System Specification, Verification and Synthesis (SSVS) – CS 4830/7485, Fall 2019

17: Formal Verification: LTL Model Checking

Stavros Tripakis

Northeastern University
Khoury College of Computer Sciences

Recall: the model-checking problem for LTL

Given:

- the implementation: a transition system (Kripke structure) $M = (\mathsf{AP}, S, S_0, L, R)$
- the specification: an LTL formula ϕ

check where M satisfies ϕ:

$$M \stackrel{?}{\models} \phi$$

i.e., check whether for **every** infinite trace generated by M satisfies ϕ.

Recall: the model-checking problem for LTL

Given:

- the implementation: a transition system (Kripke structure) $M = (\mathsf{AP}, S, S_0, L, R)$
- the specification: an LTL formula ϕ

check where M satisfies ϕ:

$$M \stackrel{?}{\models} \phi$$

i.e., check whether for **every** infinite trace generated by M satisfies ϕ.

We will assume that M is finite and has no deadlock states.
What if M has deadlocks? $\rightarrow$ **Homework.**

LTL Model-Checking: the Automata-Theoretic Approach

LTL model-checking: basic idea

To check whether $M \models \phi$:

1. Construct an automaton $A_{\neg\phi}$ which accepts all infinite traces satisfying $\neg\phi$, i.e., **violating** ϕ.
   - $A_{\neg\phi}$ is an ω-**automaton**, i.e., an automaton over infinite words.
   - There are different kinds of ω-automata: Büchi automata, Rabin automata, Street automata, parity automata, ...
2. Compute the **product** of M and $A_{\neg\phi}$, denoted $M \times A_{\neg\phi}$.
   - $M \times A_{\neg\phi}$ is usually of the same type as $A_{\neg\phi}$. For instance, if $A_{\neg\phi}$ is a Büchi automaton, then so is $M \times A_{\neg\phi}$.
   - $M \times A_{\neg\phi}$ captures the **intersection** of the set of traces generated by M, on one hand, and the set of traces accepted by $A_{\neg\phi}$, on the other.
   - Therefore, every trace accepted by $M \times A_{\neg\phi}$ contradicts $M \models \phi$, since it is a trace which can be generated by M, and at the same time violates ϕ.
3. Check that $M \times A_{\neg\phi}$ is **empty**, i.e., accepts no trace.

This is called the **automata theoretic** approach.

LTL model-checking: a language containment problem

$$M \models \phi$$

can be also restated as

$$Traces(M) \subseteq Traces(\phi)$$

where $Traces(M)$ is the set of all traces generated by M, and where $Traces(\phi)$ is the set of all traces satisfying ϕ.

LTL model-checking: a language containment problem

$$M \models \phi$$

can be also restated as

$$Traces(M) \subseteq Traces(\phi)$$

where $Traces(M)$ is the set of all traces generated by M, and where $Traces(\phi)$ is the set of all traces satisfying ϕ.

But

$$Traces(M) \subseteq Traces(\phi)$$

is equivalent to

$$Traces(M) \cap \overline{Traces(\phi)} = \emptyset$$

which is in turn equivalent to

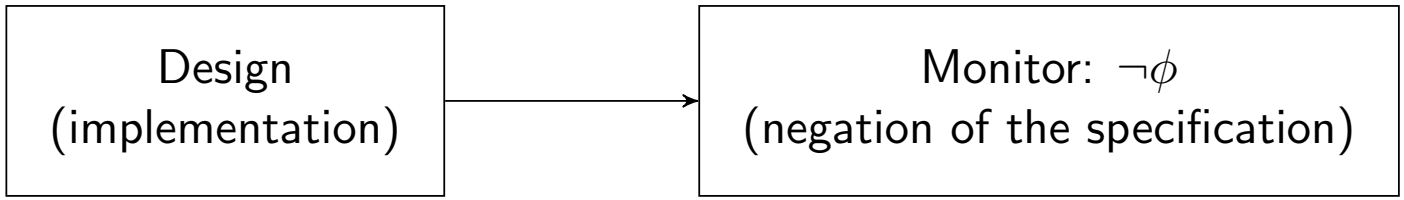
$$Traces(M) \cap Traces(\neg\phi) = \emptyset$$

$M \times A_{\neg\phi}$ is built in such a way so that we have

$$Traces(M \times A_{\neg\phi}) = Traces(M) \cap Traces(\neg\phi)$$

Recall: Monitors

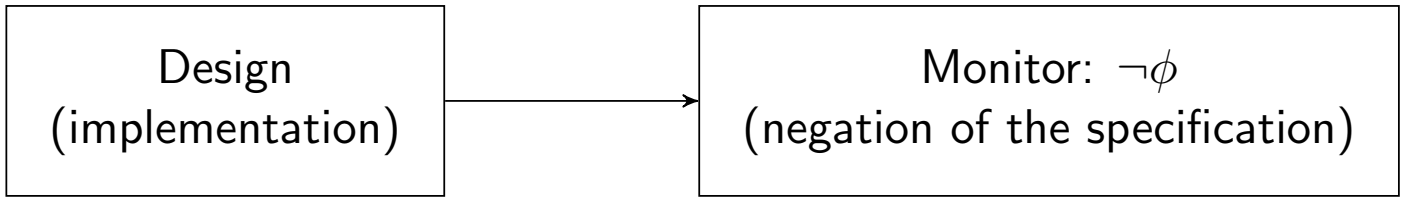
One can think of the automata theoretic approach as verification using monitors:



- The monitor is like a "watchdog": it observes the outputs of Design, and declares an error if ever Design does something wrong.
  - This analogy is appropriate for safety properties where erroneous traces are finite.
- For liveness, erroneous traces are infinite: but since we know the transition system M of Design, the system we are checking is a **white box** $\Rightarrow$ "off-line" verification has access to infinite traces.

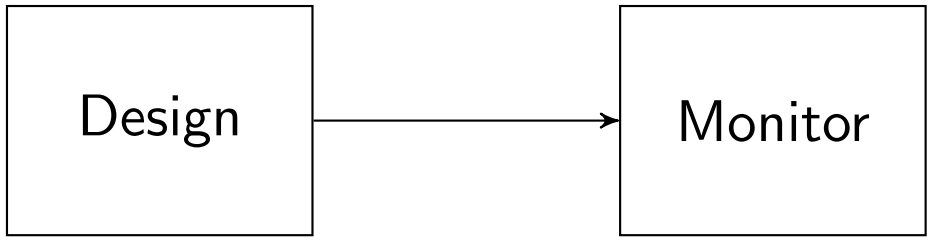
Recall: Monitors

One can think of the automata theoretic approach as verification using monitors:



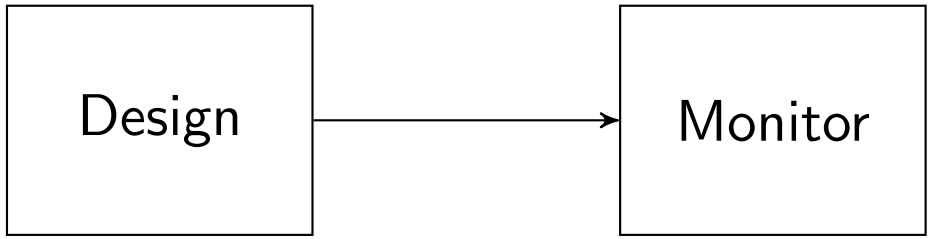
- The monitor is like a "watchdog": it observes the outputs of Design, and declares an error if ever Design does something wrong.
  - This analogy is appropriate for safety properties where erroneous traces are finite.
- For liveness, erroneous traces are infinite: but since we know the transition system M of Design, the system we are checking is a **white box** $\Rightarrow$ "off-line" verification has access to infinite traces.
- The field of **runtime verification** ("on-line") assumes that Design is a **black box**: see `https://www.runtime-verification.org/`.

Verification of safety properties using safety monitors



Using safety monitors, we reduce the model-checking problem for safety properties to checking reachability of error states in the product Design $\times$ Monitor.

What about liveness properties?



- Monitors essentially try to find **counter-examples** = behaviors that violate the property we are interested in.
- What is a counter-example to a liveness property?
  - An infinite behavior.
- Infinite behaviors represented by ω-automata (here we use Büchi).

- But how do we go from LTL into a Büchi automaton?

TRANSLATING LTL TO BÜCHI AUTOMATA

From LTL to Büchi automata

- Every LTL formula can be translated to an equivalent Büchi automaton.
- The resulting automaton is generally non-deterministic: this is unavoidable. Why?

From LTL to Büchi automata

- Every LTL formula can be translated to an equivalent Büchi automaton.
- The resulting automaton is generally non-deterministic: this is unavoidable. Why?
  - Because the LTL formula $\mathbf{FG}b$ cannot be represented as a DBA, as we saw.
- A number of LTL-to-Büchi translation algorithms exist.
- There are tools which implement such translations, e.g.,:
  - Spin
  - http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/

From LTL to Büchi automata

- Every LTL formula can be translated to an equivalent Büchi automaton.
- The resulting automaton is generally non-deterministic: this is unavoidable. Why?
  - Because the LTL formula $\mathbf{FG}b$ cannot be represented as a DBA, as we saw.
- A number of LTL-to-Büchi translation algorithms exist.
- There are tools which implement such translations, e.g.,:
  - Spin
  - http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
- The bad news: the size of the resulting automaton is generally **exponential** in the size of the formula.
- This is unavoidable: there exist LTL formulas of size $O(n^2 \cdot |\mathsf{AP}|)$ which require NBA of size $O(2^n)$.
  Example: $\bigwedge_{p \in \mathsf{AP}} \bigwedge_{0 \leq i < n} (\mathbf{X}^i p \leftrightarrow \mathbf{X}^{i+n} p)$
  Homework: explain why automaton of size $O(2^n)$ is necessary.

From LTL to Büchi automata

The translation works in two steps:

1. From an LTL formula to a GNBA: **Generalized Non-deterministic Büchi Automaton**.
2. From a GNBA to an NBA.

GNBA

Generalized Non-Deterministic Büchi Automata (GNBA)

Büchi automata have a single set of accepting states F:

$$(\Sigma, S, S_0, \Delta, F)$$

and the acceptance criterion is that (at least one state in) F is visited infinitely often.

Generalized NBA: **many** sets of accepting states:

$$(\Sigma, S, S_0, \Delta, \mathcal{F})$$

where

$$\mathcal{F} = \{F_1, F_2, ..., F_k\}$$

is a **set of sets of accepting states**, i.e., $F_i \subseteq S$ for $i = 1, ..., k$.

GNBA: acceptance criteria

GNBA:

$$(\Sigma, S, S_0, \Delta, \mathcal{F})$$

where

$$\mathcal{F} = \{F_1, F_2, ..., F_k\}$$

is a **set of sets of accepting states**, i.e., $F_i \subseteq S$ for every i.

GNBA acceptance condition: a word is accepted if there exists a run where **every** F_i is visited infinitely often.

GNBA: acceptance criteria

GNBA:

$$(\Sigma, S, S_0, \Delta, \mathcal{F})$$

where

$$\mathcal{F} = \{F_1, F_2, ..., F_k\}$$

is a **set of sets of accepting states**, i.e., $F_i \subseteq S$ for every i.

GNBA acceptance condition: a word is accepted if there exists a run where **every** F_i is visited infinitely often.

Note: $\mathcal{F}$ can be empty. What happens in that case?

GNBA: acceptance criteria

GNBA:

$$(\Sigma, S, S_0, \Delta, \mathcal{F})$$

where

$$\mathcal{F} = \{F_1, F_2, ..., F_k\}$$

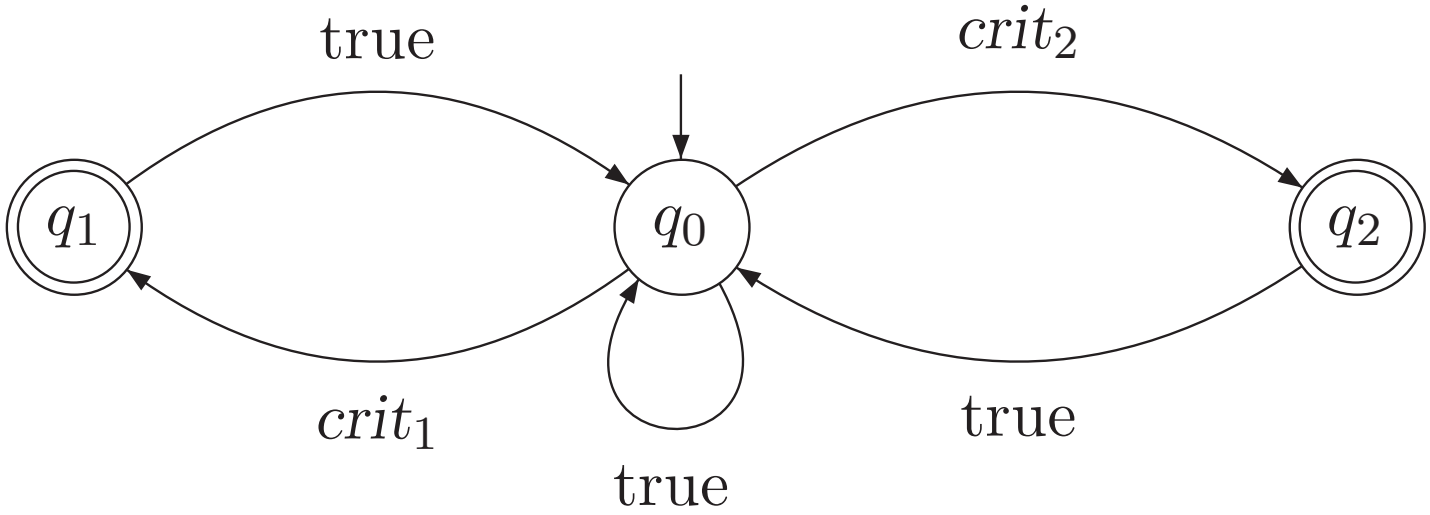
is a **set of sets of accepting states**, i.e., $F_i \subseteq S$ for every i.

GNBA acceptance condition: a word is accepted if there exists a run where **every** F_i is visited infinitely often.

Note: $\mathcal{F}$ can be empty. What happens in that case?
Every word that has a run is accepted.

GNBA: example

In the following, $\mathcal{F} = \{\{q_1\}, \{q_2\}\}$.



q_1 must be visited infinitely often, but q_2 must also be visited infinitely often $\Rightarrow$ "*every process gets access to its critical section infinitely often*".

Picture taken from [Baier and Katoen, 2008].

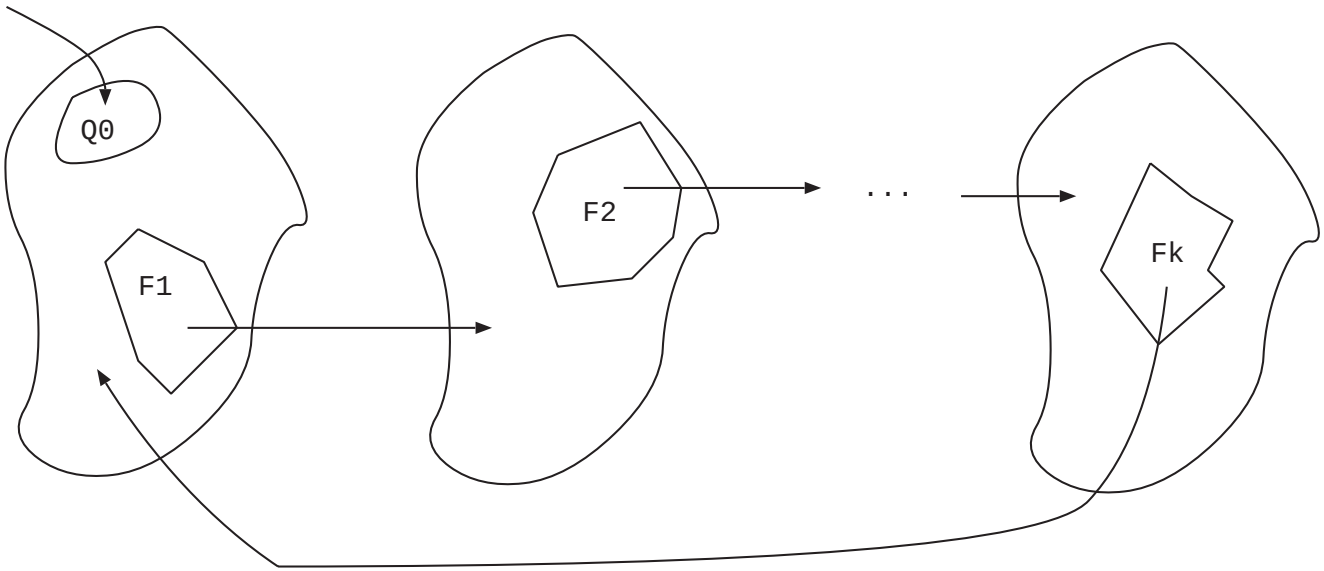
From GNBA to NBA

Every GNBA A_g can be translated to an equivalent NBA A (i.e., accepting the same language).

The translation is **polynomial**: if n is the number of states of A_g, and k is the number of acceptance sets in $\mathcal{F}$, then the number of states of A is $n \cdot k$.

From GNBA to NBA

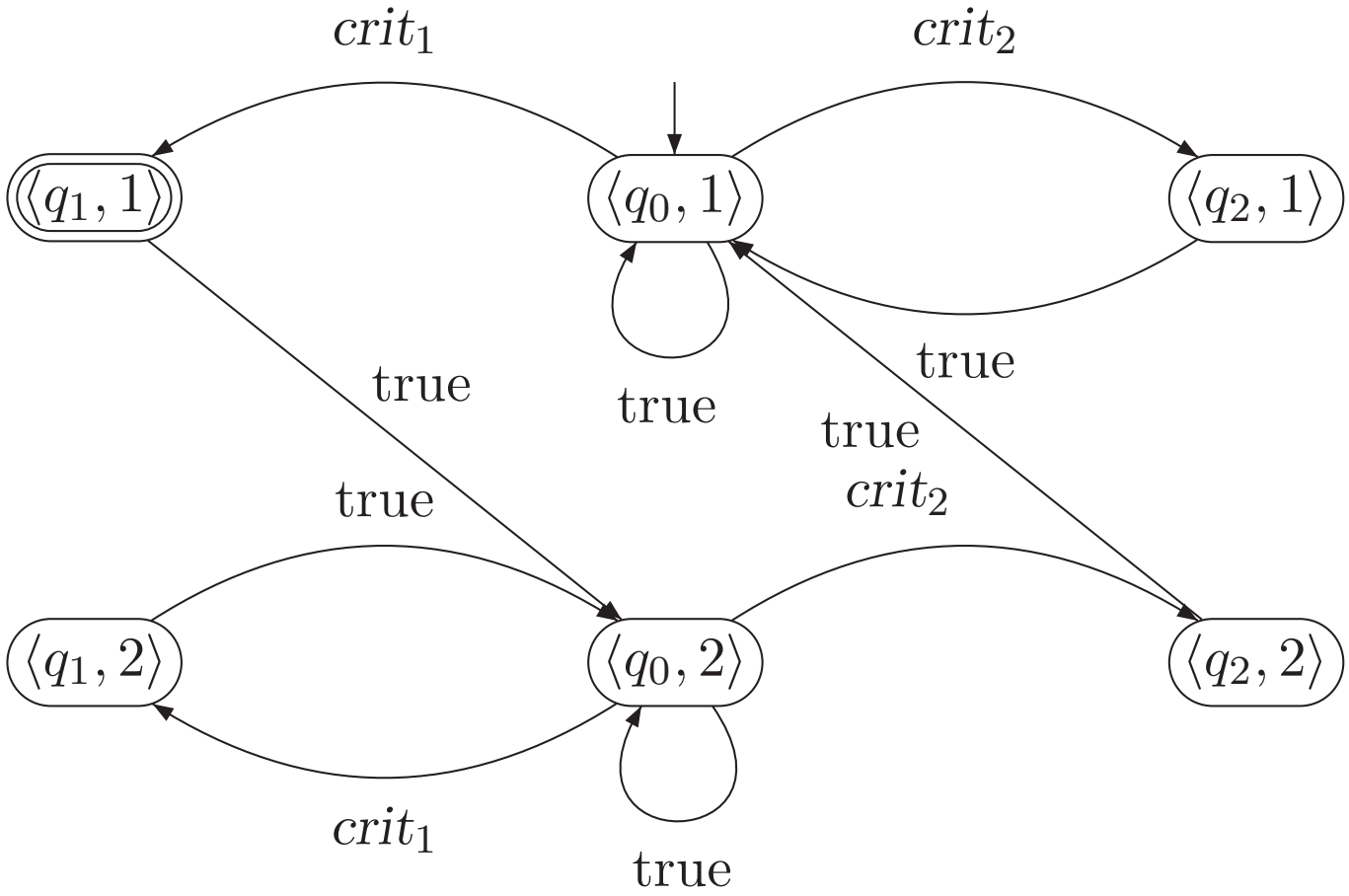
Basic idea of the translation: use a modulo-k counter $i \in \{1, ..., k\}$ which keeps track of which set F_i we visited last and makes sure sets are visited in the order $F_1, F_2, ..., F_k$.



Picture taken from [Baier and Katoen, 2008].

GNBA to NBA translation: example

For the GNBA shown earlier, this is the resulting NBA:



Picture taken from [Baier and Katoen, 2008].

TRANSLATING LTL TO GNBA

Translating an LTL formula ϕ to a GNBA A

Basic idea:

- States of A are **sets of LTL formulas** (subformulas of ϕ or their negation).
- E.g., for $p \,\mathbf{U}\, q$, a state of A could be the set

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

- Meaning of a state $s = \{\phi_1, \phi_2, ..., \phi_k\}$: **every** trace starting at s must satisfy **each** ϕ_i.
- If some ϕ_i is not satisfied already at some state s (i.e., at the first step) then s makes a *"promise"* for the future $\Rightarrow$ use GNBA acceptance criteria to avoid postponing these promises forever.
- Transitions also make sure that promises are correctly carried over to the next step.

Translating an LTL formula ϕ to a GNBA A

Basic idea:

- States of A are **sets of LTL formulas** (subformulas of ϕ or their negation).
- E.g., for $p \,\mathbf{U}\, q$, a state of A could be the set

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

- Meaning of a state $s = \{\phi_1, \phi_2, ..., \phi_k\}$: **every** trace starting at s must satisfy **each** ϕ_i.
- If some ϕ_i is not satisfied already at some state s (i.e., at the first step) then s makes a *"promise"* for the future $\Rightarrow$ use GNBA acceptance criteria to avoid postponing these promises forever.
- Transitions also make sure that promises are correctly carried over to the next step.

For simplicity, we assume that ϕ only contains atomic propositions, $\wedge$, $\neg$, $\mathbf{X}$ and $\mathbf{U}$. This is without loss of generality since we know that other operators can be derived from the above.

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

Examples:

$$\mathbf{clo}(p\,\mathbf{U}\,q) \quad = \quad \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\}$$

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

Examples:

$$\begin{aligned} \mathbf{clo}(p\,\mathbf{U}\,q) &= \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\} \\ \mathbf{clo}(p\,\mathbf{U}\,\mathbf{X}q) &= \end{aligned}$$

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

Examples:

$$\begin{aligned}
\mathbf{clo}(p\,\mathbf{U}\,q) &= \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\} \\
\mathbf{clo}(p\,\mathbf{U}\,\mathbf{X}q) &= \{p\,\mathbf{U}\,\mathbf{X}q, \neg(p\,\mathbf{U}\,\mathbf{X}q), p, \neg p, \mathbf{X}q, \neg\mathbf{X}q, q, \neg q\}
\end{aligned}$$

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

Examples:

$$\begin{aligned}
\mathbf{clo}(p\,\mathbf{U}\,q) &= \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\} \\
\mathbf{clo}(p\,\mathbf{U}\,\mathbf{X}q) &= \{p\,\mathbf{U}\,\mathbf{X}q, \neg(p\,\mathbf{U}\,\mathbf{X}q), p, \neg p, \mathbf{X}q, \neg\mathbf{X}q, q, \neg q\} \\
\mathbf{clo}(p\,\mathbf{U}\,\neg\mathbf{X}q) &=
\end{aligned}$$

The closure of an LTL formula ϕ

$$\mathbf{clo}(\phi) = \{\text{the set of all subformulas of } \phi \text{ and their negations}\}$$

If $\neg\psi$ is a subformula of ϕ we add $\neg\psi$ and ψ (instead of $\neg\neg\psi$) to $\mathbf{clo}(\phi)$.

Examples:

$$\begin{aligned}
\mathbf{clo}(p\,\mathbf{U}\,q) &= \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\} \\
\mathbf{clo}(p\,\mathbf{U}\,\mathbf{X}q) &= \{p\,\mathbf{U}\,\mathbf{X}q, \neg(p\,\mathbf{U}\,\mathbf{X}q), p, \neg p, \mathbf{X}q, \neg\mathbf{X}q, q, \neg q\} \\
\mathbf{clo}(p\,\mathbf{U}\,\neg\mathbf{X}q) &= \{p\,\mathbf{U}\,\neg\mathbf{X}q, \neg(p\,\mathbf{U}\,\neg\mathbf{X}q), p, \neg p, \mathbf{X}q, \neg\mathbf{X}q, q, \neg q\}
\end{aligned}$$

Legal subsets of $\mathbf{clo}(\phi)$

The states of the NBA will be subsets of $\mathbf{clo}(\phi)$.

But not all subsets of $\mathbf{clo}(\phi)$ are **legal** for this purpose.

For example, consider

$$\mathbf{clo}(p\,\mathbf{U}\,q) \quad = \quad \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\}$$

The subset

$$B_1 \quad = \quad \{p\,\mathbf{U}\,q, p, \neg p\}$$

is illegal, because it contains both p and $\neg p$. Since the meaning is that we promise to satisfy all formulas in B_1, and we cannot satisfy both p and $\neg p$, B_1 is illegal.

Legal subsets of $\mathbf{clo}(\phi)$

For example, consider

$$\mathbf{clo}(p\,\mathbf{U}\,q) \quad = \quad \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\}$$

Similarly, the subset

$$B_2 \quad = \quad \{p\,\mathbf{U}\,q, \neg p, \neg q\}$$

is illegal. Why?

Legal subsets of $\mathbf{clo}(\phi)$

For example, consider

$$\mathbf{clo}(p\,\mathbf{U}\,q) \quad = \quad \{p\,\mathbf{U}\,q, \neg(p\,\mathbf{U}\,q), p, \neg p, q, \neg q\}$$

Similarly, the subset

$$B_2 \quad = \quad \{p\,\mathbf{U}\,q, \neg p, \neg q\}$$

is illegal. Why?

Because in order to satisfy $p\,\mathbf{U}\,q$, either q must hold now, or p must hold now (and continuously until q holds in the future). Since neither p nor q hold in B_2, B_2 is illegal.

Recursive equivalences for the until operator

We will make use of the following fundamental equivalences for the until operator:

$$p \,\mathbf{U}\, q \quad \Leftrightarrow \quad q \vee (p \wedge \mathbf{X}(p \,\mathbf{U}\, q))$$

$$\neg(p \,\mathbf{U}\, q) \quad \Leftrightarrow \quad \neg q \wedge (\neg p \vee \mathbf{X}\neg(p \,\mathbf{U}\, q))$$

both when defining legal subsets of $\mathbf{clo}(\phi)$ and also when defining the transitions of the resulting NBA.

Legal subsets of $\mathbf{clo}(\phi)$

A subset $B \subseteq \mathbf{clo}(\phi)$ is **legal** if it satisfies the following conditions:

1. B is **logically consistent**, that is,
   - (a) For every non-negated subformula $\psi \in \mathbf{clo}(\phi)$, if $\psi \in B$ then $\neg\psi \notin B$.
   - (b) For every subformula $\psi_1 \wedge \psi_2 \in \mathbf{clo}(\phi)$, $\psi_1 \wedge \psi_2 \in B$ iff $\psi_1 \in B$ and $\psi_2 \in B$.

2. B is **maximal**, that is, for all $\psi \in \mathbf{clo}(\phi)$, if $\psi \notin B$ then $\neg\psi \in B$.
   Together with condition 1(a), this means that for each subformula ψ of ϕ, a state must promise to satisfy either ψ or its negation.

3. B is **consistent with respect to until**, that is,
   - (a) If $\psi_1 \,\mathbf{U}\, \psi_2 \in B$ then either $\psi_1 \in B$ or $\psi_2 \in B$ (or both).
   - (b) If $\neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B$ then $\neg\psi_2 \in B$.

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p\,\mathbf{U}\,(\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal?

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal?

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal? No. It is inconsistent, because it contains $\neg p \wedge q$ and thus should also contain $\neg p$.

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal? No. It is inconsistent, because it contains $\neg p \wedge q$ and thus should also contain $\neg p$.
- Let $B_5 = \{\neg p, q, \neg(\neg p \wedge q), \phi\}$. Is B_5 legal?

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal? No. It is inconsistent, because it contains $\neg p \wedge q$ and thus should also contain $\neg p$.
- Let $B_5 = \{\neg p, q, \neg(\neg p \wedge q), \phi\}$. Is B_5 legal? No. It is inconsistent, because it does not contain $\neg p \wedge q$ (it contains its negation) and therefore should not contain both $\neg p$ and q.

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal? No. It is inconsistent, because it contains $\neg p \wedge q$ and thus should also contain $\neg p$.
- Let $B_5 = \{\neg p, q, \neg(\neg p \wedge q), \phi\}$. Is B_5 legal? No. It is inconsistent, because it does not contain $\neg p \wedge q$ (it contains its negation) and therefore should not contain both $\neg p$ and q.
- Let $B_6 = \{\neg p, \neg q, \neg(\neg p \wedge q), \phi\}$. Is B_6 legal?

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p \,\mathbf{U}\, (\neg p \wedge q)$.

- Let $B_3 = \{p, q, \phi\}$. Is B_3 legal? No. It is consistent, but not maximal, because it does not contain $\neg(\neg p \wedge q)$.
- Let $B_4 = \{p, q, \neg p \wedge q, \phi\}$. Is B_4 legal? No. It is inconsistent, because it contains $\neg p \wedge q$ and thus should also contain $\neg p$.
- Let $B_5 = \{\neg p, q, \neg(\neg p \wedge q), \phi\}$. Is B_5 legal? No. It is inconsistent, because it does not contain $\neg p \wedge q$ (it contains its negation) and therefore should not contain both $\neg p$ and q.
- Let $B_6 = \{\neg p, \neg q, \neg(\neg p \wedge q), \phi\}$. Is B_6 legal? No. It is inconsistent w.r.t. until.

Examples taken from [Baier and Katoen, 2008].

Legal subsets of $\mathbf{clo}(\phi)$: more examples

Let $\phi = p\,\mathbf{U}\,(\neg p \wedge q)$.

The following subsets of $\mathbf{clo}(\phi)$ are legal:

$$
\begin{aligned}
B_7 &= \{p, q, \neg(\neg p \wedge q), \phi\} \\
B_8 &= \{p, q, \neg(\neg p \wedge q), \neg\phi\} \\
B_9 &= \{p, \neg q, \neg(\neg p \wedge q), \phi\} \\
B_{10} &= \{p, \neg q, \neg(\neg p \wedge q), \neg\phi\} \\
B_{11} &= \{\neg p, \neg q, \neg(\neg p \wedge q), \neg\phi\} \\
B_{12} &= \{\neg p, q, \neg p \wedge q, \phi\}
\end{aligned}
$$

Examples taken from [Baier and Katoen, 2008].

The LTL to GNBA construction

Let ϕ be an LTL formula over set of atomic propositions AP.
We construct from ϕ a generalized Büchi automaton $A = (\mathsf{AP}, S, S_0, L, \Delta, \mathcal{F})$ with labeling on the states:

$$L : S \to 2^{\mathsf{AP}}$$

A is constructed as follows:

- The set of states S is the set of all legal subsets of $\mathbf{clo}(\phi)$.
- The set of initial states S_0 is the set of all states that contain ϕ.

  Initially, we promise to satisfy ϕ.
- For $B \in S$, we define

$$L(B) = B \cap \mathsf{AP}$$

  The atomic propositions holding in a state B are those which are included in the set B.

The LTL to GNBA construction (continued)

From LTL formula ϕ we construct a generalized Büchi automaton $A = (\mathsf{AP}, S, S_0, L, \Delta, \mathcal{F})$.

- Δ contains a transition $B \longrightarrow B'$ iff all the following conditions hold:
  - if $\mathbf{X}\psi \in B$ then $\psi \in B'$
    if we promise to satisfy $\mathbf{X}\psi$ now, then we must promise to satisfy ψ in the next step
  - if $\neg\mathbf{X}\psi \in B$ then $\neg\psi \in B'$
    if we promise to satisfy $\neg\mathbf{X}\psi$ now, then we must promise to satisfy $\neg\psi$ in the next step
  - if $\psi_1 \,\mathbf{U}\, \psi_2 \in B$ and $\psi_2 \notin B$ then $\psi_1 \,\mathbf{U}\, \psi_2 \in B'$
    because $\psi_1 \,\mathbf{U}\, \psi_2 \Leftrightarrow \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \,\mathbf{U}\, \psi_2))$
  - if $\neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B$ and $\psi_1 \in B$ then $\neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B'$
    because $\neg(\psi_1 \,\mathbf{U}\, \psi_2) \Leftrightarrow \neg\psi_2 \wedge (\neg\psi_1 \vee \mathbf{X}\neg(\psi_1 \,\mathbf{U}\, \psi_2))$

The LTL to GNBA construction (continued)

From LTL formula ϕ we construct a generalized Büchi automaton $A = (\mathsf{AP}, S, S_0, L, \Delta, \mathcal{F})$.

- The GNBA acceptance condition is $\mathcal{F} = \{F_{\psi_1 \,\mathbf{U}\, \psi_2} \mid \psi_1 \,\mathbf{U}\, \psi_2 \in \mathbf{clo}(\phi)\}$ where

$$F_{\psi_1 \,\mathbf{U}\, \psi_2} = \{B \in S \mid \psi_2 \in B \text{ or } \neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B\}$$

For every until subformula, there is a separate acceptance set F_i in $\mathcal{F}$.
What do we want to avoid?
We should not keep postponing our promise to satisfy $\psi_1 \,\mathbf{U}\, \psi_2$.

That is, we should not accept a run $B_0, B_1, B_2, \cdots$, such that after some point i, each B_i contains $\psi_1 \,\mathbf{U}\, \psi_2$ but not ψ_2.

Taking the negation of this, we get that infinitely many B_i's must contain either $\neg(\psi_1 \,\mathbf{U}\, \psi_2)$ or ψ_2.

The LTL to GNBA construction (continued)

From LTL formula ϕ we construct a generalized Büchi automaton $A = (\mathsf{AP}, S, S_0, L, \Delta, \mathcal{F})$.

- The GNBA acceptance condition is $\mathcal{F} = \{F_{\psi_1 \,\mathbf{U}\, \psi_2} \mid \psi_1 \,\mathbf{U}\, \psi_2 \in \mathbf{clo}(\phi)\}$ where

$$F_{\psi_1 \,\mathbf{U}\, \psi_2} = \{B \in S \mid \psi_2 \in B \text{ or } \neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B\}$$

  For every until subformula, there is a separate acceptance set F_i in $\mathcal{F}$.
  What do we want to avoid?
  We should not keep postponing our promise to satisfy $\psi_1 \,\mathbf{U}\, \psi_2$.

  That is, we should not accept a run $B_0, B_1, B_2, \cdots$, such that after some point i, each B_i contains $\psi_1 \,\mathbf{U}\, \psi_2$ but not ψ_2.

  Taking the negation of this, we get that infinitely many B_i's must contain either $\neg(\psi_1 \,\mathbf{U}\, \psi_2)$ or ψ_2.

Quiz: What if $\mathbf{clo}(\phi)$ has no untils?

The LTL to GNBA construction (continued)

From LTL formula ϕ we construct a generalized Büchi automaton $A = (\mathsf{AP}, S, S_0, L, \Delta, \mathcal{F})$.

- The GNBA acceptance condition is $\mathcal{F} = \{F_{\psi_1 \,\mathbf{U}\, \psi_2} \mid \psi_1 \,\mathbf{U}\, \psi_2 \in \mathbf{clo}(\phi)\}$ where

$$F_{\psi_1 \,\mathbf{U}\, \psi_2} = \{B \in S \mid \psi_2 \in B \text{ or } \neg(\psi_1 \,\mathbf{U}\, \psi_2) \in B\}$$

  For every until subformula, there is a separate acceptance set F_i in $\mathcal{F}$.
  What do we want to avoid?
  We should not keep postponing our promise to satisfy $\psi_1 \,\mathbf{U}\, \psi_2$.

  That is, we should not accept a run $B_0, B_1, B_2, \cdots$, such that after some point i, each B_i contains $\psi_1 \,\mathbf{U}\, \psi_2$ but not ψ_2.

  Taking the negation of this, we get that infinitely many B_i's must contain either $\neg(\psi_1 \,\mathbf{U}\, \psi_2)$ or ψ_2.

Quiz: What if $\mathbf{clo}(\phi)$ has no untils? Then $\mathcal{F}$ is empty: ϕ is safety, and the GNBA accepts every word that has an infinite run.

The LTL to GNBA construction: examples

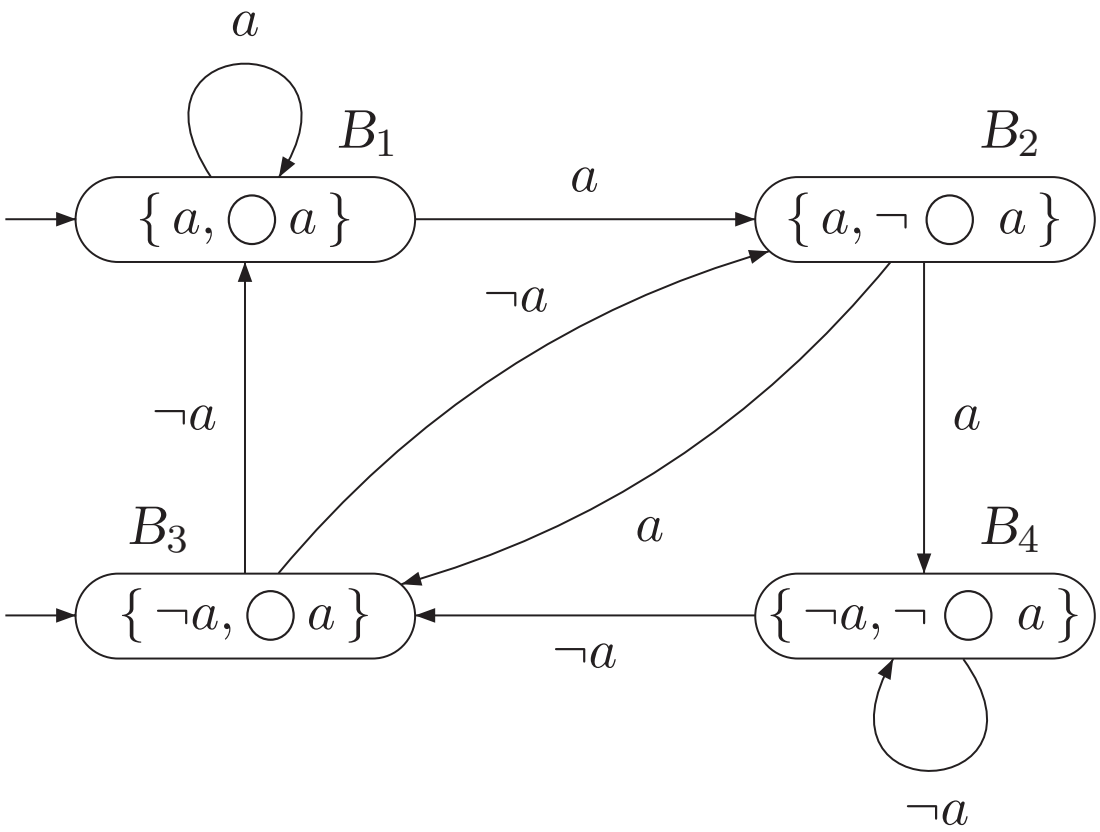
Let's construct GNBA for the following LTL formulas:

$$\mathbf{X}a$$

and

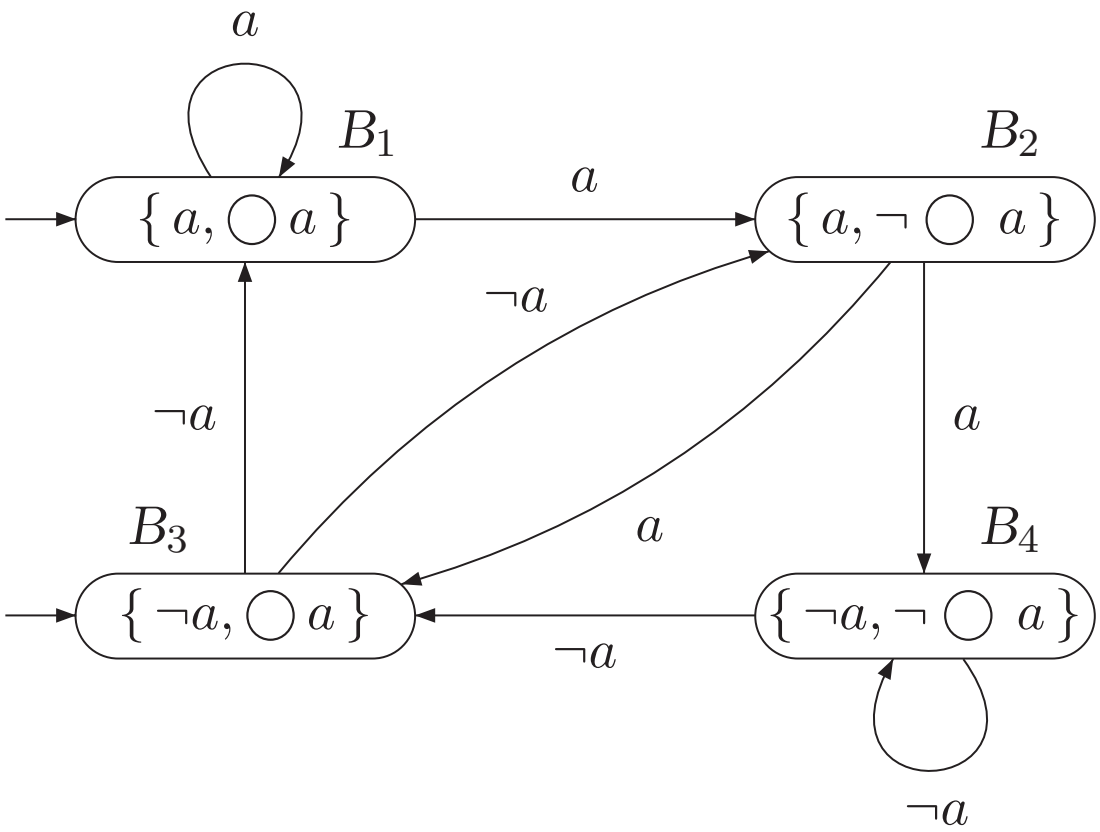
$$a\,\mathbf{U}\,b$$

The LTL to GNBA construction: the GNBA for $\mathbf{X}a$



- Ignore the labels on the transitions (observe that they are identical to the atomic propositions holding at a state).
- How many accepting state sets does this automaton have?

The LTL to GNBA construction: the GNBA for $\mathbf{X}a$

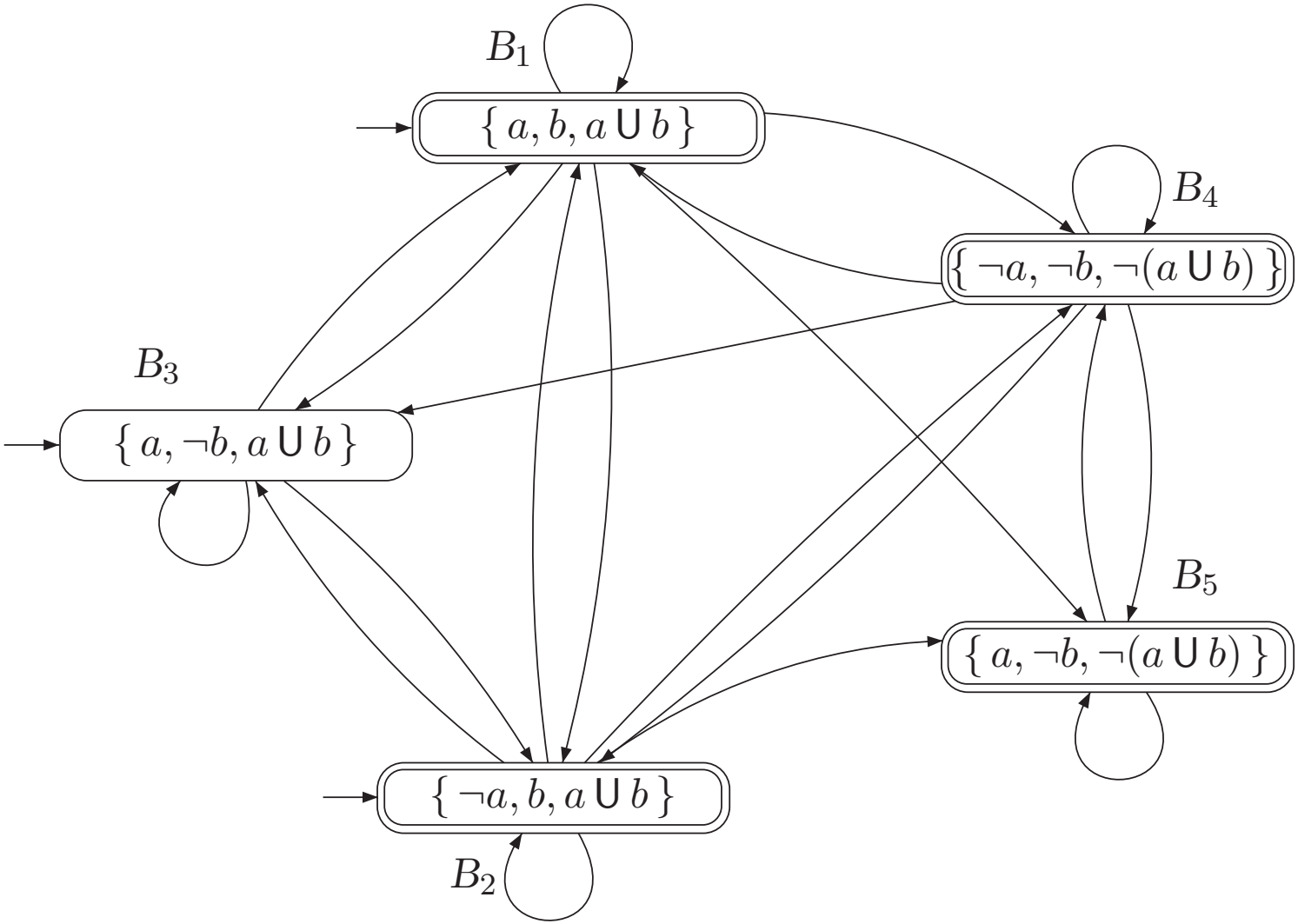


- Ignore the labels on the transitions (observe that they are identical to the atomic propositions holding at a state).
- How many accepting state sets does this automaton have? None.
  Not surprising, since $\mathbf{X}a$ is a safety property.

Picture taken from [Baier and Katoen, 2008].

The LTL to GNBA construction: the GNBA for $a \mathbf{U} b$

(actually an NBA, since there is only one until in the formula)



Picture taken from [Baier and Katoen, 2008].

BACK TO LTL MODEL-CHECKING

Recall: automata-theoretic LTL model-checking

To check whether $M \models \phi$:

1. Construct an automaton $A_{\neg\phi}$ which accepts all infinite traces satisfying $\neg\phi$, i.e., **violating** ϕ.
2. Compute the **product** of M and $A_{\neg\phi}$, denoted $M \times A_{\neg\phi}$.
3. Check that $M \times A_{\neg\phi}$ is **empty**, i.e., accepts no trace.

Computing the product $M \times A$

- AP: a set of atomic propositions.
- $M = (\text{AP}, S, S_0, L, R)$: a transition system over AP.
- $A = (\text{AP}, S^A, S_0^A, L^A, \Delta, F)$: a Büchi automaton with labeling on states. $L^A(s)$ is generally a guard (a subset of AP can also be seen as a guard).

Computing the product $M \times A$

- AP: a set of atomic propositions.
- $M = (\mathsf{AP}, S, S_0, L, R)$: a transition system over AP.
- $A = (\mathsf{AP}, S^A, S_0^A, L^A, \Delta, F)$: a Büchi automaton with labeling on states. $L^A(s)$ is generally a guard (a subset of AP can also be seen as a guard).

The **product** $M \times A$ is a Büchi automaton:

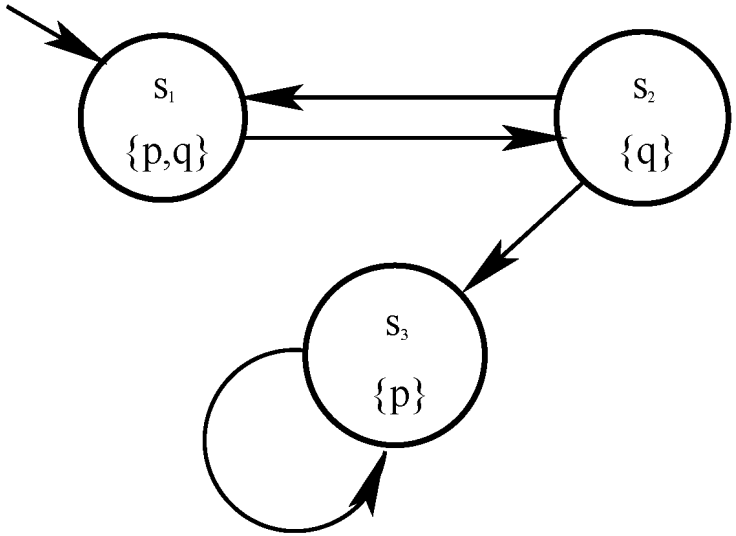
$$M \times A = (\mathsf{AP}, S', S_0', L', \Delta', F')$$

such that

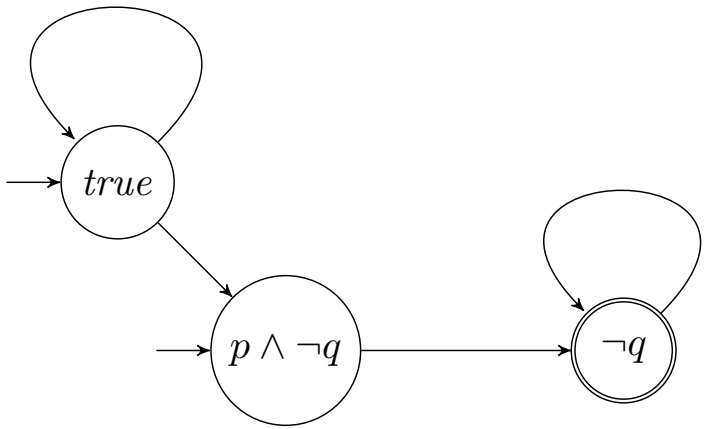
- $S' = \{(s_1, s_2) \in S \times S^A \mid L(s_1) \text{ satisfies } L^A(s_2)\}$
- $S_0' = S' \cap S_0 \times S_0^A$
- $L'(s_1, s_2) = L(s_1)$
- $\Delta' = \{((s_1, s_2), (s_1', s_2')) \in S' \times S' \mid (s_1, s_1') \in R \wedge (s_2, s_2') \in \Delta\}$
- $F' = S' \cap S \times F$

Computing the product $M \times A$: example

Let's compute the product of



with



Recall: automata-theoretic LTL model-checking

To check whether $M \models \phi$:

1. Construct an automaton $A_{\neg\phi}$ which accepts all infinite traces satisfying $\neg\phi$, i.e., **violating** ϕ.
2. Compute the **product** of M and $A_{\neg\phi}$, denoted $M \times A_{\neg\phi}$.
3. Check that $M \times A_{\neg\phi}$ is **empty**, i.e., accepts no trace.

CHECKING BÜCHI AUTOMATA EMPTINESS

Checking Büchi automata emptiness

The Büchi automaton emptiness problem:

- Given: a finite-state Büchi automaton A.
- Check: does A accept any trace?

This is equivalent to checking whether A has at least one accepting run. Why?

Accepting run = infinite run which visits the set of accepting states F infinitely often $\Rightarrow$ since F is finite, there must exist at least one accepting state $s \in F$ which is visited infinitely often.

But the set of states is finite, so what is an infinite run, really?

Viewing A as a finite directed graph (nodes = states, edges = transitions):

infinite run = a **lasso** = a finite **stem** followed by a **cycle**

Checking Büchi automata emptiness

$\Rightarrow$ The Büchi automaton emptiness problem becomes:

- Given:
  - a finite directed graph A with set of nodes S,
  - a set of initial nodes $S_0 \subseteq S$,
  - and a set of accepting nodes $F \subseteq S$.
- Find: whether A has a path

$$s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_r \longrightarrow s_{r+1} \longrightarrow \cdots \longrightarrow s_{r+k}$$

such that

- $s_{r+k} = s_r$, i.e., $s_r \longrightarrow s_{r+1} \longrightarrow \cdots \longrightarrow s_{r+k}$ forms a cycle with **root** node s_r; (this is the cycle of the lasso; the segment $s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_r$ is its tail)
- $s_0 \in S_0$, i.e., the lasso starts in some initial state
- $s_{r+i} \in F$ for some $i \in \{0, ..., k\}$, i.e., the cycle visits at least one accepting node: we call it an **accepting cycle**.

$\Rightarrow$ The problem becomes one of finding accepting cycles in graphs.

Basic algorithms for finding cycles in graphs

How do we find cycles in graphs?

Basic algorithms for finding cycles in graphs

How do we find cycles in graphs?

- Algorithms that find **Strongly-Connected Components** (e.g., Tarjan's).

We will not look at Tarjan's algorithm, as there is a simpler algorithm that works for our purpose.

But it's useful to recall a few things about SCCs.

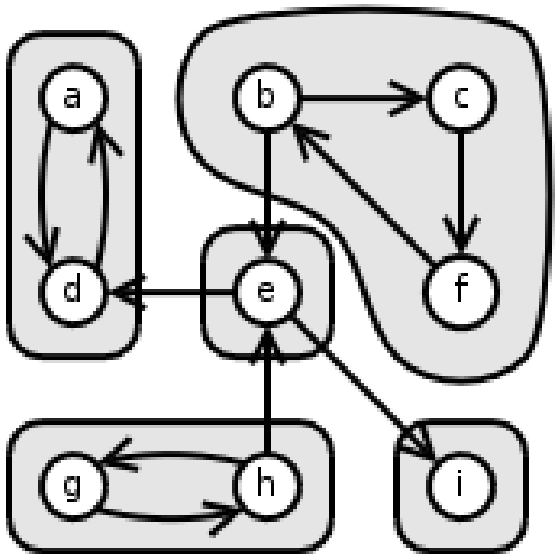
Complexity of Tarjan's SCC algorithm:

$$O(n+m), \text{ where } n = \#\text{nodes}, m = \#\text{edges}.$$

Strongly-connected components

In a directed graph $G = (V, \rightarrow)$, a **strongly-connected component** (SCC) is a subset of nodes $C \subseteq V$, such that every node in C is reachable from every other node in C.

C is called **maximal** if we cannot add more nodes to C and still preserve its SCC property, i.e., $\nexists C' \supset C$ such that C' is also a SCC.

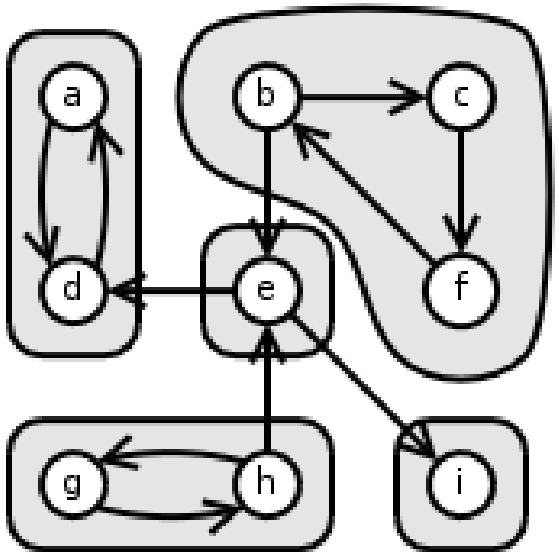


Find the (maximal) SCCs in the above graph.

The acyclic graph of maximal SCCs

The set of all maximal SCCs of a graph defines a new graph, where nodes are maximal SCCs, $C_1, C_2, ..., C_m$.

In the new graph (of maximal SCCs) an edge $C_i \to C_j$ exists iff $C_i \neq C_j$ and there is a node in $v \in C_i$ and a node $u \in C_j$ such that $v \to u$ in the original graph.

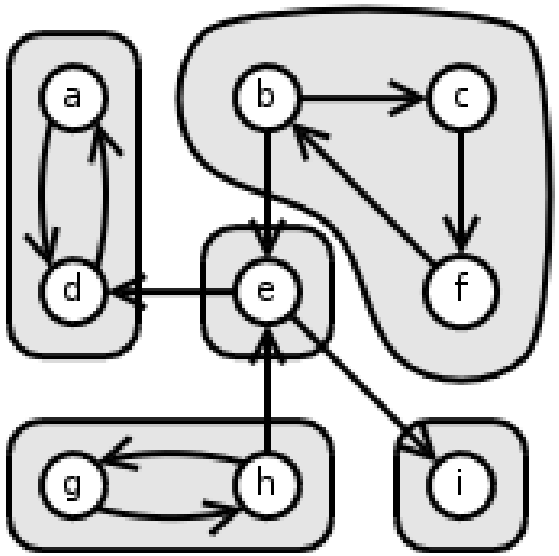


The graph of maximal SCCs is by definition acyclic: why?

The acyclic graph of maximal SCCs

A SCC C is called **terminal** if there is no C' such that $C \to C'$ in the acyclic graph of maximal SCCs.

Otherwise C is called **transient**.

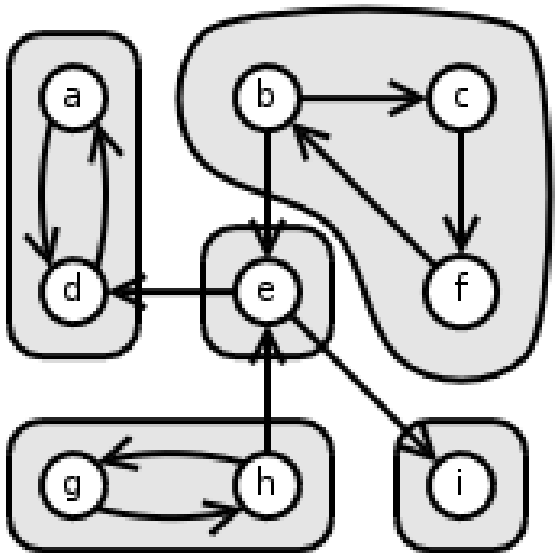


Terminal SCCs:

The acyclic graph of maximal SCCs

A SCC C is called **terminal** if there is no C' such that $C \to C'$ in the acyclic graph of maximal SCCs.

Otherwise C is called **transient**.

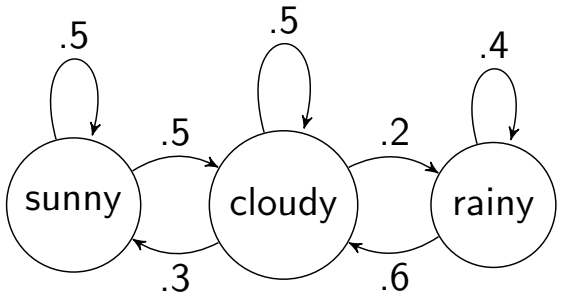


Terminal SCCs: $\{a, d\}$ and $\{i\}$.

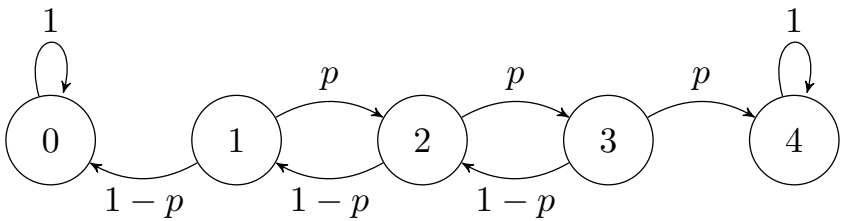
Transient SCCs: $\{b, c, f\}$, $\{e\}$, and $\{g, h\}$.

Parenthesis: probabilistic models

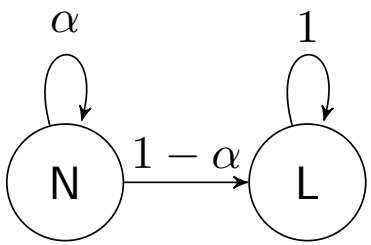
Weather model:



Gambling model:



Learning model:



Using SCC algorithm for checking emptiness

How can we use an SCC algorithm to check emptiness of a Büchi automaton?

Using SCC algorithm for checking emptiness

How can we use an SCC algorithm to check emptiness of a Büchi automaton?

1. Find all (usually maximal, but they don't have to be) SCCs in the graph.
2. All these are **reachable** from some initial state (the SCC algorithm guarantees that).
3. If some SCC contains an accepting node $\Rightarrow$ automaton is non-empty.
4. And vice versa, so:
   automaton is non-empty $\Leftrightarrow$ some SCC contains an accepting node.

Basic algorithms for finding cycles in graphs

Other algorithms except SCC for finding cycles in graphs?

- Tarjan's SCC algorithm.
- What about DFS? (depth-first search)

Using DFS to find cycles in a graph

Recall DFS:

- Maintain a set of visited nodes V.
- Starting with initial nodes, explore all successors of a node v.
- For each successor v', if $v' \notin V$ then recurse from v'; otherwise do nothing (v' already visited).
- In practice, instead of recursion, DFS is implemented using a **stack**:
  - when $v' \notin V$, v' is pushed to the stack and the search continues from v';
  - when all successors of a node u are explored, u is popped from the stack and the search backtracks.

What happens when we find a node $v' \in V$ such that v' is also somewhere in the stack?

Using DFS to find cycles in a graph

Recall DFS:

- Maintain a set of visited nodes V.
- Starting with initial nodes, explore all successors of a node v.
- For each successor v', if $v' \notin V$ then recurse from v'; otherwise do nothing (v' already visited).
- In practice, instead of recursion, DFS is implemented using a **stack**:
  - when $v' \notin V$, v' is pushed to the stack and the search continues from v';
  - when all successors of a node u are explored, u is popped from the stack and the search backtracks.

What happens when we find a node $v' \in V$ such that v' is also somewhere in the stack?

$\Rightarrow$ We found a cycle! The cycle has v' as its root.

Using DFS to find cycles in a graph

Recall DFS:

- Maintain a set of visited nodes V.
- Starting with initial nodes, explore all successors of a node v.
- For each successor v', if $v' \notin V$ then recurse from v'; otherwise do nothing (v' already visited).
- In practice, instead of recursion, DFS is implemented using a **stack**:
  - when $v' \notin V$, v' is pushed to the stack and the search continues from v';
  - when all successors of a node u are explored, u is popped from the stack and the search backtracks.

What happens when we find a node $v' \in V$ such that v' is also somewhere in the stack?

$\Rightarrow$ We found a cycle! The cycle has v' as its root.

Can we check whether the cycle is accepting?

Using DFS to find cycles in a graph

Recall DFS:

- Maintain a set of visited nodes V.
- Starting with initial nodes, explore all successors of a node v.
- For each successor v', if $v' \notin V$ then recurse from v'; otherwise do nothing (v' already visited).
- In practice, instead of recursion, DFS is implemented using a **stack**:
  - when $v' \notin V$, v' is pushed to the stack and the search continues from v';
  - when all successors of a node u are explored, u is popped from the stack and the search backtracks.

What happens when we find a node $v' \in V$ such that v' is also somewhere in the stack?

$\Rightarrow$ We found a cycle! The cycle has v' as its root.

Can we check whether the cycle is accepting? Yes: check whether some node in the stack that follows (i.e., is pushed after) v' is accepting.

Using DFS to find cycles in a graph

$\Rightarrow$ If we find a cycle during DFS, we can check whether it is accepting.

If it is, we're done: Büchi automaton is not empty.

Using DFS to find cycles in a graph

$\Rightarrow$ If we find a cycle during DFS, we can check whether it is accepting.

If it is, we're done: Büchi automaton is not empty.

But does this method find **all** cycles?

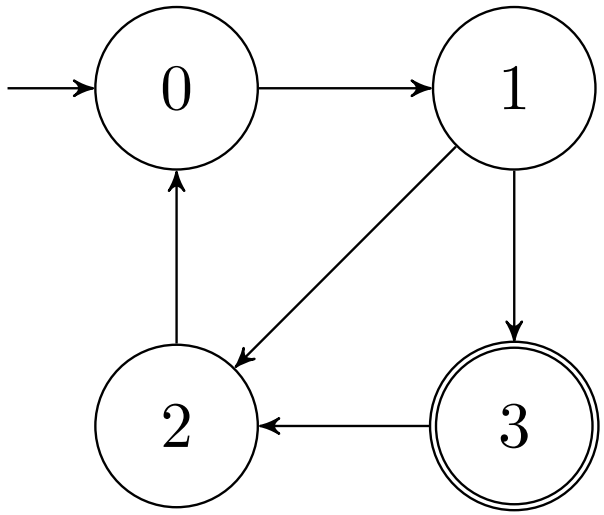
Using DFS to find cycles in a graph

$\Rightarrow$ If we find a cycle during DFS, we can check whether it is accepting.

If it is, we're done: Büchi automaton is not empty.

But does this method find **all** cycles?

No! Consider the following graph:



What happens in this case?

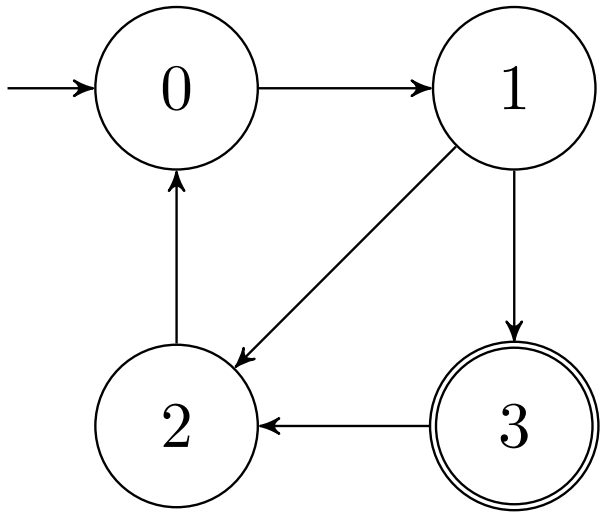
Using DFS to find cycles in a graph

$\Rightarrow$ If we find a cycle during DFS, we can check whether it is accepting.

If it is, we're done: Büchi automaton is not empty.

But does this method find **all** cycles?

No! Consider the following graph:



What happens in this case? 0, 1, and 2 are visited in that order. The cycle $0, 1, 2, 0$ is found, but it's not accepting. 2 is popped and the search backtracks to 1. 3 is explored. 2 is a successor of 3, but it's already visited. Accepting cycle $0, 1, 3, 2, 0$ is missed.

Using DFS to find cycles in a graph

DFS with a set of visited states is sound (when it finds an accepting cycle, it's a true accepting cycle), but **incomplete** (it may miss cycles).

Can DFS be made complete?

Using DFS to find cycles in a graph

DFS with a set of visited states is sound (when it finds an accepting cycle, it's a true accepting cycle), but **incomplete** (it may miss cycles).

Can DFS be made complete?

Yes:

- do not keep a set of visited nodes: only keep a stack
- do not explore nodes already in the stack

Does this algorithm terminate?

Using DFS to find cycles in a graph

DFS with a set of visited states is sound (when it finds an accepting cycle, it's a true accepting cycle), but **incomplete** (it may miss cycles).

Can DFS be made complete?

Yes:

- do not keep a set of visited nodes: only keep a stack
- do not explore nodes already in the stack

Does this algorithm terminate?

Yes: it terminates after exploring all **elementary** cycles (cycles where only their root is visited twice).

Problem: the number of such cycles is exponential in the worst case!

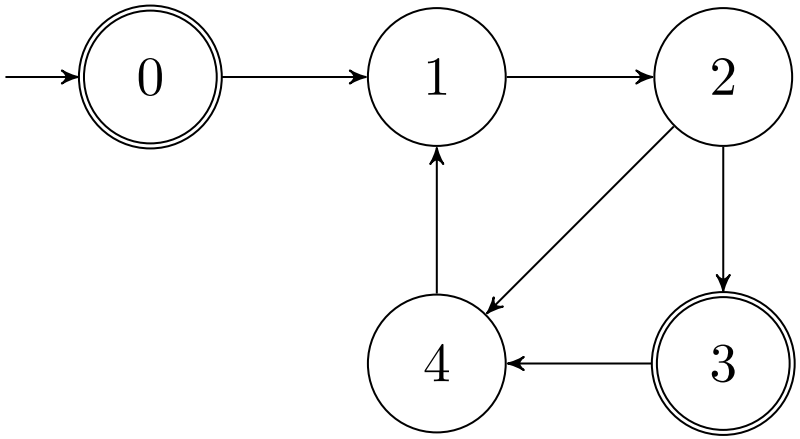
Double DFS (also called *nested* DFS)

A clever algorithm: **double DFS** [Courcoubetis et al., 1992].
A modification of standard DFS:

- Keep **two** sets of visited nodes, V_1 and V_2.
- V_1 is used as usual, for a standard **outer** DFS.
- Each time an accepting node v is visited, spawn a second (**inner**) DFS starting from v and using V_2 as the set of visited nodes.
- Important: spawn the inner DFS **only after** all successors of v have been explored in the outer DFS.[1]
- Important: V_2 is **not** reset between successive inner DFS runs $\Rightarrow$ complexity is not quadratic, but linear: each state visited at most twice (once in the outer DFS, and once in the inner DFS).
- Sound and complete!
- Details in [Baier and Katoen, 2008, Courcoubetis et al., 1992]

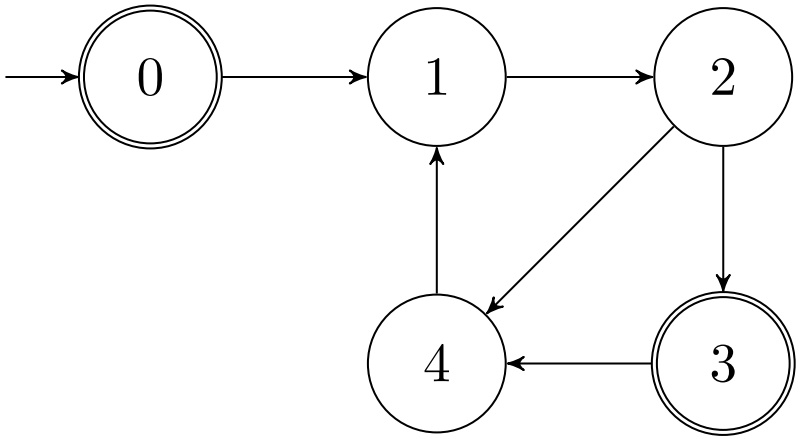
[1] This guarantees that if v' is a successor of v, and both v, v' are accepting, then the inner DFS will be called first starting at v', and then starting at v. This property is crucial for the completeness of the algorithm.

Double DFS: example



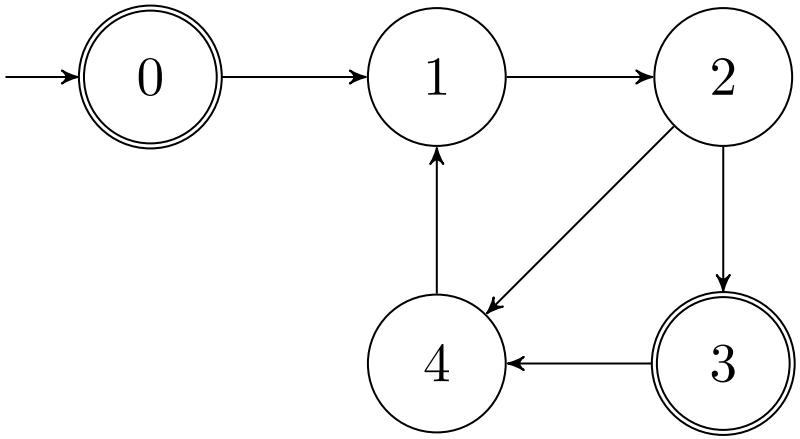
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:

Double DFS: example



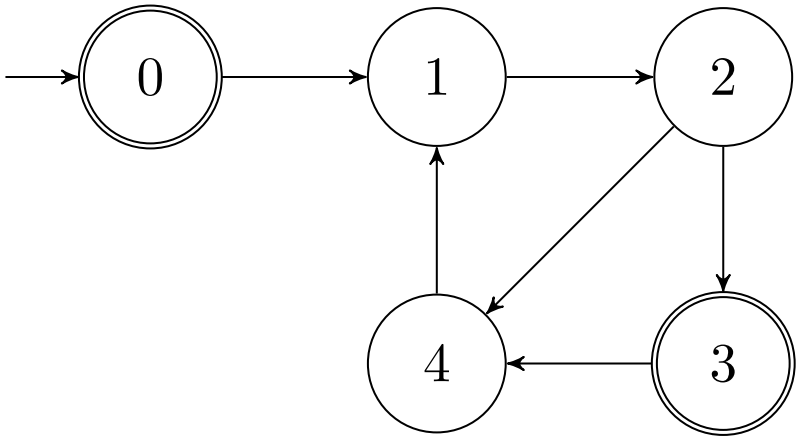
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$.

Double DFS: example



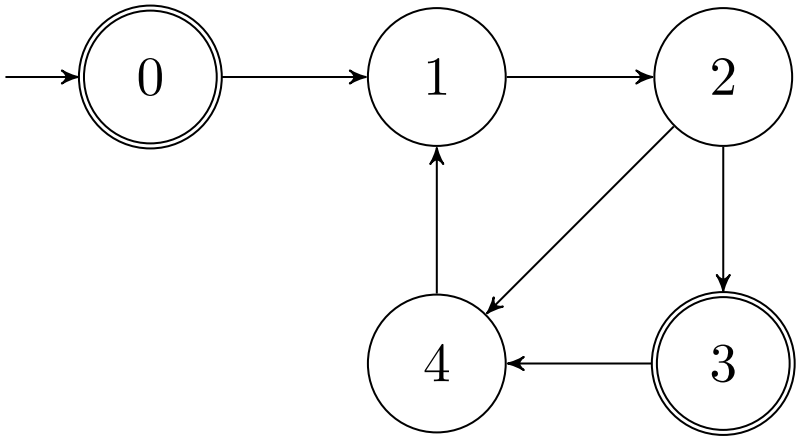
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$.

Double DFS: example



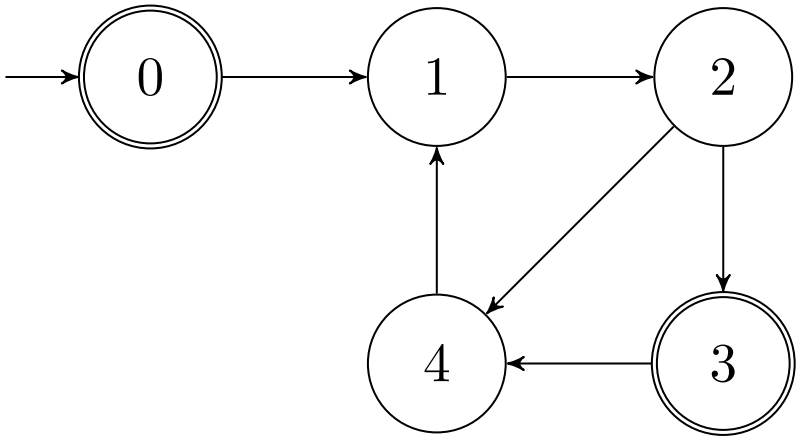
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$.

Double DFS: example



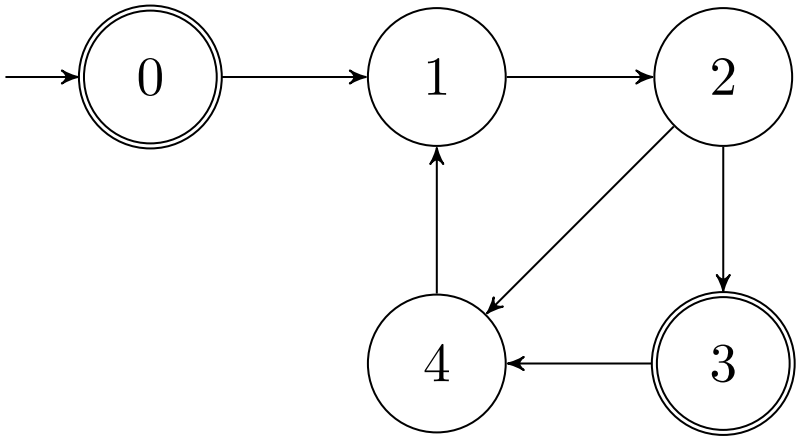
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$.

Double DFS: example



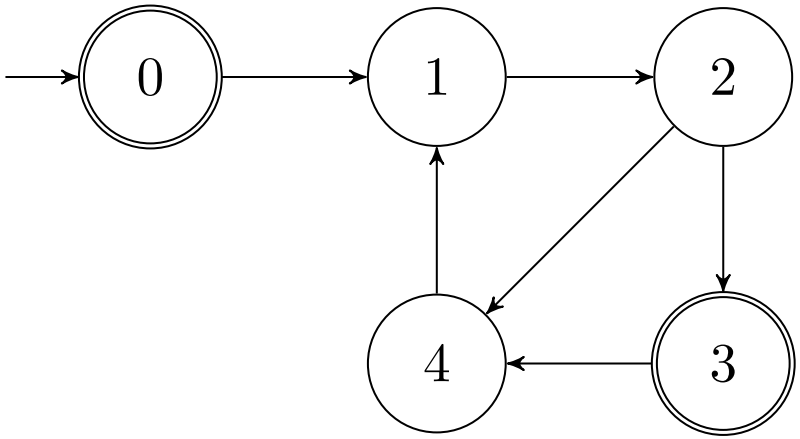
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1.

Double DFS: example



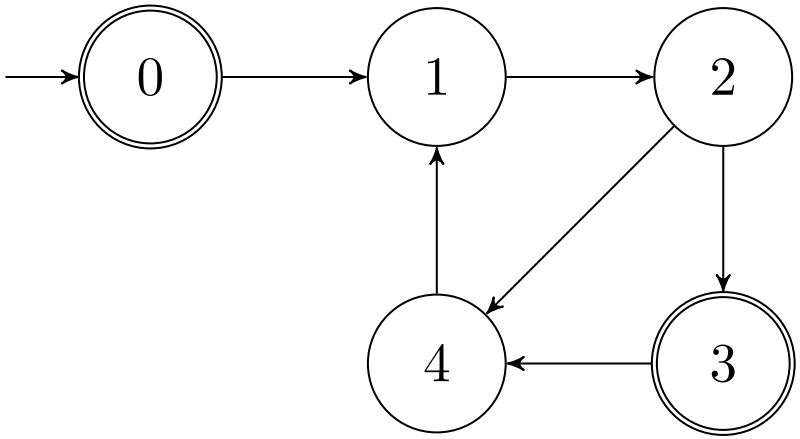
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$.

Double DFS: example



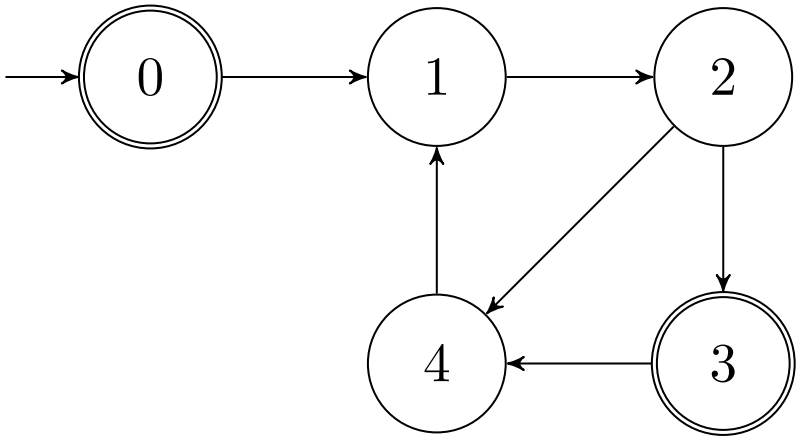
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$.

Double DFS: example



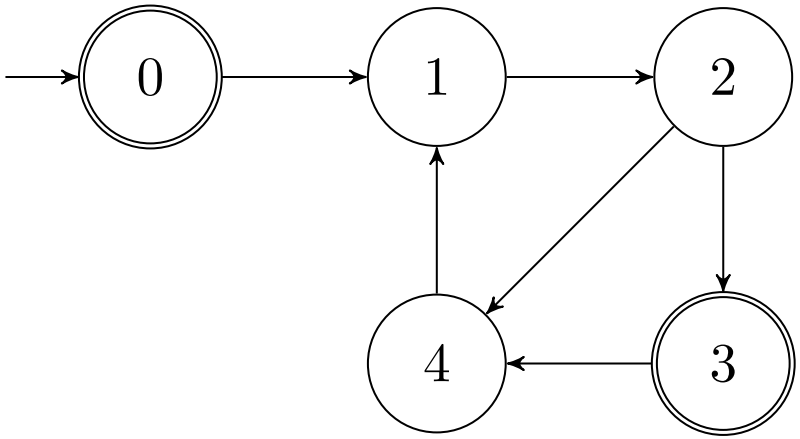
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$. Push 4 on S_2. $V_2 = \{3, 4\}$.

Double DFS: example



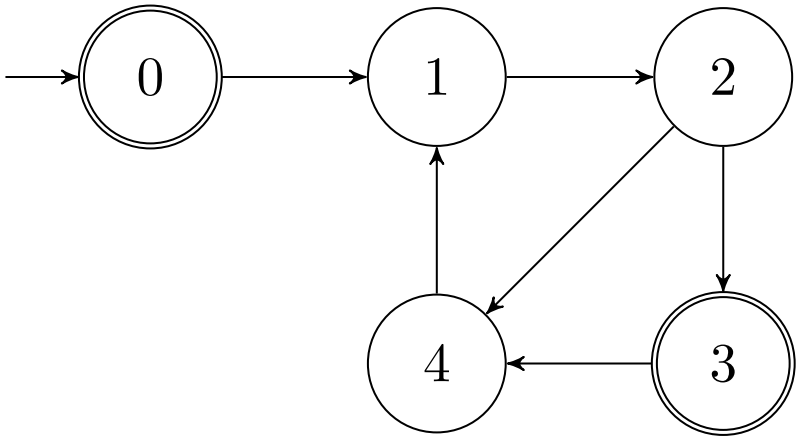
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$. Push 4 on S_2. $V_2 = \{3, 4\}$. Push 1 on S_2. $V_2 = \{3, 4, 1\}$.

Double DFS: example



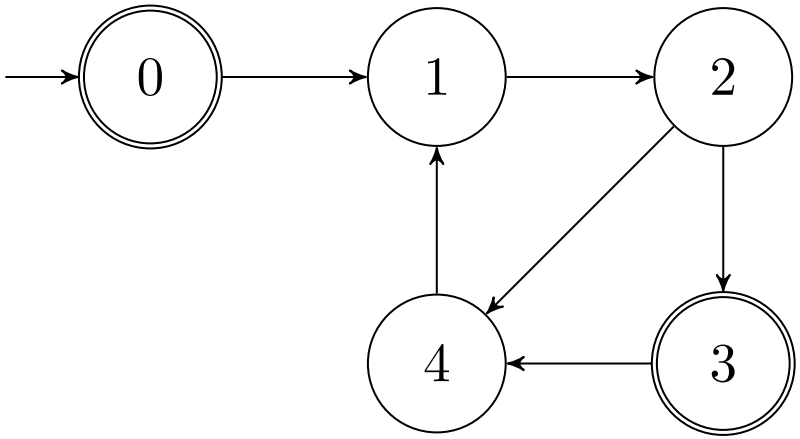
Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$. Push 4 on S_2. $V_2 = \{3, 4\}$. Push 1 on S_2. $V_2 = \{3, 4, 1\}$. Push 2 on S_2. $V_2 = \{3, 4, 1, 2\}$.

Double DFS: example



Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$. Push 4 on S_2. $V_2 = \{3, 4\}$. Push 1 on S_2. $V_2 = \{3, 4, 1\}$. Push 2 on S_2. $V_2 = \{3, 4, 1, 2\}$. Cycle $4, 1, 2, 4$ found by inner DFS, but not accepting. All successors of 4 already in V_2. Pop 4 from S_2.

Double DFS: example



Let S_1, S_2 be the two stacks of the outer and inner DFSs, respectively. Initially $V_1 = V_2 = \emptyset$ and both stacks are empty. Algorithm steps:
Push 0 on S_1. $V_1 = \{0\}$. Push 1 on S_1. $V_1 = \{0, 1\}$. Push 2 on S_1. $V_1 = \{0, 1, 2\}$. Push 4 on S_1. $V_1 = \{0, 1, 2, 4\}$. Cycle $1, 2, 4, 1$ found by outer DFS, but not accepting. All successors of 4 already in V_1. Pop 4 from S_1. Push 3 on S_1. $V_1 = \{0, 1, 2, 4, 3\}$. All successors of 3 already in V_1. Pop 3 from S_1. Call inner DFS starting from 3. Push 3 on S_2. $V_2 = \{3\}$. Push 4 on S_2. $V_2 = \{3, 4\}$. Push 1 on S_2. $V_2 = \{3, 4, 1\}$. Push 2 on S_2. $V_2 = \{3, 4, 1, 2\}$. Cycle $4, 1, 2, 4$ found by inner DFS, but not accepting. All successors of 4 already in V_2. Pop 4 from S_2. Accepting cycle $3, 4, 1, 2, 3$ found by inner DFS.

Bibliography

Baier, C. and Katoen, J.-P. (2008).
Principles of Model Checking.
MIT Press.

Clarke, E., Grumberg, O., and Peled, D. (2000).
Model Checking.
MIT Press.

Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. (1992).
Memory efficient algorithms for the verification of temporal properties.
Formal Methods in System Design, 1:275–288.

Huth, M. and Ryan, M. (2004).
Logic in Computer Science: Modelling and Reasoning about Systems.
Cambridge University Press.