Introduction to Model-Checking

Theory and Practice

Beihang International Summer School 2019

http://homepages.laas.fr/dalzilio/courses/mccourse/

Talking about computer science

Computer Science is the study of *computers* and *computational systems* this includes their theory, design, and application.

CS principal areas of study include networking, security, database systems, graphics, numerical analysis, software engineering, …

This course is about a part of Theoretical Computer Science called *formal verification*

Talking about computer science

Computer Science is the study of *computers* and *computational systems* this includes their theory, design, and application.

> *" Computer science is not about machines, in the same way that astronomy is not about telescopes."*

Talking about computer science

Computer Science is the study of *computers* and *computational systems* this includes their theory, design, and application.

Computer scientists design and analyze *algorithms* to solve problems and study their *complexity*.

Talking about computer science

Computer Science is the study of *computers* and *computational systems* this includes their theory, design, and application.

Computer scientists design and analyze *algorithms* to solve problems and study their *complexity*.

Informatique: CS is about *information* and how to store and process it

CS is concerned with information in much the same sense that physics is concerned with energy

Talking about computer science

CS principal areas of study include networking, security, database systems, graphics, numerical analysis, software engineering, …

This course is about *formal verification*, that is the domain of Theoretical Computer Science interested by questions such as:

- How do we define the (expected) behavior of a program, protocol, system, … ?
- How can we say (and trust) that a system is correct ?
- How do we prove it ? (proof ≠ testing)

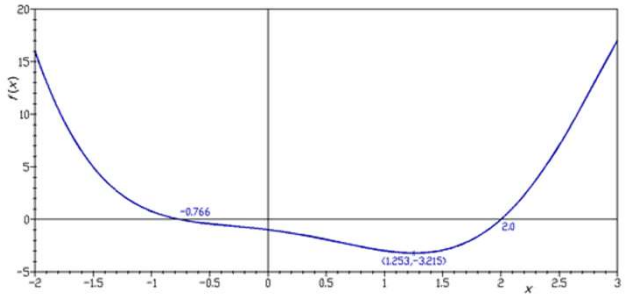
Talking about computer science

In this course, we show a small part of the underlying mathematics of Computer Science

How to model systems and describe their behavior (how they evolve)	languages and semantics	
How to reason and express properties on systems	logics	“the systematic study of the form of valid inference”
How to implement these ideas	algorithms	

The goal is explainable trust: how can we gain trust on a system

Talking about computer science



We can draw a parallel with mathematics (analysis)

mathematics has …	computer science has …
numbers, function (graphs) …	graphs and automata
equations, (stoch.) processes	formal languages
questions (how many zeroes?)	questions (does it terminate?)
theorems	… and tools
Architects use math to know if their bridge won't collapse	Architects use CS to know if their software won't collapse

Some examples: IEEE Futurebus

- In 1992 Clarke et al used SMV to verify the cache coherence protocol in the IEEE Futurebus+ Standard
- They modeled the protocol and attempted to show that it satisfied a formal specification of cache coherence.
- They found a number of previously undetected errors in the design of the protocol.
- Although development started in 1988, all previous attempts to validate Futurebus+ were based on informal techniques

see E. Clarke, J. Wing, et al. Formal methods: State of the art and future directions. ACM Computing Surveys, 28(4), 1996

Some examples: HDLC

- Design of a High-level Data Link Controller (HDLC) by AT&T.
- In 1996 researchers at Bell Labs offered to check some properties of the design. The design was almost finished, so no errors were expected.
- Within five hours, six properties were specified and five were verified, using the FormalCheck verifier.
- The sixth property failed, uncovering a bug that would have reduced throughput or caused lost transmissions.
- The error was corrected in a few minutes and formally verified.

Some Examples: Little Book of Semaphores

- A particularly difficult synchronization problem, known as the “room party problem”, has been defined by Downey in his Little Book of Semaphores.
- A student of an introductory course on operating systems, where model-checkers where used, found a bug. A discussion with the book’s author resulted in yet another proposal.
- Alas, also this model does not satisfy the required property

The need for formal methods

HACKERNOON — Follow — Sign in — Get started

HOME | DEV | REALTIME APIS, GLOBAL INFRASTRUCTURE

SECBIT Follow

Research on smart contract security, smart contract formal verification, crypto-protocols, compilation, contract analysis, game theory and crypto-economics.

Jul 16 · 11 min read

Improve Smart Contract Security by Formal Proofs

Why Formal Proofs for Smart Contracts

Ethereum improved smart contracts [...] Inevitably, bugs and security issues are introduced as well, causing tremendous financial losses.

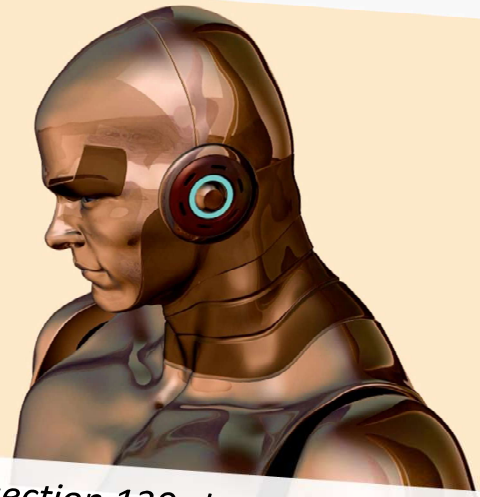
Never miss a story from **Hacker Noon**, when you sign up for Medium. Learn more

GET UPDATES

AUTOMATED RISKS

A rogue robot is blamed for a human colleague's gruesome death

By Ephrat Livni | March 13, 2017



"The robot from section 130 should have never entered section 140, and should never [...] A failure of one or more safety systems or devices had taken place, ...

Home | Our Picks | Popular | Latest | Obsessions | Search

Formal verification—Awards

- 1996 ACM Turing Award:

Pnueli *for his work on Temporal Logic*

- 2007 ACM Turing Award:

Clarke, Emerson & Sifakis *for Model-Checking*

Outline of the Course

1. Some examples of systems and problems
2. What you need to know about graphs, automata
3. Basic properties of systems
4. Modeling systems (Petri nets)
5. Model-Checking
6. Real-time models

Talking about computer science

CS principal areas of study include networking, security, database systems, graphics, numerical analysis, software engineering, …

This course is about *formal verification* but we leave out many questions:

- Behavioral equivalences (when are two models “the same”) and formal equivalence checking
- Automated theorem proving and other deductive approaches, refinement, …
- formal verification of software, abstract interpretation, …

A Tale of Four Systems

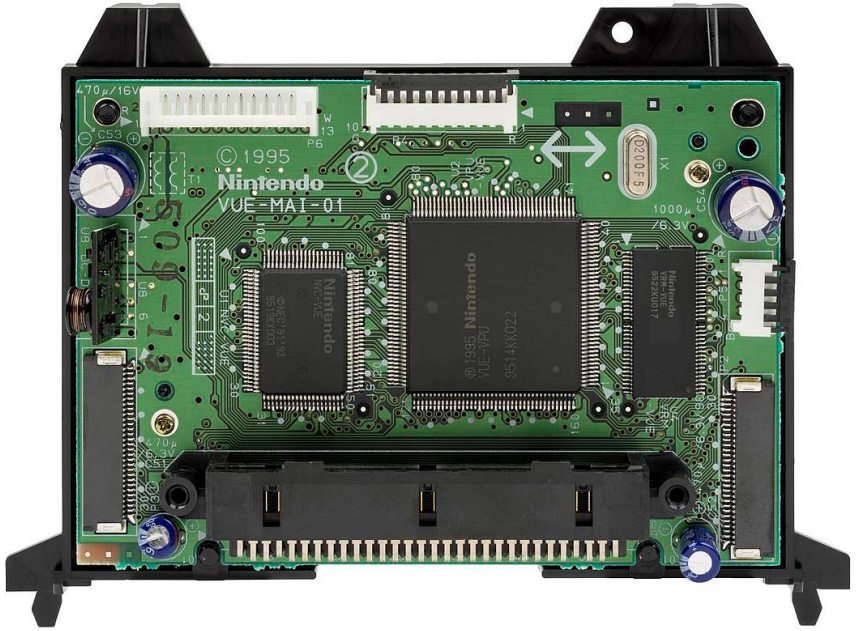
and their associated problems

Concurrent Access to Shared Memory

Example 1/4

Software System

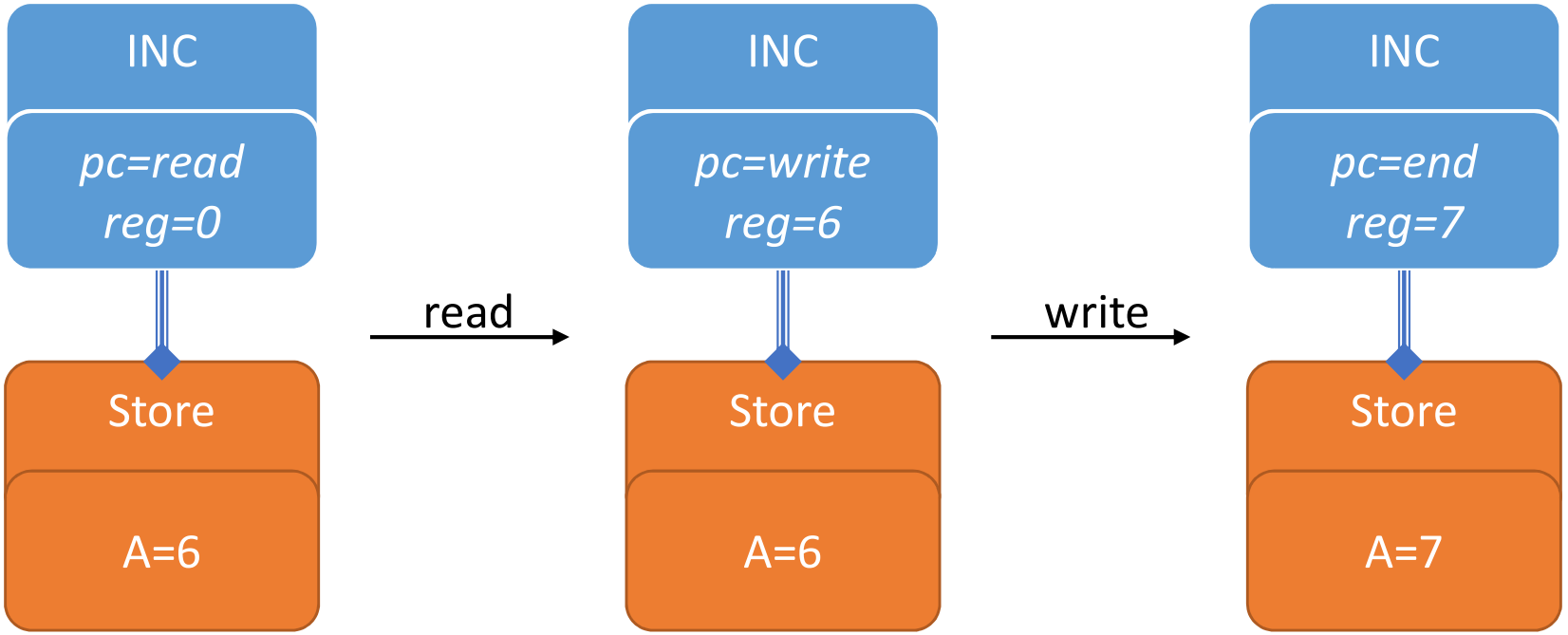
- A sequential software system performs (visible) actions by updating the memory
- Behavior depends on the *state* of the variables (the environment)



Sequential system

INC :=

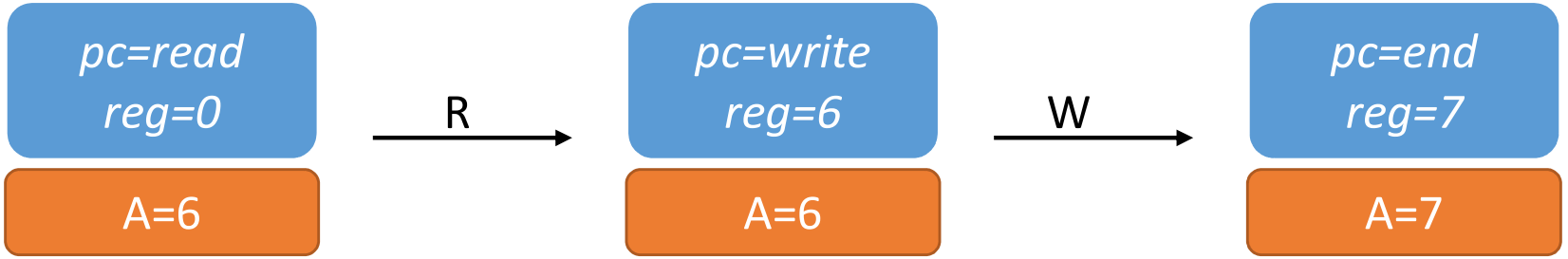
Prog. Pointer	*instruction*	*action*
read	$READ\ A\ \rightarrow reg$	R
write	$WRITE\ A\ \leftarrow reg + 1$	W
end	END	



Sequential system

INC :=

Prog. Pointer	*instruction*	*action*
read	$READ\ A\ \rightarrow reg$	R
write	$WRITE\ A\ \leftarrow reg + 1$	W
end	END	_



Sequential system

INC :=

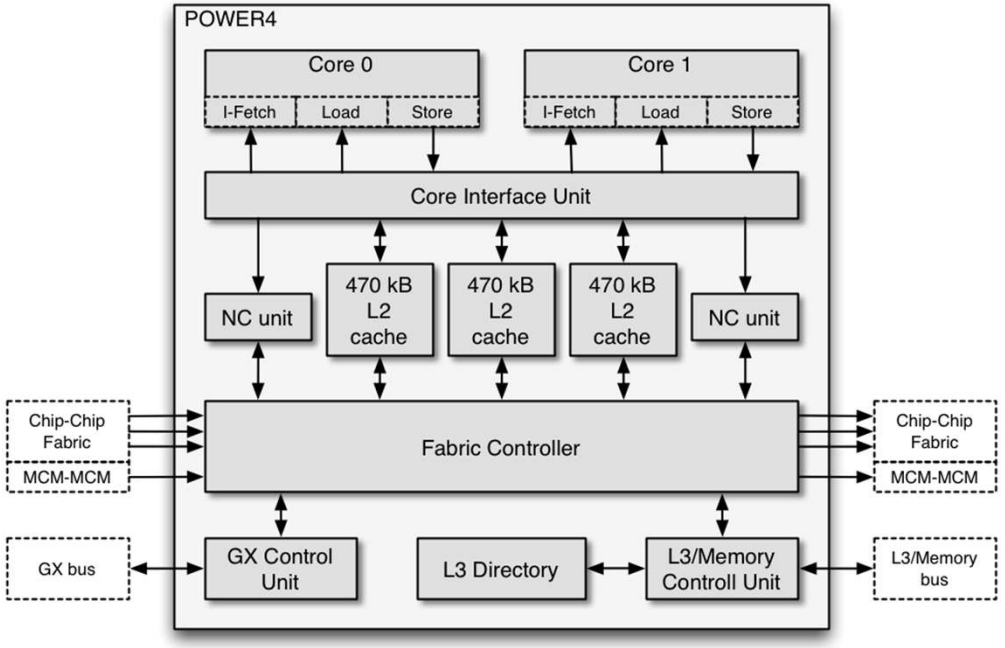
Prog. Pointer	*instruction*	*action*
read	$READ\ A\ \rightarrow reg$	R
write	$WRITE\ A\ \leftarrow reg + 1$	W
end	END	_

a sequential system performs (visible) actions in a sequence

an *execution* is a “sequence”; it is a *trace;* it is a word (R.W) …
the trace can be infinite (e.g. it has a loop)

Concurrency

- A sequential system performs (visible) actions/events
- Behavior depends on the *order* of events



Concurrent systems

INC :=

Prog. Pointer	*instruction*	*action*
read	$READ\ A\ \rightarrow reg$	R
write	$WRITE\ A\ \leftarrow reg + 1$	W
end	END	

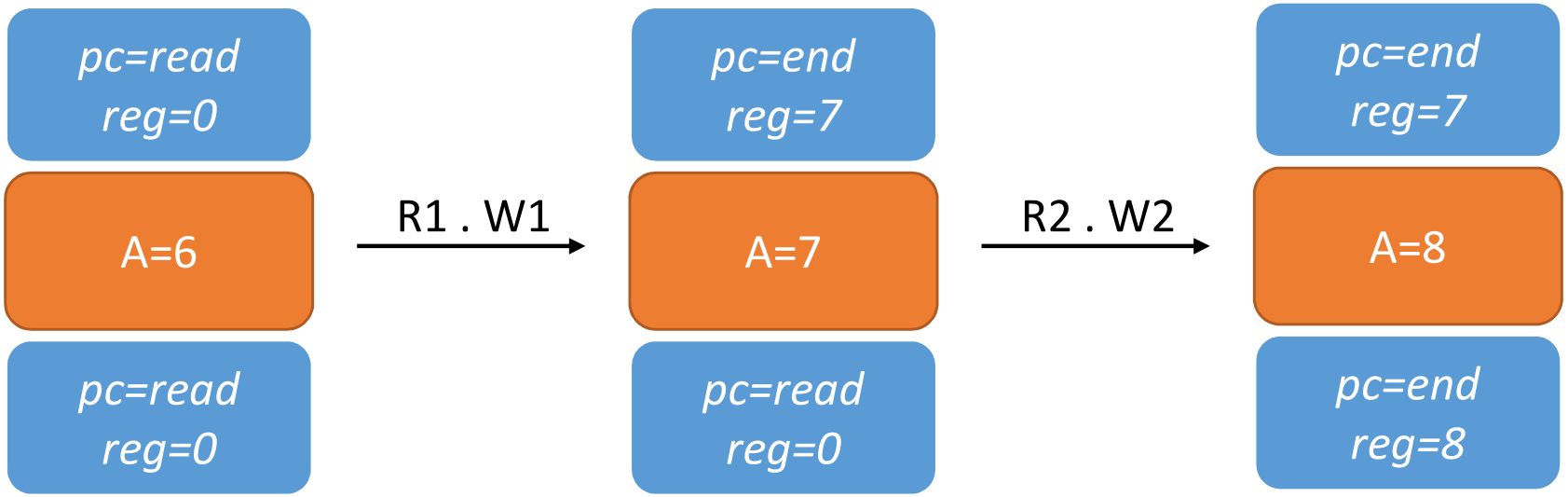
a *concurrent* system is the composition of several sequential subsystems

$$\text{INC} \parallel \text{INC} \parallel \ldots \parallel (if\ A \geq 8\ then\ \text{😊})$$

Race condition

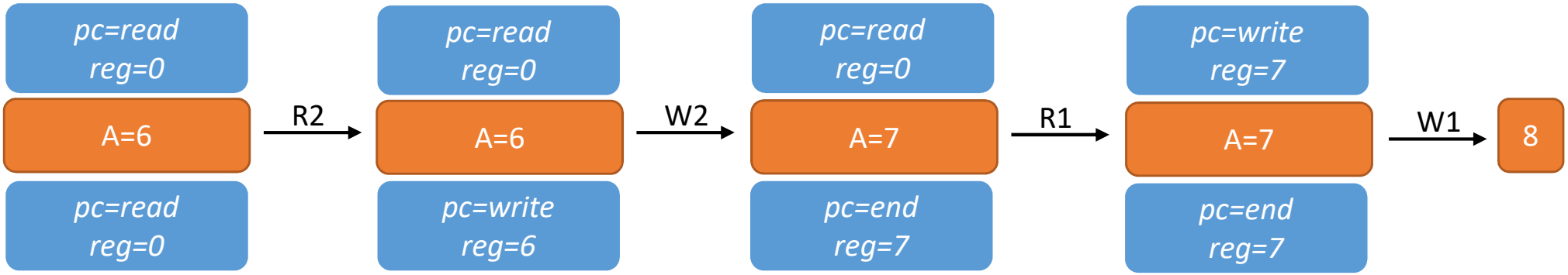
INC ‖ INC ‖ … ‖ $(if\ A \geq 8\ then$ 😊$)$

trace : R1. W1. R2. W2

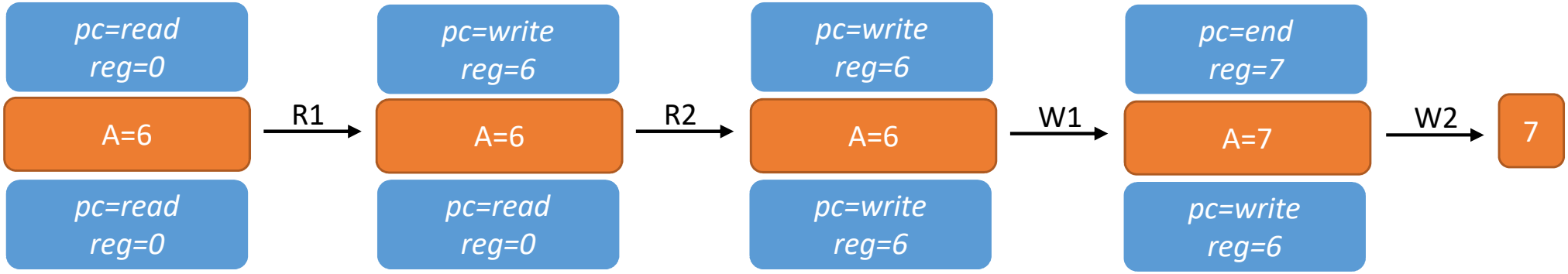


Race condition

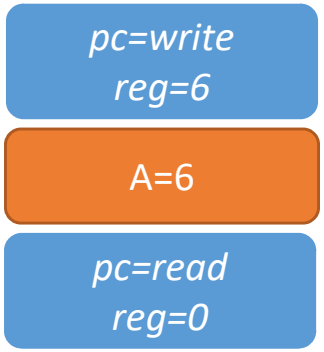
R2. W2. R1. W1



R1. R2. W1. W2

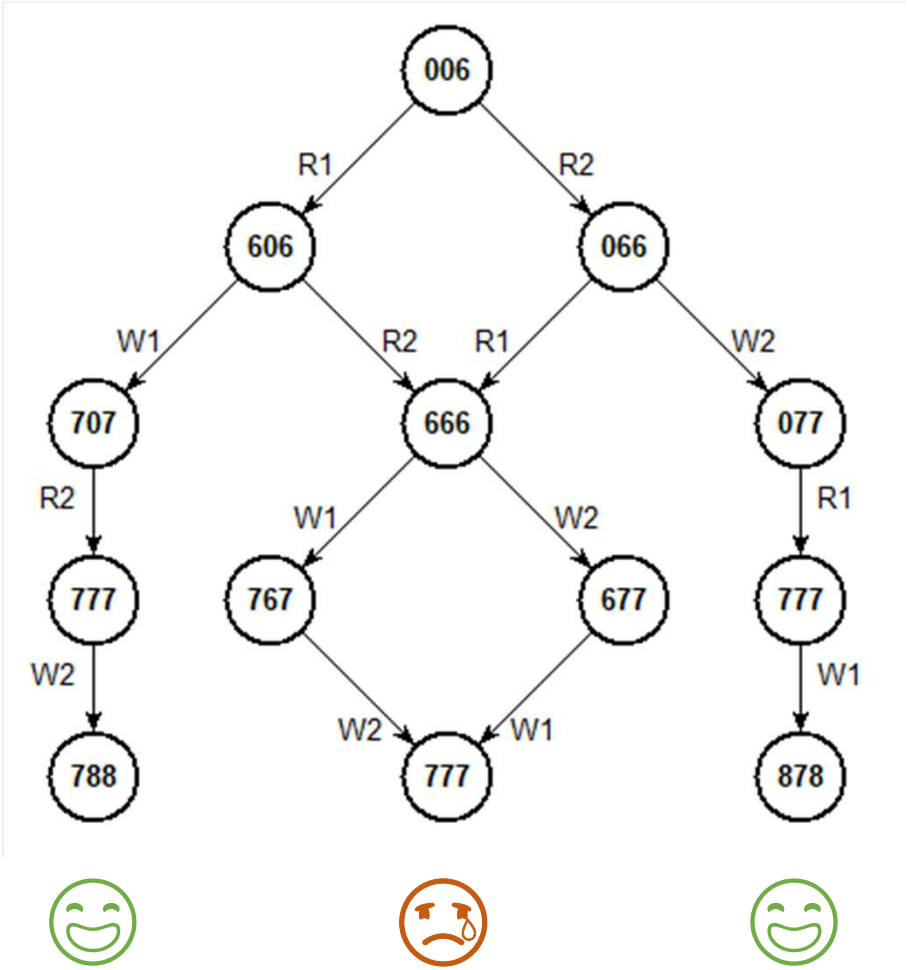


Race condition



|||

606



Race condition (alternate)

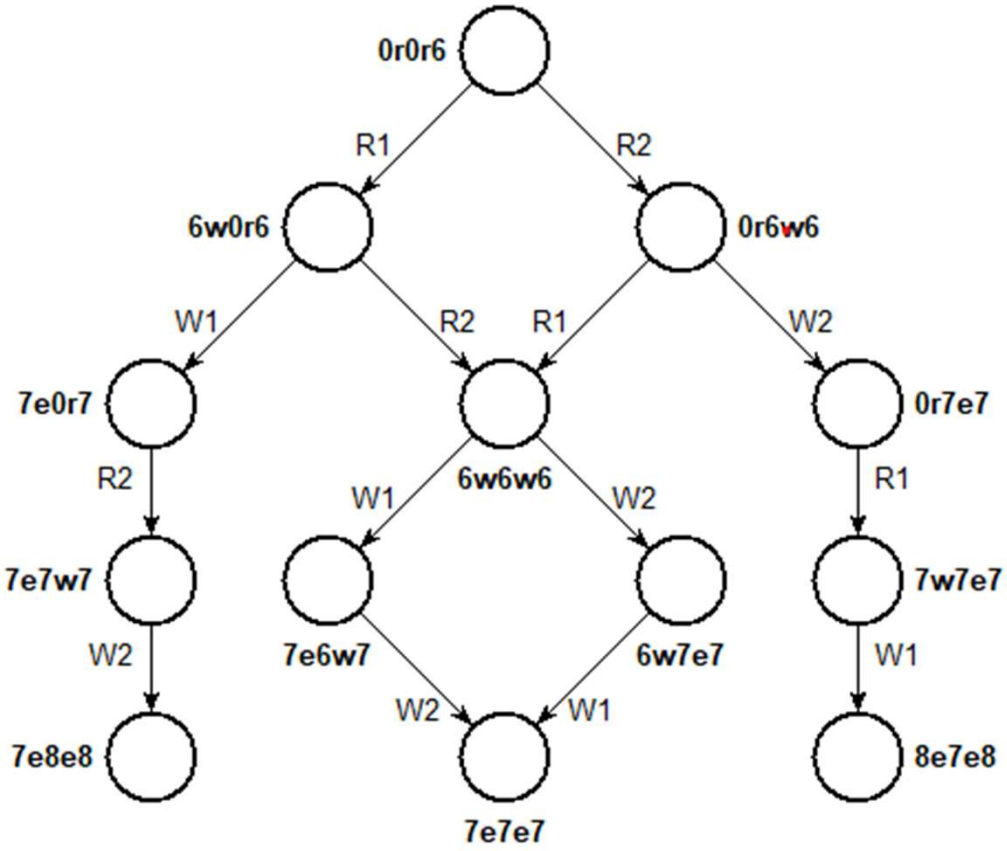
pc=write
reg=6

A=6

pc=read
reg=0

|||

6w0r6



What you need to remember

- We abstract the behavior of the system by looking at the actions/events that modifies its state

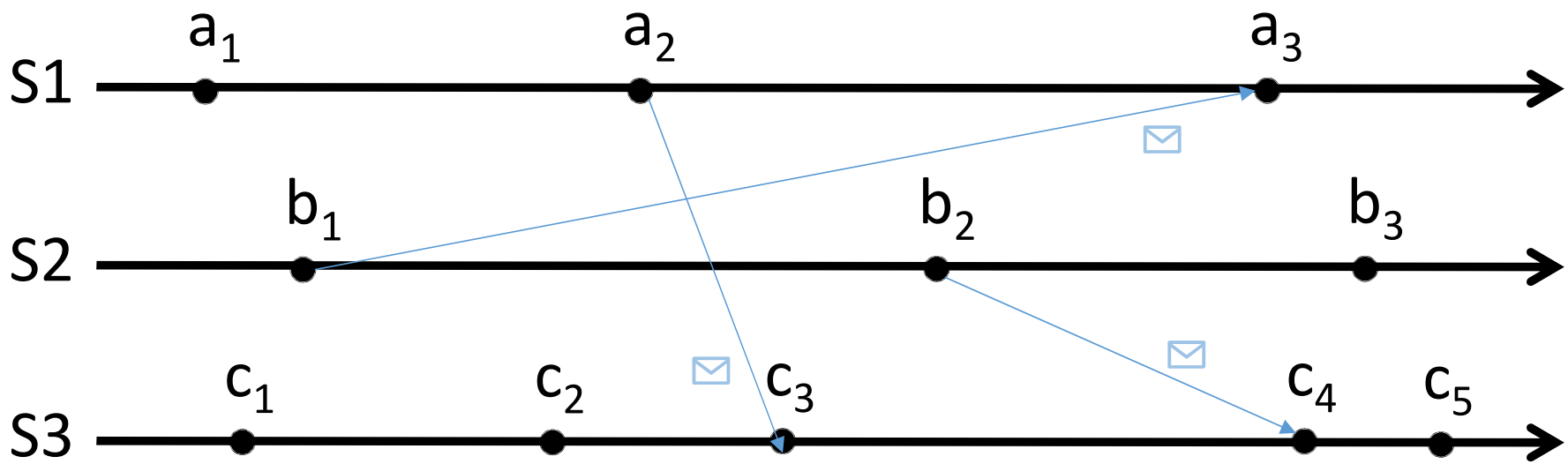
  R1. W1. R2. W2. …

- An execution is a trace (a word) on the alphabet of actions

- *Concurrency*: the order in which events occurs is important

Concurrent systems

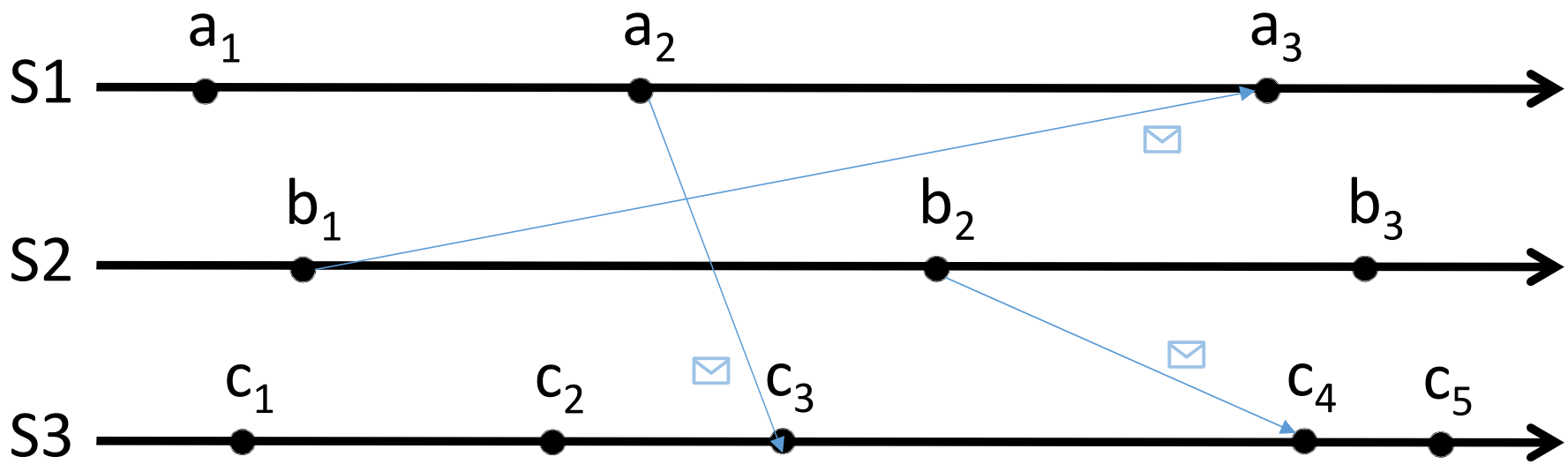
- Interactions are usually *asynchronous*
- Order of events is not fixed: *non-deterministic*
- Concurrency is generally *untractable*



Lamport's events diagrams

Concurrent systems

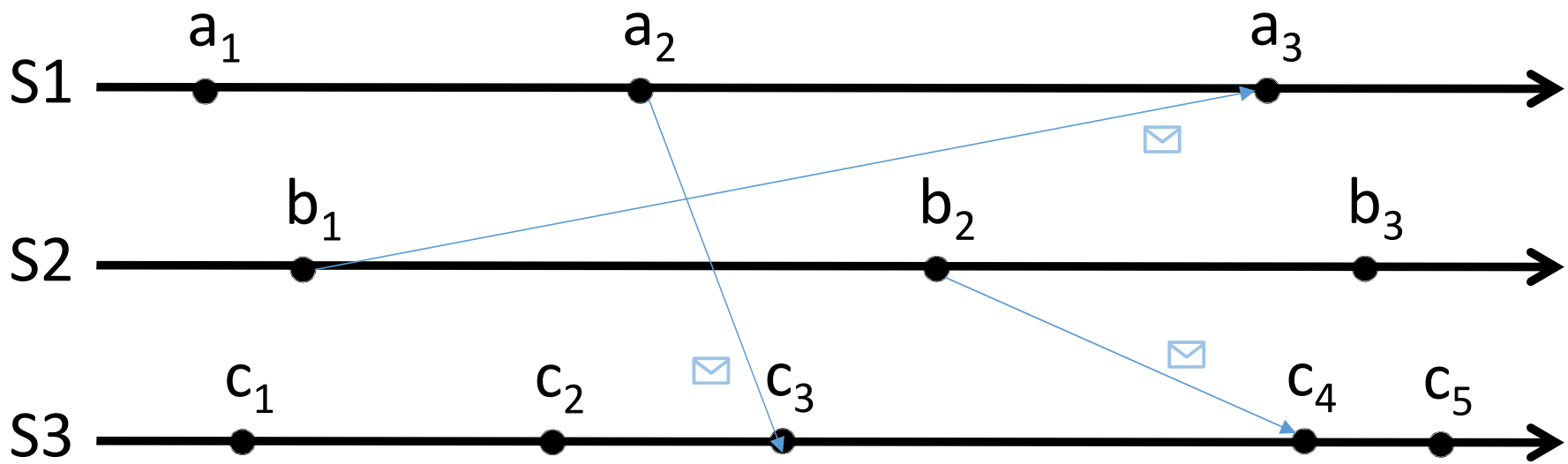
- Assume each system produces *n* events
- There are *k* systems



Concurrent systems

Then number of possible (interleaved) traces is:

$$\frac{(k\ n)!}{n!^k} \approx k^{kn}$$

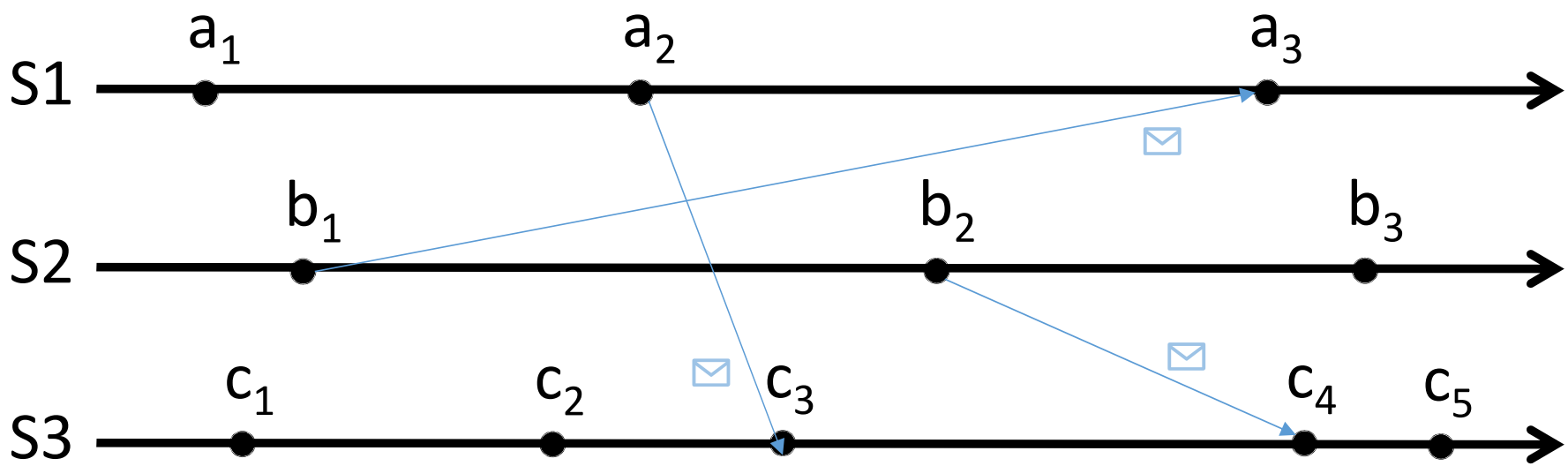


Concurrent systems

3 events, 30 systems $\approx 10^{132}$ traces

"There are about 10^{21} stars in space. This amount is about equal to the number of grains of sand on ALL of the beaches on planet Earth !!! That is a lot!!"

[some guy at NASA]



The curse of concurrent systems

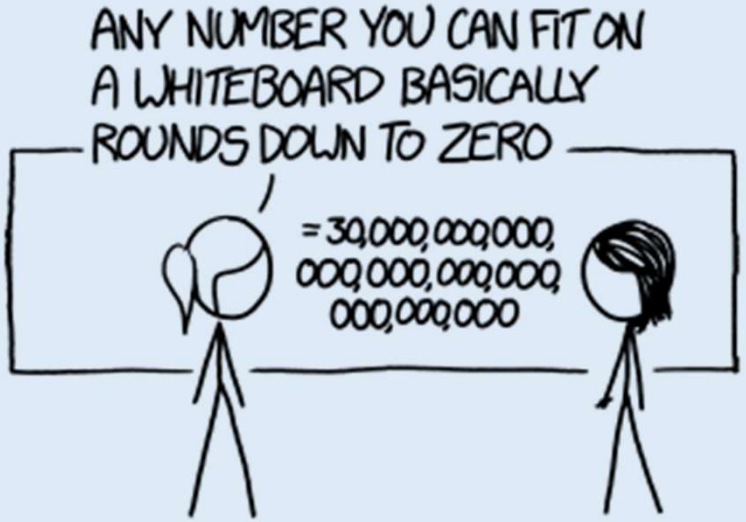
“ We’ve demonstrated how as little as a single bit flip can cause the driver to lose control of the engine speed in real cars...”

(Testimony of expert Michael Barr, trial against Toyota)

“ There are tens of millions of combinations of untested task death, any of which could happen in any possible vehicle/software state. Too many to test them all.”

What you need to remember

- This is *a lot* of states
  - It means that a *system architect* cannot (usually) consider all the possible cases.



- It also means that it is not possible to *test* all the possible cases.
  - “Program testing can be a very effective way to show the **presence of bugs**, but is inadequate for showing their absence”

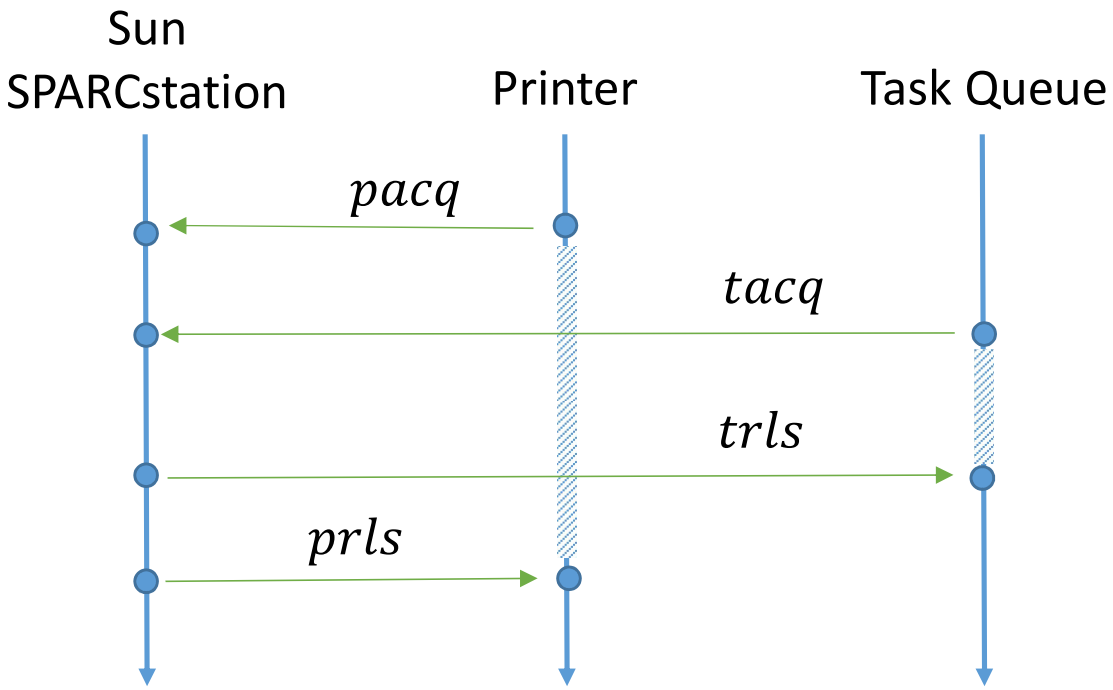
The Job/Printer Pool

Example 2/4

Job/Printer program

To print over a network you need to access a shared printer and to write in a shared task queue

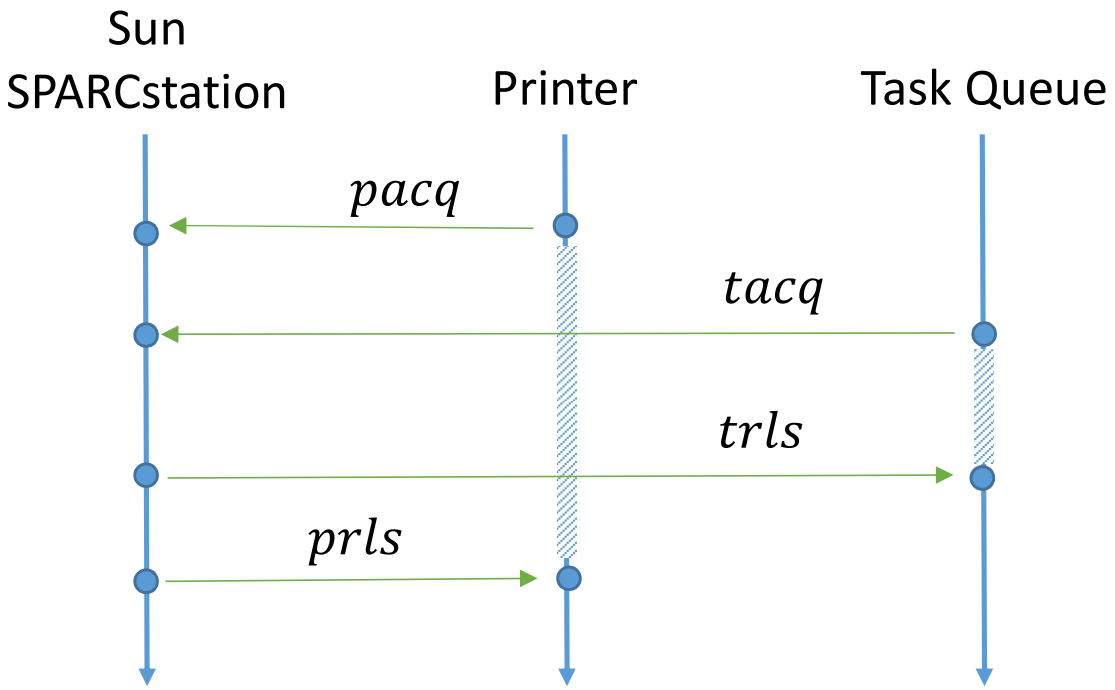
I won't be fooled again, the printer spooler and the task queue are protected by a (software) lock



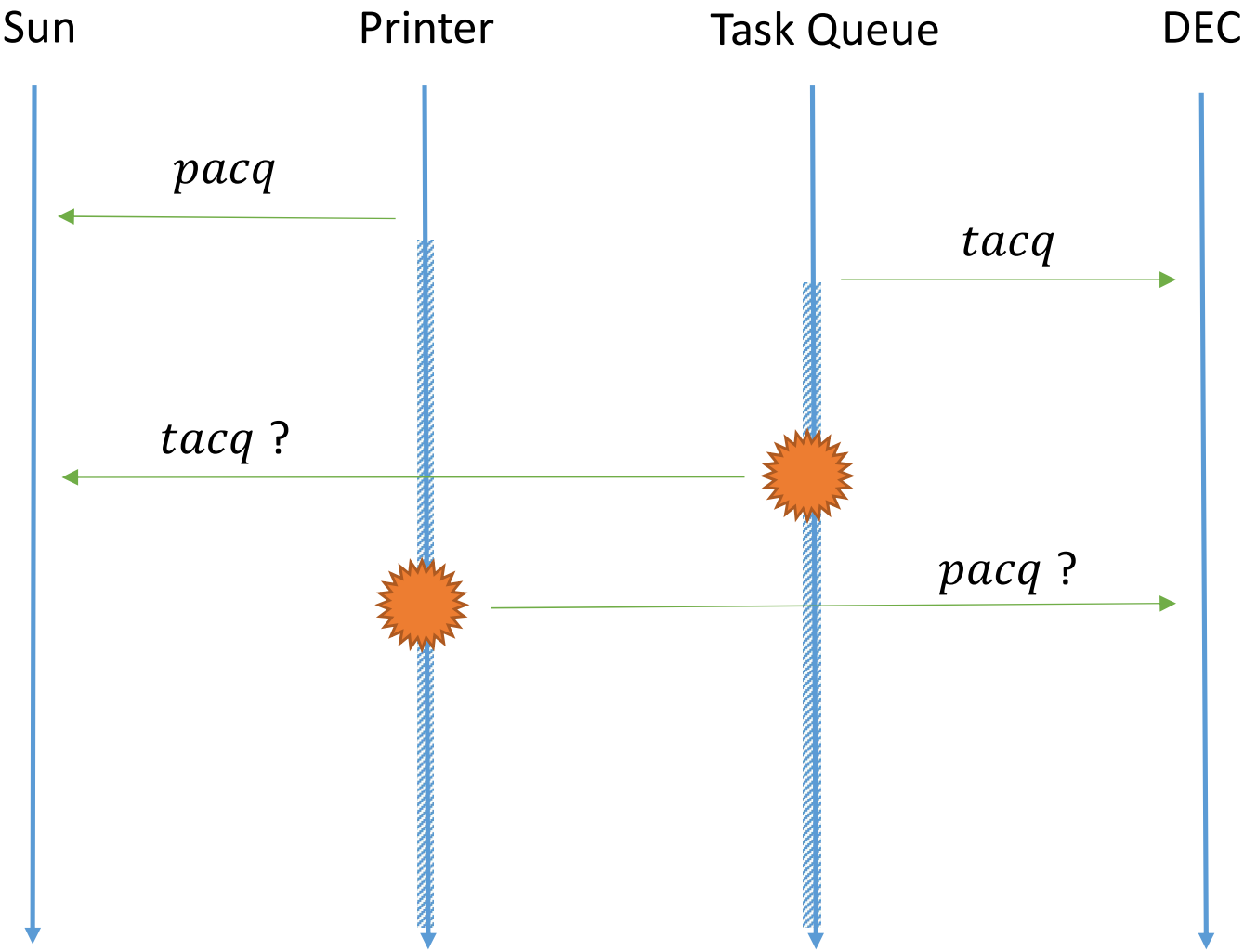
Bonus: this is yet another way engineers describe systems !

Job/Printer program

- On a SPARCstation, I make sure to reserve the printer before accessing the task queue
- On my PDP-11, I ensure first that I have access to the queue before taking the lock to the printer



$(pacq.(tacq.trls).prls)$ $[tacq.[pacq.prls].trls]$



$(pacq.(tacq.trls).[tacq.prls).[pacq.prls].trls]$ OK

$(pacq.[tacq.\ldots$ NOK

What you need to remember

- Not all programs compute results. Many of them are here to monitor and control systems

  reactive programs

  Embedded ; real-time ; distributed systems

- Not all problems are data races

  deadlocks

- We need to account for *resources* and the order in which they are acquired and released

(Classical) Sequential Programs

- Terminate
- Produce results
- Complex data but "simple" control

- Examples:

  sorting, signal processing, compilation

- Based on:

  functions or objects; conditionals; loops

Correctness: check that program terminates with the correct (expected) answer

Reactive Programs

- Not meant to stop !
- Do not generally produce results; but interact (communicate & synchronize)
- Have many independent processes

- Examples:

  communication protocols; control-command systems

- Based on:

  state machines; process algebra; communicating automata; Petri net

Correctness: safe ; live ; fairness ; …

May require an exhaustive search of their state-space

The Drink Vending Machine

Example 3/4

Drink vending machine

- We want to model a machine that sells candies and sodas
- We need to insert coins before buying a soda; but we can also ask for the money back
- However soda can only be dispensed if there are some available
- Does the machine works ? Can I be cheated (put money and not soda; take refund but have soda; …)

What you need to remember

- *Causality*: some actions depend on the occurrence of others (put coins before releasing soda)
- *global state*: the "state" of the machine depends on the local state of may of its (distributed) parts

  a *local change* can have effect on *global state;* this is a composition of interacting/communicating sub-systems
- *internal vs external choice*
- the safety of the system depends on the order in which events can occur → *temporal statements*

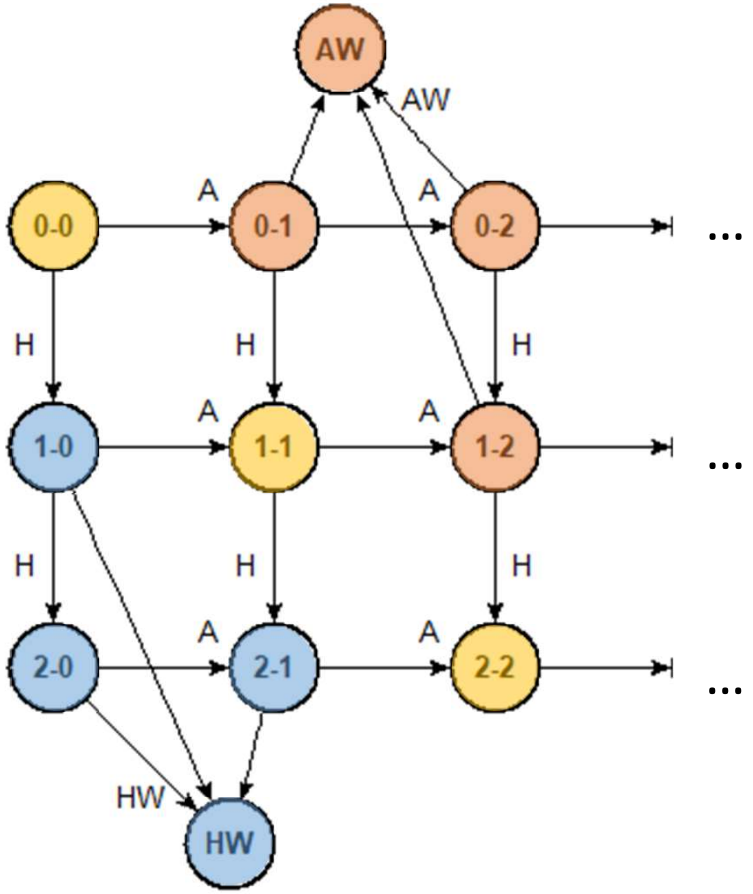
Soccer Game

Example 4/4

Basic rules of the game

- Two teams are playing one against the other
- The score can only change after a goal
- The match can end if there is a score difference ≥ 1

Basic rules of the game

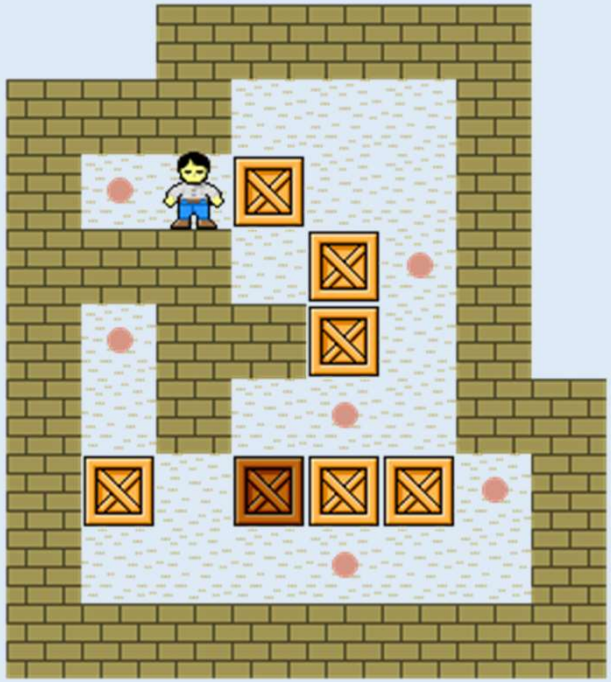


A = Away team scores
H = Home team scores
AW = A wins
HW = we win !

$A.H.A.A.AW \in \mathcal{L}$
$H.AH \in \mathcal{L}$
...

$$w.AW \in \mathcal{L} \Rightarrow \#_A(w) > \#_H(w)$$

What you need to remember



The same approach can be used to deal with different kind of systems; not only software or computing systems:

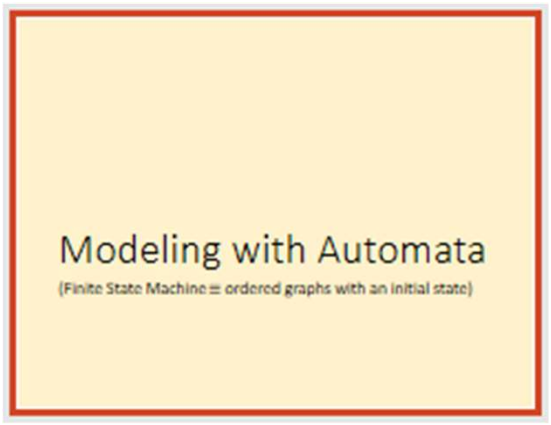
- manufacturing
- planification and optimization problems
- “real-life” protocols (healthcare scenarios, workflows, …)

Also, we may need to deal with infinite/unbounded behaviors

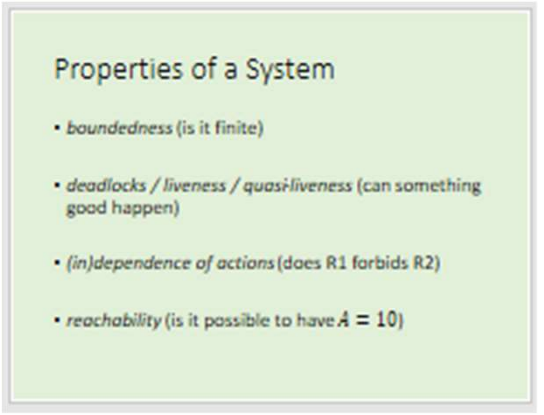
this is mostly the case in fact !

Overview

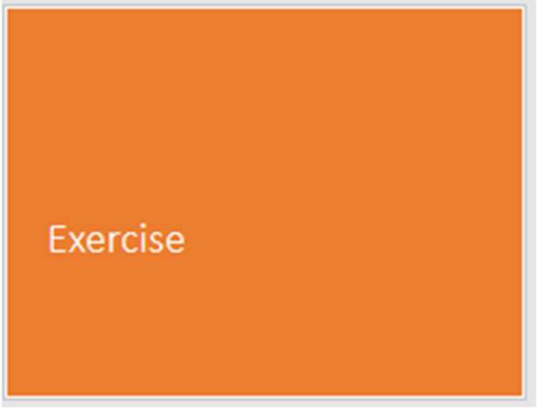
Color Code



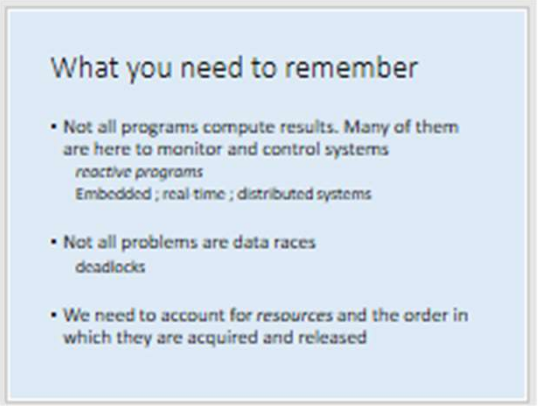
sections



definitions



practical work



reminder

Overview

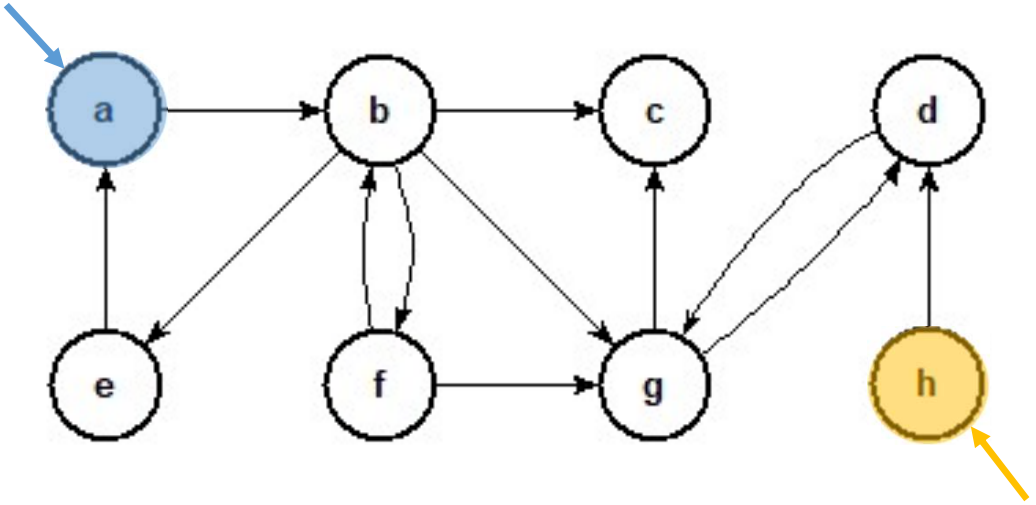
- Four models (and problems) of software systems
- Modeling with automata
- Algorithm on graphs (SCC)
- Modeling with Petri nets and reachability graph
- Model-Checking: behavioral properties, LTL, …
- Extensions: ∞-systems, partial-orders, …
- Symbolic MC using BDD
- Time Petri nets

Modeling with Automata

(Finite State Machine ≡ ordered graphs with an initial state)

Some graph theory

- A *graph* is a pair (V, E) of a set V of vertices (nodes) and E of edges (relation between nodes, in VxV)
- We often deal with *rooted graphs*
- We often deal with *connected graphs*

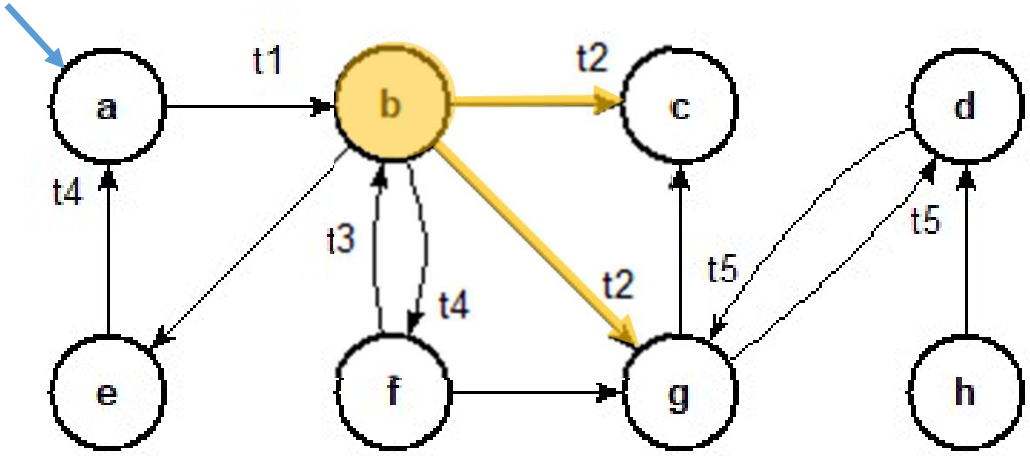


V = {a, b,…, h}

E = {(a,b), (b,c), (b,e), …}

Finite State Automata

- Edges can have labels (in a given *alphabet* Σ)
- We can interpret a (rooted) graph as an *automata* or as a set of words (a *language*)
- Automata can be deterministic or not

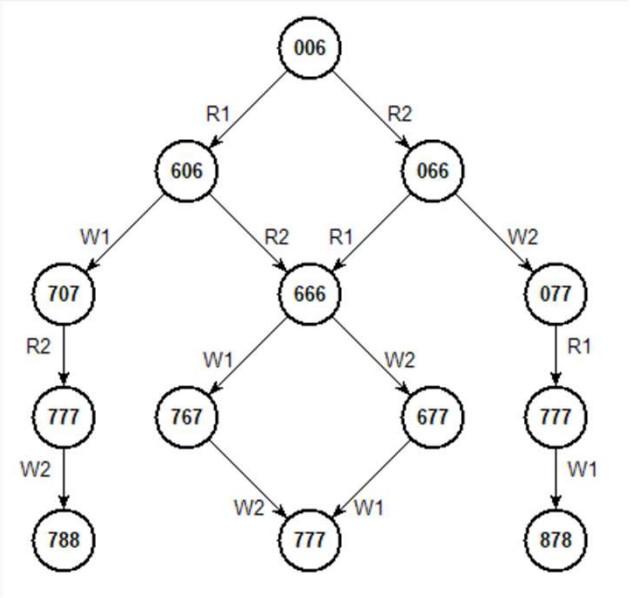


$$\Sigma = \{t_1, t_2, \dots\}$$

$$\mathcal{L} = \{\epsilon, t_1, t_1 . t_2, t_1 . t_4, \dots\}$$

Modeling Behavior with Automata

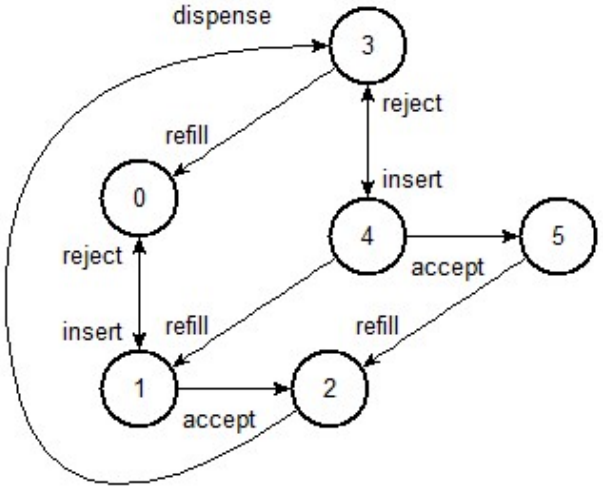
- We use *nodes* to model the state of the system
- We use *edges/labels* to model the visible actions of the system



- The evolution of the system can be viewed as the set of *traces* of the resulting automata
  - need to choose the right level of granularity
  - need to choose what are the atomic actions

Modeling Behavior with Automata

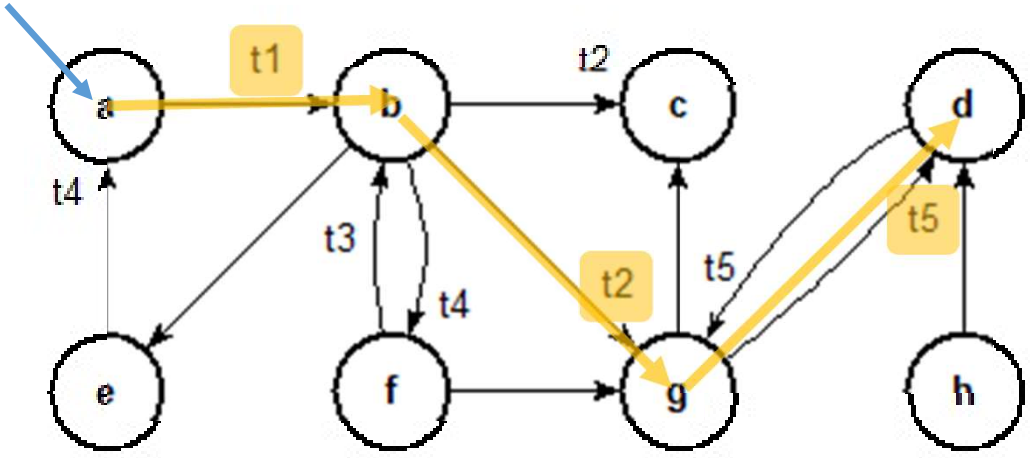
- We use *nodes* to model the state of the system
- We use *edges/labels* to model the visible actions of the system



- The evolution of the system can be viewed as the set of *traces* of the resulting automata
  - need to choose the right level of granularity
  - need to choose what are the atomic actions

Finite State Automata

- Edges can have labels (in a given *alphabet* Σ)
- The language of the automata, $\mathcal{L}$, is the set of words that you can form, starting from the initial state, by concatenating labels on the edges.
- We only need *prefix-closed languages* here !

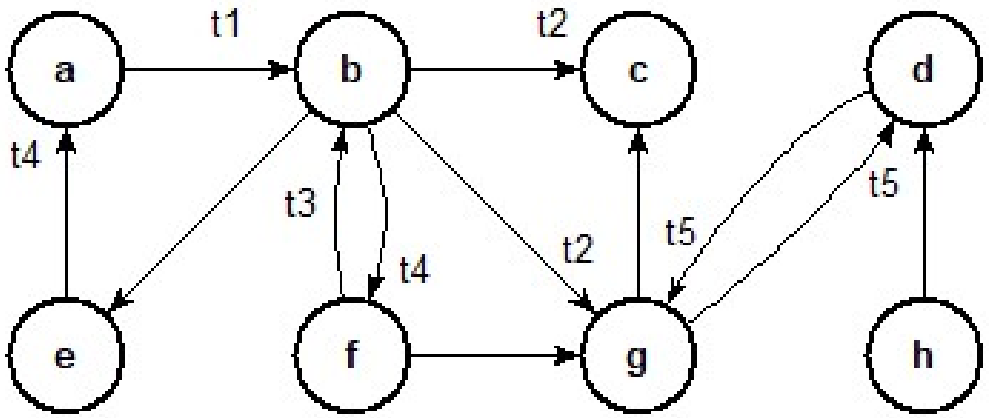


$$\Sigma = \{t_1, t_2, \dots, t_5\}$$

$$t_1.t_2.t_5 \in \mathcal{L}$$

Finite State Automata

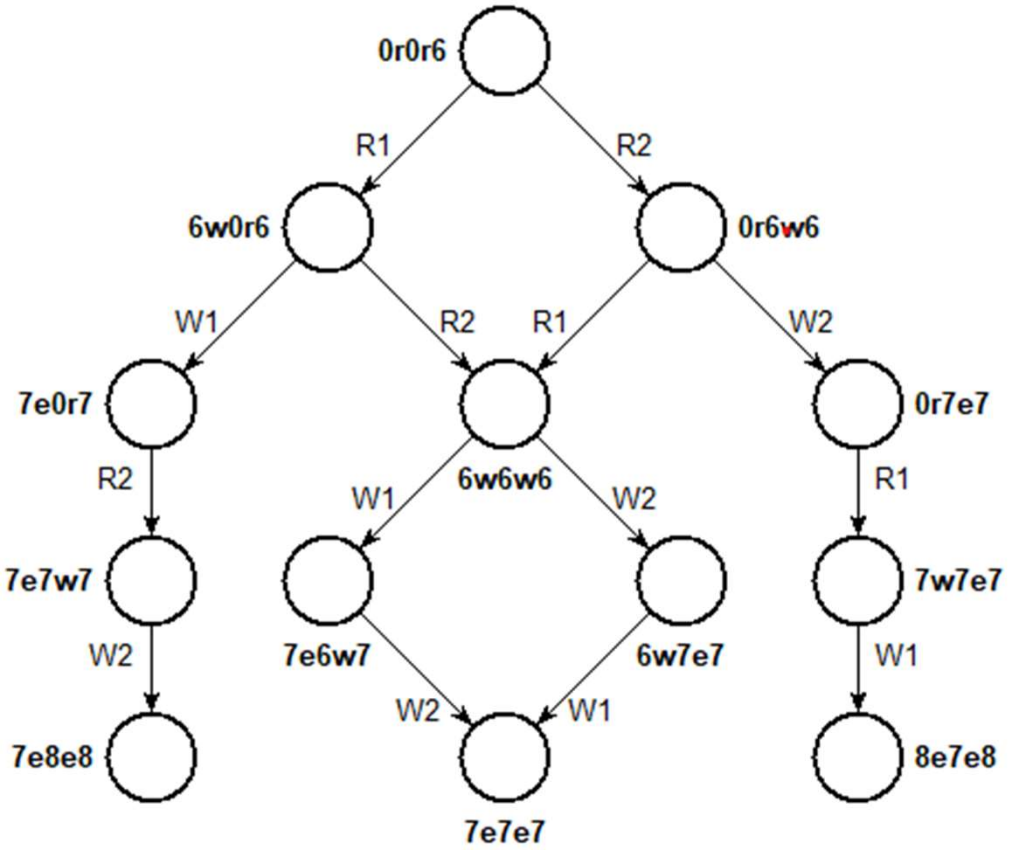
- In our case an automata $\mathcal{A}$ is defined by its sets of states Q, set of labels (alphabet) Σ, set of transitions Δ, and initial q_0
- In our running example: $\mathcal{A} = (Q, \Delta, a)$ where $Q = \{a, b, c, d, e, f, g, h\}$ and, e.g., $b \xrightarrow{t_2} c \in \Delta$



Properties of a System

- *boundedness* (is it finite)
- *deadlocks / liveness / quasi-liveness* (can something good happen)
- *(in)dependence of actions* (does R1 forbids R2)
- *reachability* (is it possible to have $A = 10$)

Race condition



pc=write
reg=6

A=6

pc=read
reg=0

=

6w0r6

pc=end
reg=7

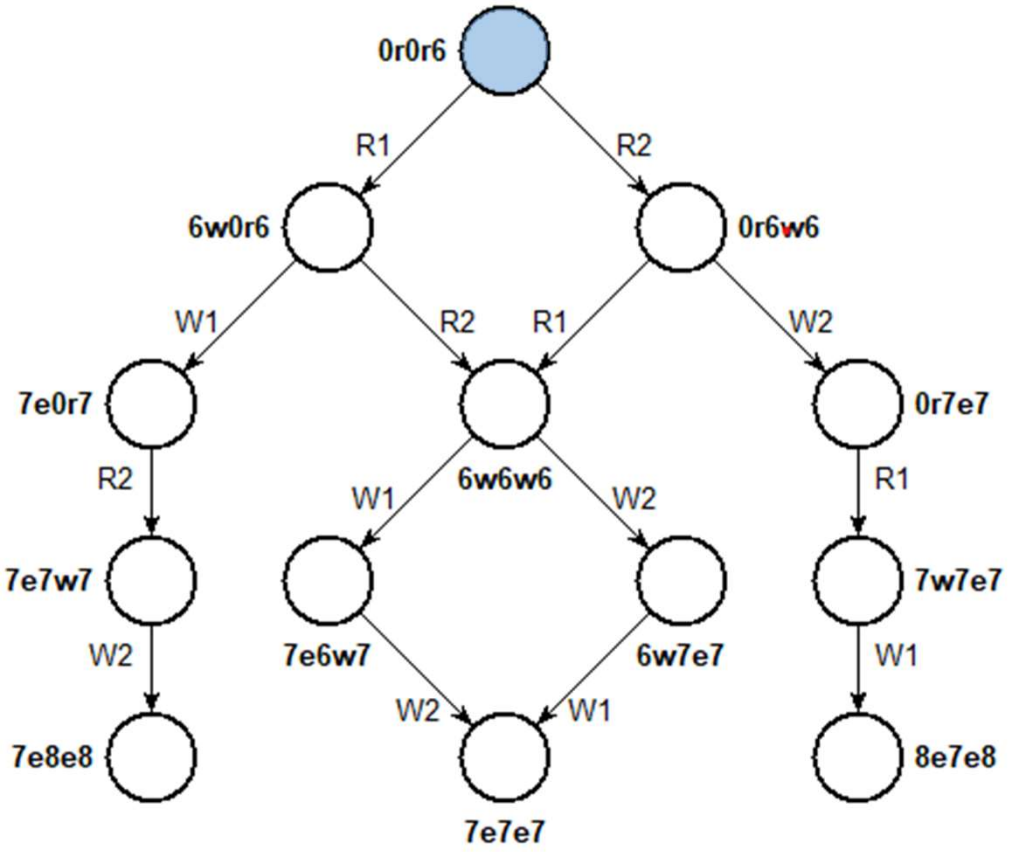
A=7

pc=write
reg=7

=

7e7w7

Race condition



$$\Sigma = \{R_1, W_1, R_2, W_2\}$$

$$\mathcal{L} \ni R_1.W_1.R_2.W_2$$

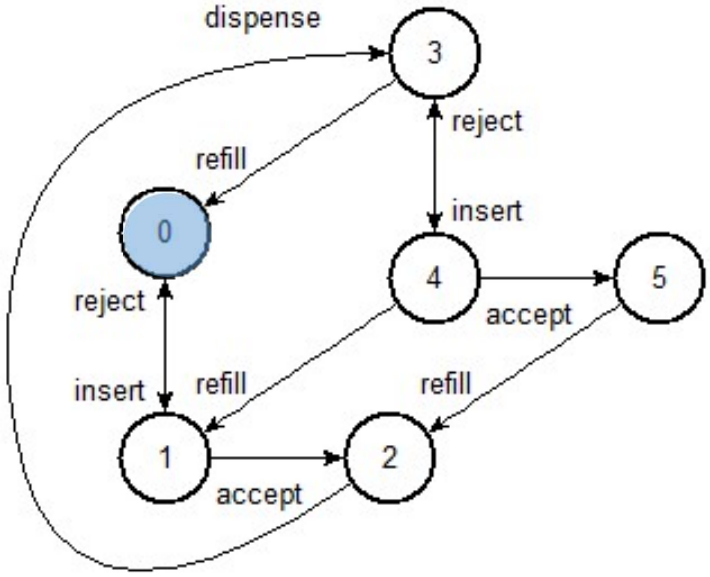
$$\mathcal{L} \ni R_1.R_2.W_1.W_2$$

$$\mathcal{L} \ni R_1.R_2$$

$\mathcal{L}$ prefix-closed

$\mathcal{L}$ is a finite set

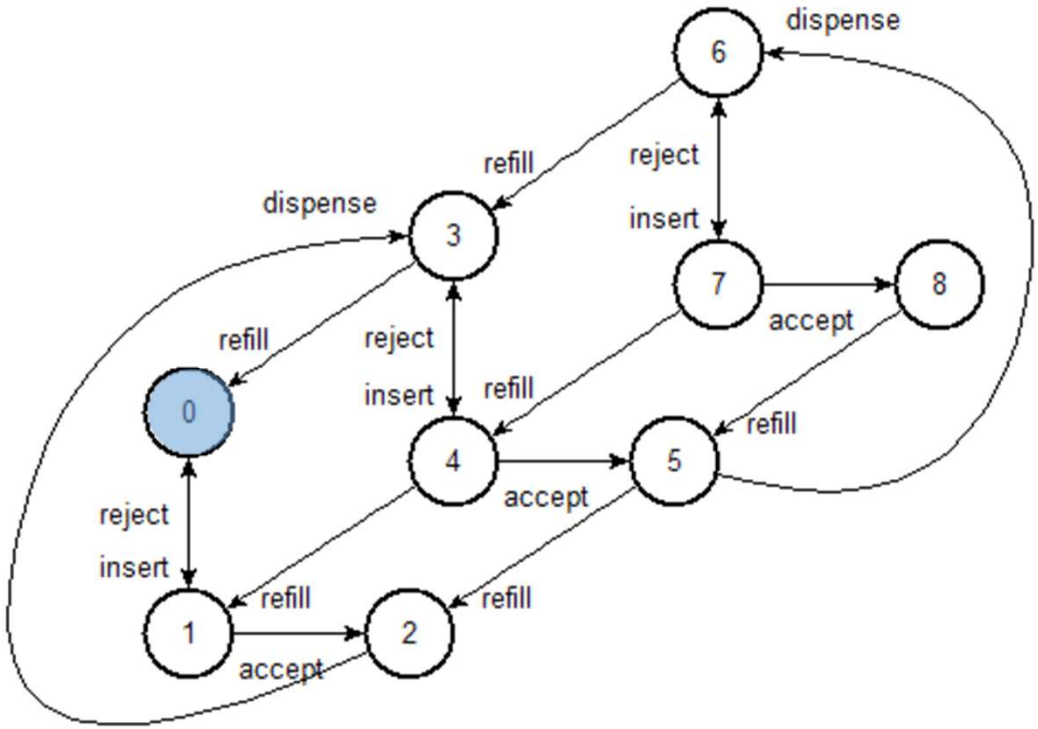
Drink vending machine



$$\Sigma = \begin{cases} \text{insert} & \text{insert coins} \\ \text{reject} & \text{recover coin} \\ \text{accept} & \text{push button} \\ \text{dispense} & \text{receive soda} \\ \text{refill} & \text{refill soda} \end{cases}$$

insert.accept.dispense.insert.refill … $\approx$ insert. …
(soda full, 1 coin inserted)

Drink vending machine



$$\Sigma = \begin{cases} \text{insert} & \text{insert coins} \\ \text{reject} & \text{recover coin} \\ \text{accept} & \text{push button} \\ \text{dispense} & \text{receive soda} \\ \text{refill} & \text{refill soda} \end{cases}$$

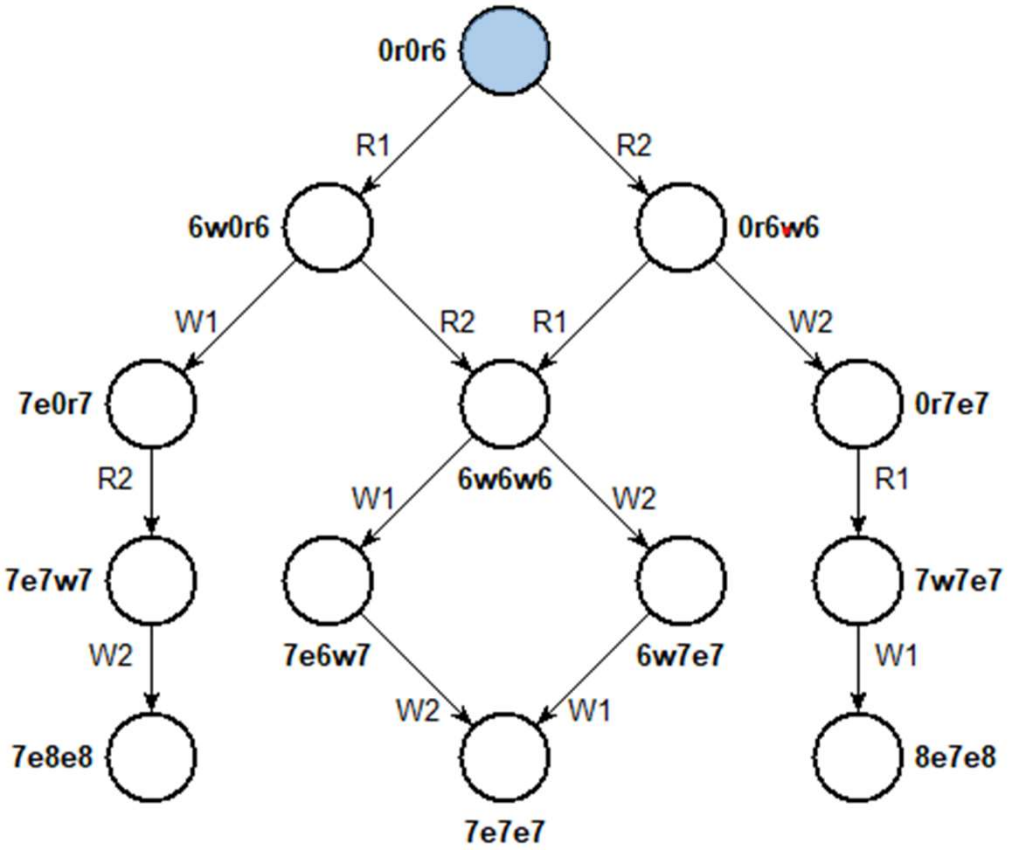
same machine but with capacity of 2 soda cans

Regular language

- *Regular languages* can be defined as languages recognized by a finite automaton (FSM)
- Alternatively, they can be expressed using regular expressions (regexp)

R ::=	ϵ	denotes the empty string
|	a	literal character ($a \in \Sigma$)
|	$R_1 \,.\, R_2$	concatenation
|	$R_1 + R_2$	alternation, choice, union
|	$R*$	Kleene star ($\epsilon + R + R.R + \cdots$)

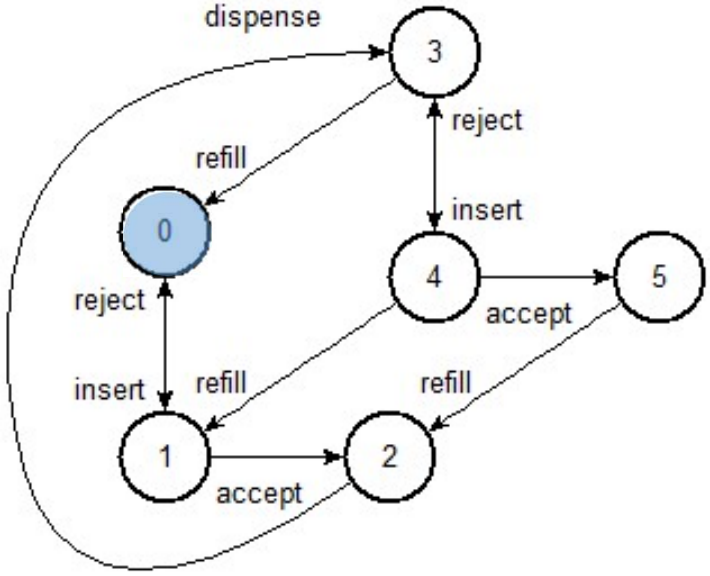
Race condition



$$\Sigma = \{R_1, W_1, R_2, W_2\}$$

$$\mathcal{L} = \begin{array}{ll} & R_1.W_1.R_2.W_2 \\ + & R_1.R_2.W_1.W_2 \\ + & R_1.R_2.W_2.W_1 \\ + & \ldots \end{array}$$

Drink vending machine

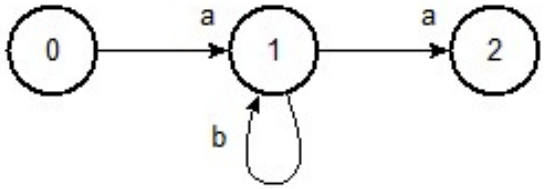


$$\Sigma = \begin{cases} \text{insert} & \text{insert coins} \\ \text{reject} & \text{recover coin} \\ \text{accept} & \text{push button} \\ \text{dispense} & \text{receive soda} \\ \text{refill} & \text{refill soda} \end{cases}$$

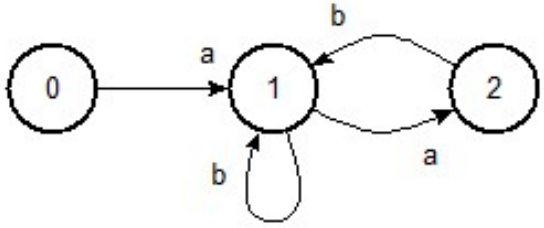
$$(insert.accept.dispense.refill + \epsilon)* \subseteq \mathcal{L}$$

Regular language

$$
\begin{aligned}
R ::= & \ \epsilon \\
& \mid a \\
& \mid R_1 . R_2 \\
& \mid R_1 + R_2 \\
& \mid R*
\end{aligned}
$$



$$\mathcal{L} \equiv \epsilon + a.b* + a.b*.a$$



$$\mathcal{L} \equiv (a.b*)*$$

$$b* \equiv \epsilon + b + b.b + b.b.b + \ \ldots$$

$$(a.b*)* \equiv \epsilon + a + a.b + a.b.b + a.b.a + \ \ldots$$

Strongly Connected Components

Computing SCCs

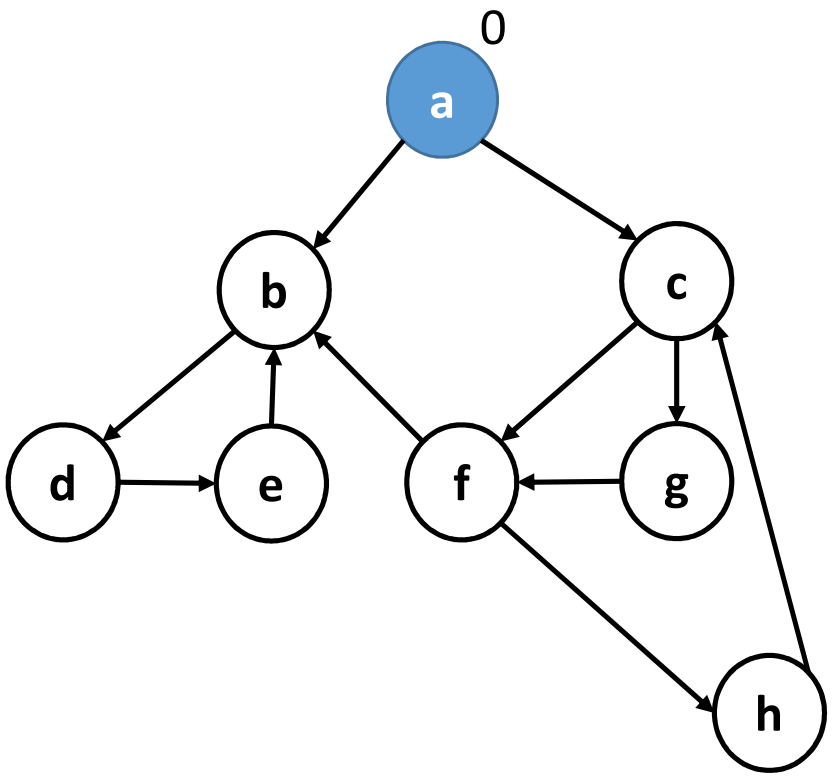
Order of exploration

- There are many different ways to “explore” the nodes of a graph, but two main strategies when you can order the “children” of a node

- Breadth-first search (BFS): explore the nodes at the present depth

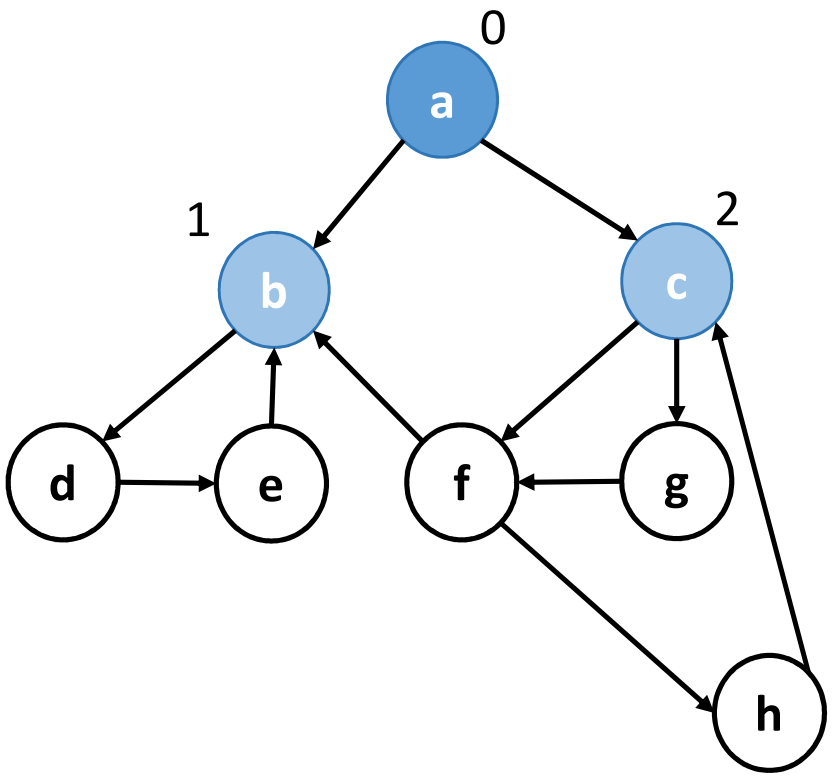
- Depth-first search (DFS): explore highest-depth nodes first

BFS



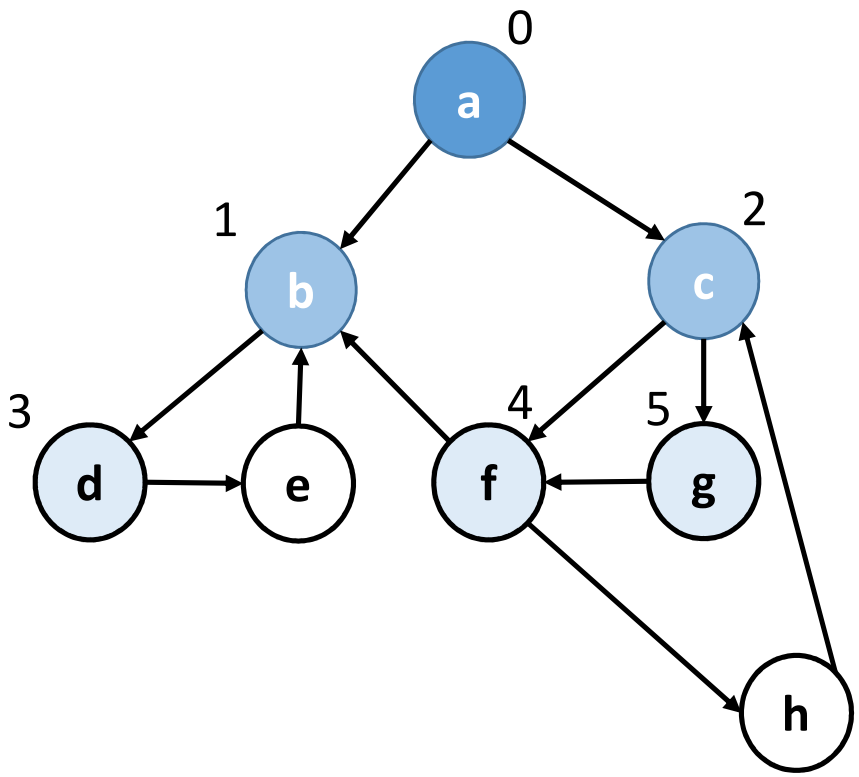
#	id	d
0	a	0
1		
2		
3		
4		
5		
6		
7		

BFS



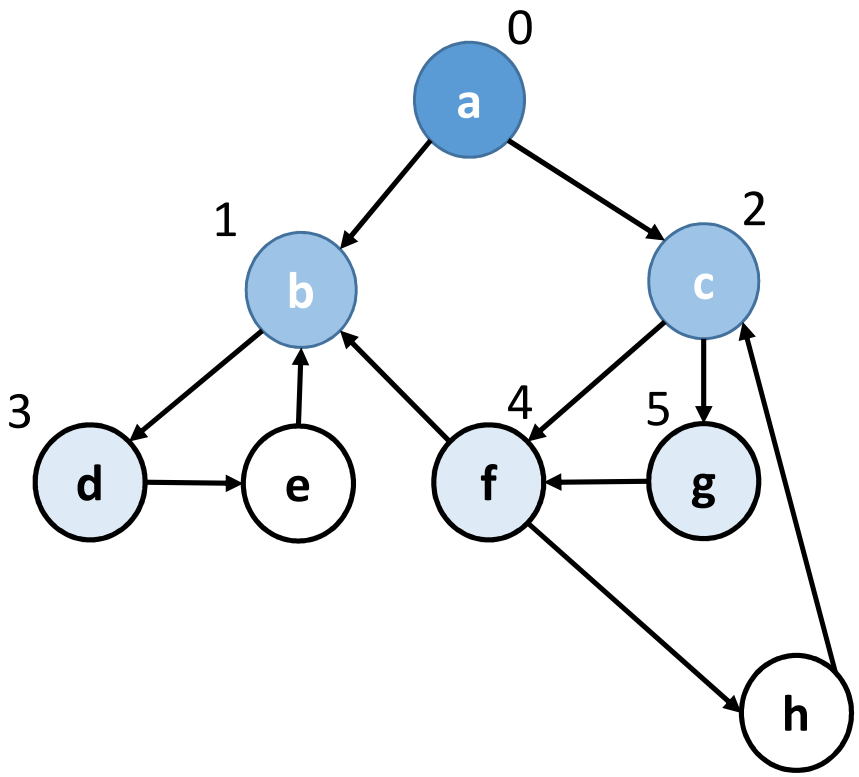
#	id	d
0	a	0
1	b	1
2	c	1
3		
4		
5		
6		
7		

BFS



#	id	d
0	a	0
1	b	1
2	c	1
3	d	2
4	f	2
5	g	2
6		
7		

BFS



#	id	d
0	a	0
1	b	1
2	c	1
3	d	2
4	f	2
5	g	2
6	e	3
7	h	3

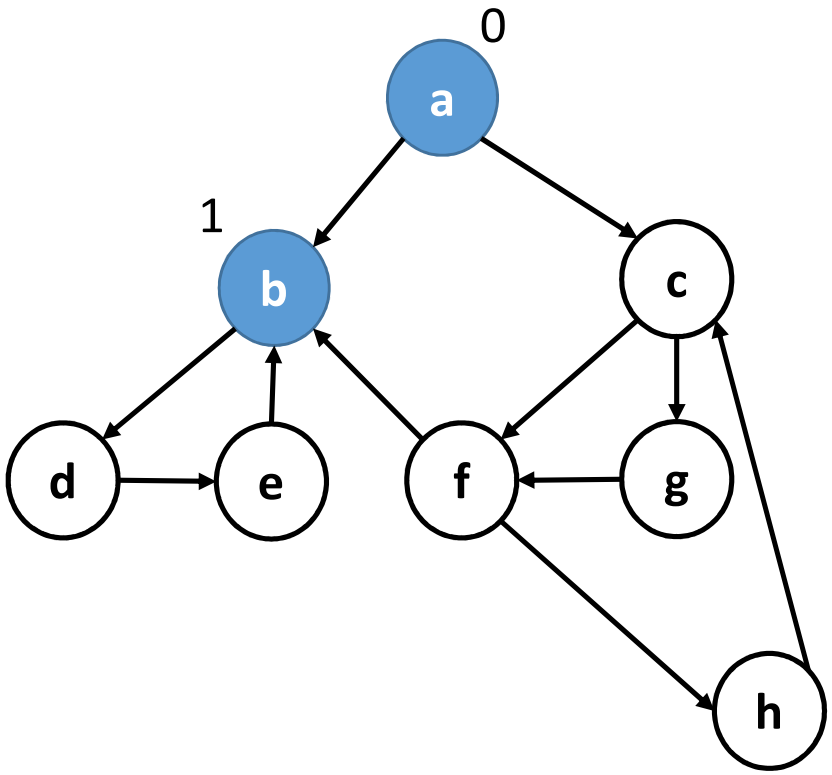
Order of exploration

- There are many different ways to “explore” the nodes of a graph, but two main strategies when you can order the “neighbors” of a node

- Breadth-first search (BFS): explore the nodes at the present depth

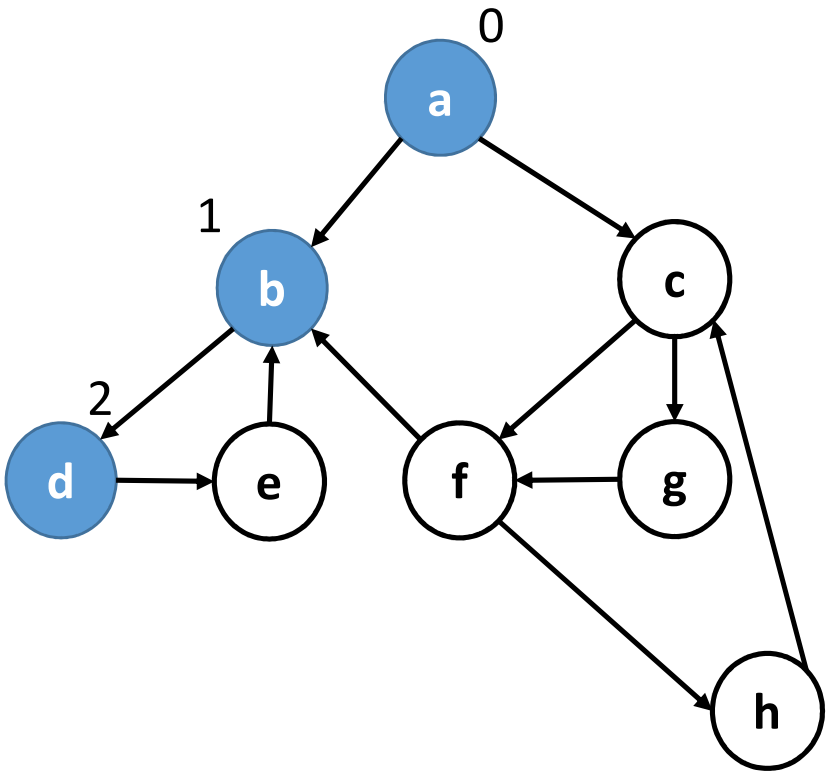
- **Depth-first search** (DFS): explore highest-depth nodes first

DFS



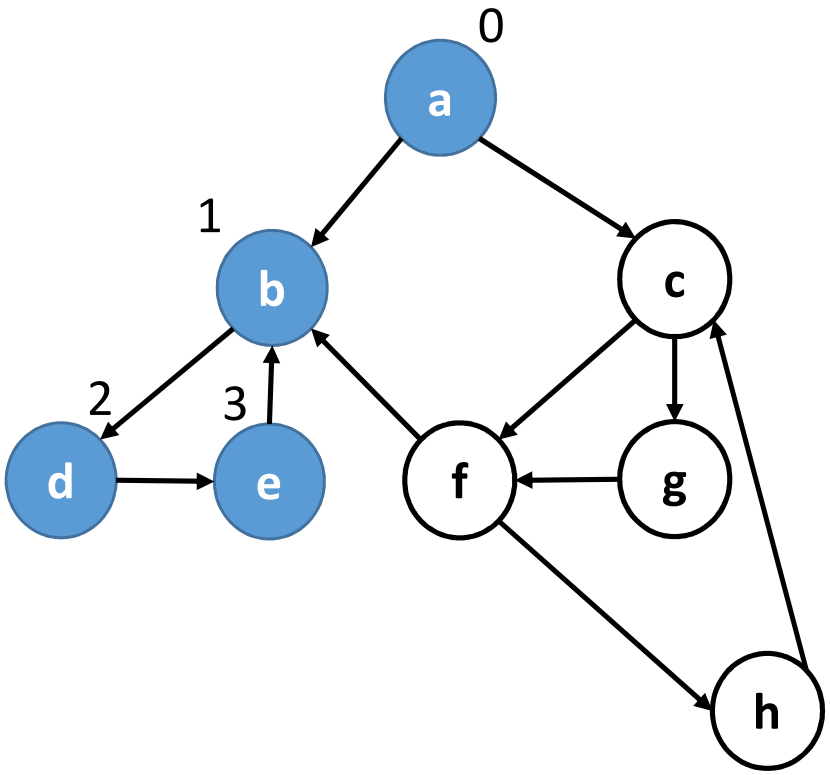
#	id	d
0	a	0
1	b	1
2		
3		
4		
5		
6		
7		

DFS



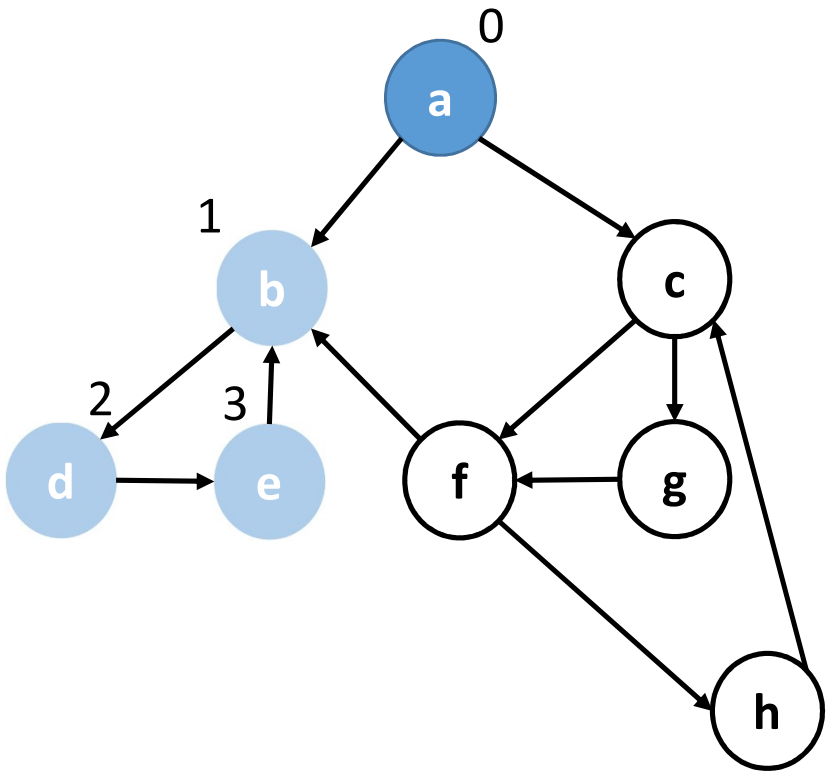
#	id	d
0	a	0
1	b	1
2	d	2
3		
4		
5		
6		
7		

DFS



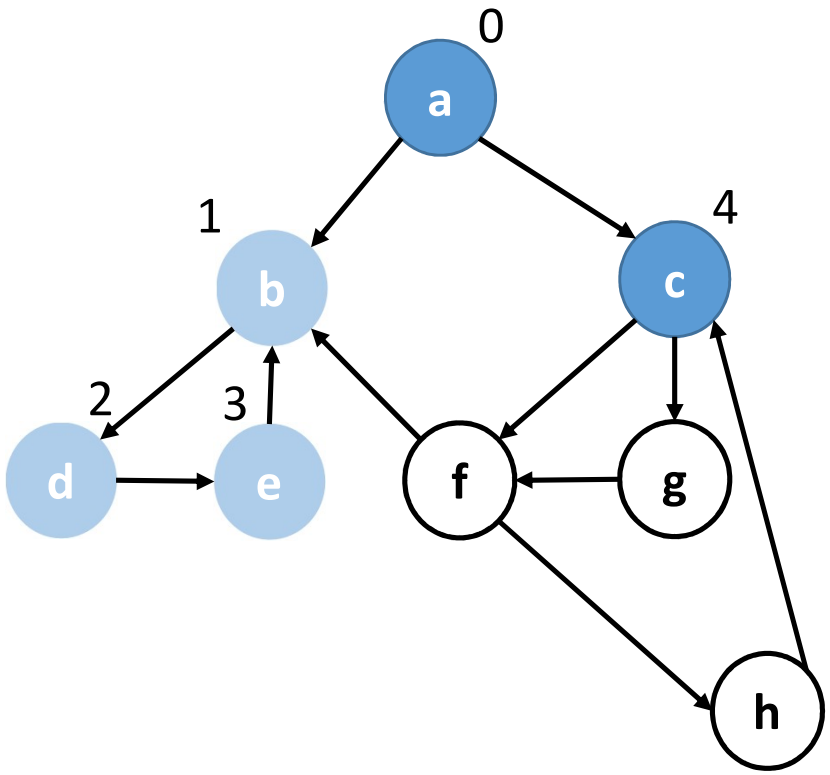
#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4		
5		
6		
7		

DFS



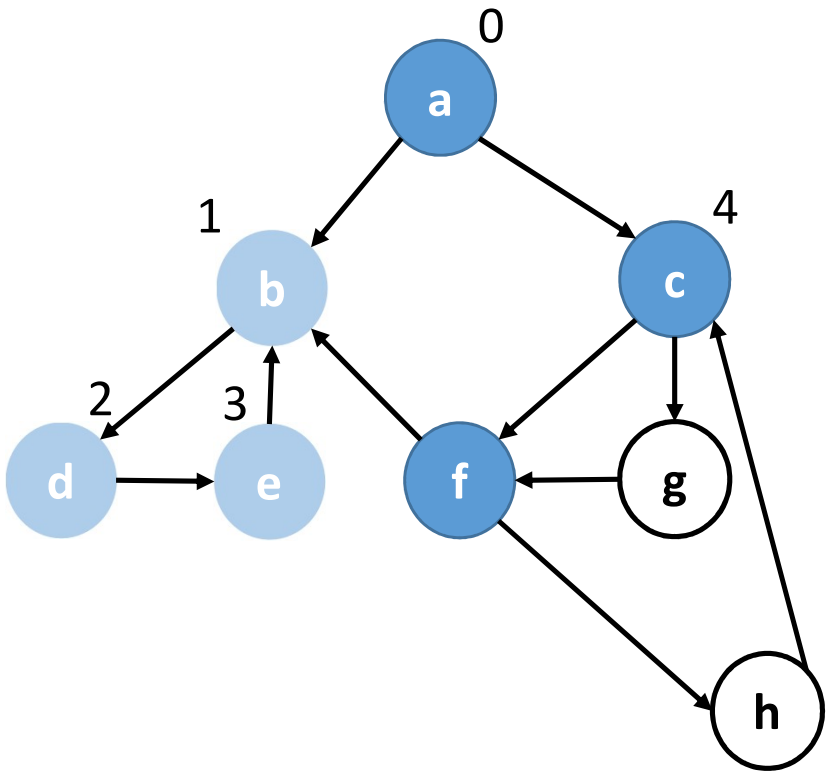
#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4		
5		
6		
7		

DFS



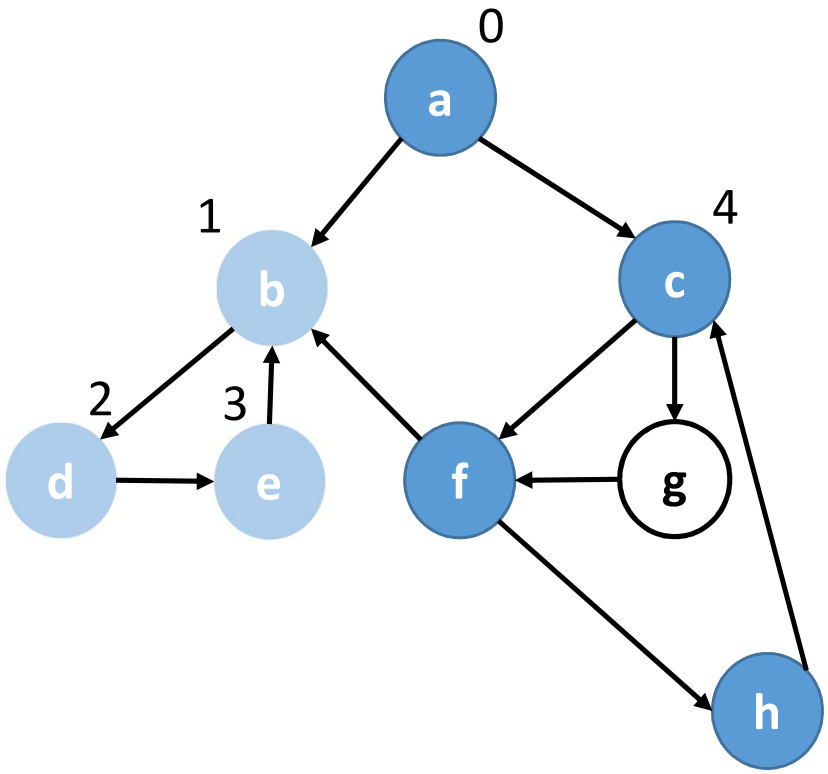
#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4	c	1
5		
6		
7		

DFS



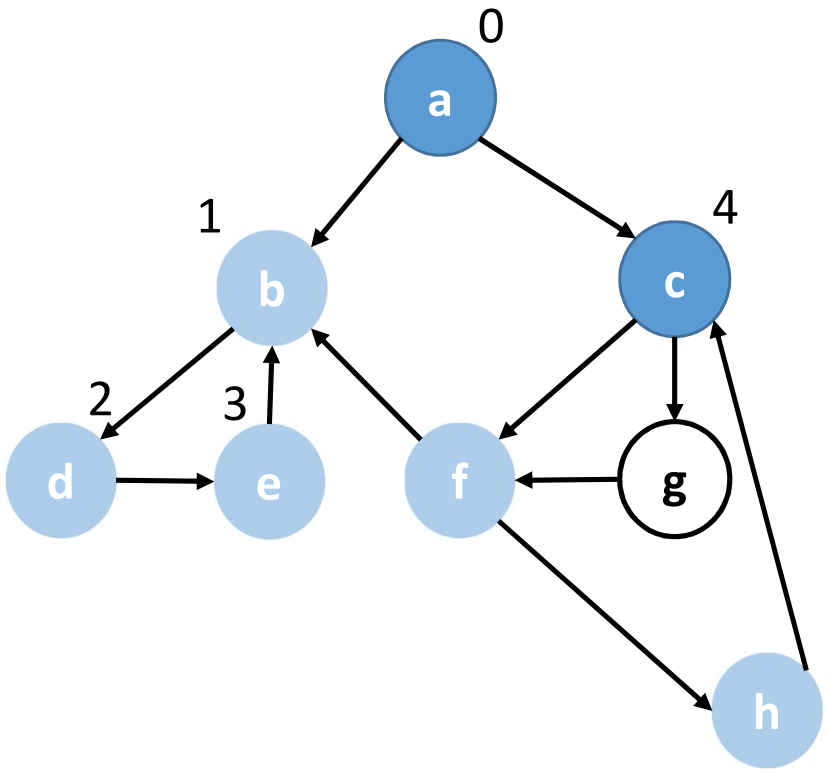
#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4	c	1
5	f	2
6		
7		

DFS



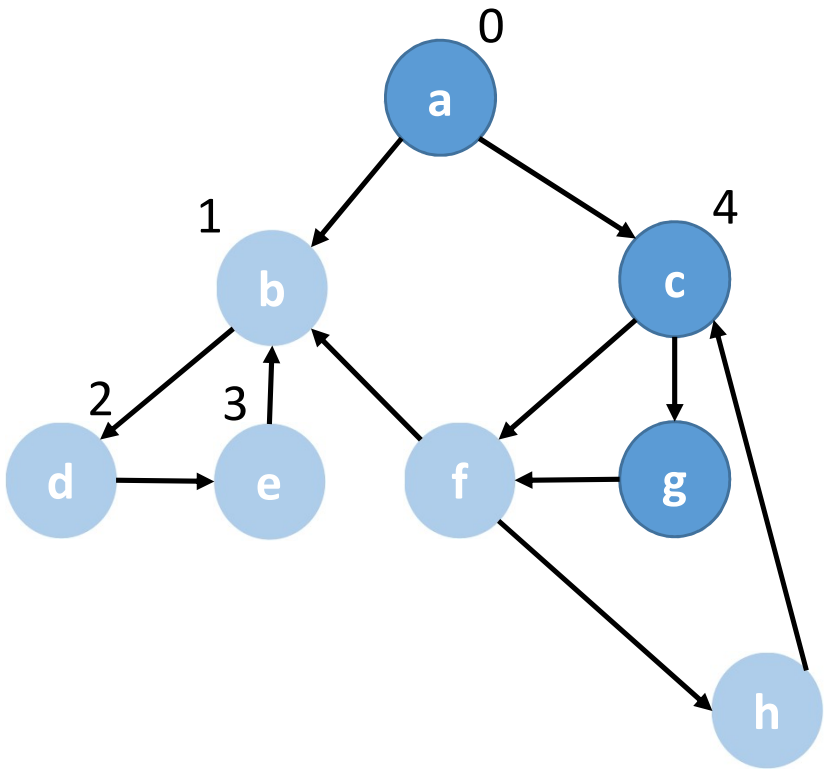
#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4	c	1
5	f	2
6	h	3
7		

DFS



#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4	c	1
5	f	2
6	h	3
7		

DFS



#	id	d
0	a	0
1	b	1
2	d	2
3	e	3
4	c	1
5	f	2
6	h	3
7	g	

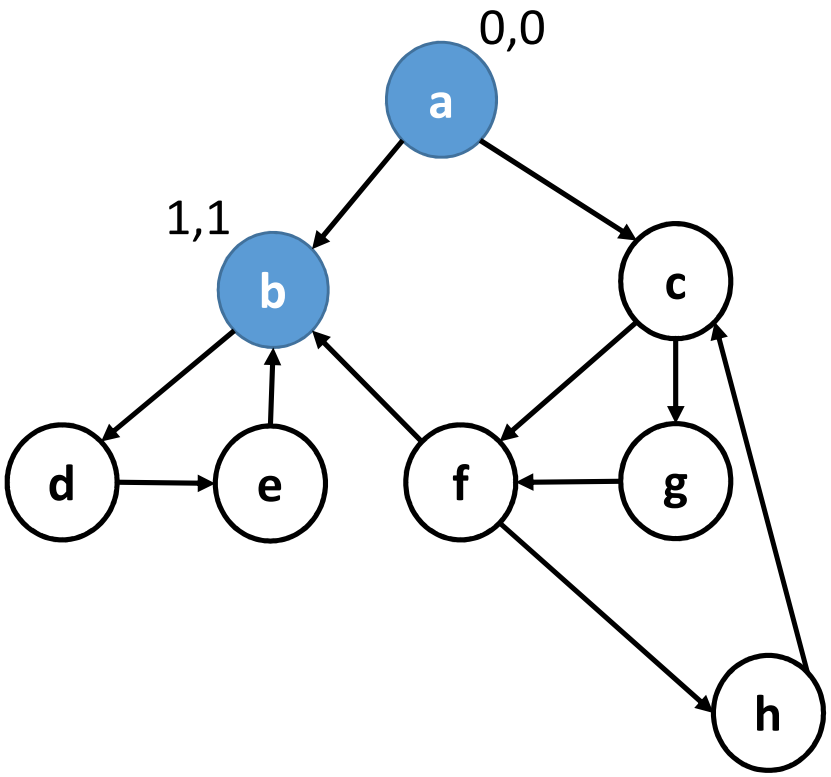
Relation between DFS and SCC

- There are two cases:
  (1) either we have no successors (we go back)
  (2) or we encounter nodes that have already been visited

- If the neighbor node has been visited:
  (2a) and was “popped” before, then it is in another SCC
  (2b) otherwise, we have a SCC

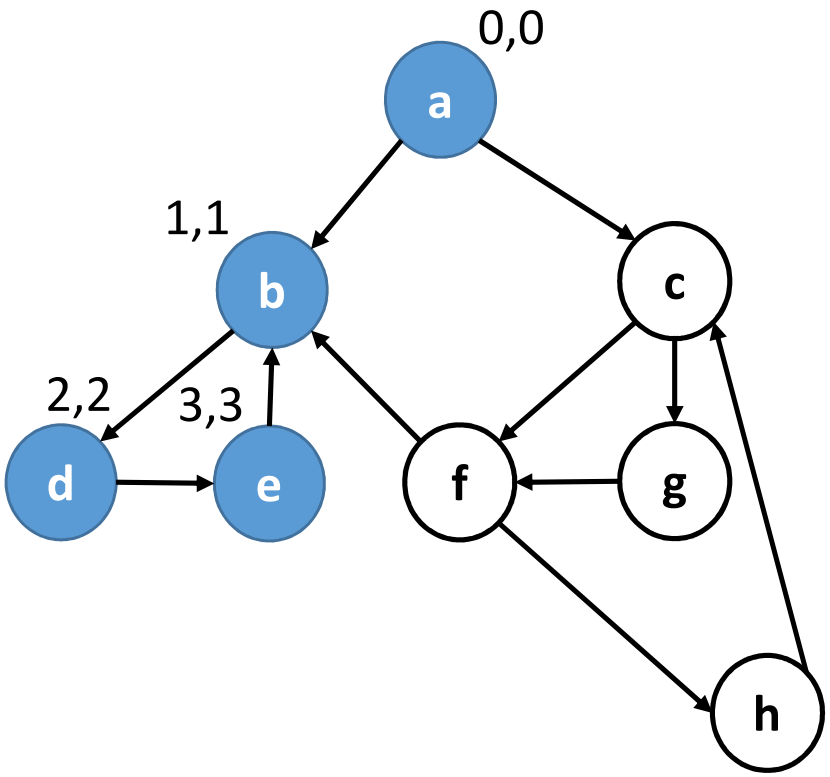
- The secret is to remember the “lowest” id of the current node explored

Tarjan's Algorithm



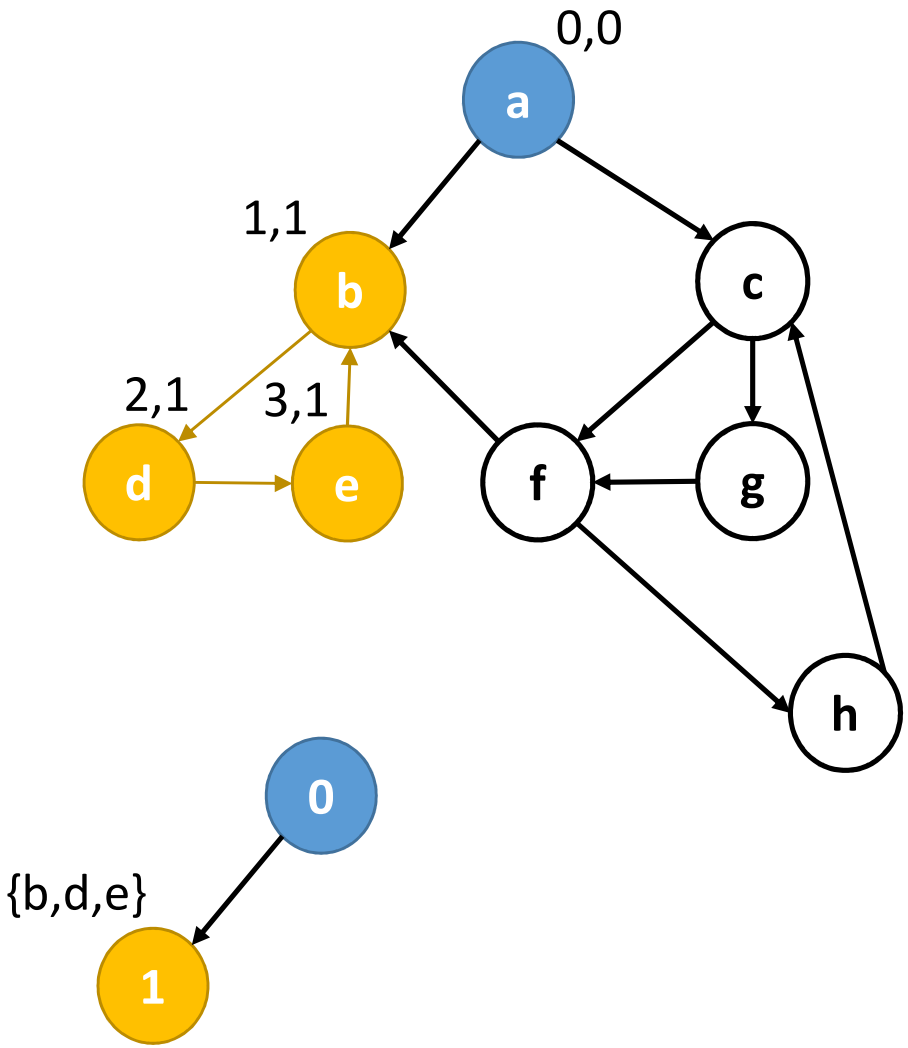
#	id	lw
0	a	0
1	b	1
2		
3		
4		
5		
6		
7		

Tarjan's Algorithm



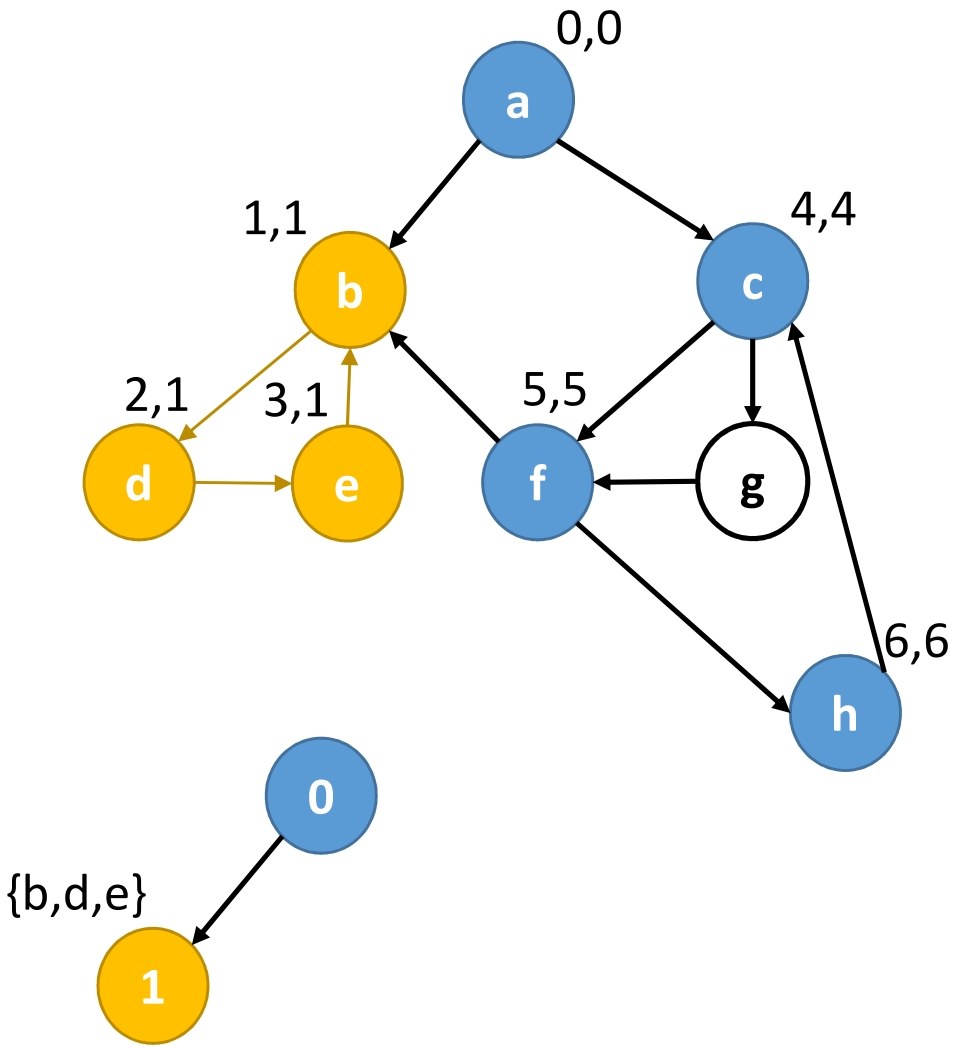
#	id	lw
0	a	0
1	b	1
2	d	2
3	e	3
4		
5		
6		
7		

Tarjan's Algorithm



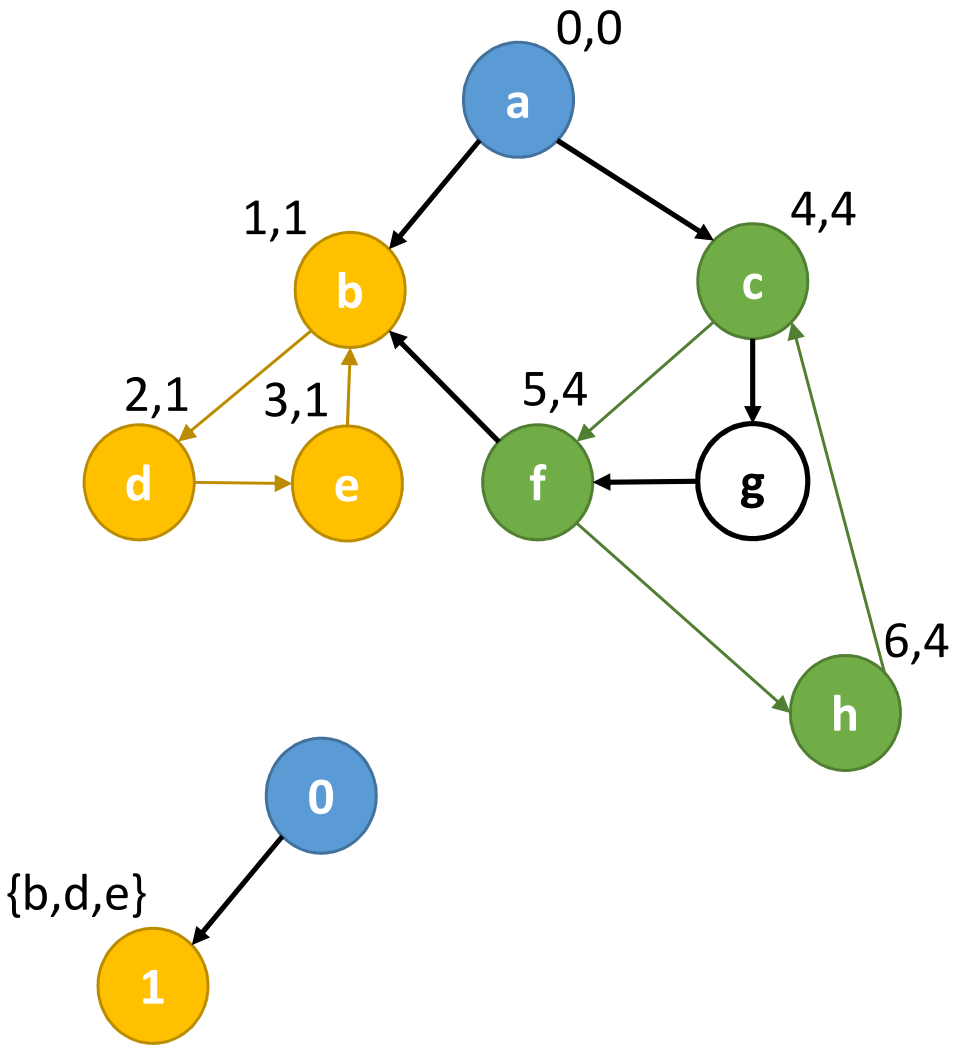
#	id	lw
0	a	0
1	b	1
2	d	1
3	e	1
4		
5		
6		
7		

Tarjan's Algorithm



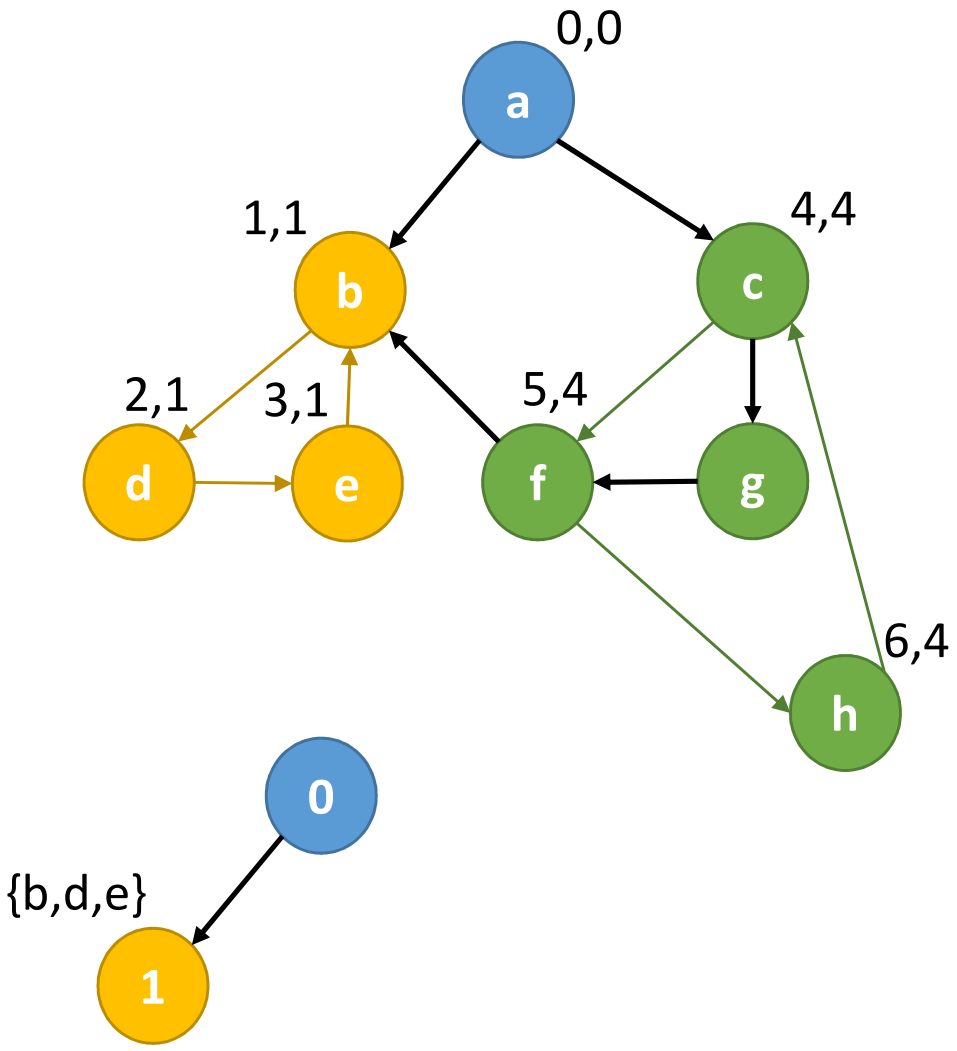
#	id	lw
0	a	0
1	b	1
2	d	1
3	e	1
4	c	4
5	f	5
6	h	6
7		

Tarjan's Algorithm



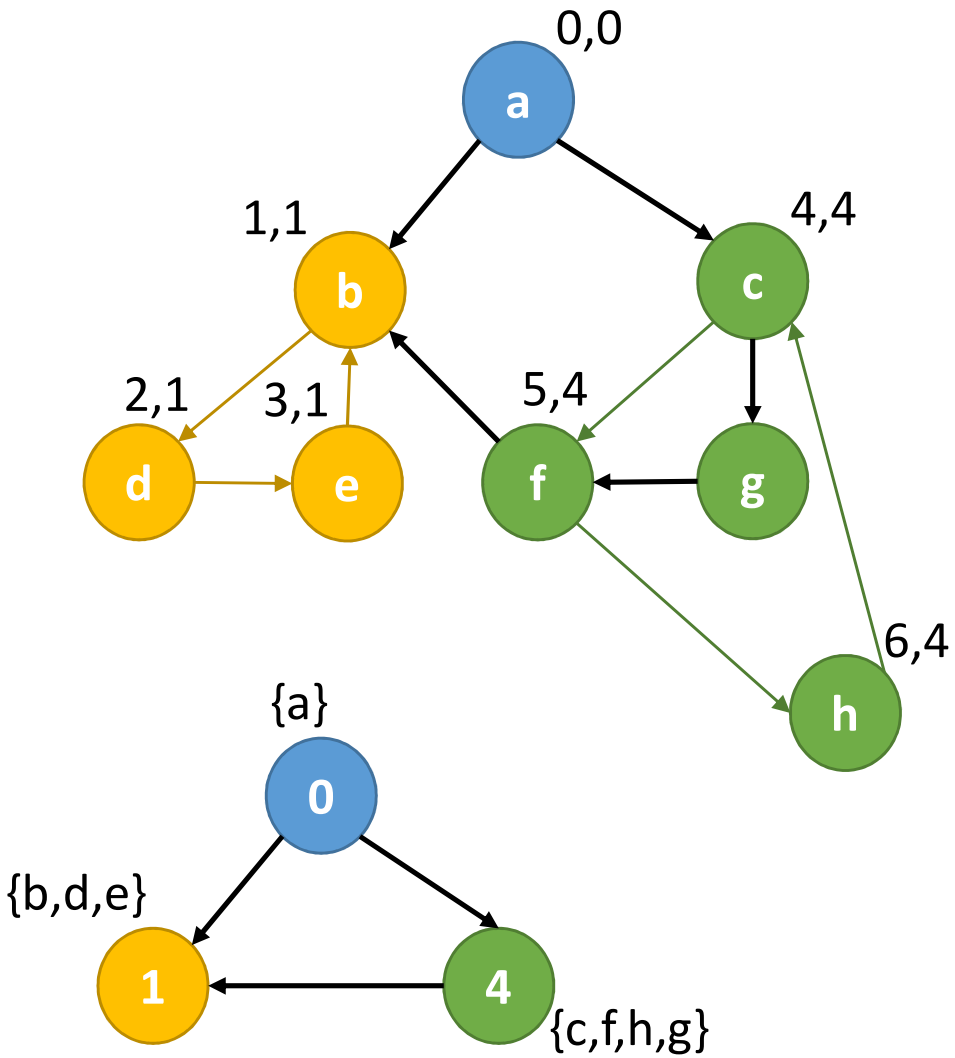
#	id	lw
0	a	0
1	b	1
2	d	1
3	e	1
4	c	4
5	f	4
6	h	4
7		

Tarjan's Algorithm



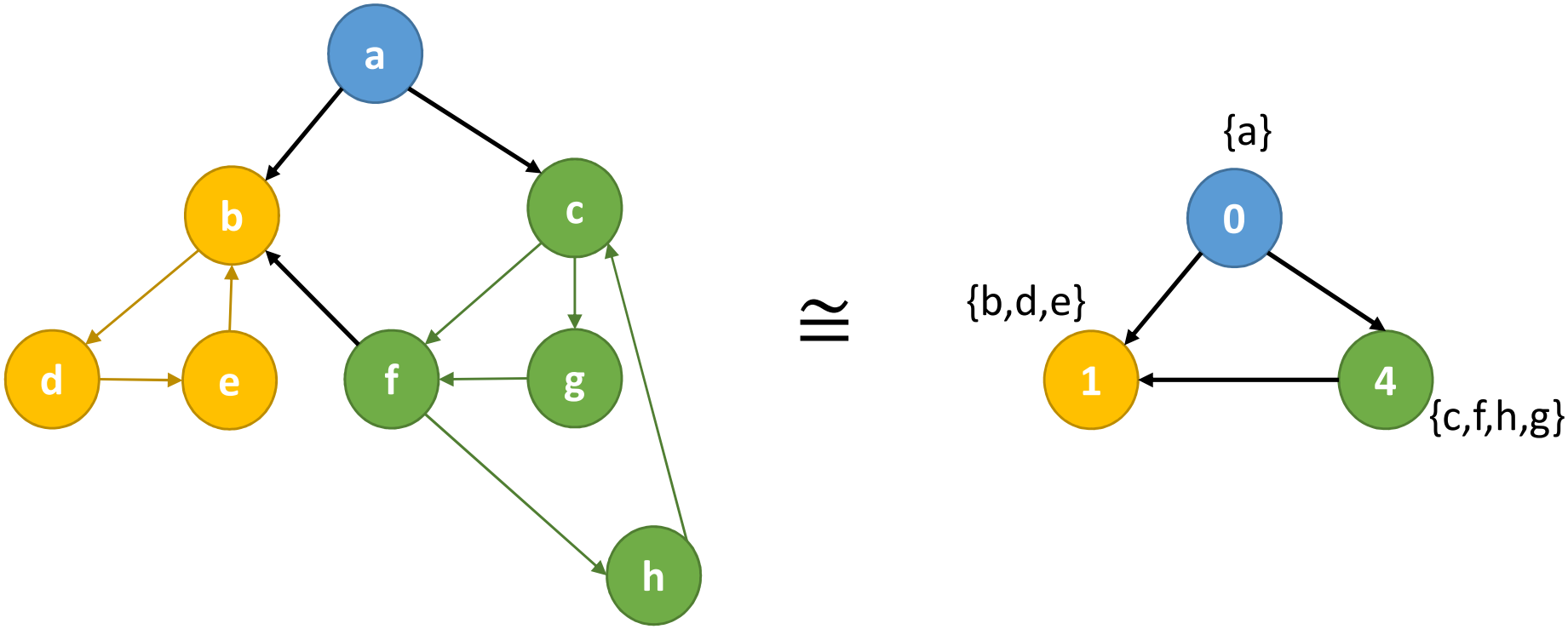
#	id	lw
0	a	0
1	b	1
2	d	1
3	e	1
4	c	4
5	f	4
6	h	4
7	g	7

Tarjan's Algorithm



#	id	lw
0	a	0
1	b	1
2	d	1
3	e	1
4	c	4
5	f	4
6	h	4
7	g	7

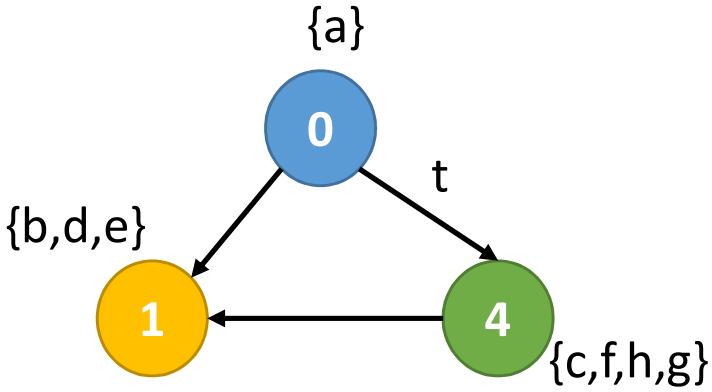
Tarjan's Algorithm



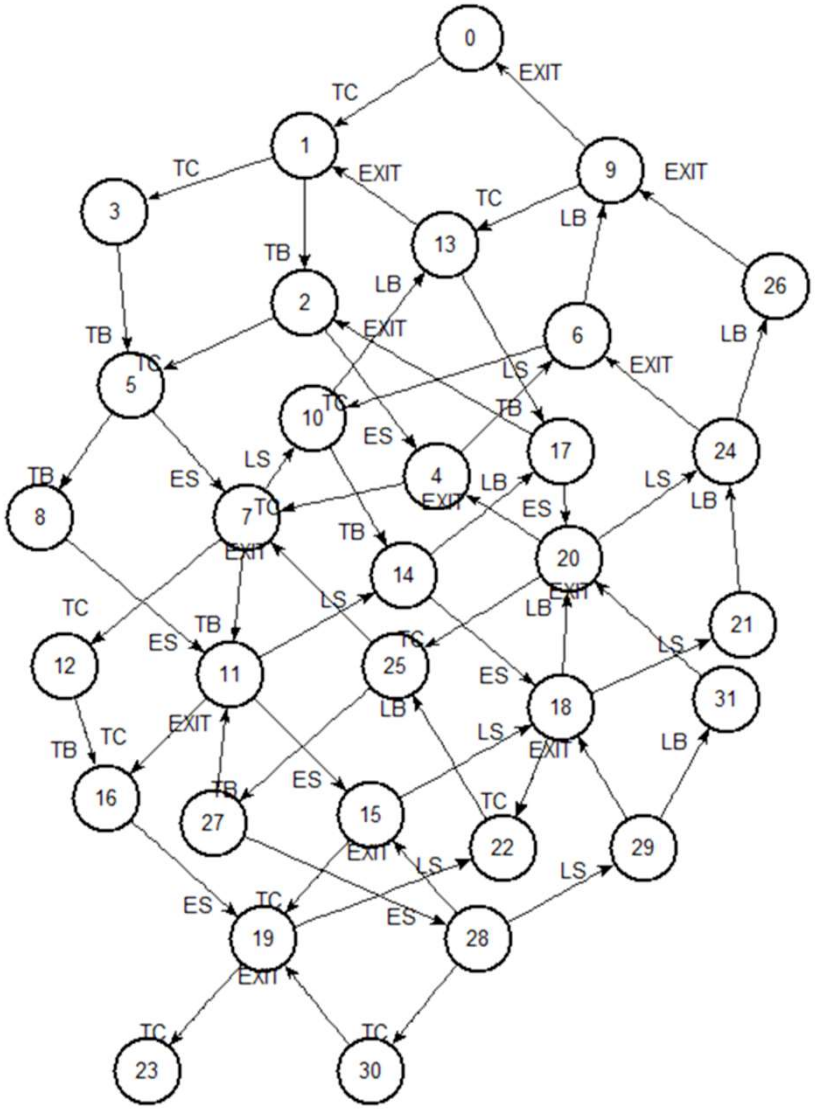
Checking Properties using SCC

- The SCC graph is (often) much more compact than the set of reachable states

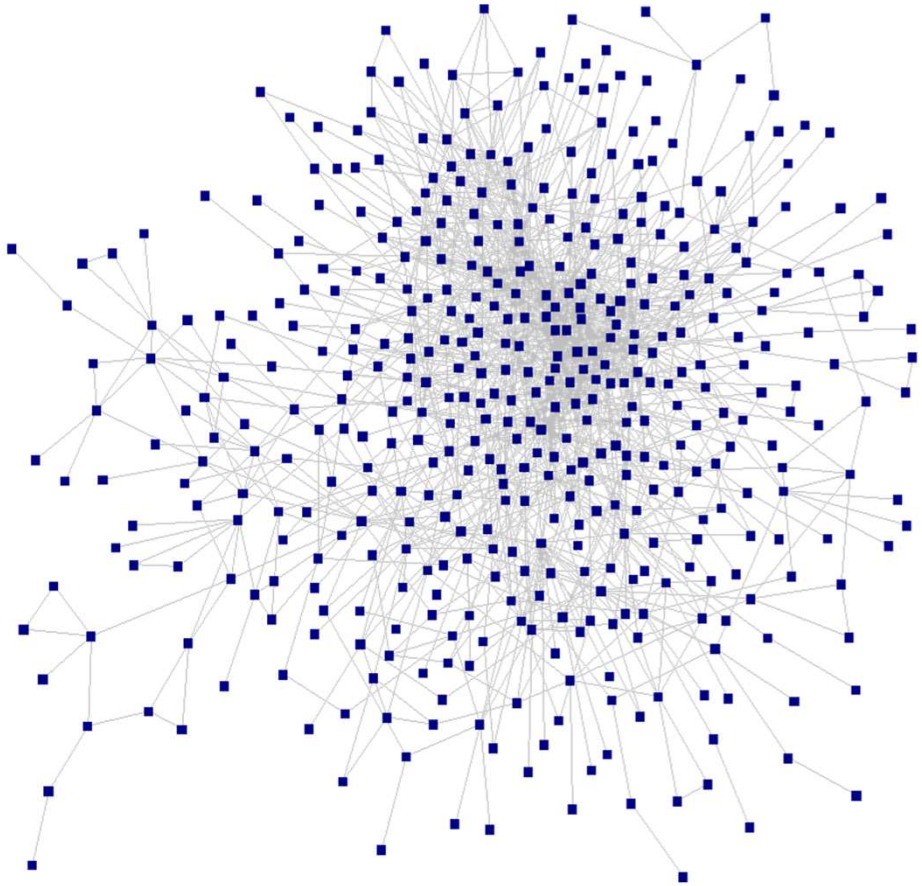
- We can use the SCC graph to check properties
  - my system is reinitializable ?
  - transition t is live ?
  - may I always avoid a deadlock ?



Checking Properties using SCC



efficient algorithm are necessary when dealing with big graphs



Tarjan's algorithm

1. Pick a starting node a (size |V|)
2. Each node is assigned its DFS number, $order(a)$
3. We also manage a stack of nodes, $link(a)$ (size |V|)
   track the "least id" among the vertex links
4. Visit the (leftmost, new) neighbor b (time |V| + |E|)
   if new then $link(b) = order(b) = n$
   else if b in stack then $link(a) = \min(link(a), link(b))$
5. If $order(a) = link(a)$ then create new SCC by popping nodes from the stack until we find a

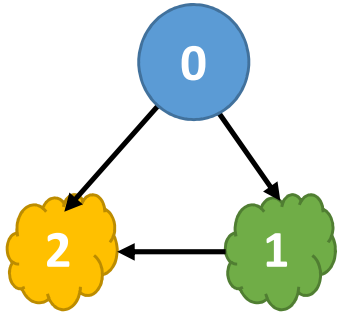
Checking Properties using SCC

- *boundedness* (is it finite)
- *deadlocks / liveness / quasi-liveness* (can something good happen)
- *(in)dependence of actions* (does R1 forbids R2)
- *reachability* (is it possible to have $A = 10$)

Checking Properties using SCC

- ~~*boundedness* (is it finite)~~
- ~~*reachability* (is it possible to have $A = 10$)~~
- ~~*(in)dependence of actions* (does R1 forbids R2)~~

- *deadlocks* ⇔ pending SCC with a single state



- *liveness* ⇔ all “transitions” are in every SCC
- *possibly always active* (∃ a SCC with two states)
- *reinitializable* ⇔ only one SCC