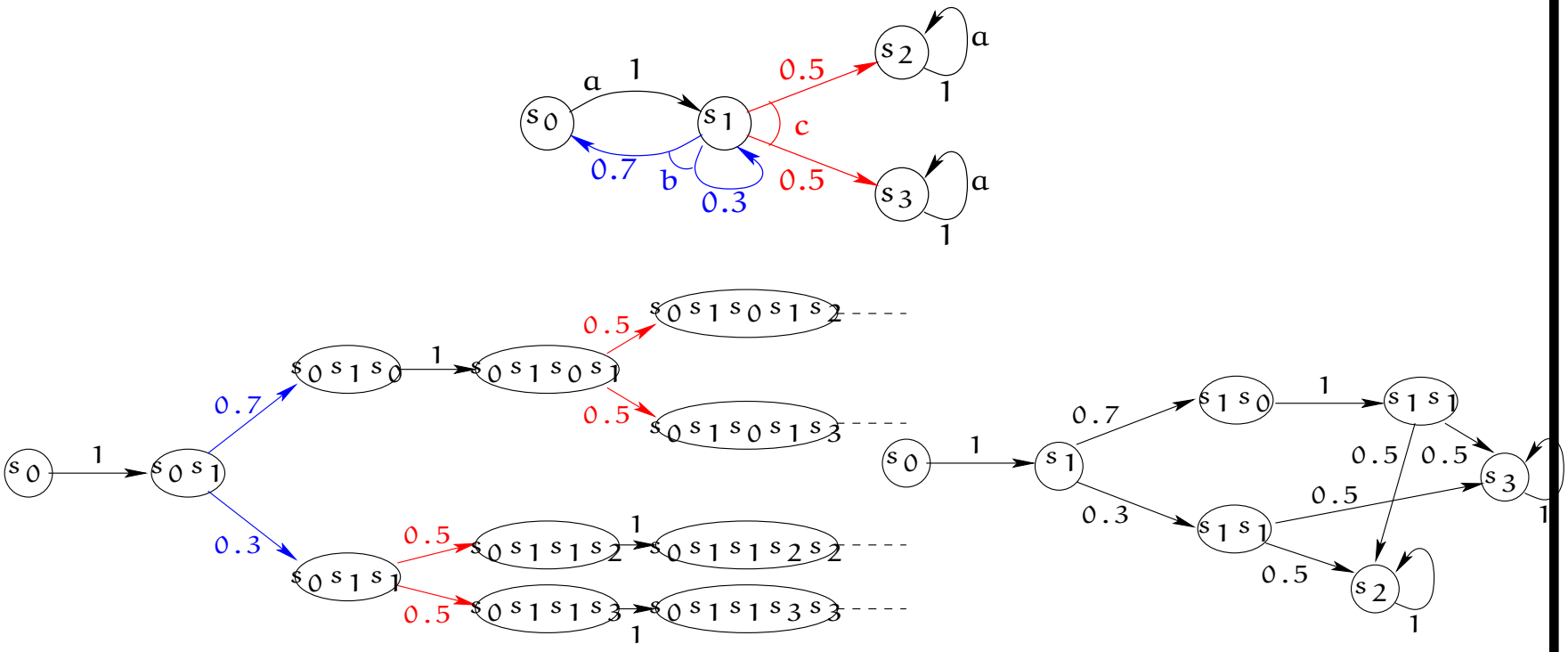
Policy Example





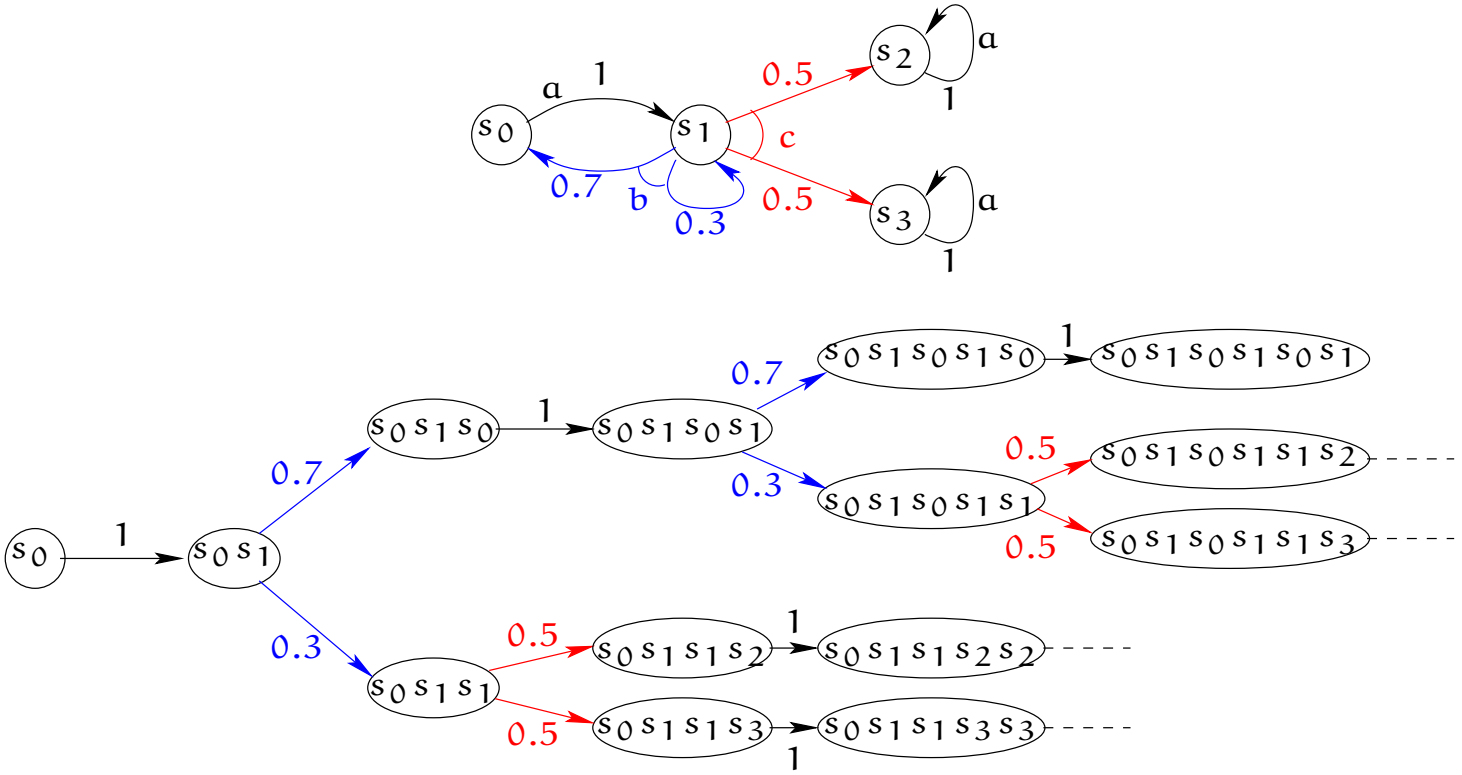
Pick b then c.

Policy Example



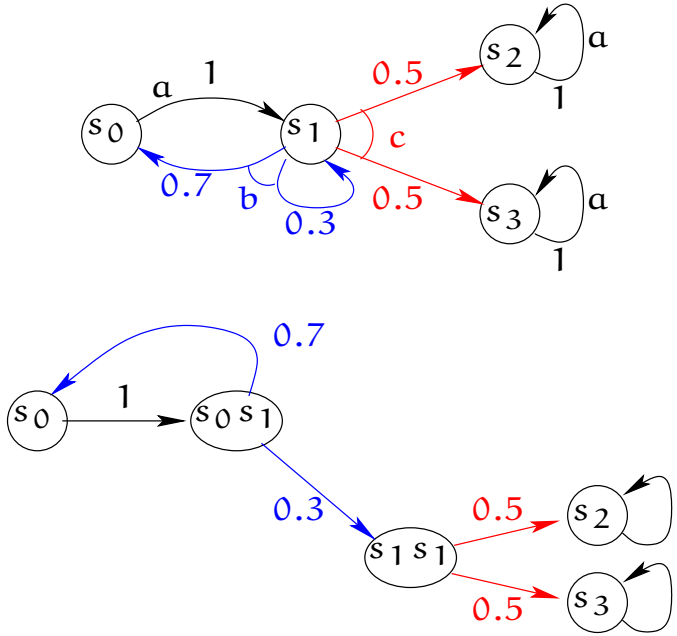
Pick b then c.

Policy Example



Pick b after arrival to s_1 from s_0 and c otherwise.

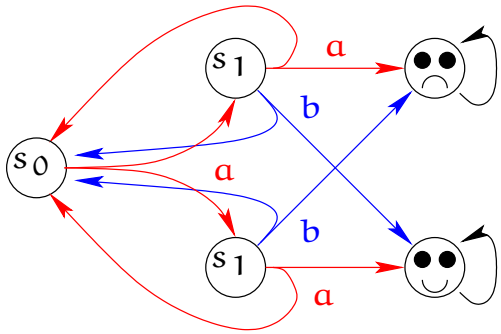
Policy Example



Pick b after arrival to s_1 from s_0 and c otherwise.

Summarized to a finite Markov chain taking into account only the memory required by the policy.

Word Example



- There is a simple policy that ensures reaching ☺ and avoiding ☹.
  - Choose b from s_1.
  - Choose a from s_2.
- Every sequence of choices aa or ab leads to ☹ and ☺ with equal probability.

Policy Gives Rise to Markov Chain

Given an MDP (Q, Σ, p) and a policy $f : Q^+ \to D(\Sigma)$ their combination is the Markov chain $M = (Q^+, p')$, where:

$$p'(\nu \cdot q, \nu \cdot q \cdot q') = \sum_{\sigma \in \Sigma} f(\nu \cdot q)(\sigma) \times p(q, \sigma, q')$$

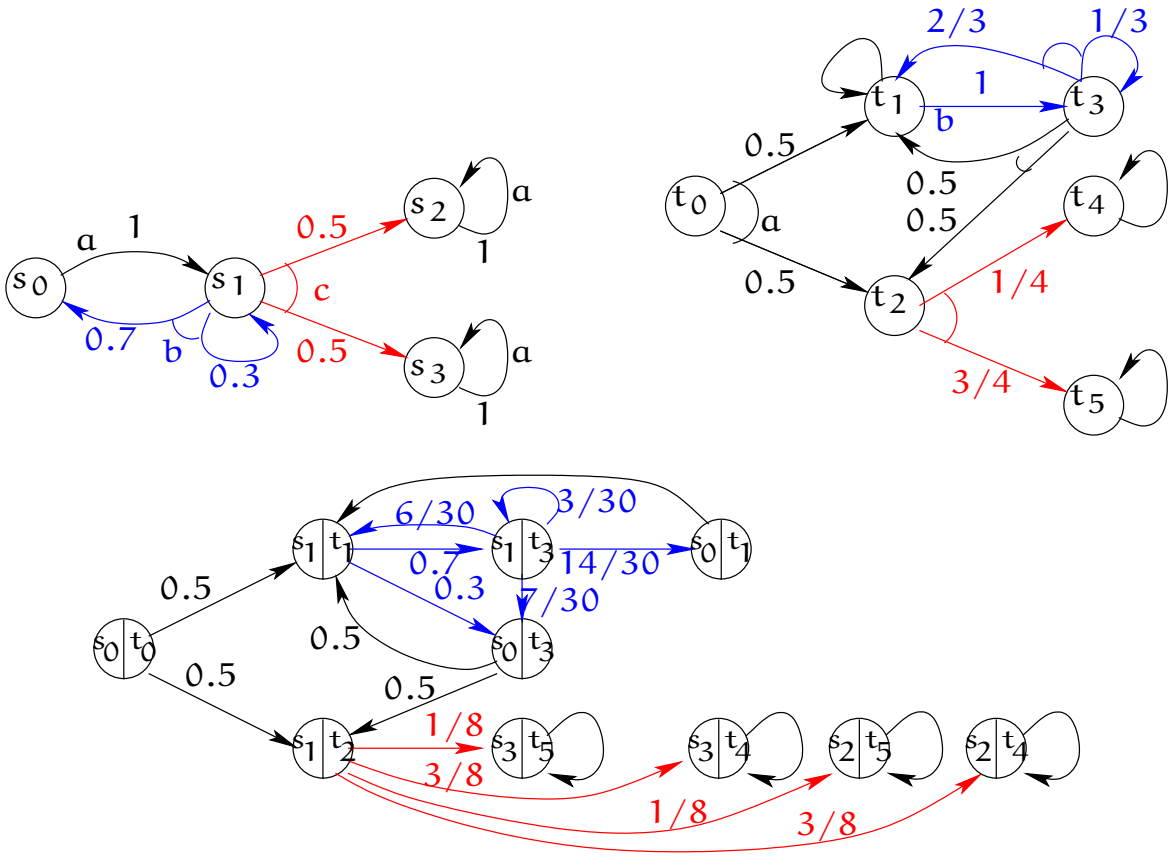
Is this a Markov chain?

$$\sum_{q' \in Q} p'(\nu \cdot q, \nu \cdot q \cdot q') = \sum_{q' \in Q} \sum_{\sigma \in \Sigma} f(\nu \cdot q)(\sigma) \times p(q, \sigma, q') =$$
$$= \sum_{\sigma \in \Sigma} f(\nu \cdot q)(\sigma) \sum_{q' \in Q} p(q, \sigma, q') = \sum_{\sigma \in \Sigma} f(\nu \cdot q)(\sigma) = 1$$

Synchronous Independent Composition of MDPs

- Generalizes synchronous composition of automata.
- MDPs not necessarily agree on alphabet.
  - Joint letters – MDPs move together. Probabilities are product of probabilities.
  - Other letters – MDPs move independently.
- Defines synchronous composition of Markov chains.

Synchronous Independent Composition of MDPs



Running MDPs in Parallel

Given two MDPs $M_1 = (Q_1, \Sigma_1, p_1)$ and $M_2 = (Q_2, \Sigma_2, p_2)$, we want to run M_1 and M_2 "in parallel".

- the state set Q is $Q_1 \times Q_2$.

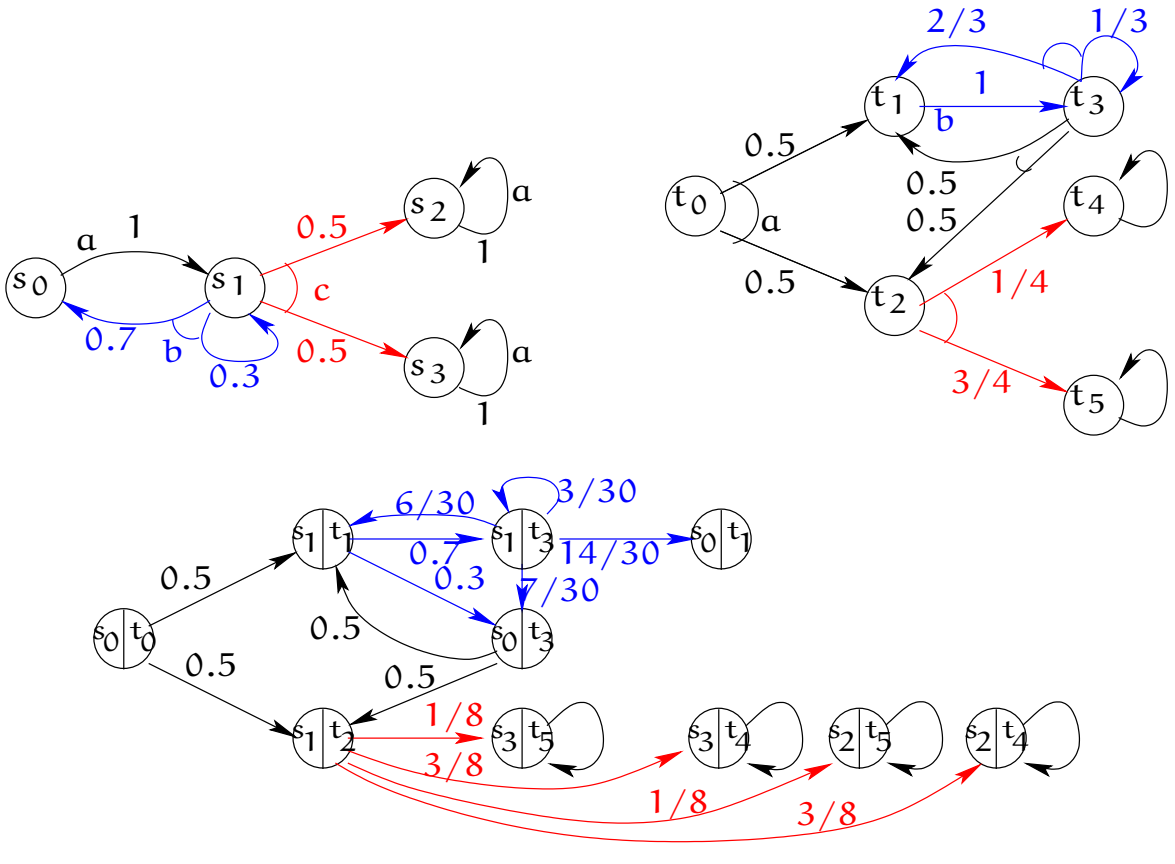
Forcing synchronous transitions:

- if $a \in \Sigma_1 \cap \Sigma_2$, a is possible from s_1, and a is possible from s_2 then $p((s_1, s_2), a, (t_1, t_2)) = p_1(s_1, a, t_1) \cdot p_2(s_2, a, t_2)$.

Implementing independent transitions:

- if $a \in \Sigma_1 - \Sigma_2$ and a possible from s_1 then $p((s_1, s_2), a, (t_1, s_2)) = p_1(s_1, a, t_1)$;
- if $a \in \Sigma' - \Sigma$ and a possible from s_2 then $p((s_1, s_2), a, (s_1, t_2) = p_2(s_2, a, t_2)$.

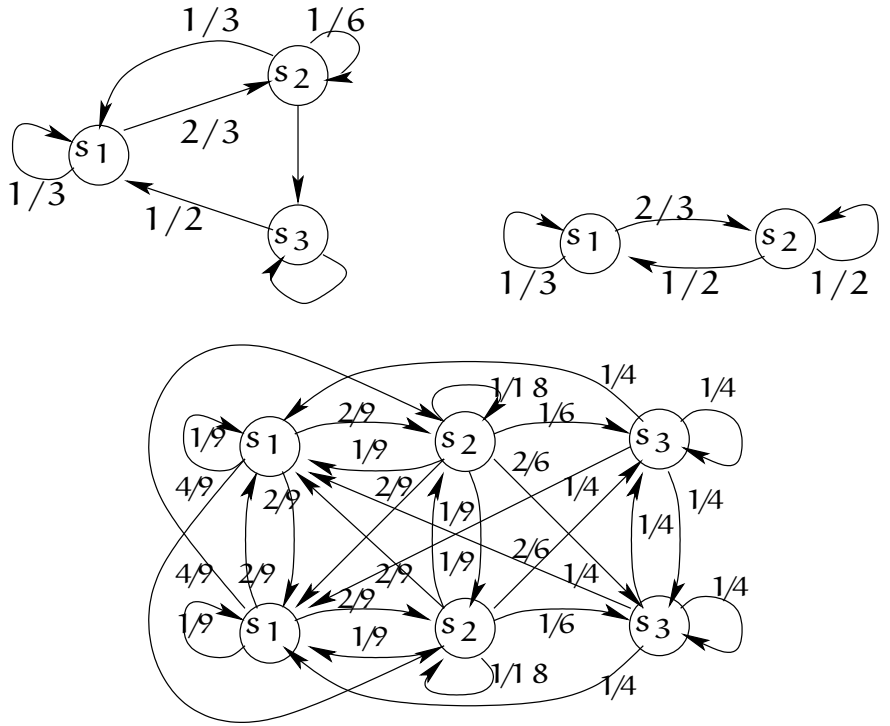
Catch - Is it MDP?



When considering Markov chains

- The single letter alphabet is joined.
- The two synchronize on every action.
- This is like taking the composition of two independent events.

Independent Product of Markov Chains



Almost Sure Reachability

- Have a certain set of states that needs to be reached.
- There are several options:
  - Sure – a strategy such that all paths reach the set.
  - Almost sure – a strategy such that reach with probilibty 1.
  - Value – a strategy such that the value to reach is $p \in (0,1)$.
- Sure analysis is similar to non-probabilistic models.
- We are going to study how to analyze almost sure reachability.
- What is the opposite of almost-sure reachability?

Setting up the Problem

Definition. Given an MDP (Q, Σ, p) and a target set of states $R \subseteq Q$ we say that:

- R is *sure* reachable from q if there is a policy such that *all* paths from q reach R.
- R is *almost-sure* reachable from q if there is a policy such that the *probability to reach* R from q *is* 1.
- R is reachable *with probability* ν if there is a policy such that the *probability to reach* R from q *is* ν.

Compute the set of states Q' such that R is sure / almost-sure / positively reachable from Q'.

In the latter case, compute the probabilities.

Solution Strategy

- Ensure that probability cannot keep us away from R:
  - In order to reach there has to be a path.
  - No path – no reachability.
  - If accidentally get there, that's the end of it.
  - Otherwise, we'll be fine!
- Ingredients:
  - Rechability analysis.
  - Attraction analysis.

Identify the impossible

- Think about (Q, Σ, p) and R as a simple graph.
- Compute the set of states that have a path to R.
- Denote this set as $\text{Reach}(R)$.
- Then, $A = Q - \text{Reach}(R)$ is the set of states from which R is not reachable.
- For every policy f, the probability to reach R from A with policy f is ???

If it is possible it will happen

- Recall: why are transient states transient?
- A repeated chance to do something, even with low probability, cannot be avoided.
- So if there is some chance for a bad thing to happen, it will!
- What about the policy?
- If there is a chance to avoid the bad thing, we'll take it!

Probabilistic attractor

Given an MDP (Q, Σ, p) and a set $S \subseteq Q$, attract to S:

- $\text{Attr}_0(S) = S$
- $\text{Attr}_{i+1}(S) =$

$$\left\{ q \in Q \;\middle|\; \begin{array}{r} \text{For every } \sigma \in \Sigma, \text{ either} \\ \text{(i) } \sigma \text{ is impossible from } q, \text{ or} \\ \text{(ii) } \exists q' \in \text{Attr}_i(S) \text{ s.t. } p(q, \sigma, q') > 0 \end{array} \right\}$$

- $\text{Attr}(S) = \bigcup_{i \geq 0} \text{Attr}_i(S)$

How to compute?

- This is a fixpoint computation.
- Add states until no more are possible.
- When computation stabilized, these are all the states.

Compute Attr(S)

1. new$:=S$
2. old$:=S$
3. while (old$!=$new) {
4. old$:=$new
5. new$:=$Attr^1(new)
6. }

$$\text{Attr}^1(T) =$$

$$\left\{ q \in Q \;\middle|\; \begin{array}{l} \forall \sigma \in \Sigma, \text{ either} \\ \text{(i) } \forall q' \,.\, p(q,\sigma,q') \text{ is Undefined,} \\ \text{or (ii) } \exists q' \in T \text{ s.t. } p(q,\sigma,q') > 0 \end{array} \right\}$$

Final Algorithm

- Start: R.
- Compute: Reach(R).
- Complement: $\mathrm{A} = \mathrm{Q} - \mathrm{Reach}(\mathrm{R})$.
- Attract: Attr(A).
- Complement: $\mathrm{Q} - \mathrm{Attr}(\mathrm{A})$.
- All together $\mathrm{ASure}(\mathrm{R}) := \mathrm{Q} - \mathrm{Attr}(\mathrm{Q} - \mathrm{Reach}(\mathrm{R}))$.

Theorem. From every state $\mathrm{q} \in \mathrm{Q}$ there is a policy f such that the probability to reach R from q is 1.

Summary

- MDPs combie Markov chains with nondeterministic choice.
- Analysis corresponds to evaluation of policies.
- We have seen policy to achieve sure reachability.
- Common analysis – computation of max/min value achievable given an associated cost function with states / transitions.
- Questions regarding words are a lot more complicated.