Model Checking (IN2050)

Summer 2010

Andrey Rybalchenko

TUM

Slides courtesy of Stefan Schwoon, ENS Cachan

Part 3: Linear-time logic

Preliminaries

Linear-time logic in general:

- any logic working with sequences of valuations
- model: time progresses in discrete steps and in linear fashion, each point in time has exactly one possible future
- origins in philosophy/logic

Most prominent species: LTL

- in use for verification since end of the 1970s
- spezification of correctness properties

Recap

Let AP be a set of atomic propositions.

2^{AP} denotes the powerset of AP, i.e. its set of subsets.

$(2^{AP})^{\omega}$ denotes the set of (infinite) sequences of valuations (of AP).

Syntax of LTL

Let *AP* be a set of atomic propositions. The set of LTL formulae over *AP* is inductively defined as follows:

If $p \in AP$ then p is a formula.

If ϕ_1, ϕ_2 are formulae then so are

$$\neg\phi_1, \qquad \phi_1 \vee \phi_2, \qquad \mathbf{X}\,\phi_1, \qquad \phi_1\,\mathbf{U}\,\phi_2$$

Intuitive meaning: $\mathbf{X} \mathrel{\widehat{=}}$ “next”, $\mathbf{U} \mathrel{\widehat{=}}$ “until”.

Remarks

This is a minimal syntax that we will use for proofs etc.

For added expressiveness, we will later define some abbreviations based on the minimal syntax.

Comparision of propositional logic (PL) and LTL:

	PL	LTL
Syntax	atomic proposition, logic operators	+ temporal operators
Evaluated on…	valuations	sequences of valuations
Semantics	set of valuations	set of valuation sequences

Semantics of LTL

Let ϕ be an LTL formula and σ a valuation sequence.
We write $\sigma \models \phi$ for "σ satisfies ϕ."

$$
\begin{array}{lll}
\sigma \models & p & \text{if } p \in AP \text{ and } p \in \sigma(0) \\
\sigma \models & \neg\phi & \text{if } \sigma \not\models \phi \\
\sigma \models & \phi_1 \vee \phi_2 & \text{if } \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models & \mathbf{X}\,\phi & \text{if } \sigma^1 \models \phi \\
\sigma \models & \phi_1 \,\mathbf{U}\, \phi_2 & \text{if } \exists i\colon \left(\sigma^i \models \phi_2 \ \wedge\ \forall k < i\colon \sigma^k \models \phi_1\right)
\end{array}
$$

Semantics of ϕ: $[\![\phi]\!] = \{\, \sigma \mid \sigma \models \phi \,\}$

Examples

Let $AP = \{p, q, r\}$. Find out whether the sequence

$$\sigma = \{p\}\ \{q\}\ \{p\}^\omega$$

satisfies the following formulae:

$$p$$

$$q$$

$$\mathbf{X}\, q$$

$$\mathbf{X}\, \neg p$$

$$p\ \mathbf{U}\ q$$

$$q\ \mathbf{U}\ p$$

$$(p \vee q)\ \mathbf{U}\ r$$

Extended syntax

We will commonly use the following abbreviations:

$$
\begin{aligned}
\phi_1 \wedge \phi_2 &\equiv \neg(\neg\phi_1 \vee \neg\phi_2) & \mathbf{F}\,\phi &\equiv \mathbf{true}\,\mathbf{U}\,\phi \\
\phi_1 \rightarrow \phi_2 &\equiv \neg\phi_1 \vee \phi_2 & \mathbf{G}\,\phi &\equiv \neg\,\mathbf{F}\,\neg\phi \\
\mathbf{true} &\equiv a \vee \neg a & \phi_1\,\mathbf{W}\,\phi_2 &\equiv (\phi_1\,\mathbf{U}\,\phi_2) \vee \mathbf{G}\,\phi_1 \\
\mathbf{false} &\equiv \neg\mathbf{true} & \phi_1\,\mathbf{R}\,\phi_2 &\equiv \neg(\neg\phi_1\,\mathbf{U}\,\neg\phi_2)
\end{aligned}
$$

Meaning: $\mathbf{F} \mathrel{\widehat{=}}$ “finally” (eventually), $\mathbf{G} \mathrel{\widehat{=}}$ “globally” (always),
$\mathbf{W} \mathrel{\widehat{=}}$ “weak until”, $\mathbf{R} \mathrel{\widehat{=}}$ “release”.

Some example formulae

Invariant: $\mathbf{G}\,\neg(cs_1 \wedge cs_2)$

> cs_1 and cs_2 are never true at the same time.

Remark: This particular form of invariant is also called mutex property ("mutual exclusion").

Safety: $(\neg x)\ \mathbf{W}\ y$

> x does not occur before y has happend.

Remark: It may happen that y never happens in which case x also never happens.

Liveness: $(\neg x)\ \mathbf{U}\ y$

> x does not occur before y and y eventually happens.

More examples

$\mathbf{G}\,\mathbf{F}\,p$

p occurs infinitely often.

$\mathbf{F}\,\mathbf{G}\,p$

At some point *p* will continue to hold forever.

$\mathbf{G}(\mathit{try}_1 \rightarrow \mathbf{F}\,\mathit{cs}_1)$

For mutex algorithms: Whenever process 1 tries to enter its critical section it will eventually succeed.

Tautology, equivalence

Certain oncepts from propositional logic can be transferred to LTL.

Tautology: A formula ϕ with $[\![\phi]\!] = (2^{AP})^\omega$ is called tautology.

Unsatisfiability: A formula ϕ with $[\![\phi]\!] = \emptyset$ is called unsatisfiable.

Equivalence: Two formulae ϕ_1, ϕ_2 are called equivalent iff $[\![\phi_1]\!] = [\![\phi_2]\!]$.
Denotation: $\phi_1 \equiv \phi_2$

Equivalences: relations between operators

$$\begin{aligned}
\mathbf{X}(\phi_1 \vee \phi_2) &\equiv \mathbf{X}\,\phi_1 \;\vee\; \mathbf{X}\,\phi_2 \\
\mathbf{X}(\phi_1 \wedge \phi_2) &\equiv \mathbf{X}\,\phi_1 \;\wedge\; \mathbf{X}\,\phi_2 \\
\mathbf{X}\,\neg\phi &\equiv \neg\,\mathbf{X}\,\phi \\[1ex]
\mathbf{F}(\phi_1 \vee \phi_2) &\equiv \mathbf{F}\,\phi_1 \;\vee\; \mathbf{F}\,\phi_2 \\
\neg\,\mathbf{F}\,\phi &\equiv \mathbf{G}\,\neg\phi \\[1ex]
\mathbf{G}(\phi_1 \wedge \phi_2) &\equiv \mathbf{G}\,\phi_1 \;\wedge\; \mathbf{G}\,\phi_2 \\
\neg\,\mathbf{G}\,\phi &\equiv \mathbf{F}\,\neg\phi \\[1ex]
(\phi_1 \wedge \phi_2)\,\mathbf{U}\,\psi &\equiv (\phi_1\,\mathbf{U}\,\psi) \;\wedge\; (\phi_2\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{U}\,(\psi_1 \vee \psi_2) &\equiv (\phi\,\mathbf{U}\,\psi_1) \;\vee\; (\phi_1\,\mathbf{U}\,\psi_2)
\end{aligned}$$

Equivalences: idempotence and recursion laws

$$\begin{aligned}
\mathbf{F}\,\phi &\equiv \mathbf{F}\,\mathbf{F}\,\phi \\
\mathbf{G}\,\phi &\equiv \mathbf{G}\,\mathbf{G}\,\phi \\
\phi\,\mathbf{U}\,\psi &\equiv \phi\,\mathbf{U}\,(\phi\,\mathbf{U}\,\psi)
\end{aligned}$$

$$\begin{aligned}
\mathbf{F}\,\phi &\equiv \phi \vee \mathbf{X}\,\mathbf{F}\,\phi \\
\mathbf{G}\,\phi &\equiv \phi \wedge \mathbf{X}\,\mathbf{G}\,\phi \\
\phi\,\mathbf{U}\,\psi &\equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{U}\,\psi)) \\
\phi\,\mathbf{W}\,\psi &\equiv \psi \vee (\phi \wedge \mathbf{X}(\phi\,\mathbf{W}\,\psi))
\end{aligned}$$

Interpretation of LTL on Kripke structures

Let $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ be a Kripke structure.
We are interested in the valuation sequences generated by $\mathcal{K}$.

Let ρ in S^ω be an infinite path of $\mathcal{K}$.

We assign to ρ an “image” $\nu(\rho)$ in $(2^{AP})^\omega$; for all $i \geq 0$ let

$$\nu(\rho)(i) = \nu(\rho(i))$$

i.e. $\nu(\rho)$ is the corresponding valuation sequence.

Let $[\![\mathcal{K}]\!]$ denote the set of all such sequences:

$$[\![\mathcal{K}]\!] = \{\, \nu(\rho) \mid \rho \text{ is an infinite path of } \mathcal{K} \,\}$$

The LTL model-checking problem

Problem: Given a Kripke structure $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ and an LTL formula ϕ over AP, does $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ hold?

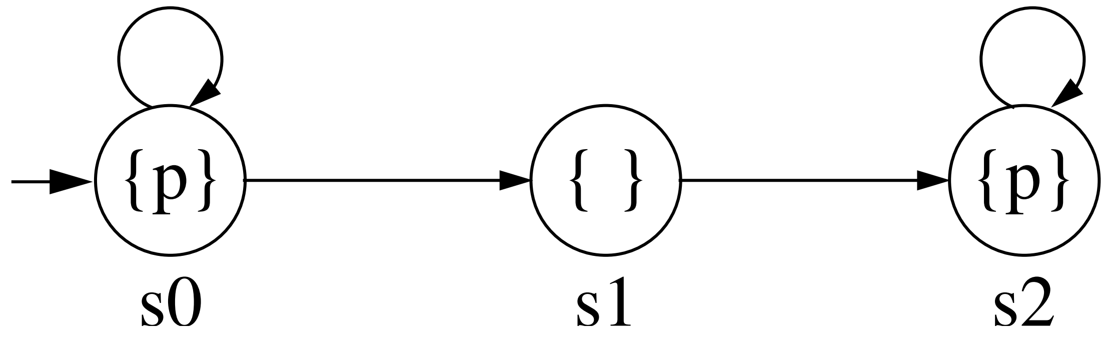
Definition: If $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ then we write $\mathcal{K} \models \phi$.

Interpretation: Every execution of $\mathcal{K}$ must satisfy ϕ for $\mathcal{K} \models \phi$ to hold.

Remark: We may have $\mathcal{K} \not\models \phi$ and $\mathcal{K} \not\models \neg\phi$!

Example

Consider the following Kripke structure $\mathcal{K}$ with $AP = \{p\}$:



There are two classes of infinite paths in $\mathcal{K}$:

(i) Either the system stays in s_0 forever,

(ii) or it eventually reaches s_2 via s_1 and remains there.

We have:

$\mathcal{K} \models \mathbf{F}\,\mathbf{G}\,p$ because all runs eventually end in a state satisfying p.

$\mathcal{K} \not\models \mathbf{G}\,p$ because executions of type (ii) contain a non-p state.

Dealing with deadlocks

The definition of the model-checking problem only considers the infinite sequences!

Thus, executions reaching a deadlock (i.e. a state without any successor) will be ignored, with possibly unforeseen consequences:

> Suppose $\mathcal{K}$ contains an error, so that every execution reaches a deadlock.
>
> Then $[\![\mathcal{K}]\!] = \emptyset$, so $\mathcal{K}$ satisfies *every* formula, according to the definition.

Possible alternatives

Remove deadlocks by design:

equip deadlock states with a self loop

Interpretation: system stays in deadlock forever

adapt formula accordingly, if necessary

Treat deadlocks specially:

Check for deadlocks before LTL model checking, deal with them separately.

Part 4: Büchi automata

Preview

Model-checking problem: $[\![\mathcal{K}]\!] \subseteq [\![\phi]\!]$ – how can we check this algorithmically?

(Historically) first approach: Translate $\mathcal{K}$ into an LTL formula $\psi_{\mathcal{K}}$, check whether $\psi_{\mathcal{K}} \rightarrow \phi$ is a tautology. Problem: very inefficient.

Language-/automata-theoretic approach: $[\![\mathcal{K}]\!]$ and $[\![\phi]\!]$ are languages (of infinite words).

Find a suitable class of automata for representing these languages.

Define suitable operations on these automata for solving the problem.

This is the approach we shall follow.

Büchi automata

A Büchi automaton is a tuple

$$\mathcal{B} = (\Sigma, S, s_0, \Delta, F),$$

where:

Σ	is a finite alphabet;
S	is a finite set of states;
$s_0 \in S$	is an initial state;
$\Delta \subseteq S \times \Sigma \times S$	are transitions;
$F \subseteq S$	are accepting states.

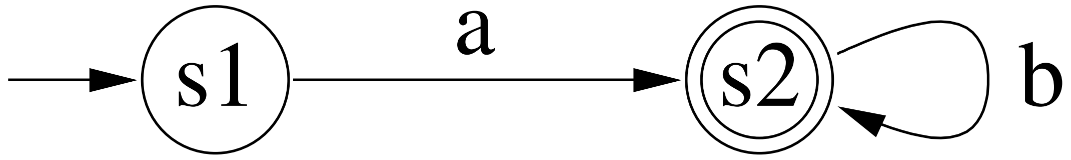
Remarks:

Definition and graphical representation like for finite automata.

However, Büchi automata are supposed to work on *infinite* words, requiring a different acceptance condition.

Example

Graphical representation of a Büchi automaton:



The components of this automaton are $(\Sigma, S, s_1, \Delta, F)$, where:

- $\Sigma = \{a, b\}$ (symbols on the edges)
- $S = \{s_1, s_2\}$ (circles)
- s_1 (indicated by arrow)
- $\Delta = \{(s_1, a, s_2), (s_2, b, s_2)\}$ (edges)
- $F = \{s_2\}$ (with double circle)

Language of a Büchi automaton

Let $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ be a Büchi automaton.

A run of $\mathcal{B}$ over an infinite word $\sigma \in \Sigma^\omega$ is an infinite sequences of states $\rho \in S^\omega$ where $\rho(0) = s_0$ and $(\rho(i), \sigma(i), \rho(i+1)) \in \Delta$ for $i \geq 0$.

We call ρ accepting iff $\rho(i) \in F$ for infinitely many values of i.

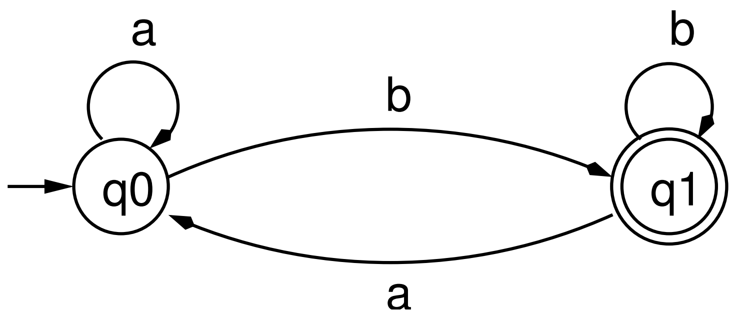
> I.e., ρ infinitely often visits accepting states.
> (By the pigeon-hole principle: at least one accepting state is visited infinitely often.)

$\sigma \in \Sigma^\omega$ is accepted by $\mathcal{B}$ iff there exists an accepting run over σ in $\mathcal{B}$.

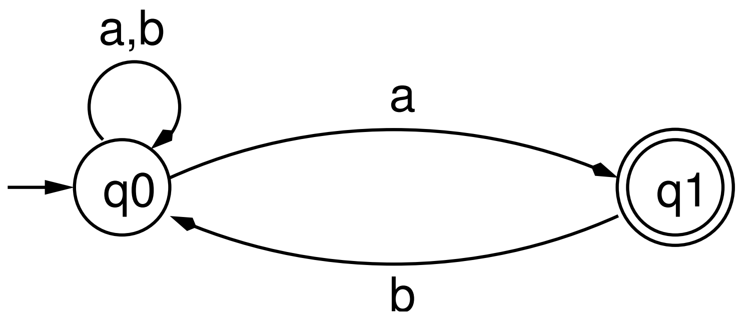
The language of $\mathcal{B}$, denoted $\mathcal{L}(\mathcal{B})$, is the set of all words accepted by $\mathcal{B}$.

Büchi automata: examples

"infinitely often b"



"infinitely often ab"



Büchi automata and LTL

Let AP be a set of atomic propositions.

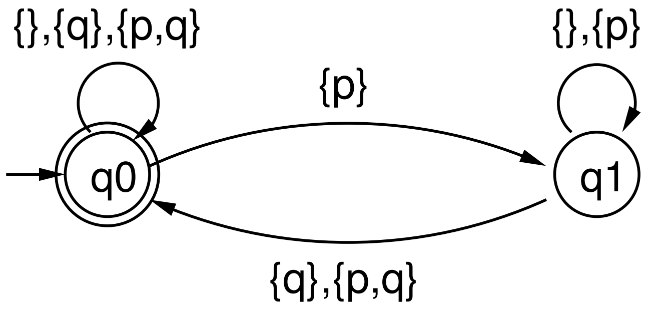
A Büchi automaton with alphabet 2^{AP} accepts a sequence of valuations.

Claim: For every LTL formula ϕ there exists a Büchi automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$.

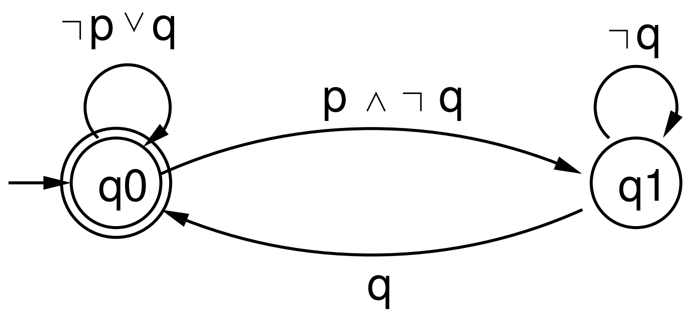
(We shall prove this claim later.)

Examples: $\mathbf{F}\, p$, $\mathbf{G}\, p$, $\mathbf{G}\,\mathbf{F}\, p$, $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, $\mathbf{F}\,\mathbf{G}\, p$

Example automaton for $\mathbf{G}(p \to \mathbf{F}\, q)$, with $AP = \{p, q\}$.



Alternatively we can label edges with formulae of propositional logic; in this case, a formula F stands for all elements of $[\![F]\!]$. In this case:



Operations on Büchi automata

The languages accepted by Büchi automata are also callled ω-regular languages.

Like the usual regular languages, ω-regular languages are also closed under Boolean operations.

I.e., if $\mathcal{L}_1$ and $\mathcal{L}_2$ are ω-regular, then so are

$$\mathcal{L}_1 \cup \mathcal{L}_2, \qquad \mathcal{L}_1 \cap \mathcal{L}_2, \qquad \mathcal{L}_1^c.$$

We shall now define operations that take Büchi automata accepting some languages $\mathcal{L}_1$ and $\mathcal{L}_2$ and produce automata for their union or intersection.

In the following slides we assume $\mathcal{B}_1 = (\Sigma, S, s_0, \Delta_1, F)$ and $\mathcal{B}_2 = (\Sigma, T, t_0, \Delta_2, G)$ (with $S \cap T = \emptyset$).

Union

“Juxtapose” $\mathcal{B}_1$ and $\mathcal{B}_2$ and add a new initial state.

In other words, the automaton $\mathcal{B} = (\Sigma,\ S \cup T \cup \{u\},\ u,\ \Delta_1 \cup \Delta_2 \cup \Delta_u,\ F \cup G)$ accepts $\mathcal{L}(\mathcal{B}_1) \cup \mathcal{L}(\mathcal{B}_2)$, where

- u is a “fresh” state ($u \notin S \cup T$);
- $\Delta_u = \{\, (u, \sigma, s) \mid (s_0, \sigma, s) \in \Delta_1 \,\} \cup \{\, (u, \sigma, t) \mid (t_0, \sigma, t) \in \Delta_2 \,\}$.

Intersection I (a special case)

We first consider the case where all states in $\mathcal{B}_2$ are accepting, i.e. $G = T$.

Idea: Construct a cross-product automaton (like for FA), check whether F is visited infinitely often.

Let $\mathcal{B} = (\Sigma,\ S \times T,\ \langle s_0, t_0\rangle,\ \Delta,\ F \times T)$, where

$$\Delta = \{\, (\langle s,t\rangle, a, \langle s',t'\rangle) \mid a \in \Sigma,\ (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2 \,\}.$$

Then: $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$.

Intersection II (the general case)

Principle: We again construct a cross-product automaton.

Problem: The acceptance condition needs to check whether *both* accepting sets are visited infinitely often.

Idea: create two copies of the cross product.

– In the first copy we wait for a state from F.

– In the second copy we wait for a state from G.

– In both copies, once we've found one of the states we're looking for, we switch to the other copy.

We will choose the acceptance condition in such a sway that an accepting run switches back and forth between the copies infinitely often.

Let $\mathcal{B} = (\Sigma, U, u, \Delta, H)$, where

$$U = S \times T \times \{1,2\}, \quad u = \langle s_0, t_0, 1 \rangle, \quad H = F \times T \times \{1\}$$

$$\begin{array}{lll}
(\langle s,t,1\rangle, a, \langle s',t',1\rangle) \in \Delta & \text{iff} & (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \notin F \\
(\langle s,t,1\rangle, a, \langle s',t',2\rangle) \in \Delta & \text{iff} & (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ s \in F \\
(\langle s,t,2\rangle, a, \langle s',t',2\rangle) \in \Delta & \text{iff} & (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \notin G \\
(\langle s,t,2\rangle, a, \langle s',t',1\rangle) \in \Delta & \text{iff} & (s,a,s') \in \Delta_1,\ (t,a,t') \in \Delta_2,\ t \in G
\end{array}$$

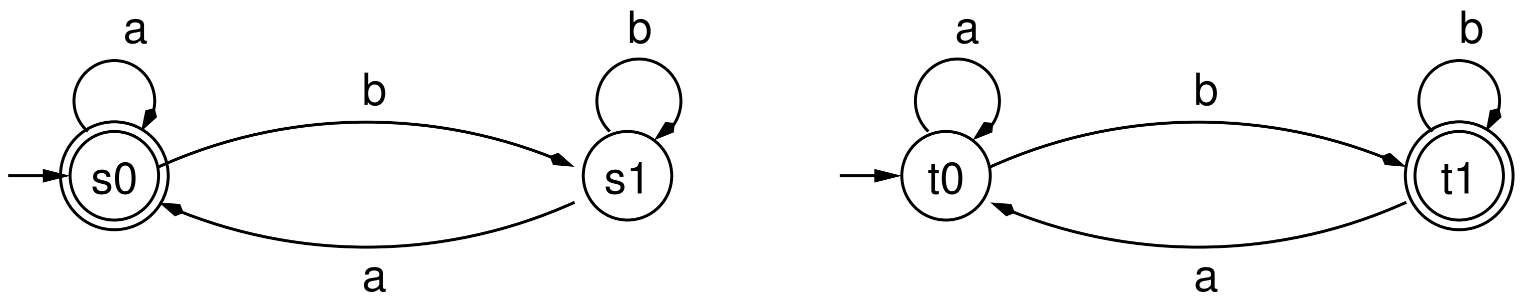
Remarks:

The automaton starts in the first copy.

We could have chosen other acceptance conditions such as $S \times G \times \{2\}$.

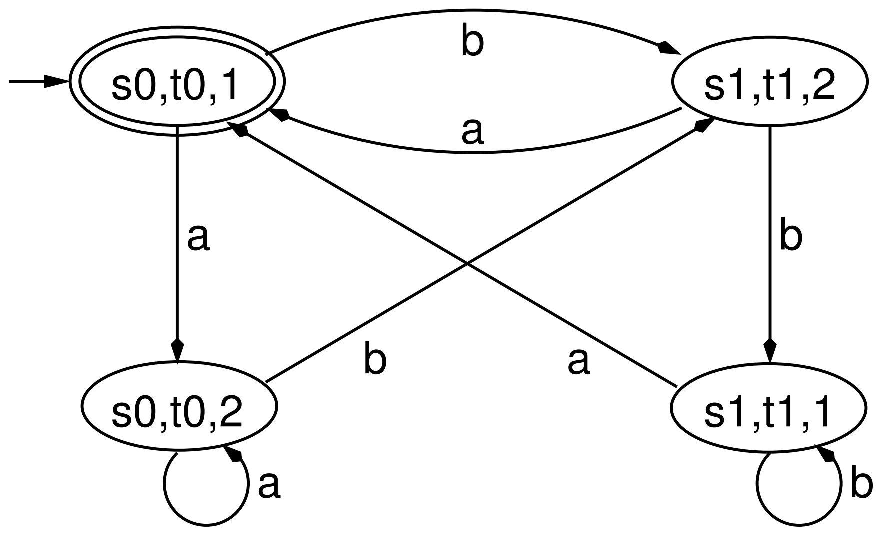
The construction can be generalized to intersecting n automata.

Intersection: example



B1 B2

B1 x B2



Complement

Problem: Given $\mathcal{B}_1$, construct $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}_1)^c$.

Such a construction is possible (but rather complicated). We will not require it for the purpose of this course.

Additional literature:

Wolfgang Thomas, *Automata on Infinite Objects*,
Chapter 4 in *Handbook of Theoretical Computer Science*,

Igor Walukiewicz, lecture notes on *Automata and Logic*, chapter 3,
`www.labri.fr/Perso/~igw/Papers/igw-eefss01.ps`

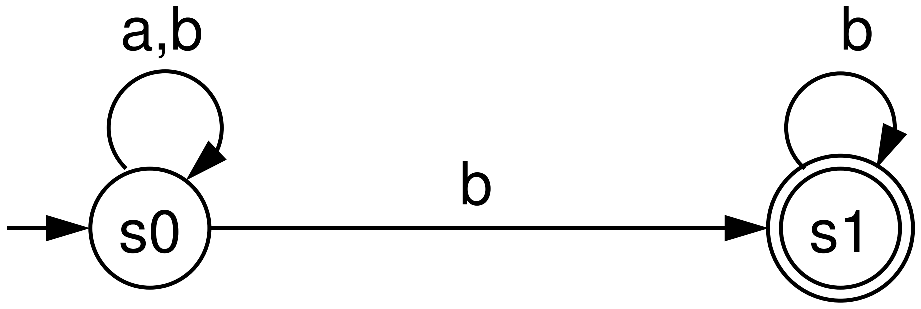
Deterministic Büchi automata

For finite automata (known from *regular language theory*), it is known that every language expressible by a finite automaton can also be expressed by a deterministic automaton, i.e. one where the transition relation Δ is a function $S \times \Sigma \to S$.

Such a procedure does not exist for Büchi automata.

In fact, there is no *deterministic* Büchi automaton accepting the same language as the automaton below:

"Only finitely many as."



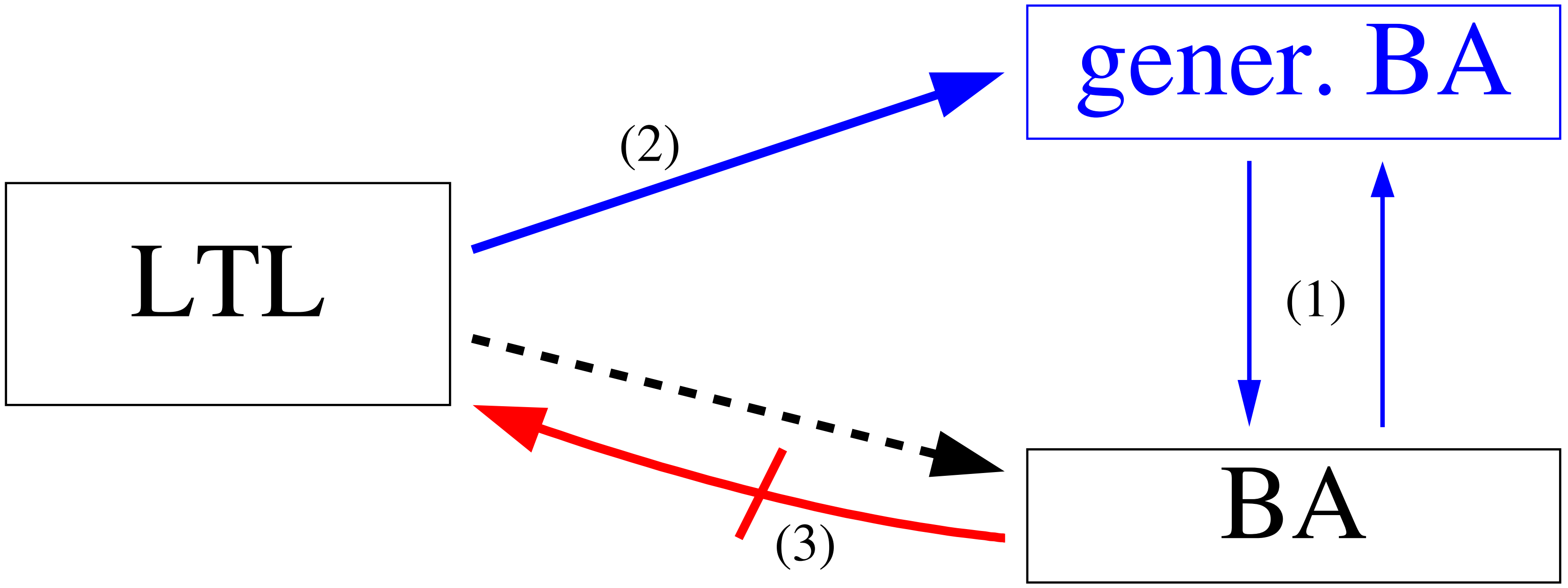
Proof: Let $\mathcal{L}$ be the language of infinite words over $\{a, b\}$ containing only finitely many as. Assume that a deterministic Büchi automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = \mathcal{L}$ exists, and let n be the number of states in $\mathcal{B}$.

We have $b^\omega \in \mathcal{L}$, so let α_1 be the (unique) accepting run for b^ω. Suppose that an accepting state is first reached after n_1 letters, i.e. $s_1 := \alpha_1(n_1)$ is the first accepting state in α_1.

We now regard the word $b^{n_1}ab^\omega$, which is still in $\mathcal{L}$, therefore accepted by some run α_2. Since $\mathcal{B}$ is deterministic, α_1 and α_2 must agree on the first n_1 states. Now, watch for the second occurrence of an accepting state in α_2, i.e. let $s_2 := \alpha_2(n_1 + 1 + n_2)$ be an accepting state for n_2 minimal. Then, $s_1 \neq s_2$ because otherwise there would be a loop around an accepting state containing a transition with an a.

We now repeat the argument for $b^{n_1}ab^{n_2}ab^\omega$, derive the existence of a third distinct state, etc. After doing this $n + 1$ times, we conclude that $\mathcal{B}$ must have more than n distinct states, a contradiction.

Preview



We shall proceed in the order indicated above.

Generalized Büchi automata

A generalized Büchi automaton (GBA) is a tuple $\mathcal{G} = (\Sigma, S, s_0, \Delta, \mathcal{F})$.

There is only one difference w.r.t. normal BA:

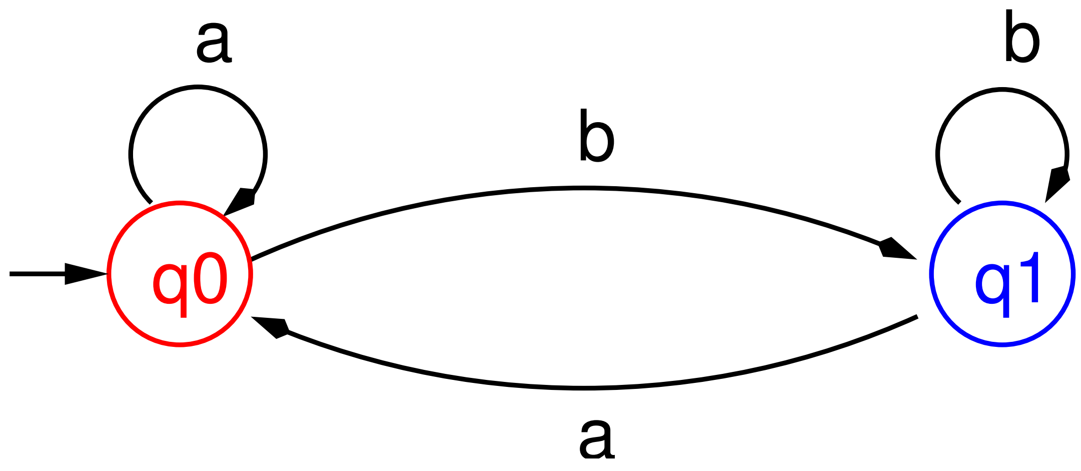
The acceptance condition $\mathcal{F} \subseteq 2^S$ is a *set of sets of states*.

E.g., let $\mathcal{F} = \{F_1, \ldots, F_n\}$. A run ρ of $\mathcal{G}$ is called accepting iff for every F_i ($i = 1, \ldots, n$), ρ visits infinitely many states of F_i.

Put differently: many acceptance conditions at once.

GBA: Example

For the GBA shown below, let $\mathcal{F} = \{ \{q_0\}, \{q_1\} \}$.



Language of the automaton: “infinitely often *a and* infinitely often *b*”

Note: In general, the acceptance conditions need not be pairwise disjoint.

Advantage: GBA may be more succinct than BA.

Translations BA $\leftrightarrow$ GBA

GBA accept the same class of languages as BA.

I.e., for every BA there is a GBA accepting the same language, and vice versa.

Part 1 of the claim (BA $\rightarrow$ GBA):

Let $\mathcal{B} = (\Sigma, S, s_0, \Delta, F)$ be a BA.

Then $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F\})$ is a GBA with $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{B})$.

Part 2 of the claim (GBA $\rightarrow$ BA):

Let $\mathcal{G} = (\Sigma, S, s_0, \Delta, \{F_1, \ldots, F_n\})$ be a GBA.

We construct $\mathcal{B} = (\Sigma, S', s_0', \Delta', F)$ as follows:

$S' = S \times \{1, \ldots, n\}$

$s_0' = (s_0, 1)$

$F = F_1 \times \{1\}$

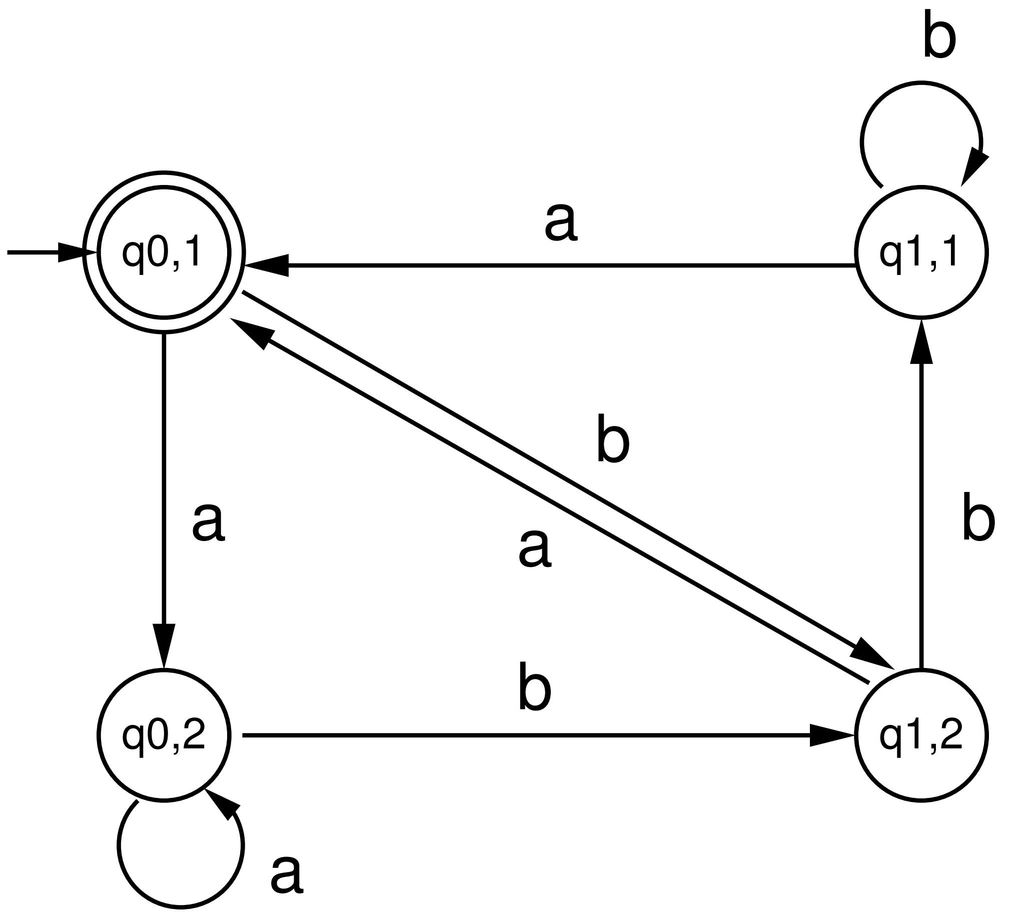
$((s, i), a, (s', k)) \in \Delta'$ iff $1 \leq i \leq n,\ (s, a, s') \in \Delta$

and $k = \begin{cases} i & \text{if } s \notin F_i \\ (i \bmod n) + 1 & \text{if } s \in F_i \end{cases}$

Then we have $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{G})$. (Idea: *n*-fold intersection)

GBA $\rightarrow$ BA: example

The BA corresponding to the previous GBA (“infinitely often *a and* infinitely often *b*”) is as follows:



Remark: Multiple initial states

Our definitions of BA and GBA require exactly one initial state.

For the translation LTL $\rightarrow$ BA it will be convenient to use GBA with multiple initial states.

> Intended meaning: A word is regarded as accepted if it is accepted starting from *any* initial state.

Obviously, every (G)BA with multiple initial states can easily be converted into a (G)BA with just one initial state.

Part 5: LTL and Büchi automata

Overview

In this part, we shall solve the following problem:

Given an LTL formula ϕ over *AP*, we shall construct a GBA $\mathcal{G}$ (with multiple initial states) such that $\mathcal{L}(\mathcal{G}) = [\![\phi]\!]$.

($\mathcal{G}$ can then be converted to a normal BA.)

Remarks:

Analogous operation for regular languages: reg. expression $\rightarrow$ NFA

The crucial difference: it is not possible to provide an LTL $\rightarrow$ BA translation in modular fashion.

The automaton may have to check multiple subformulae *at the same time* (e.g.: $(\mathbf{G}\,\mathbf{F}\,p) \rightarrow (\mathbf{G}(q \rightarrow \mathbf{F}\,r))$ or $(p\,\mathbf{U}\,q)\,\mathbf{U}\,r$).

More remarks:

The construction shown in the following is comparatively simplistic.

It will produce rather suboptimal automata (size *always* exponential in $|\phi|$).

Obviously, this is quite inefficient, and not meant to be done by pen and paper, only as a “proof of concept”.

There are far better translation procedures but the underlying theory is rather beyond the scope of the course.

Interesting, on-going research area!

Structure of the construction

1. We first convert ϕ into a certain normal form.

2. States will be “responsible” for some set of subformulae.

3. The transition relation will ensure that “simple” subformulae such as p or $\mathbf{X}\, p$ are satisfied.

4. The acceptance condition will ensure that $\mathbf{U}$-subformulae are satisfied.

Negation normal form

Let *AP* be a set of atomic propositions. The set of NNF formulae over *AP* is inductively defined as follows:

If $p \in AP$ then p and $\neg p$ are NNF formulae.
(Remark: Negations occur *exclusively* in front of atomic propositions.)

If ϕ_1 and ϕ_2 are NNF formulae then so are

$$\phi_1 \vee \phi_2, \quad \phi_1 \wedge \phi_2, \quad \mathbf{X}\,\phi_1, \quad \phi_1\,\mathbf{U}\,\phi_2, \quad \phi_1\,\mathbf{R}\,\phi_2.$$

Claim: For every LTL formula ϕ there is an equivalent NNF formula:

$$\begin{array}{rcl@{\qquad}rcl}
\neg(\phi_1\,\mathbf{R}\,\phi_2) & \equiv & \neg\phi_1\,\mathbf{U}\,\neg\phi_2 & \neg(\phi_1\,\mathbf{U}\,\phi_2) & \equiv & \neg\phi_1\,\mathbf{R}\,\neg\phi_2 \\
\neg(\phi_1 \wedge \phi_2) & \equiv & \neg\phi_1 \vee \neg\phi_2 & \neg(\phi_1 \vee \phi_2) & \equiv & \neg\phi_1 \wedge \neg\phi_2 \\
\neg\,\mathbf{X}\,\phi & \equiv & \mathbf{X}\,\neg\phi & \neg\neg\phi & \equiv & \phi
\end{array}$$

NNF: Example

Translation into an NNF formula:

$$
\begin{aligned}
\mathbf{G}(p \rightarrow \mathbf{F}\, q) &\equiv \neg\, \mathbf{F}\, \neg(p \rightarrow \mathbf{F}\, q) \\
&\equiv \neg(\mathbf{true}\, \mathbf{U}\, \neg(p \rightarrow \mathbf{F}\, q)) \\
&\equiv \neg\mathbf{true}\, \mathbf{R}\, (p \rightarrow \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee \mathbf{F}\, q) \\
&\equiv \mathbf{false}\, \mathbf{R}\, (\neg p \vee (\mathbf{true}\, \mathbf{U}\, q))
\end{aligned}
$$

Remark: Because of this, we shall henceforth assume that the LTL formula in the translation procedure is given in NNF.

Subformulae

Let ϕ be an NNF formula. The set $Sub(\phi)$ is the smallest set satisfying:

- $\phi \in Sub(\phi)$;
- $\mathbf{true} \in Sub(\phi)$;
- if $\phi_1 \in Sub(\phi)$ then $\neg\phi_1 \in Sub(\phi)$, and vice versa;
- if $\mathbf{X}\,\phi_1 \in Sub(\phi)$ then $\phi_1 \in Sub(\phi)$;
- if $\phi_1 \vee \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;
- if $\phi_1 \wedge \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;
- if $\phi_1 \mathbf{U}\, \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$;
- if $\phi_1 \mathbf{R}\, \phi_2 \in Sub(\phi)$ then $\phi_1, \phi_2 \in Sub(\phi)$.

Note: We have $|Sub(\phi)| = \mathcal{O}(|\phi|)$ (one subformula per syntactic element).

Consistent sets

Recall item 2 of the construction:

> Every state will be labelled with a subset of $Sub(\phi)$.

Idea: A state labelled by set *M* will accept a sequence iff it satisfies every single subformula contained in *M* and violates every single subformula contained in $Sub(\phi) \setminus M$.

For this reason, we will a priori exclude some sets *M* which would obviously lead to empty languages.

The other states will be called consistent.

Definition: We call a set $M \subset Sub(\phi)$ *consistent* if it satisfies the following conditions:

$\mathbf{true} \in M$

if $\phi_1 \in Sub(\phi)$ then $\neg\phi_1 \in M$ gdw. $\phi_1 \notin M$;

if $\phi_1 \wedge \phi_2 \in Sub(\phi)$ then $\phi_1 \wedge \phi_2 \in M$ iff $\phi_1 \in M$ and $\phi_2 \in M$;

if $\phi_1 \vee \phi_2 \in Sub(\phi)$ then $\phi_1 \vee \phi_2 \in M$ iff $\phi_1 \in M$ or $\phi_2 \in M$.

By $CS(\phi)$ we denote the set of all consistent subsets of $Sub(\phi)$.

Translation (1)

Let ϕ be an NNF formula and $\mathcal{G} = (\Sigma, S, S_0, \Delta, \mathcal{F})$ be a GBA such that:

$\Sigma = 2^{AP}$
(i.e. the valuations over *AP*)

$S = CS(\phi)$
(i.e. every state is a consistent set)

$S_0 = \{ M \in S \mid \phi \in M \}$
(i.e. the initial states admit sequences satisfying ϕ)

Δ and $\mathcal{F}$: see next slide

Translation (2)

Transitions: $(M, \sigma, M') \in \Delta$ iff $\sigma = M \cap AP$ and:

- if $\mathbf{X}\, \phi_1 \in Sub(\phi)$ then $\mathbf{X}\, \phi_1 \in M$ iff $\phi_1 \in M'$;
- if $\phi_1 \,\mathbf{U}\, \phi_2 \in Sub(\phi)$ then $\phi_1 \,\mathbf{U}\, \phi_2 \in M$
  iff $\phi_2 \in M$ or ($\phi_1 \in M$ and $\phi_1 \,\mathbf{U}\, \phi_2 \in M'$);
- if $\phi_1 \,\mathbf{R}\, \phi_2 \in Sub(\phi)$ then $\phi_1 \,\mathbf{R}\, \phi_2 \in M$
  iff $\phi_1 \wedge \phi_2 \in M$ or ($\phi_2 \in M$ and $\phi_1 \,\mathbf{R}\, \phi_2 \in M'$).

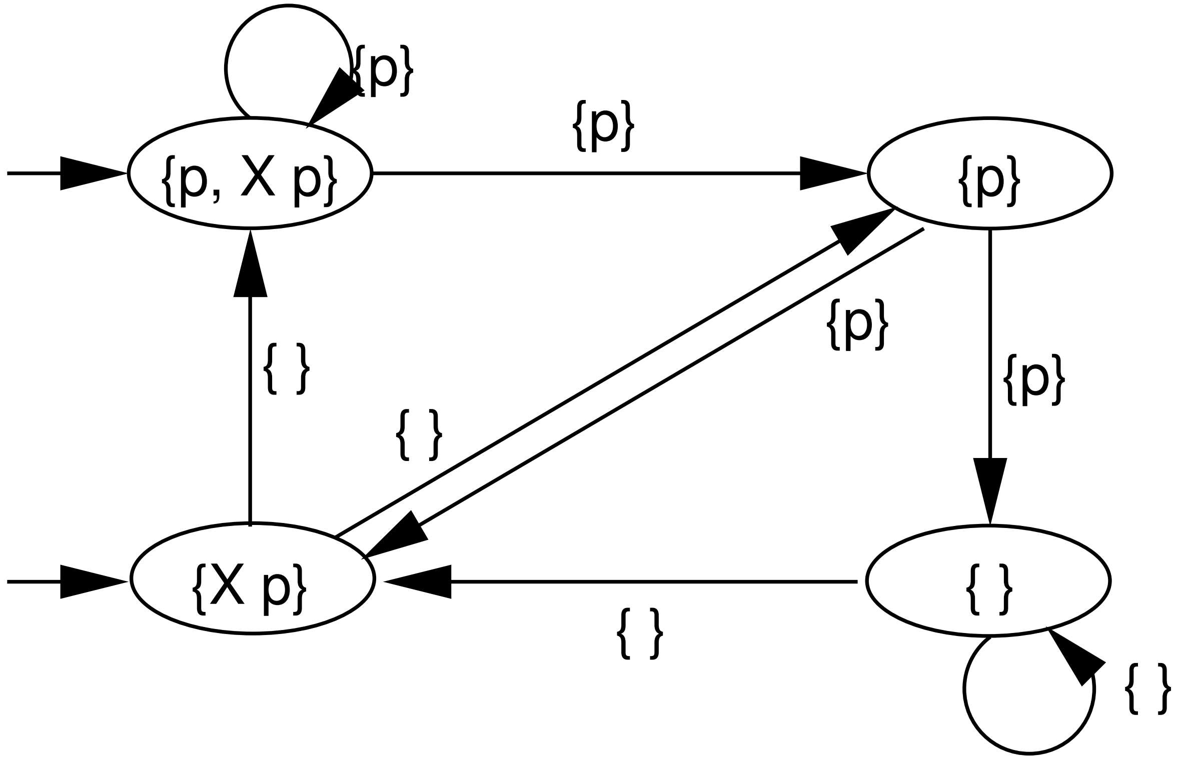
Acceptance condition:

$\mathcal{F}$ contains a set F_ψ for every subformula ψ of the form $\phi_1 \,\mathbf{U}\, \phi_2$, where

$$F_\psi = \{\, M \in CS(\phi) \mid \phi_2 \in M \text{ or } \neg(\phi_1 \,\mathbf{U}\, \phi_2) \in M \,\}.$$

Translation: Example 1

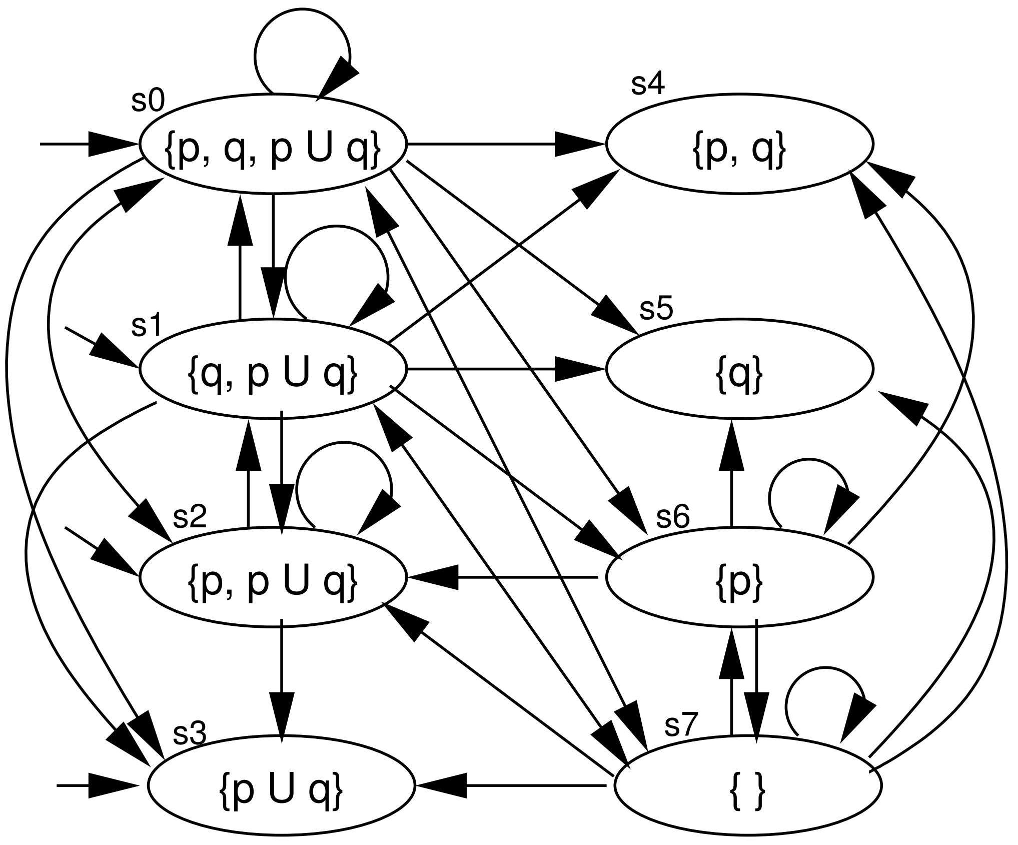
$\phi = \mathbf{X}\, p$



This GBA has got two initial states and the acceptance condition $\mathcal{F} = \emptyset$, i.e. every infinite run is accepting. (Negated Formulas omitted from state labels.)

Translation: Example 2

$\phi \equiv p \mathbf{U} q$



GBA with $\mathcal{F} = \{\{s_0, s_1, s_4, s_5, s_6, s_7\}\}$, transition labels also omitted.

Proof of correctness

We want to prove the following:

$$\sigma \in \mathcal{L}(\mathcal{G}) \quad \text{gdw.} \quad \sigma \in [\![\phi]\!]$$

To this aim, we shall prove the following stronger property:

Let α be a sequence of consistent sets (i.e., states of $\mathcal{G}$) and let σ be a sequence of valuations over *AP*.

α is an accepting run of $\mathcal{G}$ over σ
iff $\sigma^i \in [\![\psi]\!]$ for all $i \geq 0$ and $\psi \in \alpha(i)$.

The desired proof then follows from the choice of initial states.

Correctness (2)

Remark: By construction, we have $\sigma(i) = \alpha(i) \cap AP$ for all $i \geq 0$.

Proof via structural induction over ψ:

for $\psi = p$ and $\psi = \neg p$ if $p \in AP$:
obvious since $\sigma^i \in [\![p]\!]$ iff $p \in \sigma(i)$ iff $p \in \alpha(i)$.

for $\psi_1 \vee \psi_2$ and $\psi_1 \wedge \psi_2$: follows from consistency of $\alpha(i)$ and from the induction hypothesis for ψ_1 and ψ_2, resp.

for $\mathbf{X}\,\psi_1$: follows from the construction of Δ and induction hypothesis for ψ_1.

Correctness (3)

for $\psi = \psi_1 \mathbf{R}\, \psi_2$:

Follows from the construction of Δ, the recursion equation for $\mathbf{R}$ and the induction hypothesis.

for $\psi = \psi_1 \mathbf{U}\, \psi_2$:

Analogous to $\mathbf{R}$, but additionally we must ensure that $\psi_2 \in \alpha(k)$ for some $k \geq i$. Assume that this is not the case, then we have $\psi_1 \mathbf{U}\, \psi_2 \in \alpha(k)$ for all $k \geq i$. However, none of these states is in F_ψ, therefore α cannot be accepting, which is a contradiction.

Complexity of the translation

The translation procedure produces an automaton of size $\mathcal{O}(2^{|\phi|})$, for a formula ϕ.

Question: Is there a better translation procedure?

Answer 1: No (not in general). There exist formulae for which any Büchi automaton has necessarily exponential size.

Example: The following LTL formula over $\{p_0, \ldots, p_{n-1}\}$ simulates an *n*-bit counter.

$$\mathbf{G}(p_0 \not\leftrightarrow \mathbf{X}\, p_0) \wedge \bigwedge_{i=1}^{n-1} \mathbf{G}\left(\left(p_i \not\leftrightarrow \mathbf{X}\, p_i\right) \leftrightarrow \left(p_{i-1} \wedge \neg \mathbf{X}\, p_{i-1}\right)\right)$$

The formula has size $\mathcal{O}(n)$. Obviously, any automaton for this formula must have at least 2^n states.

Answer 2: Yes (sometimes). There are translation procedures that produce smaller automata *for most cases*.

Some tools:

Spin (Aufruf: `spin -f 'p U q'`)

LTL2BA (web applet)

Literature:

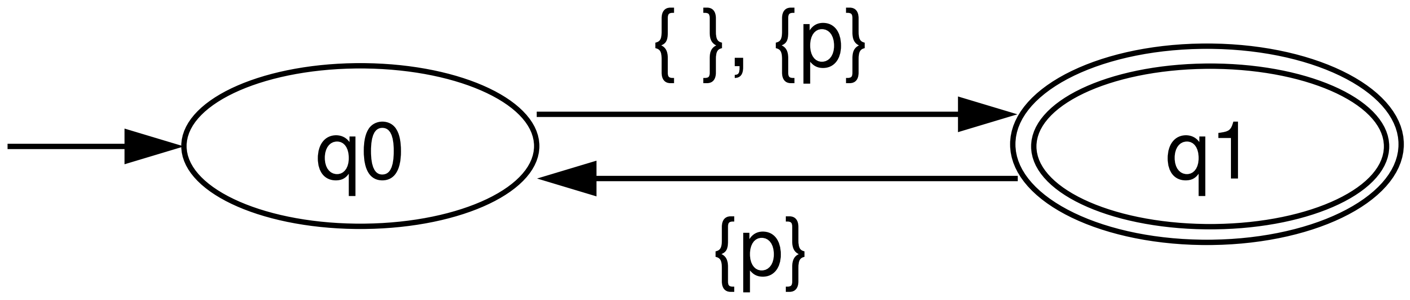
Gerth, Peled, Vardi, Wolper: *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, 1996

Oddoux, Gastin: *Fast LTL to Büchi Automata Translation*, 2001

Translation BA $\rightarrow$ LTL

The reverse translation (BA $\rightarrow$ LTL) is not possible in general.

I.e., there are Büchi automata $\mathcal{B}$ such that there is no formula ϕ with $\mathcal{L}(\mathcal{B}) = [\![\phi]\!]$ (Wolper, 1983).



The property "*p* holds in every second step" is not expressible in LTL (proof: next slide).

Proof (BA $\not\to$ LTL)

We first show a more general lemma:

Let ϕ be an arbitrary LTL formula over AP and n the number of $\mathbf{X}$ operators in ϕ. We regard the sequences

$$\sigma_i = \{p\}^i \, \emptyset \, \{p\}^\omega$$

for $i \geq 0$. For all pairs $i, k > n$ we have: $\sigma_i \models \phi$ iff $\sigma_k \models \phi$.

Proof by structural induction over ϕ:

If $\phi = p$, for $p \in AP$, then $n = 0$ and $i, k \geq 1$.
Thus, $\sigma_i \models p$ and $\sigma_k \models p$.

For the other cases, the induction hypothesis assumes that the property holds for ϕ_1 und ϕ_2, i.e. if ϕ_1, ϕ_2 contain n_1 and n_2 occurrences of $\mathbf{X}$, resp., then for all $i_1, k_1 > n_1$ we have $\sigma_{i_1} \models \phi_1$ iff $\sigma_{k_1} \models \phi_1$, and analogously for ϕ_2.

If $\phi = \neg\phi_1$, then the proof follows directly from the induction hypothesis.

For $\phi = \phi_1 \vee \phi_2$: same

If $\phi = \mathbf{X}\,\phi_1$, then $n_1 = n - 1$. Since $i-1, k-1 > n-1 = n_1$, the induction hypothesis implies: $\sigma_i^1 = \sigma_{i-1} \models \phi_1$ iff $\sigma_k^1 = \sigma_{k-1} \models \phi_1$, which implies the proof.

For $\phi = \phi_1 \,\mathbf{U}\, \phi_2$: Let $m > n$. We have:

$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2))$$

Applying this law recursively we obtain:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_m \models \phi_2 \vee (\sigma_m \models \phi_1 \wedge (\sigma_{m-1} \models \phi_2 \vee (\ldots$$
$$(\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

According to the induction hypothesis, we can replace indices bigger than n equivalently by $n+1$:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge (\sigma_{n+1} \models \phi_2 \vee (\ldots \\ (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2))))$$

This can be simplified to the following:

$$\sigma_m \models \phi \quad \text{gdw.} \quad \sigma_{n+1} \models \phi_2 \vee (\sigma_{n+1} \models \phi_1 \wedge \sigma_n \models \phi_1 \,\mathbf{U}\, \phi_2)$$

Thus, the validity of $\sigma_m \models \phi$ is completely independent of m, leading to the desired property for i and k, which concludes the proof of the lemma.

Let us now assume that there exists an LTL formula ϕ expressing the property of the aforementioned BA ("*p* holds in every second step"). Let *n* be the number of occurrences of $\mathbf{X}$ in ϕ.

Let us consider the sequences σ_{n+1} and σ_{n+2}.

If *n* is even then $\sigma_{n+1} \not\models \phi$ and $\sigma_{n+2} \models \phi$. If *n* is odd, then vice versa.

However, the previous lemma tells us that this is impossible: either σ_{n+1} and σ_{n+2} both satisfy ϕ, or none of them does. Therefore, such a formula ϕ cannot exist.

The model-checking problem for LTL (preview)

Problem: Given a Kripke structure $\mathcal{K} = (S, \rightarrow, r, AP, \nu)$ and an LTL formula ϕ over AP, we ask whether $\mathcal{K} \models \phi$.

Solution: (sketch)

We re-interpret $\mathcal{K}$ as a Büchi automaton $\mathcal{B}_{\mathcal{K}}$:

$\mathcal{B}_{\mathcal{K}} = (2^{AP}, S, r, \Delta, S)$, where $\Delta = \{\, (s, \nu(s), t) \mid s \rightarrow t \,\}$

Obviously, $[\![\mathcal{K}]\!] = \mathcal{L}(\mathcal{B}_{\mathcal{K}})$.

Moreover, we translate $\neg\phi$ into a Büchi automaton $\mathcal{B}_{\neg\phi}$.

We have:

$$\begin{array}{rrcll} & \mathcal{K} & \models & \phi & \\ \iff & [\![\mathcal{K}]\!] & \subseteq & [\![\phi]\!] & \\ \iff & [\![\mathcal{K}]\!] & \cap & [\![\neg\phi]\!] & = \emptyset \\ \iff & \mathcal{L}(\mathcal{B}_{\mathcal{K}}) & \cap & \mathcal{L}(\mathcal{B}_{\neg\phi}) & = \emptyset \end{array}$$

Therefore:

We construct Büchi automata $\mathcal{B}_{\mathcal{K}}$ and $\mathcal{B}_{\neg\phi}$.

We intersect both automata (using the special-case construction).

Thus, the model-checking problem reduces to the problem of deciding whether the product automaton accepts the empty language.

Part 6: Efficient Emptiness Test for Büchi Automata

Overview

As we have seen, the model-checking problem reduces to checking whether the language of a certain Büchi automaton $\mathcal{B}$ is *empty*.

Reminder: $\mathcal{B}$ arises from the intersection of a Kripke structure $\mathcal{K}$ with a BA for the *negation* of ϕ.

If $\mathcal{B}$ accepts some word, we call such a word a counterexample.

$\mathcal{K} \models \phi$ iff $\mathcal{B}$ accepts the empty language.

Typical instances:

Size of $\mathcal{K}$: between several hundreds to millions of states.

Size of $\mathcal{B}_{\neg\phi}$: usually just a couple of states

Typical setting (e.g., in Spin):

$\mathcal{K}$ indirectly given in some description language (C, Java / in Spin: Promela); model-checking tools will generate $\mathcal{K}$ internally.

$\mathcal{B}_{\neg\phi}$ generated from ϕ before start of emptiness check.

Typical setting:

$\mathcal{B}$ generated “on-the-fly” from (the description of) $\mathcal{K}$ and from $\mathcal{B}_{\neg\phi}$ and tested for emptiness *at the same time*.

As a consequence, the size of $\mathcal{K}$ (and of $\mathcal{B}$) is not known initially!

At the beginning, only the initial state is known, and we have a function $\mathtt{succ}\colon S \to 2^S$ for computing the immediate successors of a given state (the function implements the semantics of the description).

Memory requirements

Transitions not stored explicitly, will be explored “on demand” by calling succ (calls to succ will be comparatively costly).

Hash table for explored states.

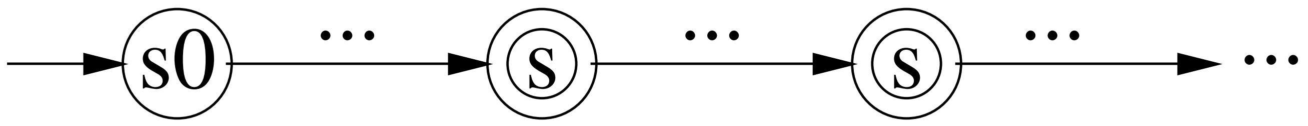
Information stored for each state:

> Descriptor: program counter, variable values, active processes, etc (often dozens or hundreds of bytes)
>
> Auxiliary information: Data needed by the emptiness test (a couple of bytes)

Simple solution I: Check for Lassos

Let $\mathcal{B} = (\Sigma, S, s_0, \delta, F)$ be a Büchi automaton.

$\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff there is $s \in F$ such that $s_0 \rightarrow^* s \rightarrow^+ s$



Naïve solution:

Check for each $s \in F$ whether there is a cycle around s; let $F_\circ \subseteq F$ denote the set of states with this property.

Check whether s_0 can reach some state in $F_\circ$.

Time requirement: Each search takes linear time in the size of $\mathcal{B}$, altogether quadratic run-time $\rightarrow$ unacceptable for millions of states.

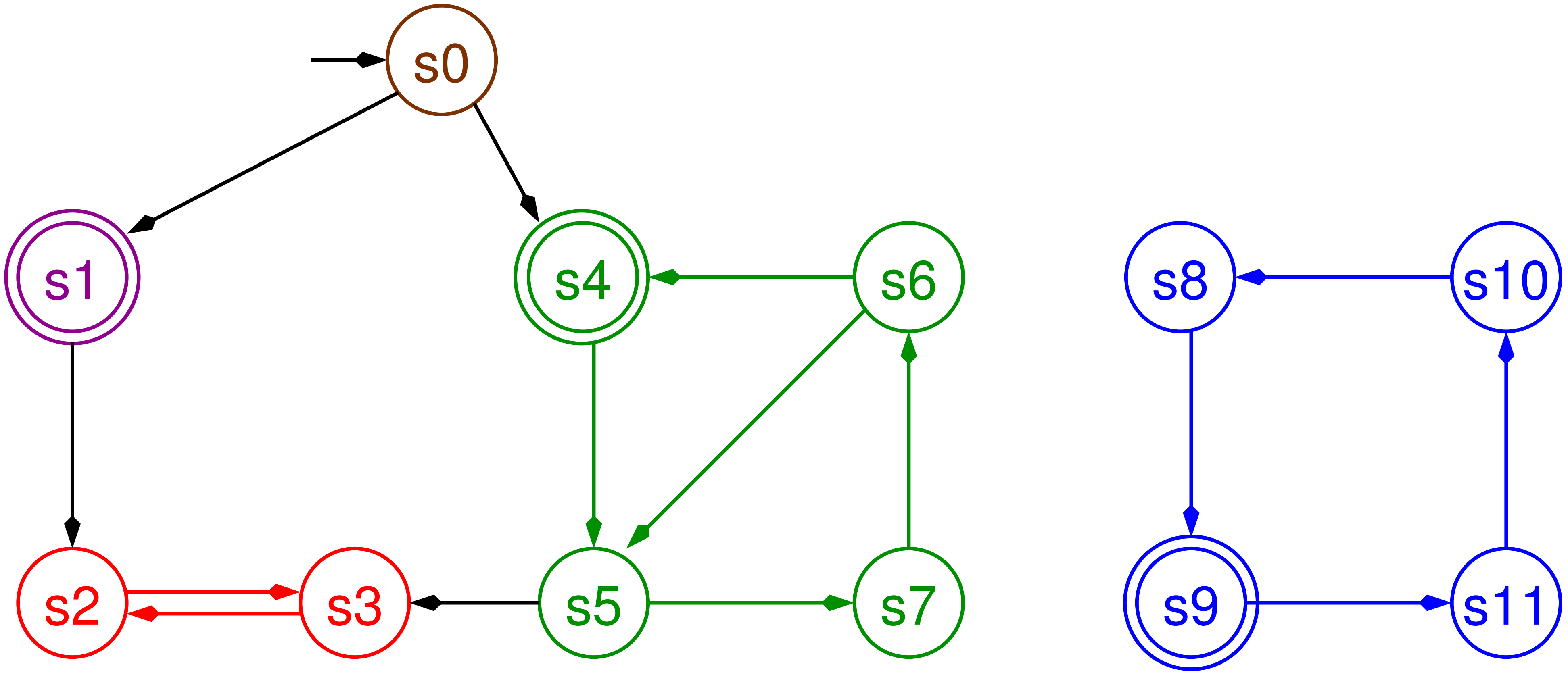
Strongly connected components

$C \subseteq S$ is called a strongly connected component (SCC) iff

- $s \rightarrow^* s'$ for all $s, s' \in C$;
- C is maximal w.r.t. the above property, i.e. there is no proper superset of C satisfying the above.

An SCC C is called trivial if $|C| = 1$ and for the unique state $s \in C$ we have $s \not\rightarrow s$ (single state without loop).

Example: SCCs



The SCCs $\{s_0\}$ and $\{s_1\}$ are trivial.

Simple algorithm II: SCCs

Observation: $\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff $\mathcal{B}$ has a non-trivial SCC that is reachable from s_0 and contains an accepting state.

Simple algorithm: for every accepting state s

compute the set V_s of the predecessors of s;

compute the set N_s of the successors of s;

$V_s \cap N_s$ is the SCC containing s;

test whether $V_s \cap N_s \supset \{s\}$ or $s \to s$.

Running time: again quadratic

Efficient solution

In the following, we shall discuss a solution whose run-time is linear in $|\mathcal{B}|$ (i.e. proportional to $|S| + |\delta|$).

The solution is based on depth-first search (DFS) and on partitioning $\mathcal{B}$ into its SCCs.

Literature: [Tarjan 1972], Couvreur 1999, Gabow 2000

Depth-first search (basic version)

```
nr = 0;
hash = {};
dfs(s0);
exit;

dfs(s) {
   add s to hash;
   nr = nr+1;
   s.num = nr;

   for (t in succ(s)) {
      // deal with transition s -> t
      if (t not yet in hash) { dfs(t); }
   }
}
```

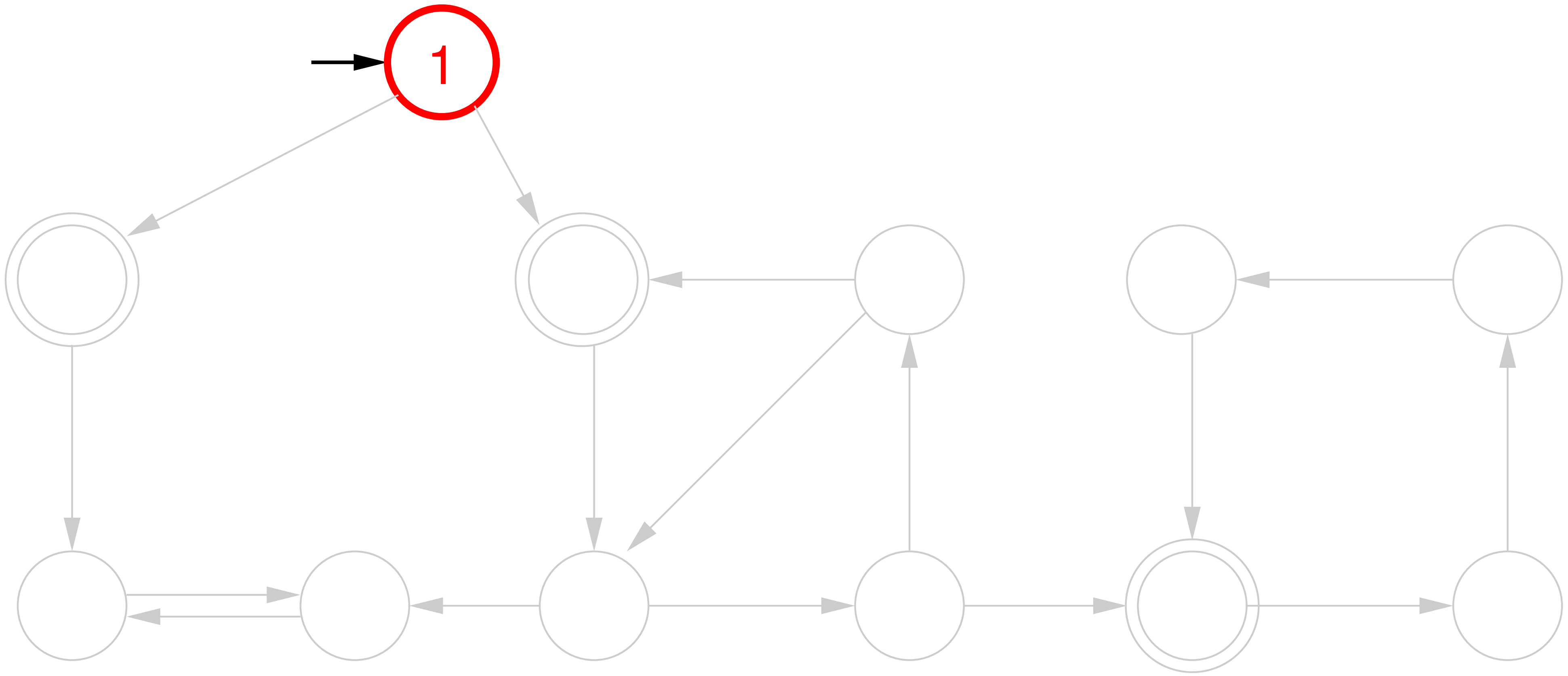
Memory usage

Global variables: counter *nr*, hash table for states

Auxiliary information: “DFS number” *s.num*

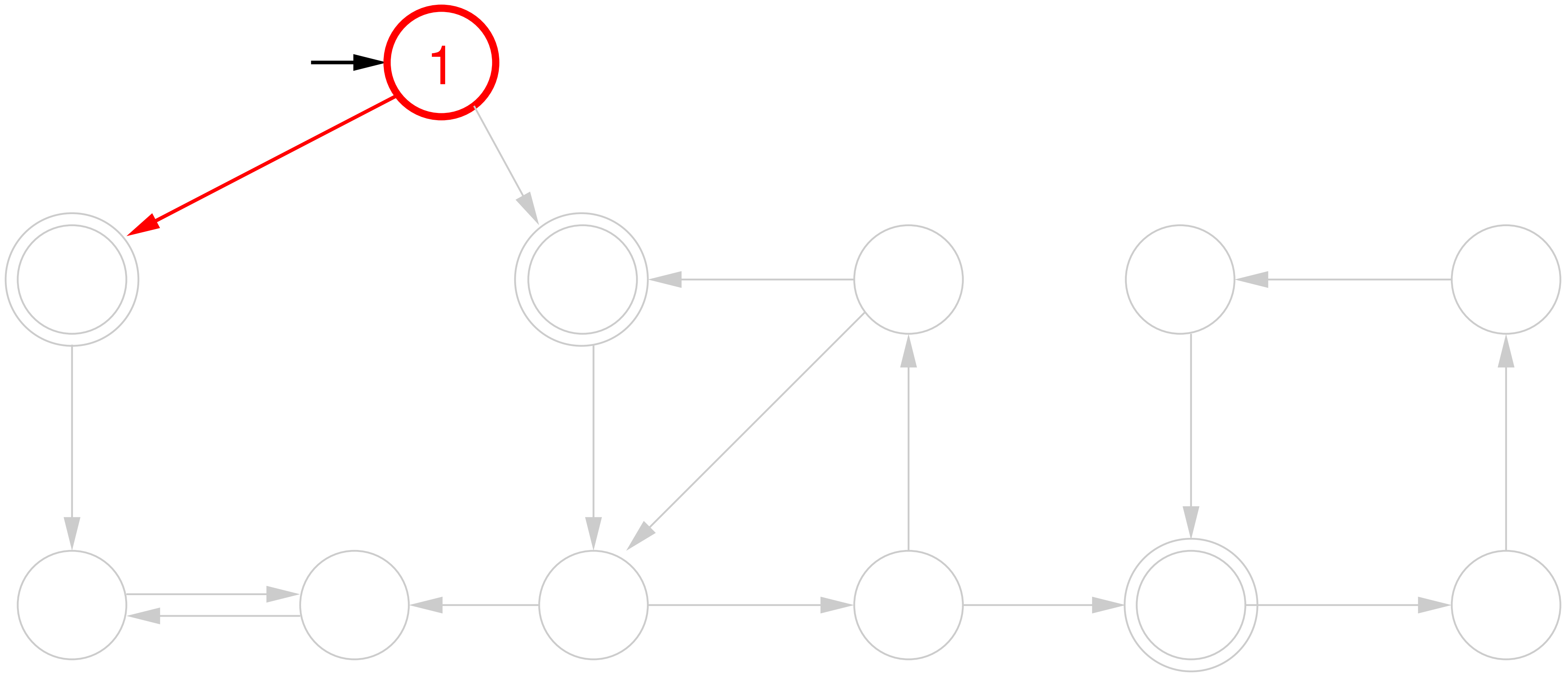
search path: Stack for memorizing the “unfinished” calls to *dfs*

Example: Depth-first search



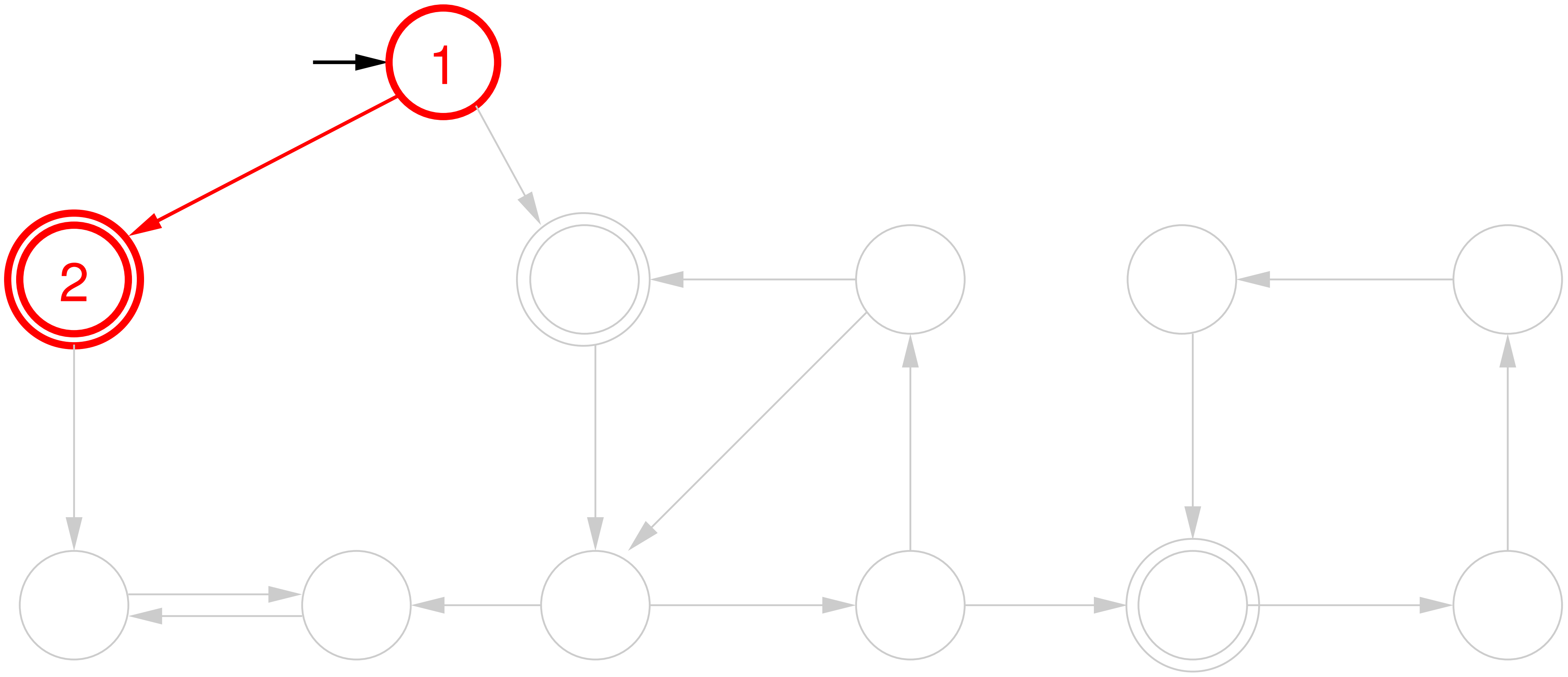
Search path shown in red, other visited states black, states not yet seen grey.

Example: Depth-first search



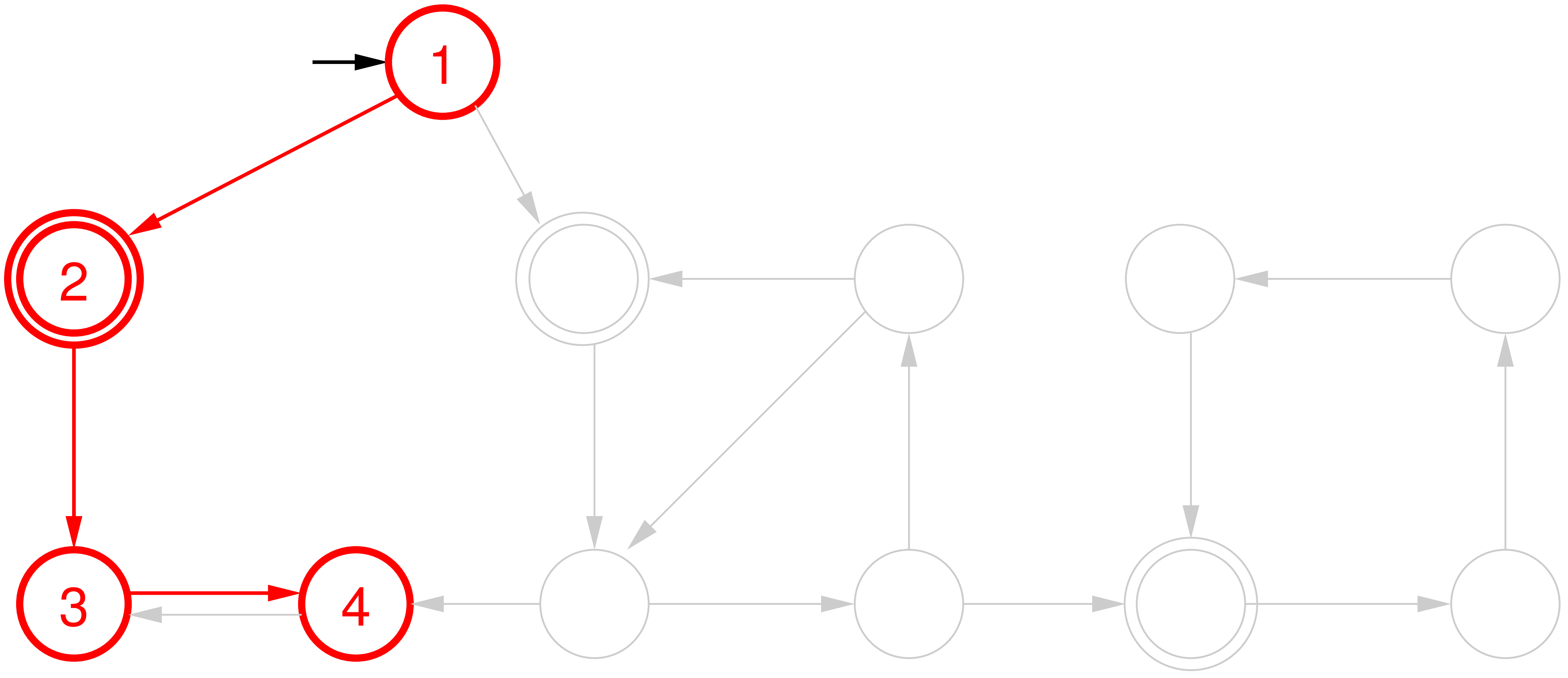
DFS starts at initial state and explores some immediate successor.

Example: Depth-first search



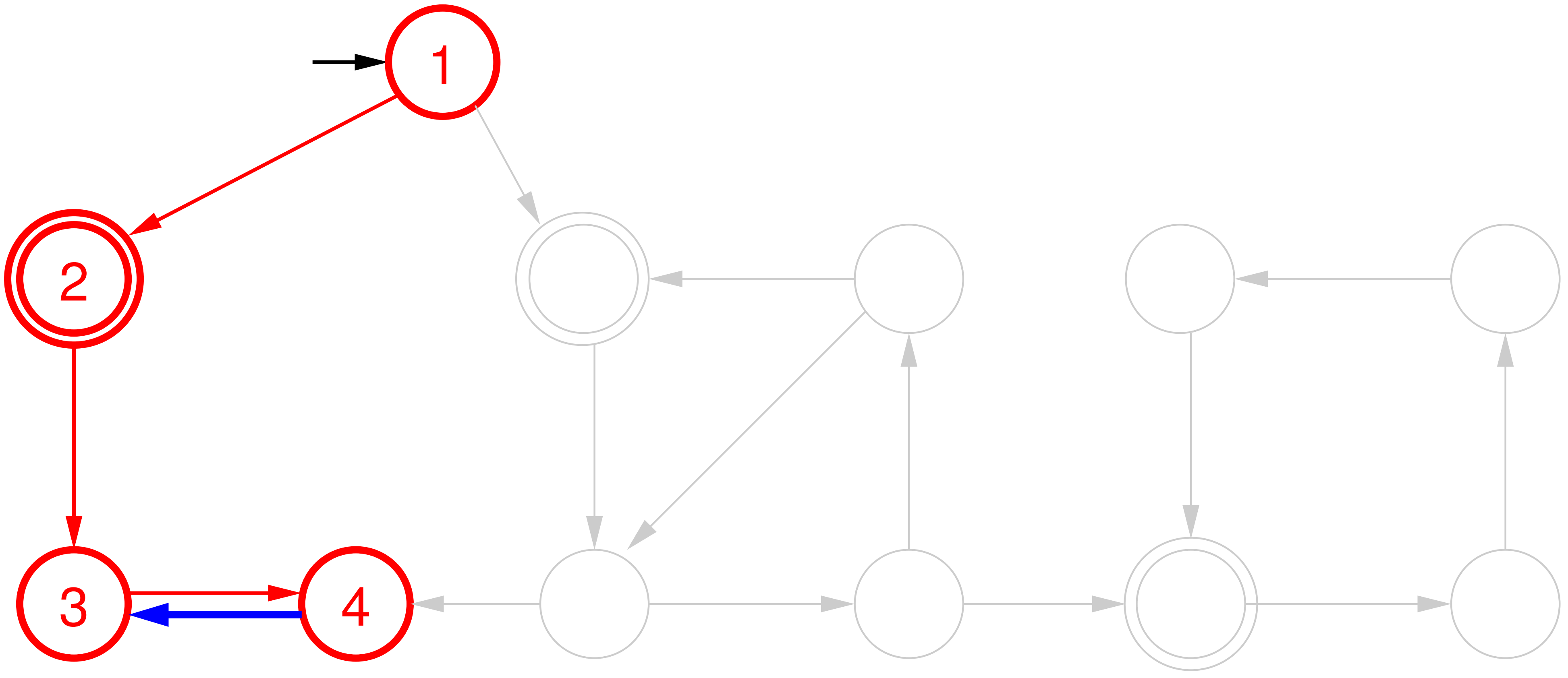
Successor state not yet visited; recursive call, assigned to number 2.

Example: Depth-first search



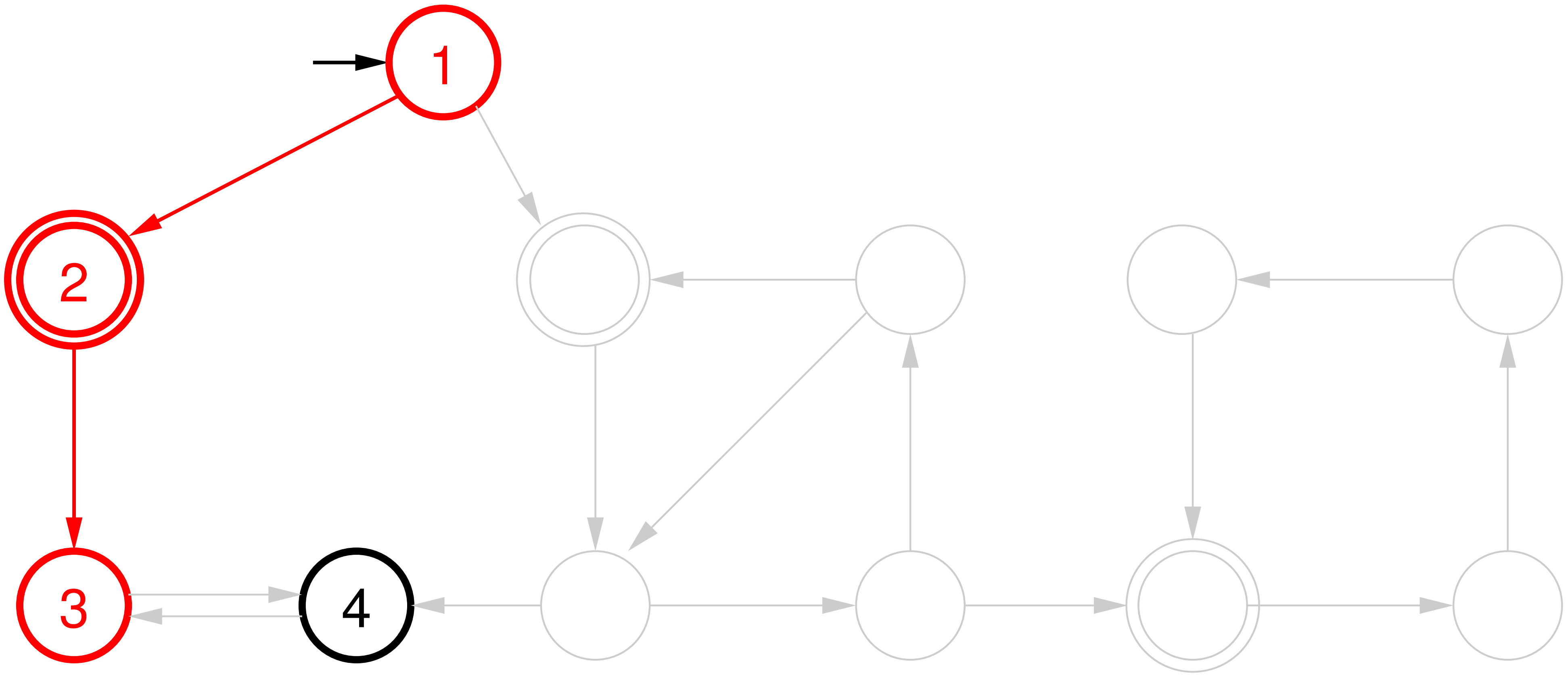
More unvisited states are being explored. . .

Example: Depth-first search



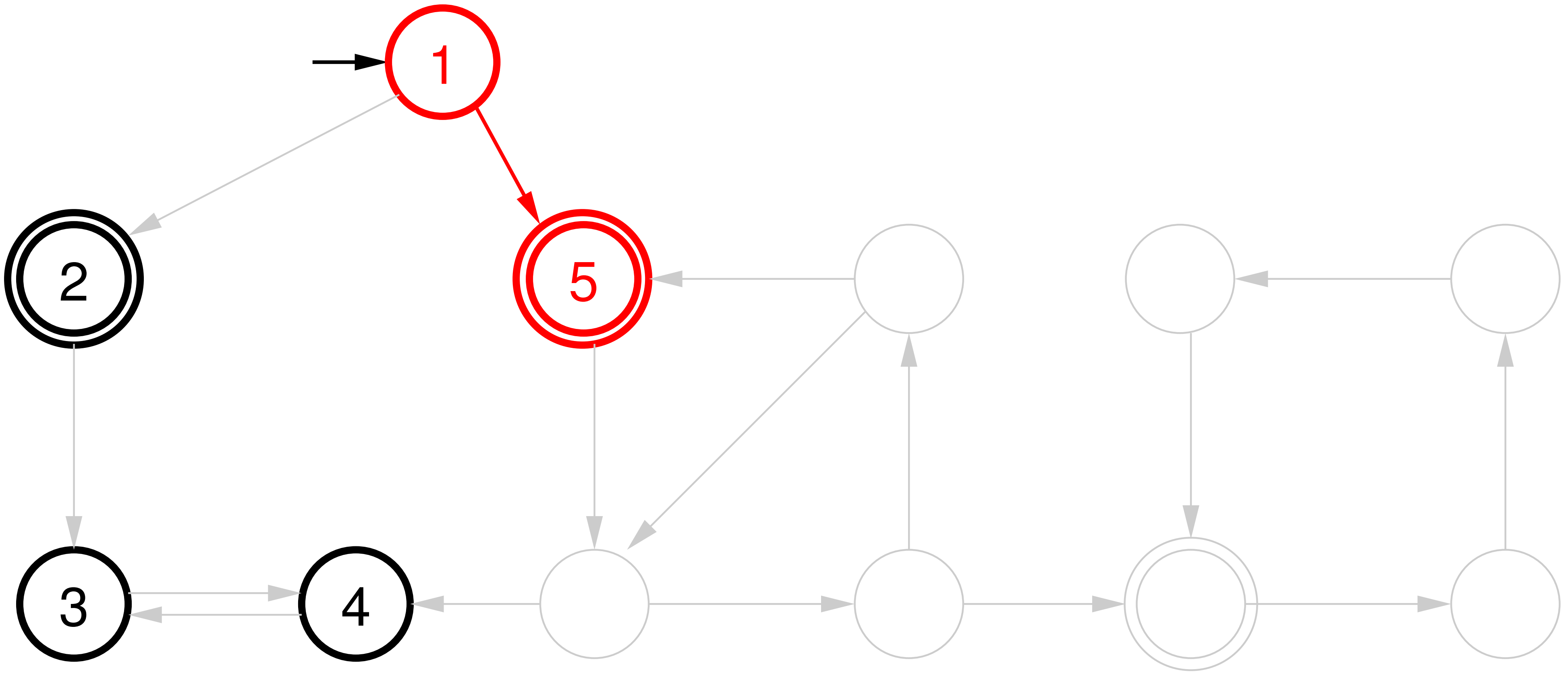
Edge from 4 to 3: target state already known, no recursive call

Example: Depth-first search



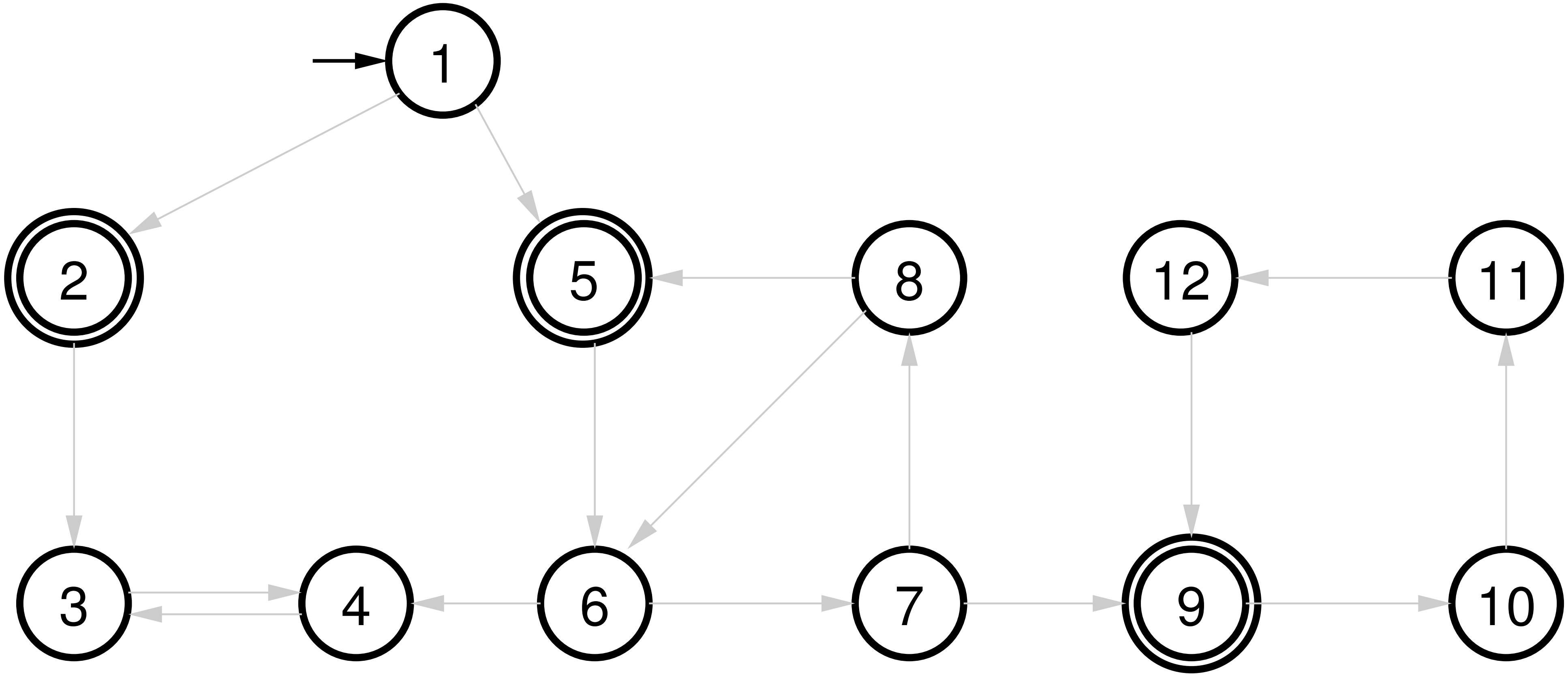
All immediate successors of 4 have been explored; backtrack.

Example: Depth-first search



Backtracking proceeds to state 1, next successor gets number 5.

Example: Depth-first search



Possible numbering at the end of DFS.

Properties of the search path

(1) Let $s_0 s_1 \ldots s_n$ be the search path at some point during DFS.
Then we have $s_i.num < s_j.num$ iff $i < j$.
Moreover, $s_i \rightarrow^* s_j$ if $i < j$.

Proof: follows from the logic of the program and the order of recursive calls.

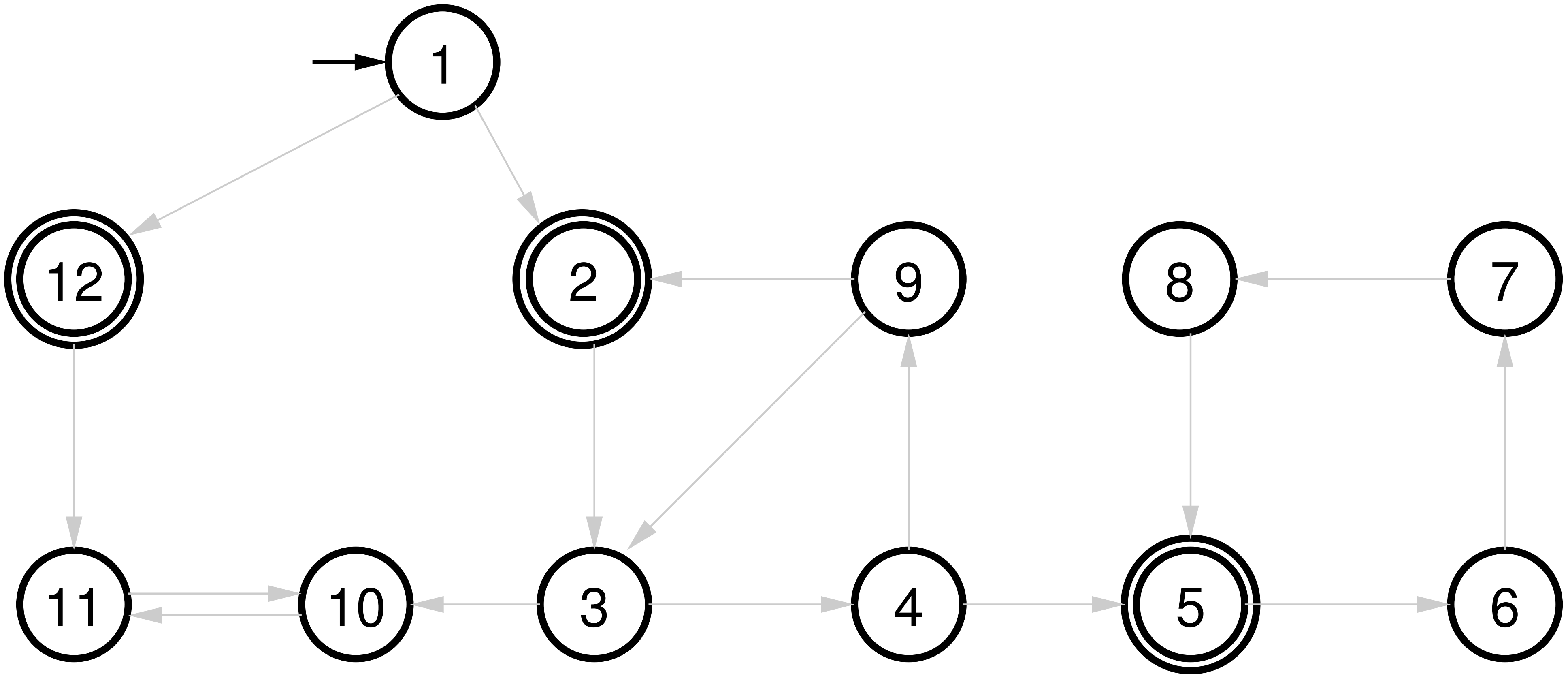
Search order

If a state has got multiple immediate successors, they must be explored in *some* order.

The DFS numbering therefore depends on the order in which these successors are explored; multiple different numberings are possible.

The search order may influence how quickly a counterexample is found (if one exists)!

Example: Search order



Possible alternative numbering for a different search order.

Search order

If a state has got multiple immediate successors, they must be explored in *some* order.

The DFS numbering therefore depends on the order in which these successors are explored; multiple different numberings are possible.

The search order may influence how quickly a counterexample is found (if one exists)!

Assumption: search-order non-deterministic (or fixed from outside)

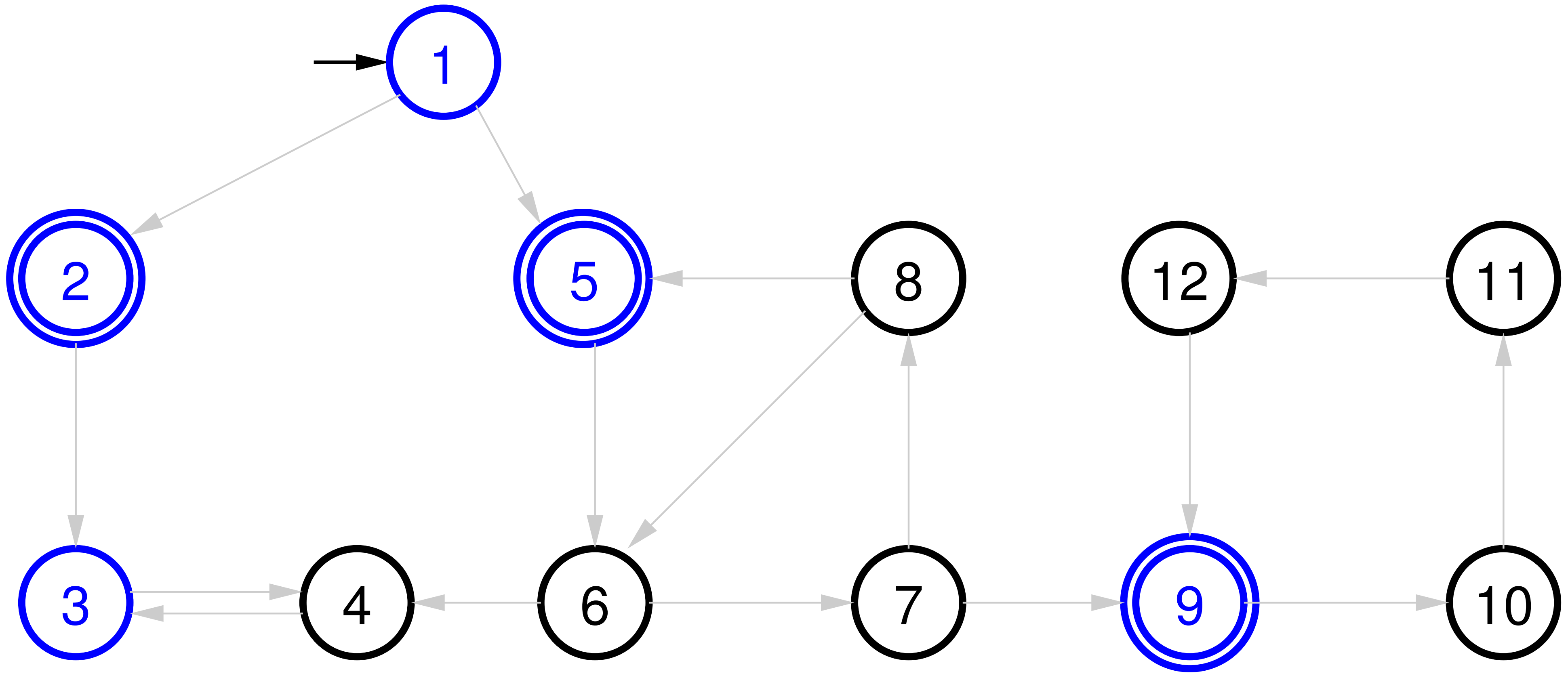
Possible extension: “intelligent” search order exploiting additional knowledge about the model to find counterexamples more quickly.

Roots

The unique (w.r.t. a fixed search order) state of an SCC that is visited first during DFS is called its root.

Remark: Different search orders may lead to different states designated as roots.

Example: Search order



Roots shown in blue when using the previous search order.

Properties of roots

(2) A root has the smallest DFS number within its SCC.

Proof: obvious

(3) Within each SCC, the root is the last state from which DFS backtracks, and, at that point, the SCC has been explored completely (i.e., all states and edges have been considered).

Proof: Suppose the DFS first reaches a root r. At that point, no other state of the SCC has been visited so far, and all are reachable from r. Therefore, the DFS will visit all those states (and possibly others) and backtrack from them before it can backtrack from r.

Explored/active Subgraph

At each point during the DFS, let us distinguish two specific subgraphs of $\mathcal{B}$.

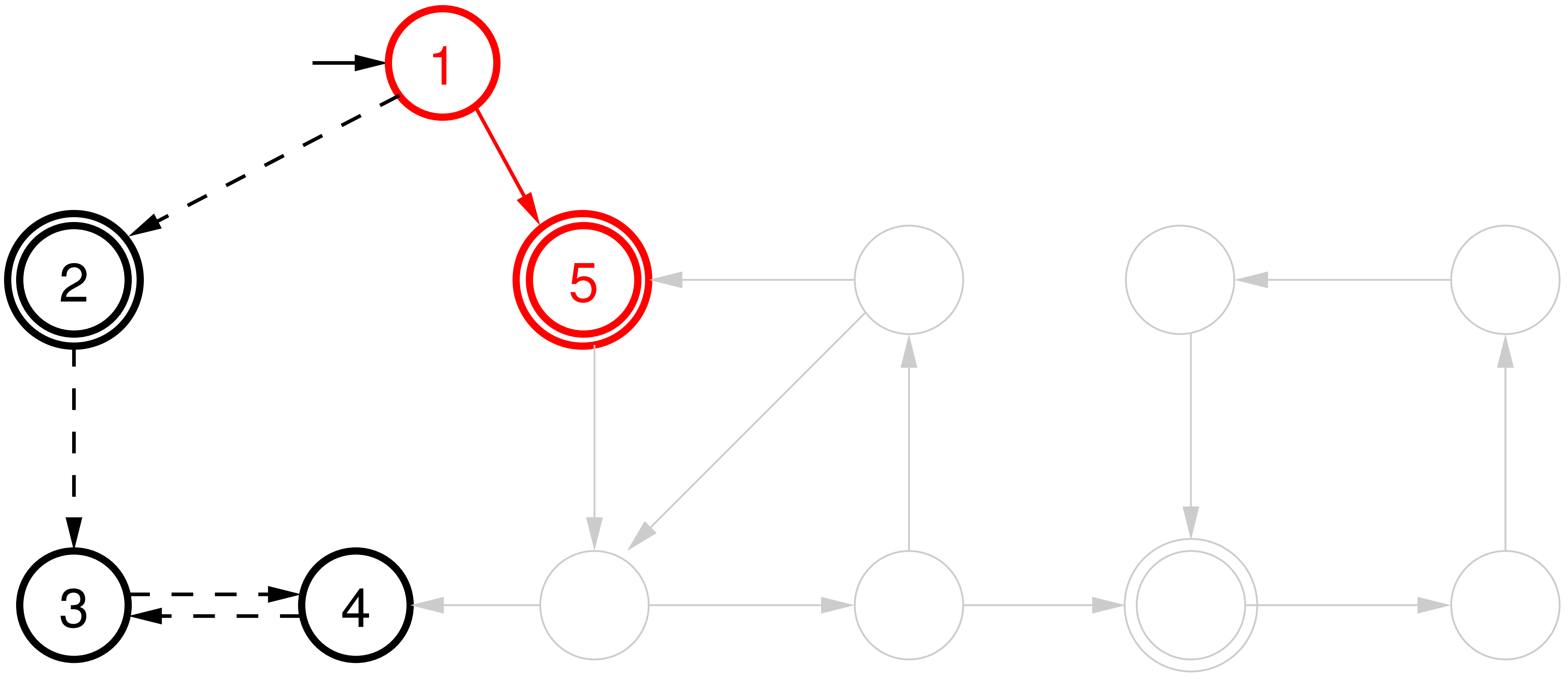
The explored graph of $\mathcal{B}$ denotes the subgraph containing all visited states and explored transitions.

We call an SCC *of the explored graph*(!) active, if the search path contains at least one of its states (whose DFS call has not yet terminated).

A state is called active if it is part of an active SCC (it is not necessary for the state itself to be on the search path).

The active graph is the subgraph of the explored graph induced by the active states.

Example: Explored/active subgraph



Here: explored graph shown in red and black, active SCCs: $\{1\}$ and $\{5\}$, inactive SCCs $\{2\}$ and $\{3, 4\}$.

Properties of the active graph

(4) An SCC becomes inactive when we backtrack from its root.

Proof: follows from (3).

(5) An inactive SCC of the explored graph is also an SCC of $\mathcal{B}$.

Proof: Follows immediately from (3) and (4).

(6) The roots of the active graph are a subsequence of the search path.

Proof: Follows from (4) because the root of an active SCC must be on the search path.

(7) Let s be an active state and t (where $t.num \leq s.num$) the root of its SCC in the active graph. Then there is no active root u with $t.num < u.num < s.num$.

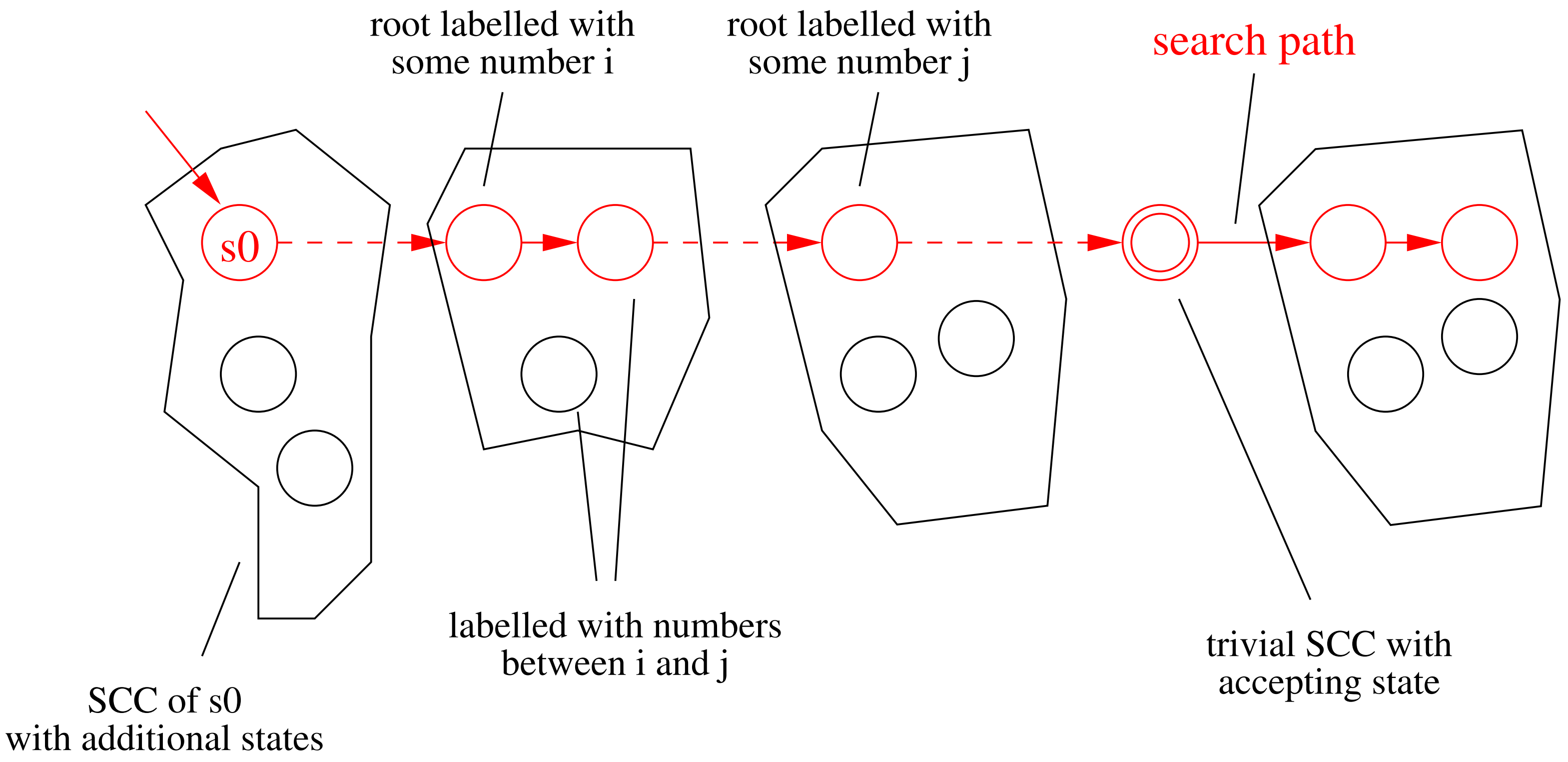
Proof: Assume that such an active root u exists. Since u is active, it is on the search stack, just like t, see (4). Then, because of (1), we have $t \rightarrow^* u$. As `dfs(u)` has not yet terminated and $u.num < s.num$, s must have been reached from u, i.e. $u \rightarrow^* s$. Because s, t are in the same SCC, $s \rightarrow^* t$ holds. But then, t, u are in the same SCC and cannot both be its root.

(8) Let s and t be two active states with $s.num \leq t.num$. Then $s \rightarrow^* t$.

Let s', t' be the (active) roots for s and t, resp. Because of (7) we have $s'.num \leq t'.num$, thus because of (1) $s' \rightarrow^* t'$, and therefore $s \rightarrow^* t$.

Visualization

From the properties we've just proved, it follows that the active graph and its SCCs are always of the following form, at any time during DFS:


root labelled with some number i
root labelled with some number j
search path
s0
SCC of s0 with additional states
labelled with numbers between i and j
trivial SCC with accepting state

Properties of our emptiness-checking algorithm

Run-time linear in $|S| + |\delta|$.

Explores $\mathcal{B}$ using DFS; reports a counterexample *as soon as the explored graph contains one*. (*)

For every explored state s the algorithm computes $\mathsf{succ}(s)$ only once.

$\rightarrow$ saves time because succ is the most expensive operation in practice.

Additional memory usage

Stack W with elements of the form (s, C), where

- s is the root of an active SCC;
- C is the set of state in the SCC of s.

(C may be implemented as a linked list, one additional pointer for each state.)

One bit per state indicating whether a state is active or not.

How the algorithm works

Actions of the algorithm:

- Initialization
- Discovering new edges (to old or new states)
- Backtracking

With each action, we

- update the contents of W and the “active” bits;
- check whether the explored graph contains a counterexample.

Initialization

Explored graph consists just of the initial state s_0, no edges.

One single element in W: the tuple $(s_0, \{s_0\})$

s_0 is active.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 1: t was never seen before:

The explored graph is extended by the state t and the edge $s \to t$.

t is active and forms a trivial SCC within the active graph.

Extend W by $(t, \{t\})$.

Recursively start DFS on t.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 2: t has been visited before and is inactive.

If t is inactive, then its SCC has been completely explored, see (3) and (4). Therefore, s, t must belong to different SCCs, in particular, $t \to^* s$ cannot hold. Therefore, the edge $s \to t$ cannot be part of a lasso, and we can ignore it.

No recursive call, W and the “active” bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \rightarrow t$. We distinguish five cases:

Case 3: t was visited before and is active, and $t.num > s.num$.

From (8) we already know that $s \rightarrow^* t$ holds, therefore the SCCs of the active graph do not change, and no new counterexample can be generated in this way. Thus, we ignore the edge.

No recursive call, W and the "active" bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \rightarrow t$. We distinguish five cases:

Case 4: t was visited before and $t.num = s.num$.

Then $s = t$.

A counterexample has been discovered iff s is accepting.

Otherwise: no recursive call, W and the “active” bits remain unchanged.

Dealing with new edges

Suppose we discover an edge $s \to t$. We distinguish five cases:

Case 5: t was seen before and is active, $t.num < s.num$.

Then because of (8) we have $t \to^* s$. Thus, s, t belong to the same SCC. Let u, with $u.num \leq t.num$, be the root of the SCC to which t belongs. Since s is the latest element on the search path, it follows from (1) that all SCCs stored on W from u downwards must be merged into one SCC.

We find u by removing elements from W until we find a root whose number is no larger than $t.num$, compare (7).

A new counterexample is generated iff one of the merged SCCs was hitherto trivial. Therefore, while removing elements from W we simply check whether any of the roots is an accepting state.

Backtracking

Suppose that all elements in $\mathtt{succ}(s)$ have been explored.

Case 1: s is a root.

> Then s and its entire SCC become inactive, see (4).
>
> Moreover, we remove the topmost element from W.

Case 2: s is not a root.

> Then the root of its SCC is still active.
>
> W and the “active” bits remain unchanged.

Et voilà. . .

```
nr = 0; hash = {}; W = {}; dfs(s0); exit;

dfs(s) {
   add s to hash; s.active = true;
   nr = nr+1; s.num = nr;
   push (s,{s}) onto W;
   for (t in succ(s)) {
      if (t not yet in hash) { dfs(t); }
      else if (t.active) {
         D = {};
         repeat
            pop (u,C) from W;
            if u is accepting { report success; halt; }
            merge C into D;
         until u.num <= t.num;
         push (u,D) onto W;
   }  }
   if s is the top root in W {
      pop (s,C) from W;
      for all t in C { t.active = false; }
   }
}
```

Remarks on the algorithm

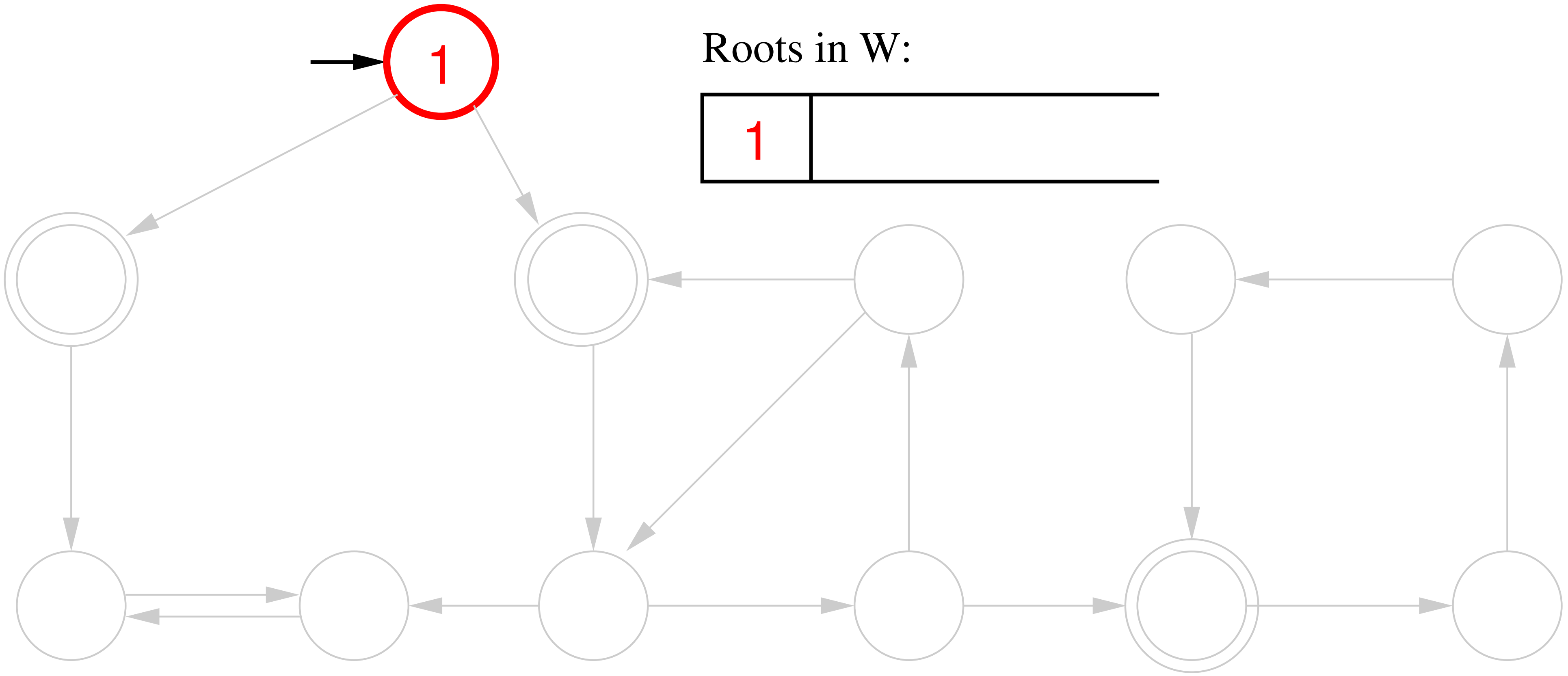
Cases 3 to 5 for handling edges are dealt with uniformly in the repeat-until loop.

The statement `report success` symbolizes the discovery of a counterexample.

If `dfs(s0)` terminates, no counterexample exists.

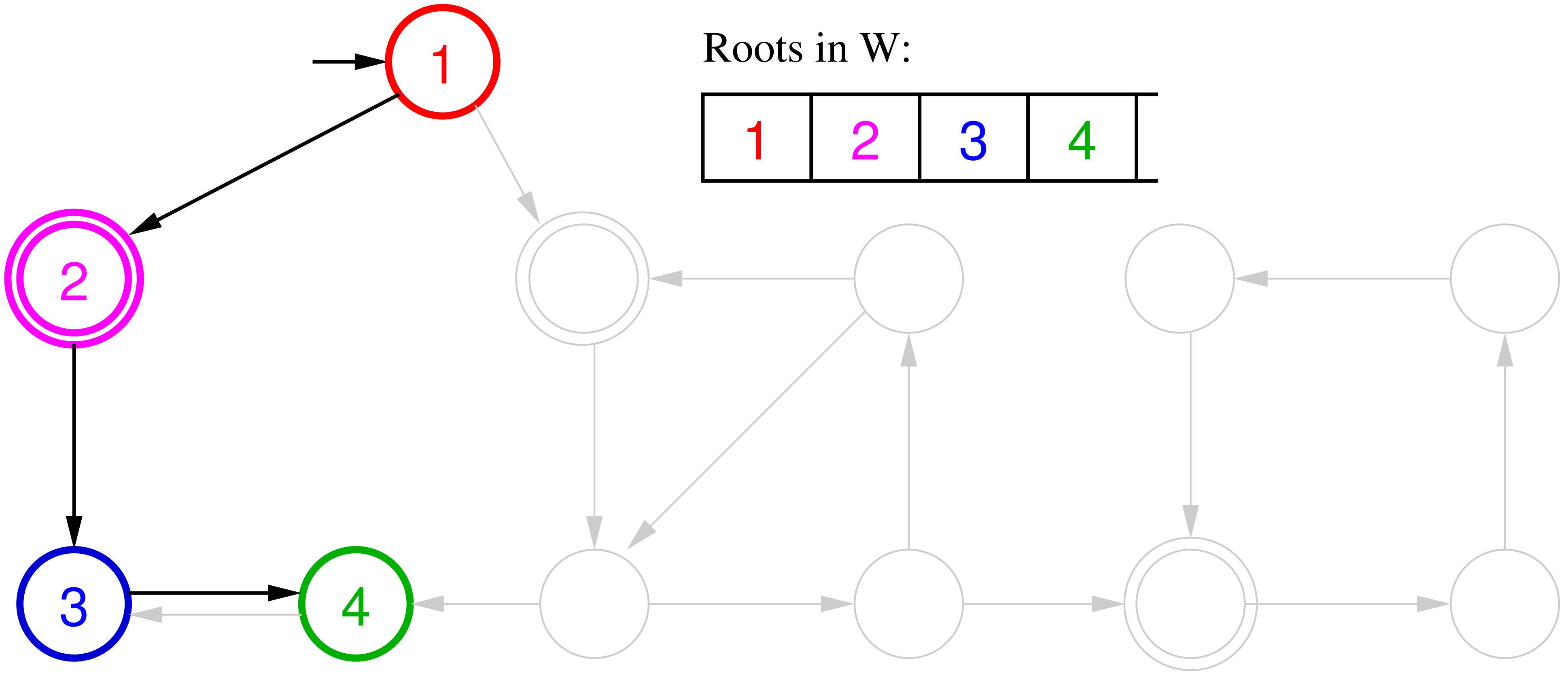
Run-time linear in number of states plus number of transitions.

Example: Execution of the algorithm



Situation at the beginning, only s_0 explored.

Example: Execution of the algorithm

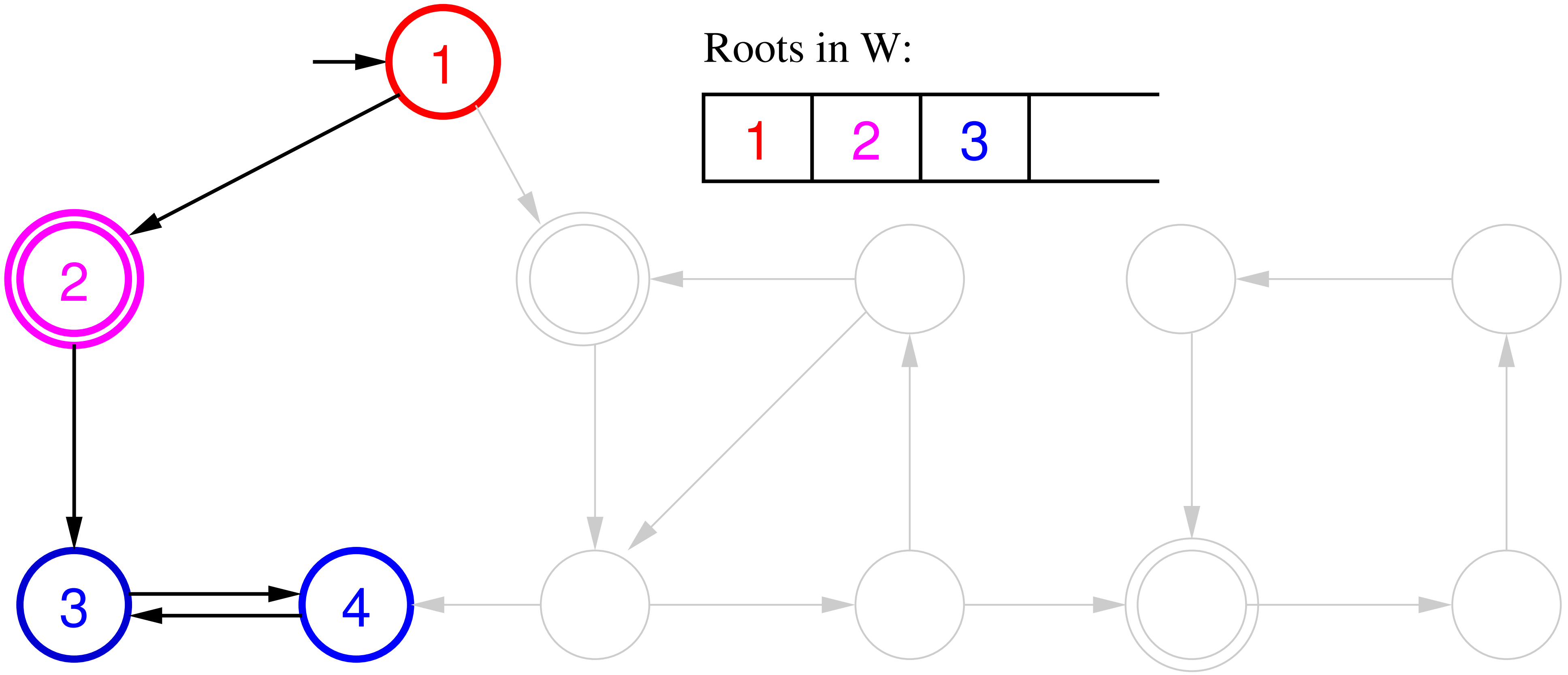


Roots in W:

1	2	3	4	

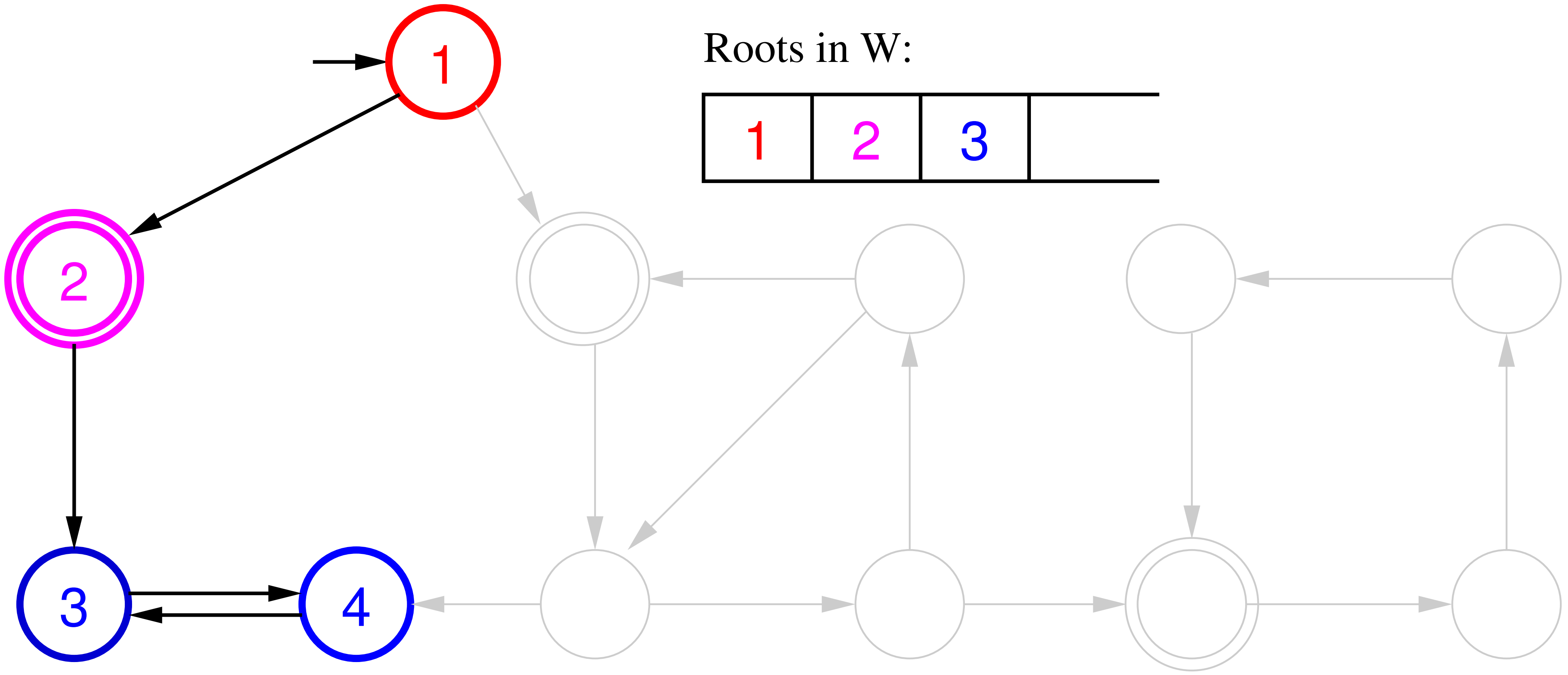
Situation after discovering three edges.

Example: Execution of the algorithm



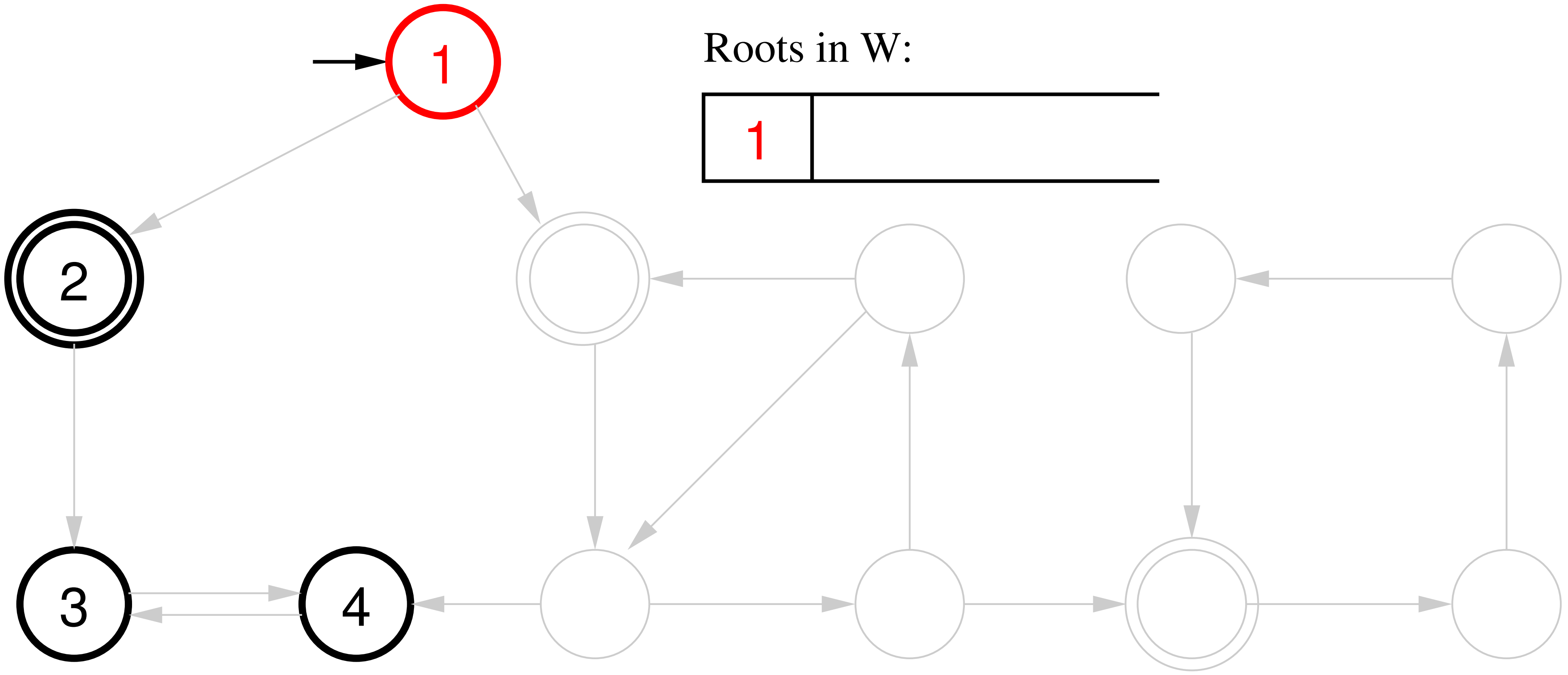
Edge $4 \rightarrow 3$ leads to merger of two SCCs.

Example: Execution of the algorithm



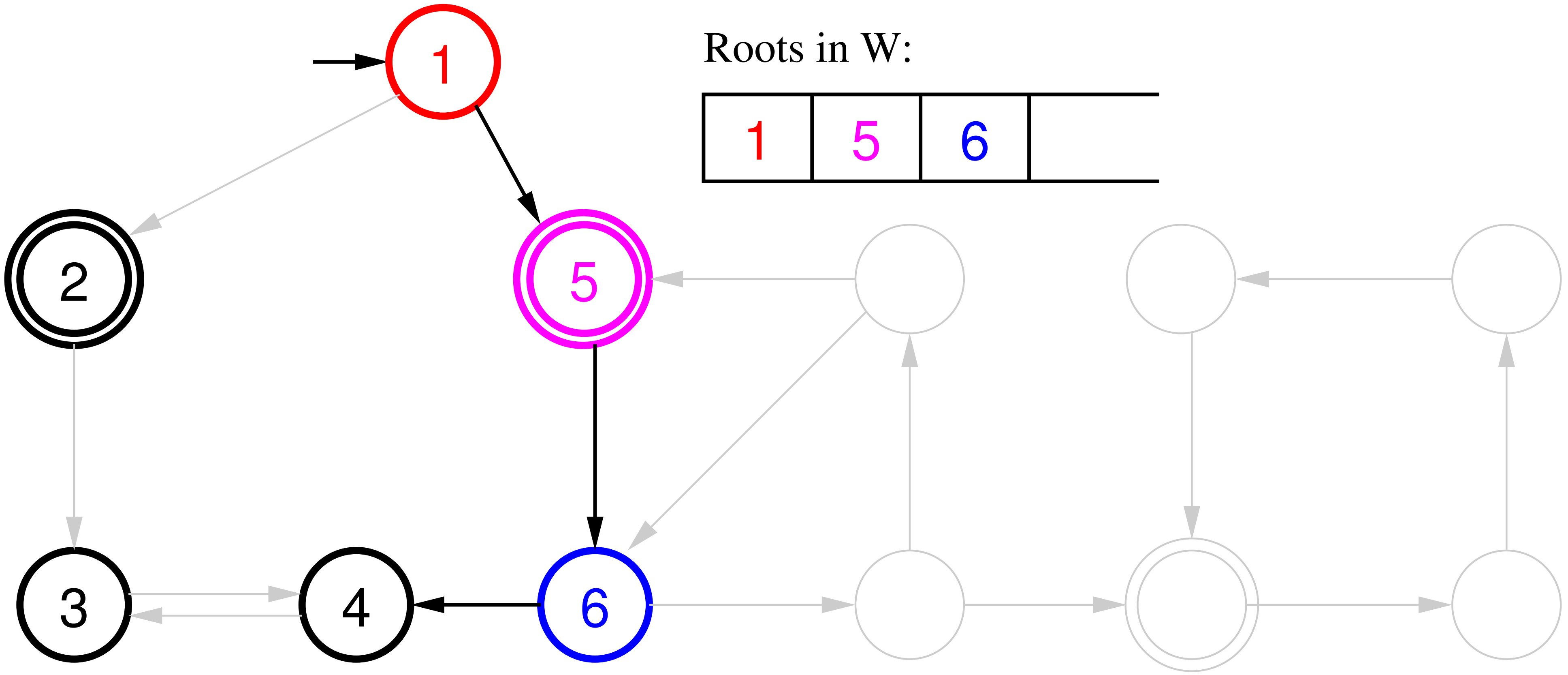
(set component of each entry in *W* indicated by colours)

Example: Execution of the algorithm



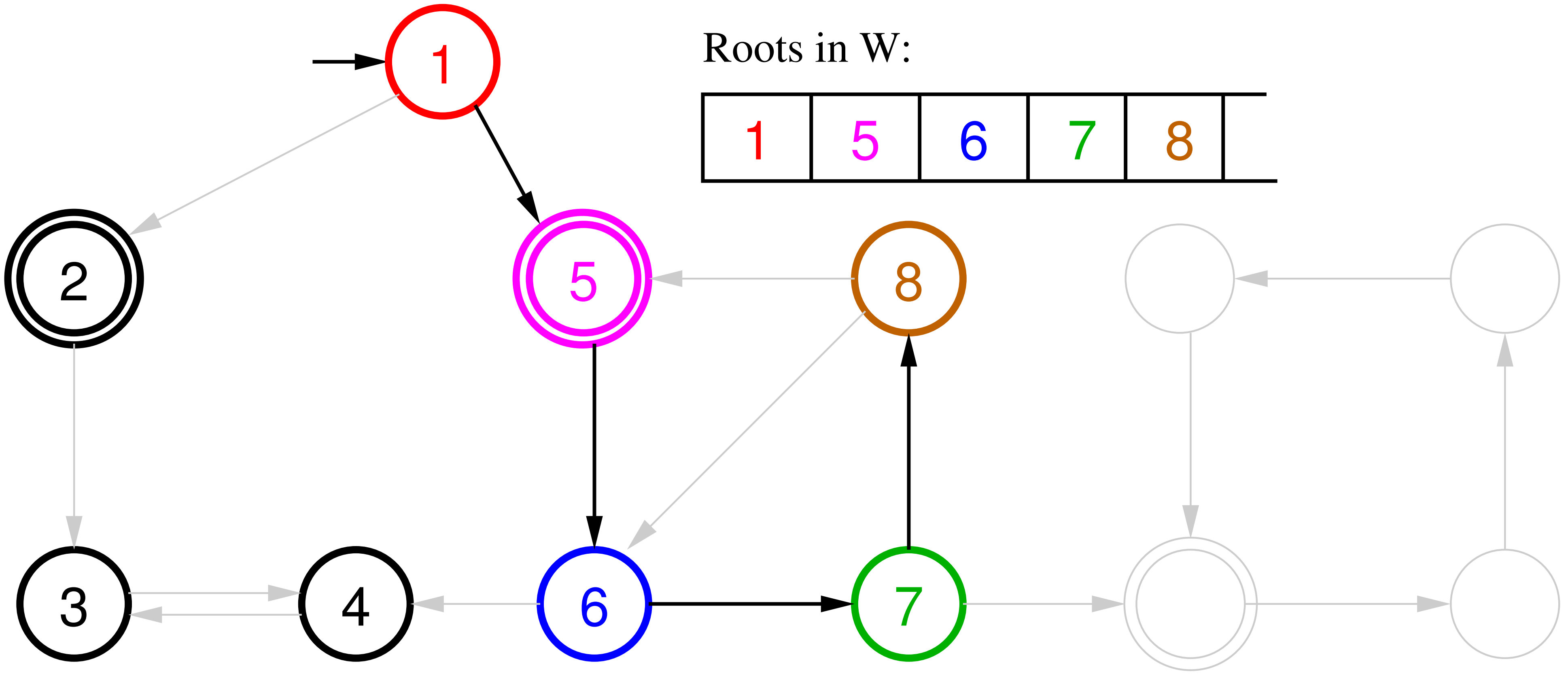
Backtracking makes 2, 3, and 4 inactive (shown in black).

Example: Execution of the algorithm



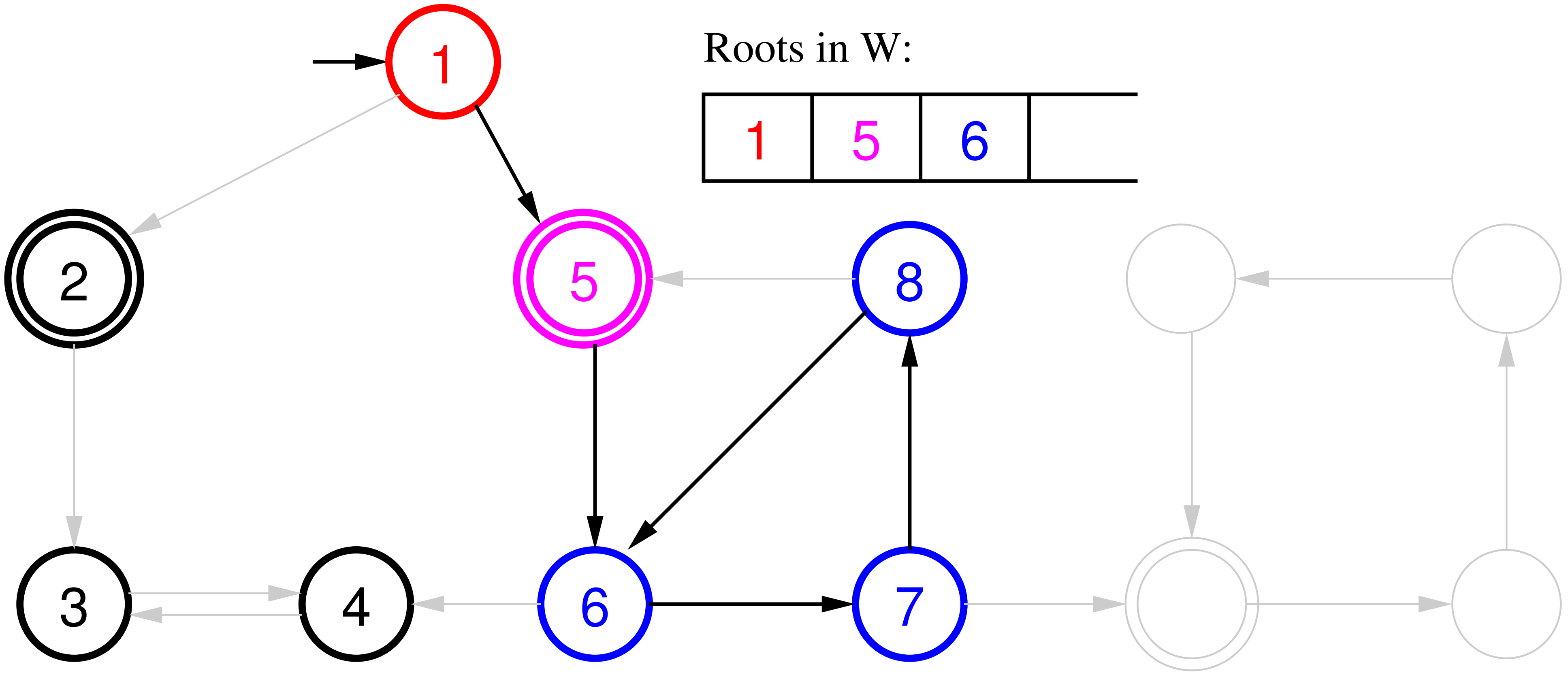
Edge $6 \rightarrow 4$ is an example of Case 2 and may be ignored.

Example: Execution of the algorithm



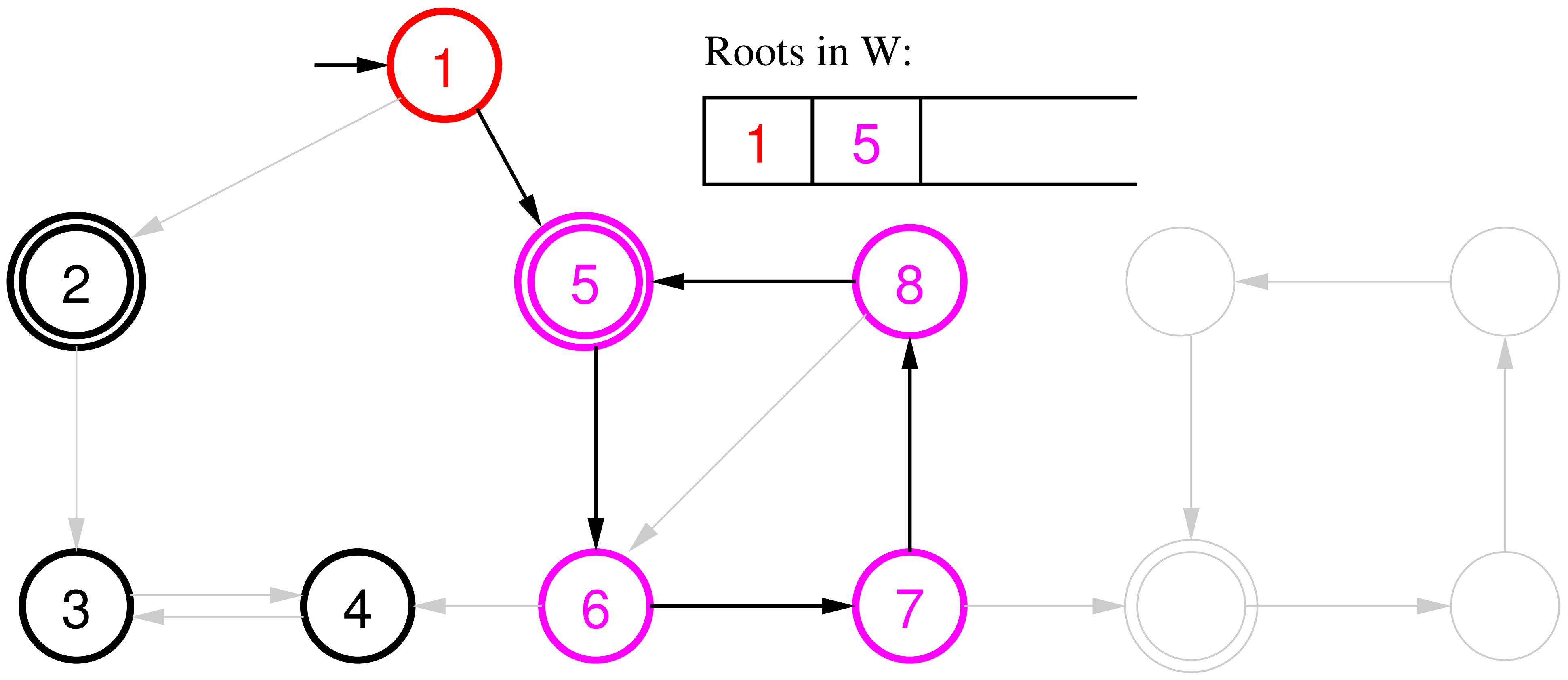
Situation when reaching 8.

Example: Execution of the algorithm



Edge 8 $\rightarrow$ 6 leads to a merger.

Example: Execution of the algorithm



Edge 8 $\rightarrow$ 5: Counterexample discovered because root 5 is accepting.

Extension to generalized Büchi automata

Let $\mathcal{G}$ be a GBA with n acceptance sets $F_1, \ldots, F_n$.

$\mathcal{L}(\mathcal{G})$ is non-empty iff there exists a non-trivial SCC intersecting each set F_i ($1 \leq i \leq n$).

Let us label each state s with the index set of the acceptance sets it is contained in, denoted M_s. (E.g., if s in F_1 and in F_3, but in no other acceptance set, then $M_s = \{1, 3\}$.)

We extend W by a third component, an index set, i.e. a subset of $\{1, \ldots, n\}$.

During the algorithm, we uphold the following invariant: if W has an entry (s, C, M), then $M = \bigcup_{t \in C} M_t$.

When two SCCs are merged, we take the union of the index sets.

A counterexample is discovered if this leads to an index set $\{1, \ldots, n\}$.

If n is “small”, the required operations can be implemented using bit vectors (constant time).

Modification for computing SCCs

The algorithm can also be used to partition the BA (or, in fact, any directed graph) into its SCCs.

For this, we simply omit the acceptance test when merging active SCCs.

The algorithm may output a complete SCC as soon as one backtracks from its root.