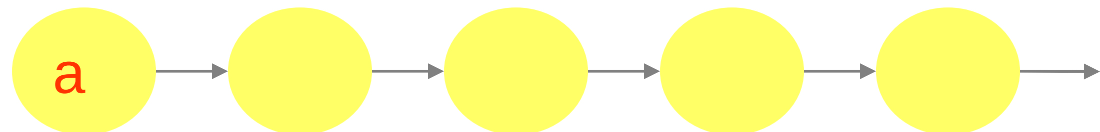
Automata-Theoretic LTL Model-Checking

Arie Gurfinkel
arie@cmu.edu

SEI/CMU

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions

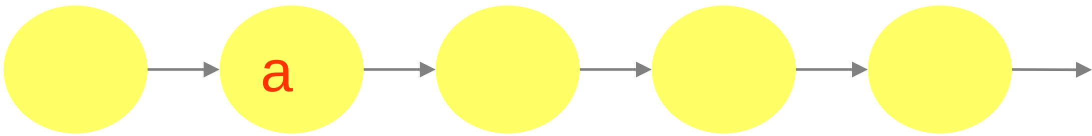
Boolean Operations

Temporal operators

a	"a is true now"
X a	"a is true in the neXt state"
Fa	"a will be true in the Future"
Ga	"a will be Globally true in the future"
a U b	"a will hold true Until b becomes true"

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions

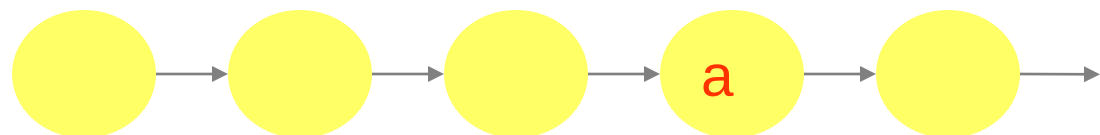
Boolean Operations

Temporal operators

a	“a is true now”
X a	“a is true in the neXt state”
Fa	“a will be true in the Future”
Ga	“a will be Globally true in the future”
a U b	“a will hold true Until b becomes true”

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions

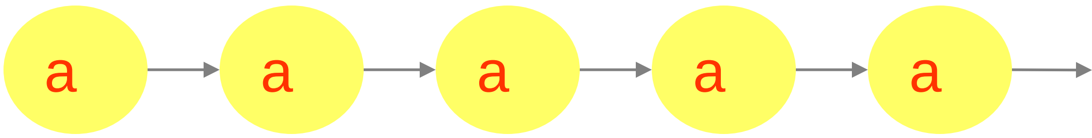
Boolean Operations

Temporal operators

a	“a is true now”
X a	“a is true in the neXt state”
Fa	**“a will be true in the Future”**
Ga	“a will be Globally true in the future”
a U b	“a will hold true Until b becomes true”

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions

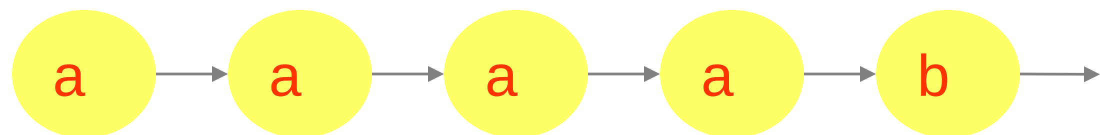
Boolean Operations

Temporal operators

a	"a is true now"
X a	"a is true in the neXt state"
Fa	"a will be true in the Future"
Ga	**"a will be Globally true in the future"**
a U b	"a will hold true Until b becomes true"

LTL - Linear Time Logic (Pn 77)

Determines Patterns on Infinite Traces



Atomic Propositions

Boolean Operations

Temporal operators

a	“a is true now”
X a	“a is true in the neXt state”
Fa	“a will be true in the Future”
Ga	“a will be Globally true in the future”
a U b	“a will hold true Until b becomes true”

Outline

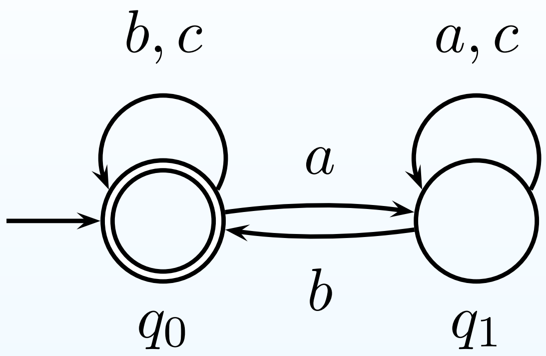
- Automata-Theoretic Model-Checking
  - Finite Automata and Regular Languages
  - Automata over infinite words: Büchi Automata
  - Representing models and formulas with automata
  - Model checking as language emptiness

Finite Automata

A finite automaton $\mathcal{A}$ (over finite words) is a tuple $(\Sigma, Q, \Delta, Q^0, F)$, where

- Σ is a finite alphabet
- Q is a finite set of states
- $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation
- $Q^0 \subseteq Q$ is a set of initial states
- $F \subseteq Q$ is a set of final states

Finite Automaton: An Example

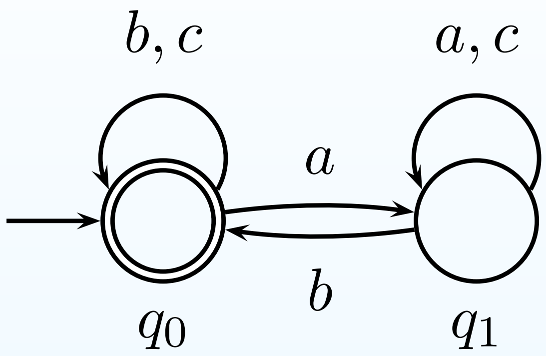


$$\Sigma = \{a, b, c\}, Q = \{q_0, q_1\}, Q^0 = \{q_0\}, F = \{q_1\}$$

A Run

- A *run* of $\mathcal{A}$ over a word $v \in \Sigma^*$ of length $|v|$ is a mapping $\rho : \{0, 1, ..., |v|\} \to Q$ s.t.
    - First state is the initial state: $\rho(0) \in Q^0$
    - States are related by transition relation: $\forall 0 \leq i \leq |v| \cdot (\rho(i), v(i), \rho(i+1)) \in \Delta$
- A run is a path in $\mathcal{A}$ from q_0 to a state $\rho(|v|)$ s.t. the edges are labeled with letters in v
- A run is *accepting* if it ends in an accepting state: $\rho(|v|) \in F$.
- $\mathcal{A}$ *accepts* v iff exists an accepting run of $\mathcal{A}$ on v.

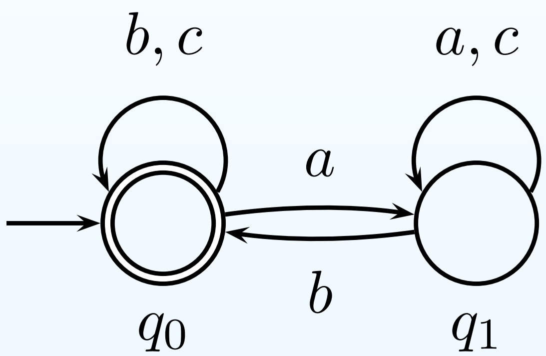
An Example of a Run



- A run q_0, q_1, q_1, q_1, q_0 on $aacb$ is accepting
- A run q_0, q_0, q_0, q_0, q_0 on $bbbb$ is accepting
- A run q_0, q_0, q_1, q_1, q_1 on $baac$ is rejecting

Language

The *language* $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ is the set of all words in Σ^* accepted by $\mathcal{A}$.



The language is $\{\epsilon, b, bb, ccc, bab, \ldots\}$

That is, a regular expression: $\epsilon + a(a+c)^*b(b+c)^*$

Regular Languages

- A set of strings is *regular* if is a language of a finite automaton (i.e., recognizable by a finite automaton)
- An automaton is *deterministic* if the transition relation is deterministic for every letter in the alphabet:

$$\forall a \cdot (q, a, q') \in \Delta \ \wedge \ (q, a, q'') \in \Delta \Rightarrow q' = q''$$

  otherwise, it is *non-deterministic*.
- NFA = DFA: Every non-deterministic finite automaton (NFA) can be translated into a language-equivalent deterministic automaton (DFA)

Automata on Infinite Words

- Reactive programs execute forever – need infinite sequences of states to model them!
- Solution: finite automata over infinite words.
- Simplest case: Büchi automata
  - Same structure as automata on finite words
    - ... but different notion of acceptance
  - Recognize words from Σ^ω (not Σ^*!)
    - $\Sigma = \{a, b\} \quad v = abaabaaab...$
    - $\Sigma = \{a, b, c\}$
      $\mathcal{L}_1 = \{v \mid$ in v after every a there is a $b\}$
      Some words in $\mathcal{L}_1$:
      $ababab\cdots \qquad aaabaaab\cdots$
      $abbabbabb\cdots \qquad accbaccb\cdots$

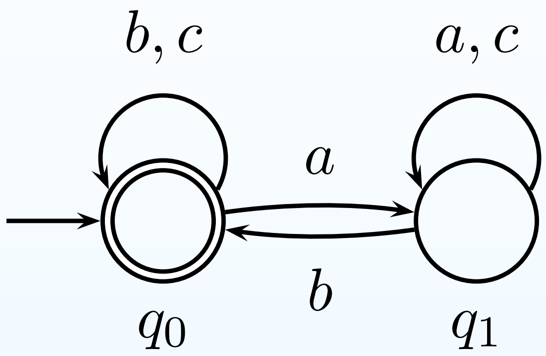
Infinite Run and Acceptance

- Recall, F is the set of *accepting* states
- A *run* ρ of a Büchi automaton $\mathcal{A}$ is over an infinite word $v \in \Sigma^\omega$. Domain of the run is the set of all natural numbers.
- Let *inf*(ρ) be the set of states that appear infinitely often in ρ:

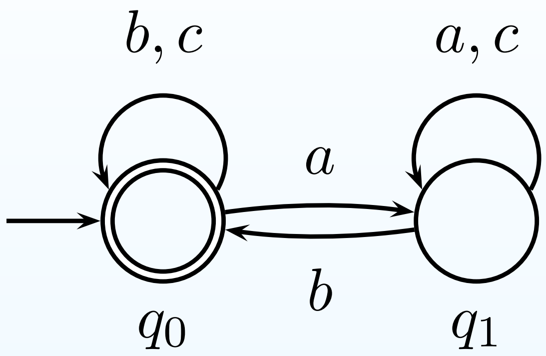
$$\textit{inf}(\rho) = \{q \mid \forall i \in \mathbb{N} \cdot \exists j \geq i \cdot \rho(j) = q\}$$

- A run ρ is *accepting* (Büchi accepting) iff *inf*$(\rho) \cap F \neq \emptyset$.
- A set of strings is ω-regular iff it is recognizable by a Büchi automaton

Example



Example



Language of the automaton is: $((b+c)^\omega a(a+c)^* b)^\omega$

This is an ω-regular expression

Examples

Let $\Sigma = \{0, 1\}$. Define Büchi automata for the following languages:

1. $L = \{v \mid$ 0 occurs in v exactly once$\}$
2. $L = \{v \mid$ after each 0 in v there is a 1$\}$
3. $L = \{v \mid v$ contains finitely many 1's$\}$
4. $L = (01)^n \Sigma^\omega$
5. $L = \{v \mid$ 0 occurs in every even position of $v\}$

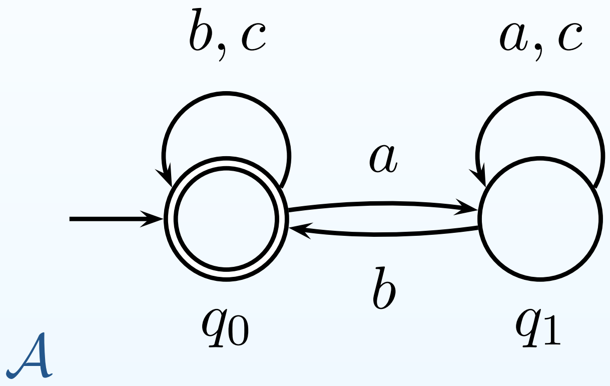
Closure Properties

Büchi-recognizable languages are closed under . . .

- (alphabet) projection and union
  - Same algorithms as Finite Automata
- intersection
  - Different construction from Finite Automata
- complement
  - i.e., from a Büchi automaton $\mathcal{A}$ recognizing $\mathcal{L}$ one can construct an automaton $\overline{\mathcal{A}}$ recognizing $\Sigma^\omega - \mathcal{L}$.
  - $\overline{\mathcal{A}}$ has order of $O(2^{QlogQ)})$ states, where Q are states in $\mathcal{A}$ [Safra's construction]

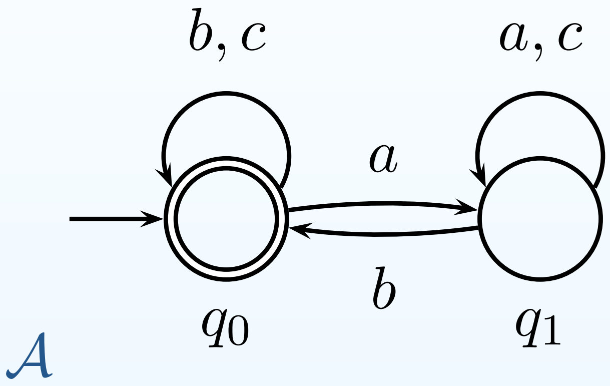
Complementation: Example

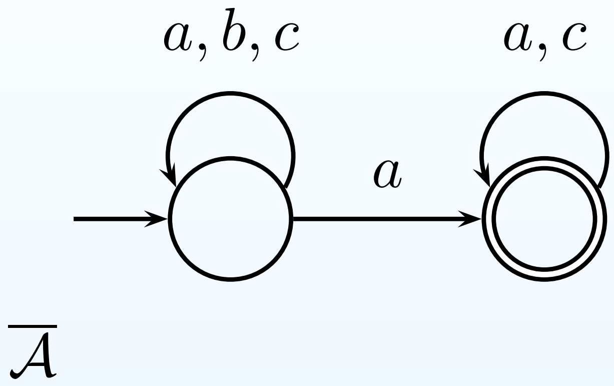
Complement is easy for deterministic Büchi automata:



Complementation: Example

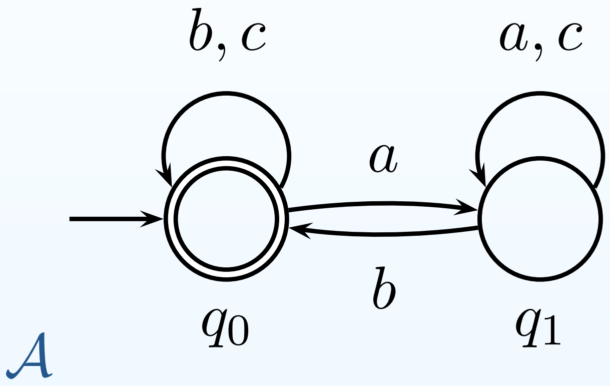
Complement is easy for deterministic Büchi automata:

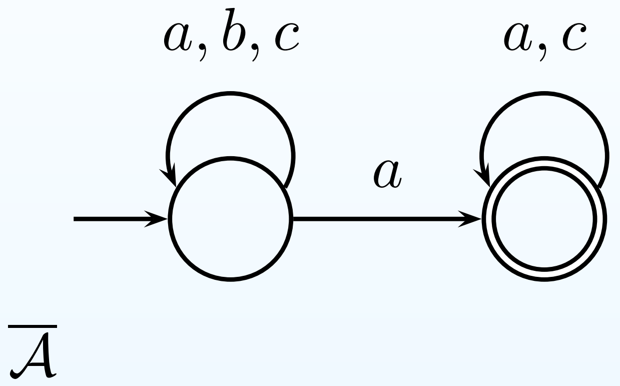




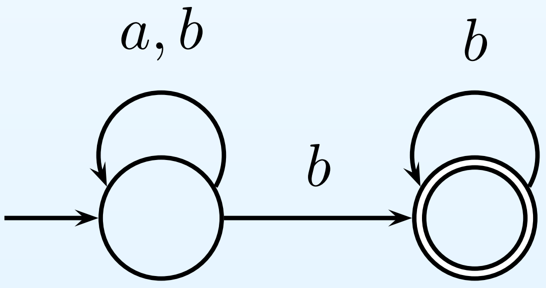
Complementation: Example

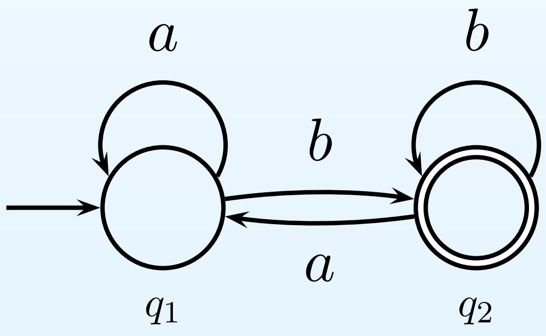
Complement is easy for deterministic Büchi automata:





But, Büchi automata are not closed under determinization!!!





Intersection (Special Case)

Büchi automata are closed under intersection [Chouka74]:

- given two Büchi automata (note all states of $\mathcal{B}_1$ are accepting):

$$\mathcal{B}_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, Q_1) \qquad \mathcal{B}_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$$

- Define $\mathcal{B}_\cap = (\Sigma, Q_1 \times Q_2, \Delta', Q_1^0 \times Q_2^0, Q_1 \times F_2)$, where
  - $((s_1, s_2), a, (s_1', s_2')) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$
- Then, $\mathcal{L}(\mathcal{B}_\cap) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$

Intersection (General Case)

- Main problem: determining accepting states
  - need to go through accepting states of $\mathcal{B}_1$ and $\mathcal{B}_2$ infinite number of times

Intersection (General Case)

- Main problem: determining accepting states
  - need to go through accepting states of $\mathcal{B}_1$ and $\mathcal{B}_2$ infinite number of times
- Key idea: make 3 copies of the automaton:
  - 1st copy: start and accept here
  - 2nd copy: move from here from 1 when accepting state from B_1 has been seen
  - 3rd copy: move here from 2 when accepting state from B_2 has been seen, then go back to 1

Intersection (General Case)

Given two Büchi automata:

$$\mathcal{B}_1 = (\Sigma, Q_1, \Delta_1, Q_1^0, F_2) \qquad \mathcal{B}_2 = (\Sigma, Q_2, \Delta_2, Q_2^0, F_2)$$

Define
$\mathcal{B}_\cap = (\Sigma, Q_1 \times Q_2 \times \{0,1,2\}, \Delta', Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times Q_2 \times \{2\})$,
where

- $((s_1, s_2, 0), a, (s_1', s_2', 0)) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$, and $s_1' \notin F_1$
- $((s_1, s_2, 1), a, (s_1', s_2', 1)) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$, and $s_2' \notin F_2$
- $((s_1, s_2, 0), a, (s_1', s_2', 1)) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$, and $s_1' \in F_1$
- $((s_1, s_2, 1), a, (s_1', s_2', 2)) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$, and $s_2' \in F_2$
- $((s_1, s_2, 2), a, (s_1', s_2', 0)) \in \Delta'$ iff $(s_i, a, s_i') \in \Delta_i$, $i = 1, 2$

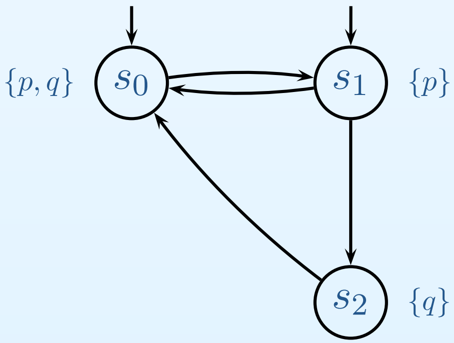
Then, $\mathcal{L}(\mathcal{B}_\cap) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$

Complexity

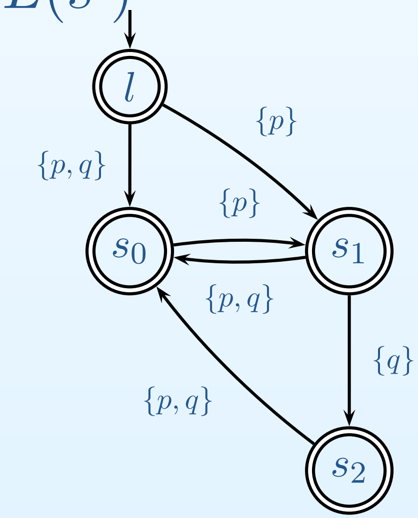
- The emptiness problem for Büchi automata is decidable
  - $\mathcal{L}(\mathcal{A}) \neq \emptyset$
  - logspace-complete for NLOGSPACE, i.e., solvable in linear time [Vardi, Wolper]) – see later in the lecture.
- Nonuniversality problem for Büchi automata is decidable
  - $\mathcal{L}(\mathcal{A}) \neq \Sigma^\omega$
  - logspace-complete for PSPACE [Sistla, Vardi, Wolper]

Modeling Systems Using Automata

- A system is a set of all its executions. So, every state is accepting!
- Transform Kripke structure (S, R, S_0, L), where $L : S \to 2^{AP}$
- ...into automaton $\mathcal{A} = (\Sigma, S \cup \{\ell\}, \Delta, \{\ell\}, S \cup \{\ell\})$,
  - where $\Sigma = 2^{AP}$
  - $(\ell, \alpha, s') \in \Delta$ iff $s \in S_0$ and $\alpha = L(s)$
  - $(s, \alpha, s) \in \Delta$ iff $(s, s') \in R$ and $\alpha = L(s')$

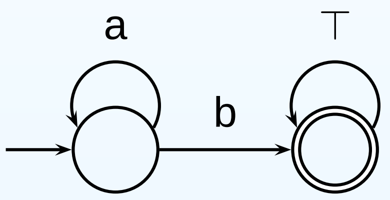


Kripke structure



Automaton

LTL and Büchi Automata

- Specification – also in the form of an automaton!
- Büchi automata can encode all LTL properties.
- Examples:
  - 

    $a \; U \; b$
- Other examples:
  - $\Box \Diamond p$
  - $\Box \Diamond (p \vee q)$
  - $\neg \Box \Diamond (p \vee q)$
  - $\neg (\Box (p \; U \; q))$

LTL to Büchi Automata

- Theorem [Wolper, Vardi, Sistla 83]: Given an LTL formula ϕ, one can build a Büchi automaton $\mathcal{S} = (\Sigma, Q, \Delta, Q_0, F)$, where
  - $\Sigma = 2^{\mathrm{Prop}}$
    - the number of automatic propositions, variables, etc. in ϕ
  - $|Q| \leq 2^{O(|\phi|)}$, where $|\phi|$ is the length of the formula
- ... s.t. $\mathcal{L}(\mathcal{S})$ is exactly the set of computations satisfying the formula ϕ.
- Algorithm: see Section 9.4 of Model Checking book or try one of the online tools:
  - `http://spot.lip6.fr/ltl2tgba.html`
  - `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`
- But Büchi automata are more expressive than LTL!

Automata-theoretic Model Checking

- The system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when
  - $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S})$
  - ... each behavior of the system is among the allowed behaviours
- Alternatively,
  - let $\overline{\mathcal{L}(\mathcal{S})}$ be the language $\Sigma^\omega - \mathcal{L}(\mathcal{S})$. Then,
    - $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S}) \iff \mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$
    - no behavior of $\mathcal{A}$ is prohibited by $\mathcal{S}$
  - If the intersection is not empty, any behavior in it corresponds to a counterexample.
  - Counterexamples are always of the form uv^ω, where u and v are finite words.

Complexity

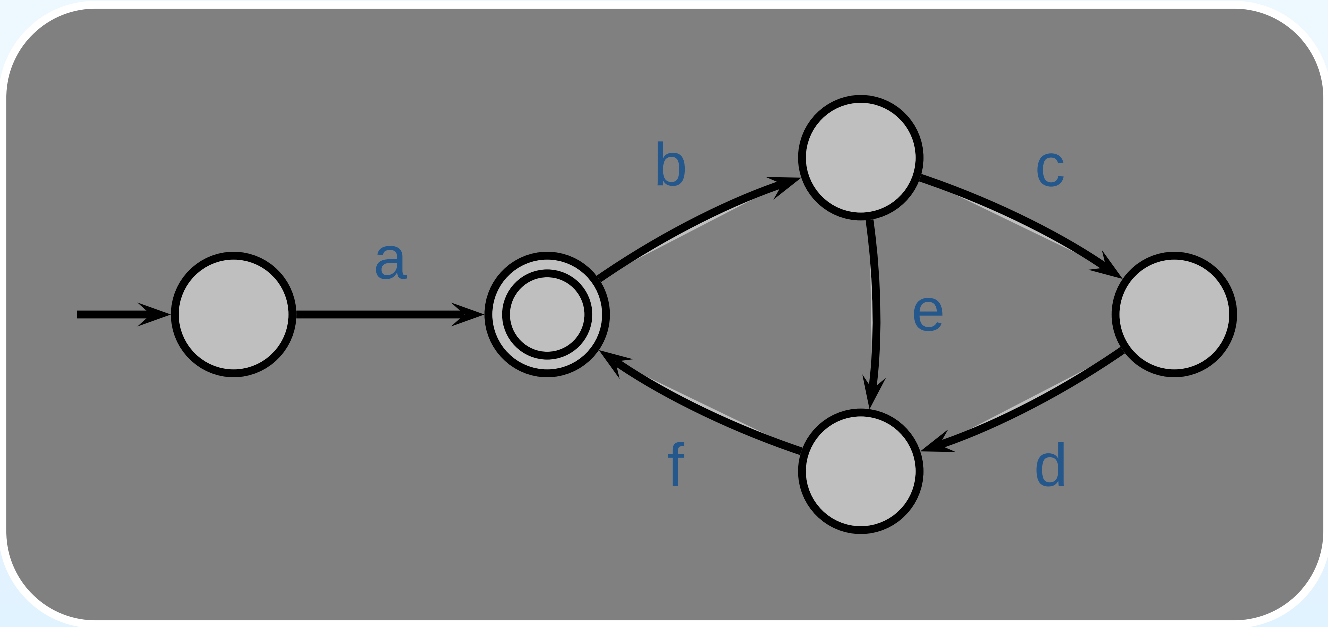
- Checking whether a formula ϕ is satisfied by a finite-state model K can be done in time $O(||K|| \times 2^{O(|\phi|)})$ or in space $O((log||K|| + \|\phi\|)^2)$.
- i.e., checking is polynomial in the size of the model and exponential in the size of the specification.

Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state

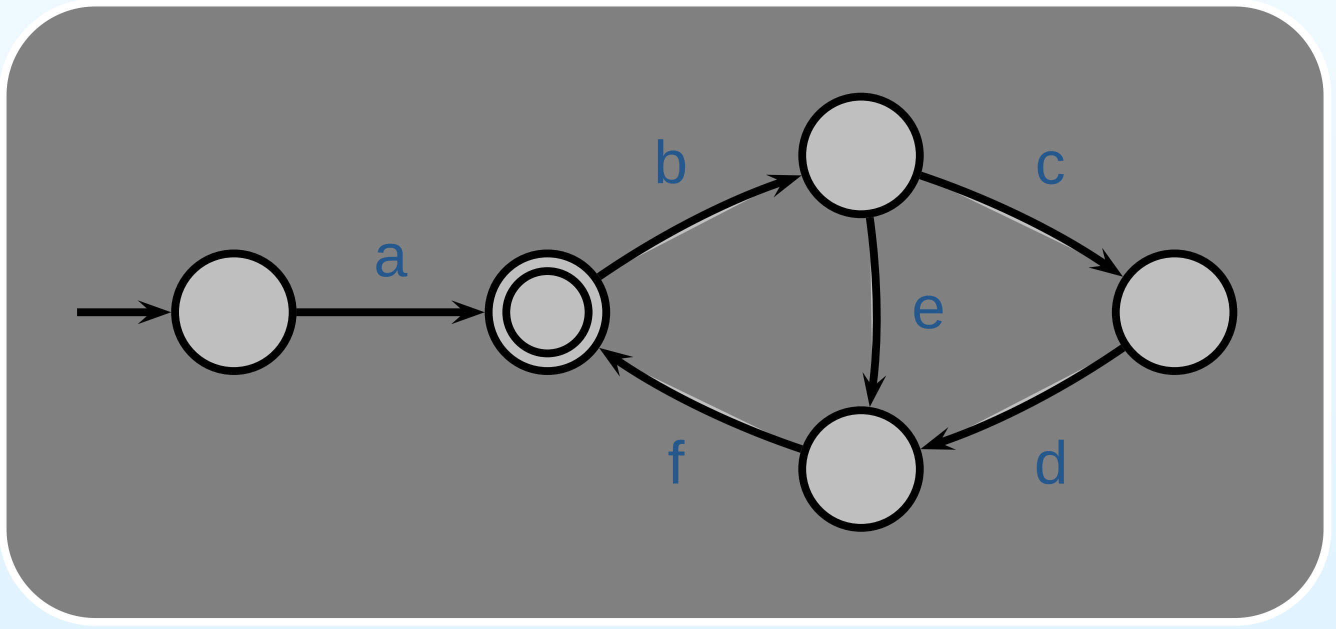
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?



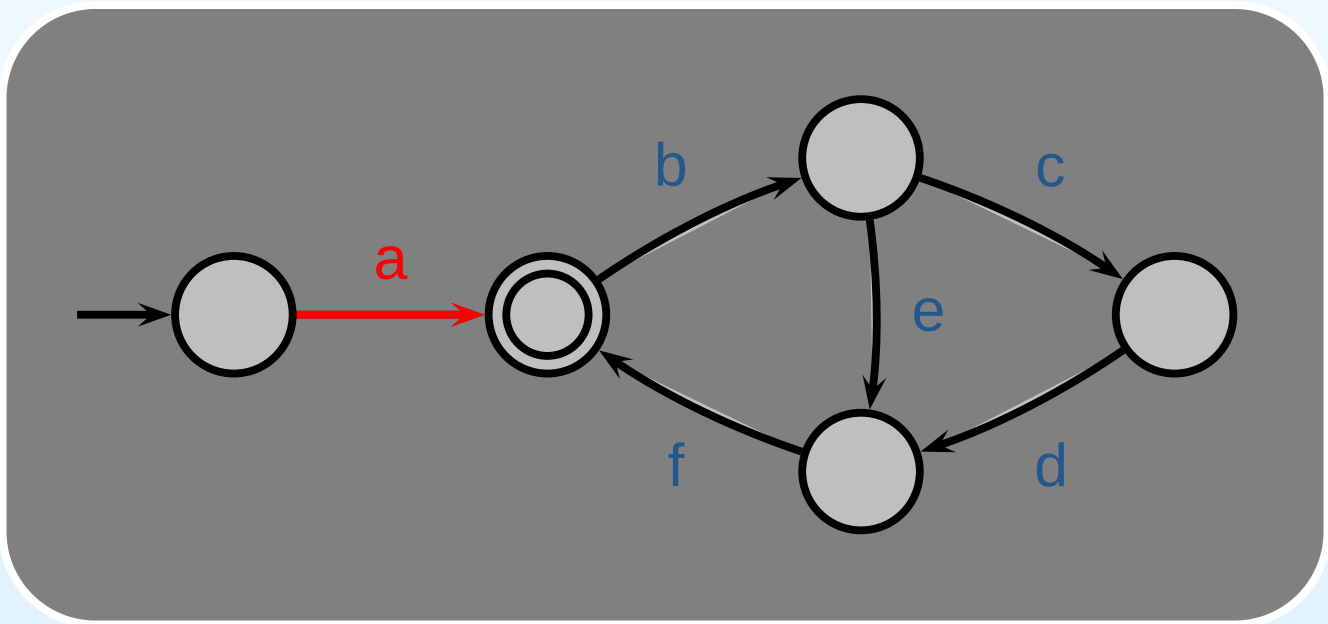
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



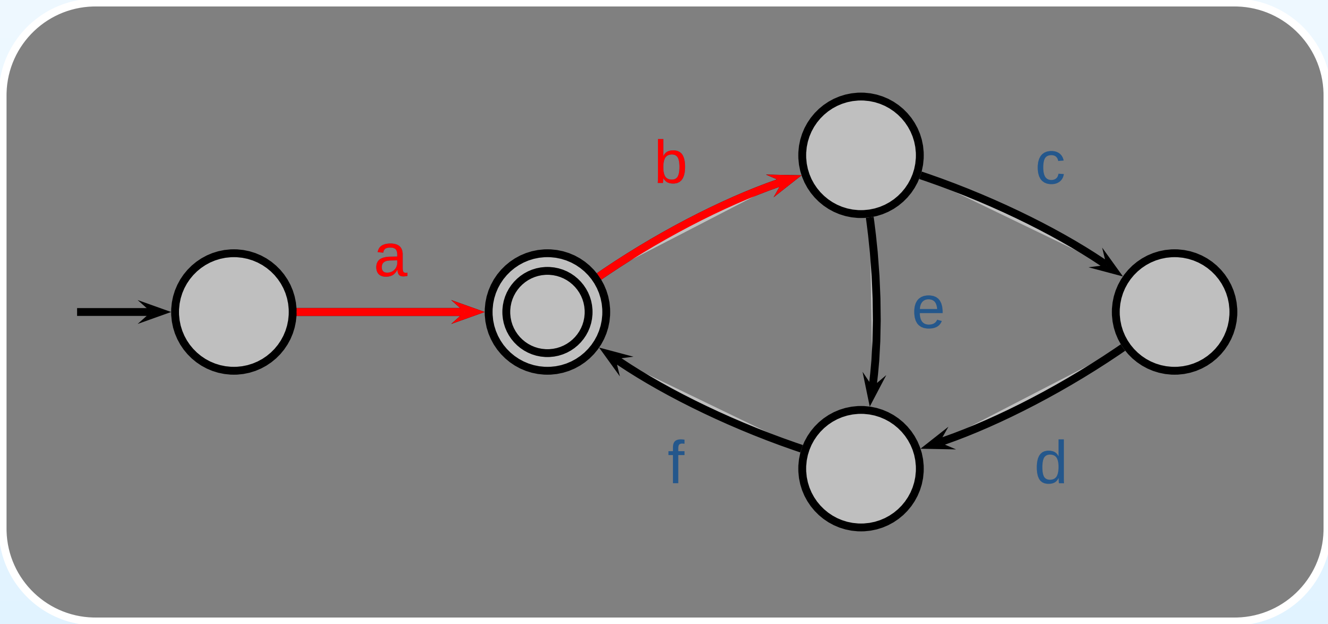
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



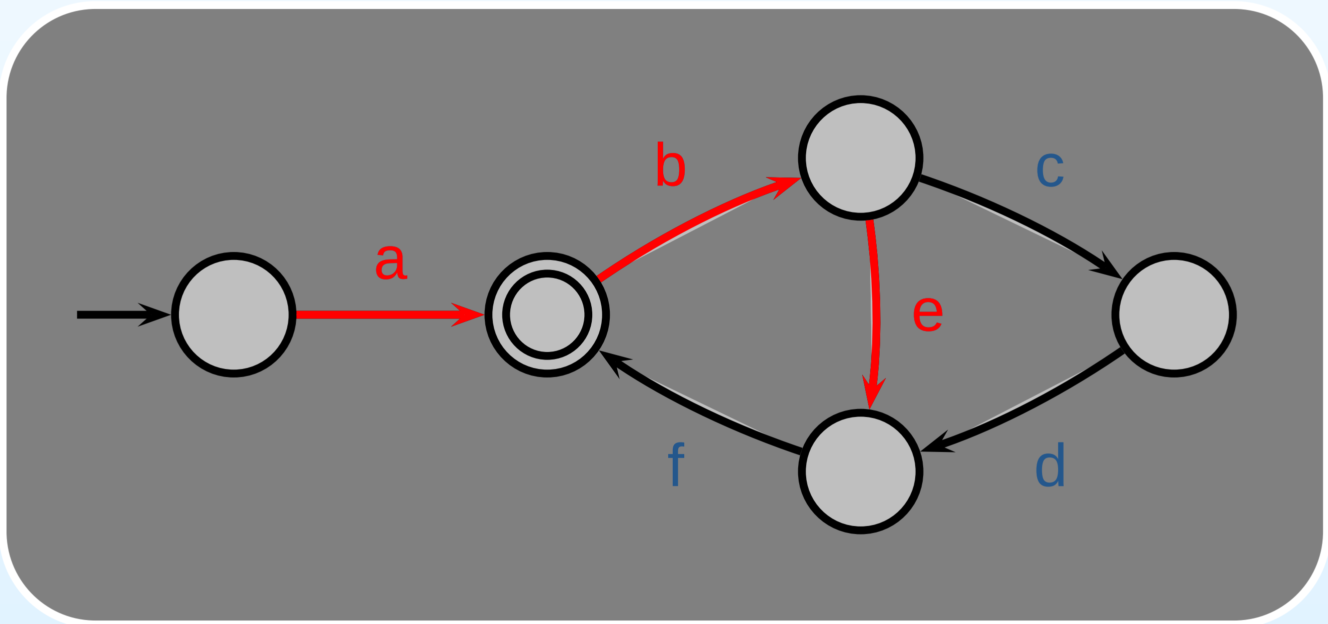
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



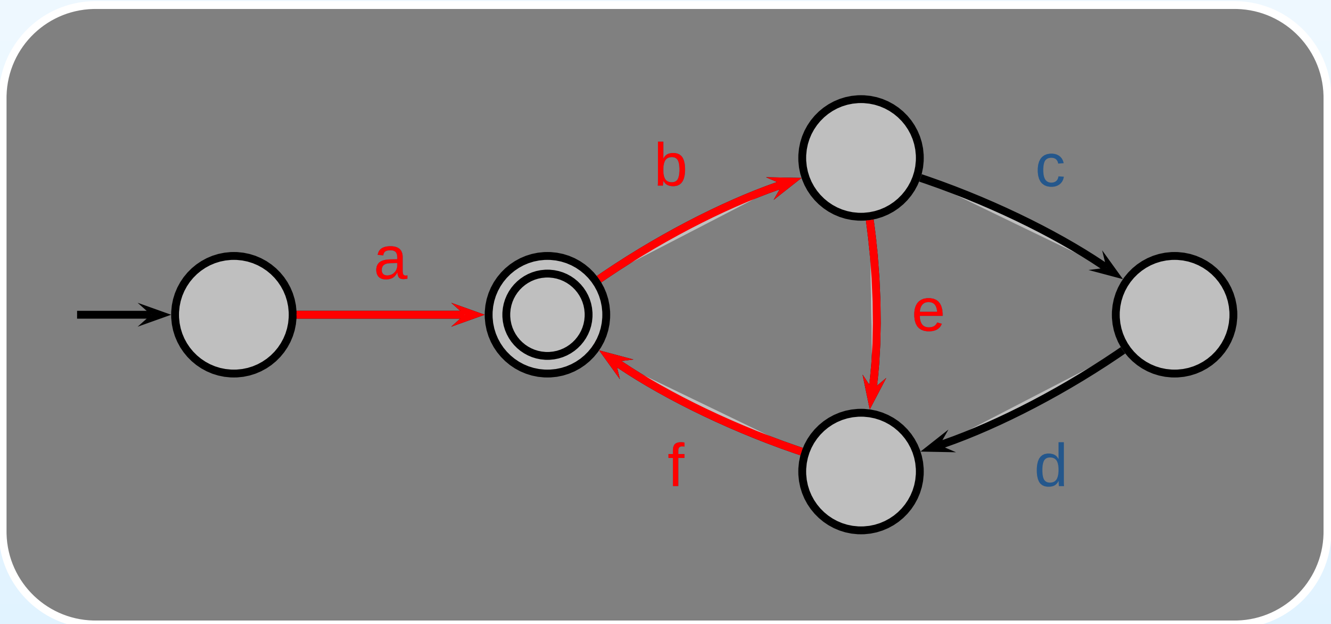
Emptiness of Büchi Automata

- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



Emptiness of Büchi Automata

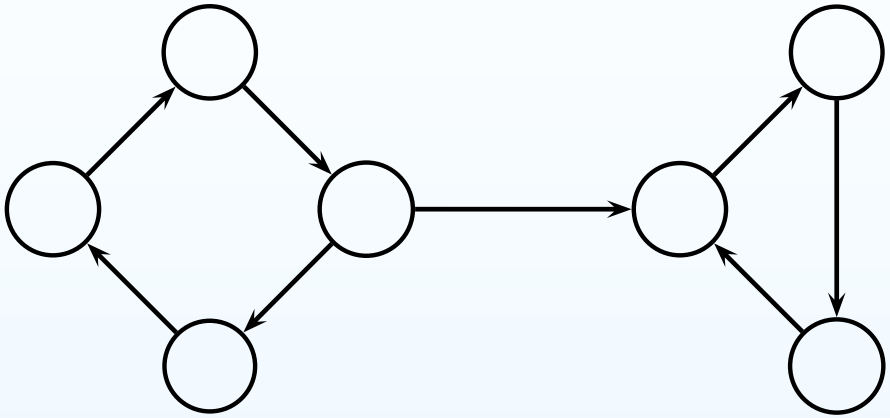
- An automation is non-empty iff
  - there exists a path to a cycle containing an accepting state
- Is this automaton empty?
  - No – it accepts $a(bef)^\omega$



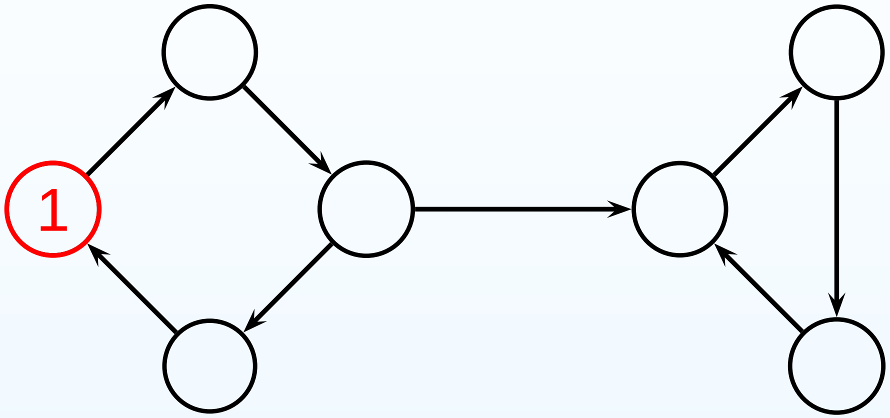
LTL Model-Checking

- LTL Model-Checking = Emptiness of Büchi automata
  - a tiny bit of automata theory +
  - trivial graph-theoretic problem
    - typical solution – use depth-first search (DFS)
- Problem: state-explosion
  - the graph is *HUGE*
- End result:
  - LTL model-checking a very elaborate DFS

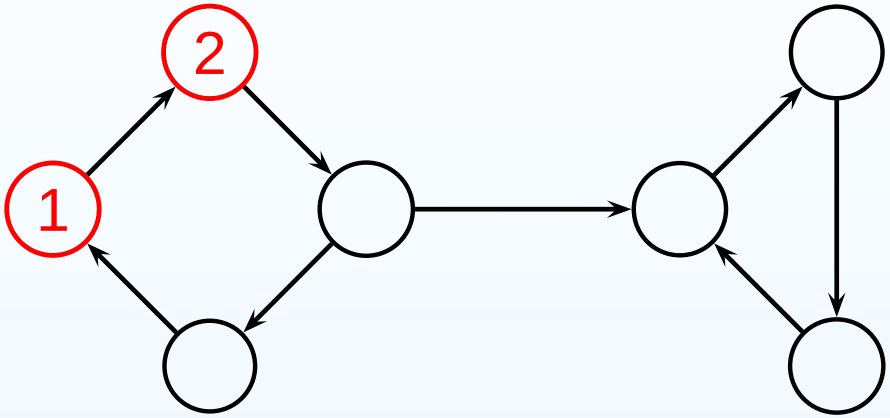
Depth-First Search – Refresher



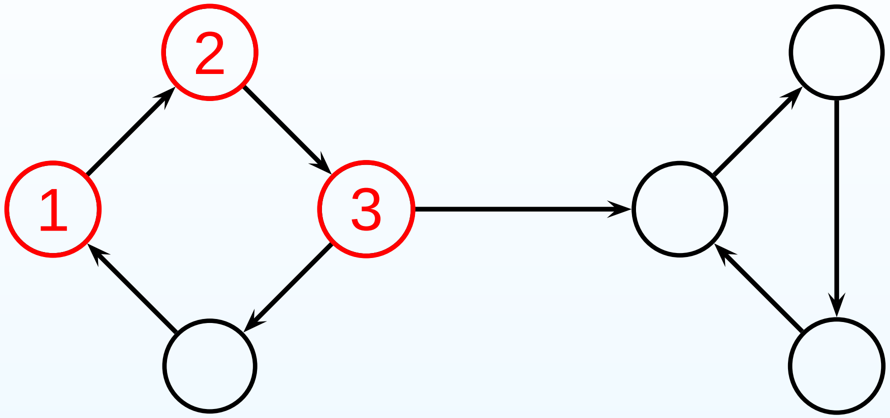
Depth-First Search – Refresher



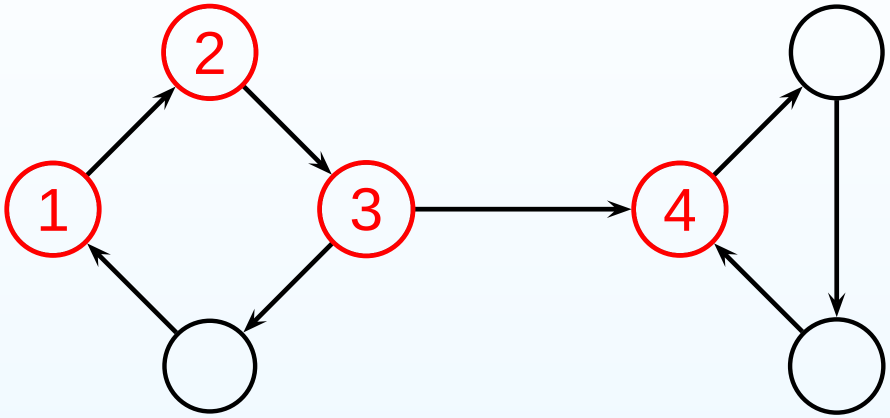
Depth-First Search – Refresher



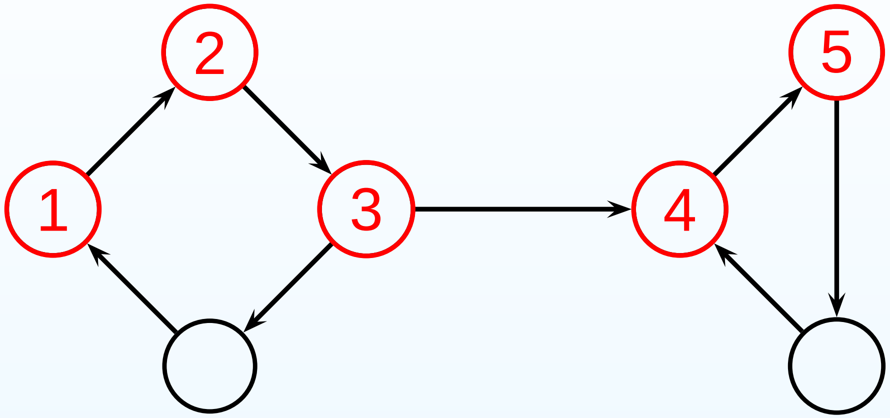
Depth-First Search – Refresher



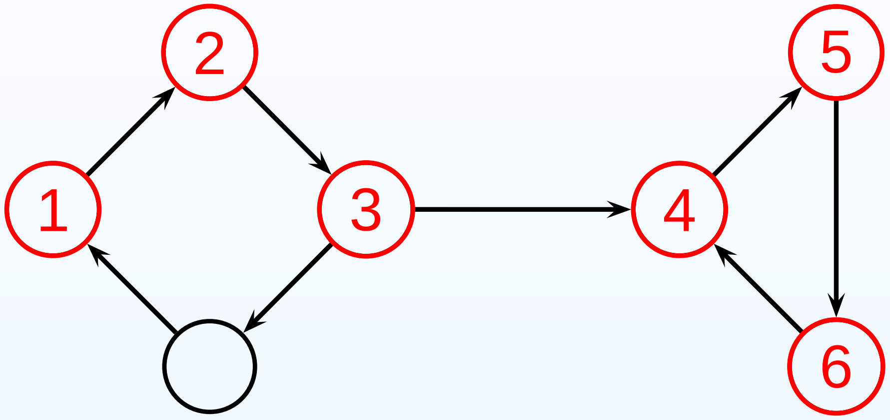
Depth-First Search – Refresher



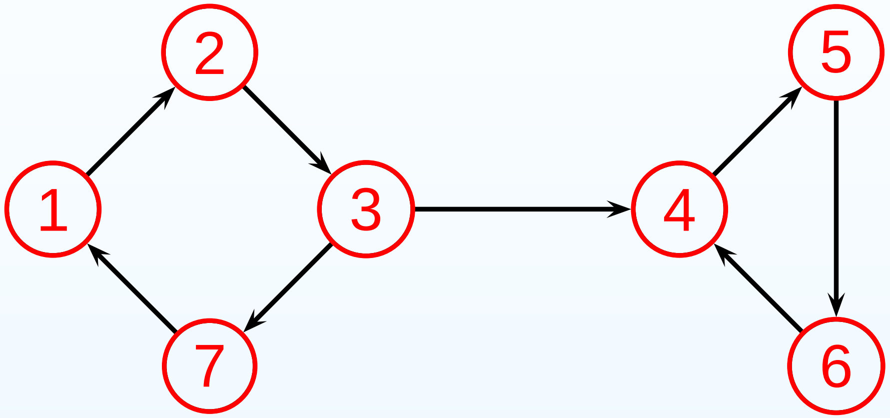
Depth-First Search – Refresher



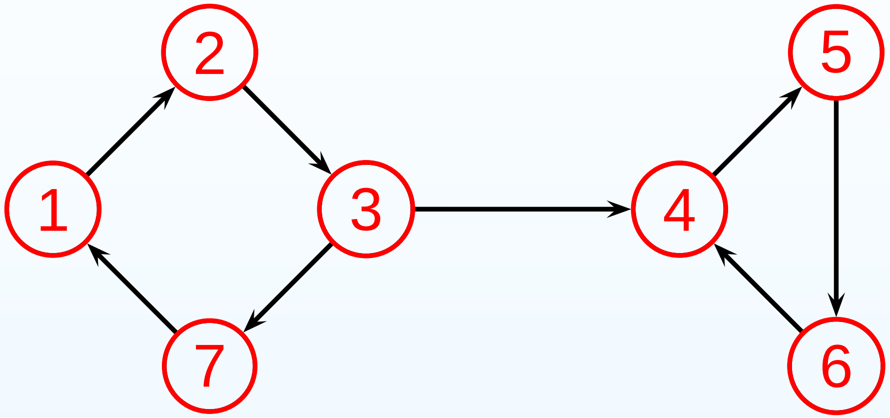
Depth-First Search – Refresher



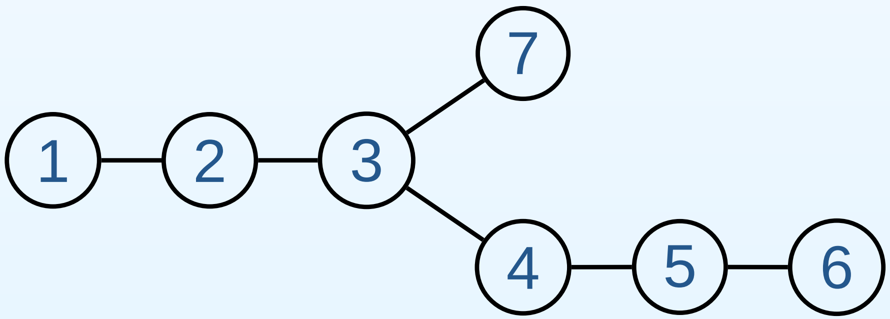
Depth-First Search – Refresher



Depth-First Search – Refresher



Depth-first tree



DFS – The Algorithm

```
time := 0
proc DFS(v)
    add v to Visited
    d[v] := time
    time := time + 1
    for all w ∈ succ(v) do
        if w ∉ Visited then
            DFS(w)
        end if
    end for
    f[v] := time
    time := time + 1
end proc
```

DFS – Data Structures

- implicit STACK
  - stores the current path through the graph
- *Visited* table
  - stores visited nodes
  - used to avoid cycles
- for each node
  - *discovery time* – array d
  - *finishing time* – array f

What we want

- Running time
  - at most linear — anything else is not feasible
- Memory requirements
  - sequentially accessed – (for the STACK)
    - disk storage is good enough
    - assume unlimited supply – so can ignore
  - randomly accessed – (for hash tables)
    - must use RAM
    - limited resource – minimize
    - why cannot use virtual memory?

Additionally...

- Counterexamples
  - an automaton is non-empty iff exists an accepting run
  - this is the counterexample – we want it
- Approximate solutions
  - partial result is better than nothing!

DFS – Complexity

- Running time
  - each node is visited once
  - linear in the size of the graph
- Memory
  - the STACK
    - accessed sequentially
    - can store on disk – ignore
  - *Visited* table
    - randomly accessed – important
    - $|\textit{Visited}| = S \times n$
    - n – number of nodes in the graph
    - S – number of bits needed to represent each node

Take 1 – Tarjan's SCC algorithm

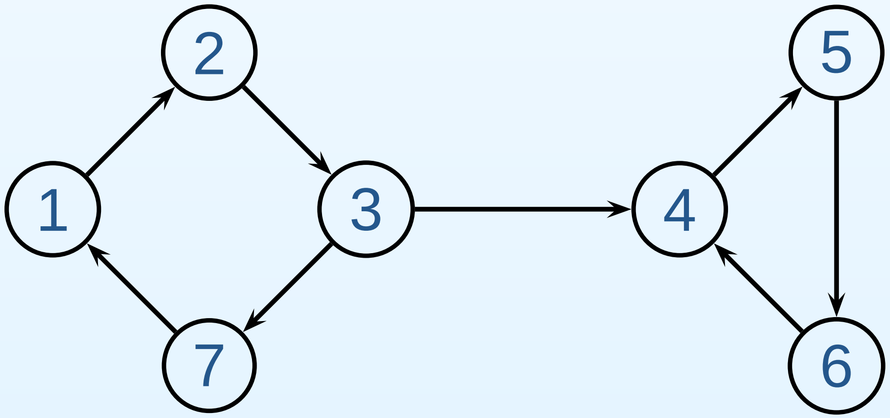
- Idea: find all maximal SCCs: SCC_1, SCC_2, etc.
  - an automaton is non-empty iff exists SCC_i containing an accepting state

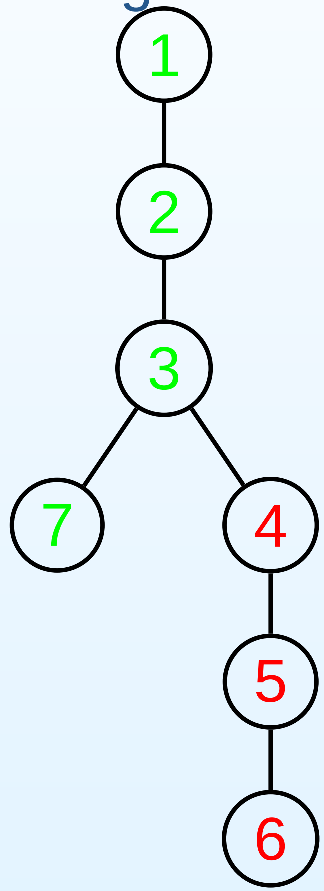
Take 1 – Tarjan's SCC algorithm

- Idea: find all maximal SCCs: SCC_1, SCC_2, etc.
  - an automaton is non-empty iff exists SCC_i containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees

Take 1 – Tarjan's SCC algorithm

- Idea: find all maximal SCCs: SCC_1, SCC_2, etc.
  - an automaton is non-empty iff exists SCC_i containing an accepting state
- Fact: each SCC is a sub-tree of DFS-tree
  - need to find roots of these sub-trees





Finding a Root of an SCC

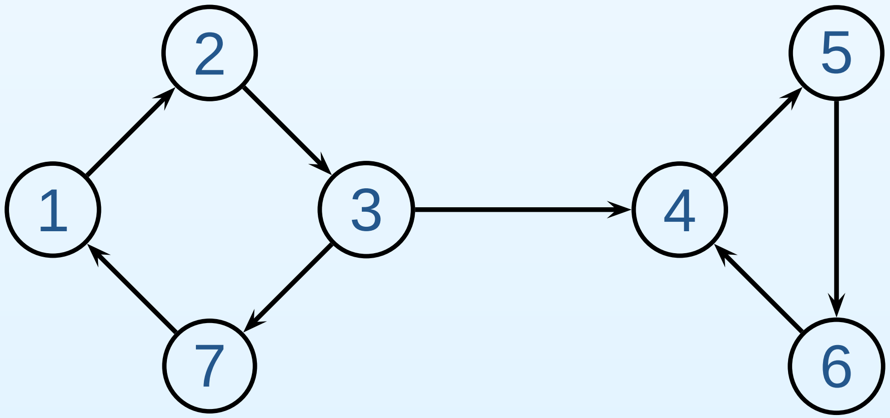
- For each node v, compute $lowlink[v]$
- $lowlink[v]$ is the minimum of
  - discovery time of v
  - discovery time of w, where
    - w belongs to the same SCC as v
    - the length of a path from v to w is at least 1
- Fact: v is a root of an SCC iff
  - $d[v] = lowlink[v]$

Finally: the algorithm

```
proc SCC_SEARCH(v)
  add v to Visited
  d[v] := time
  time := time + 1
  lowlink[v] := d[v]
  push v on STACK
  for all w ∈ succ(v) do
    if w ∉ Visited then
      SCC_SEARCH(w)
      lowlink[v] := min(lowlink[v], lowlink[w])
    else if d[w] < d[v] and w is on STACK then
      lowlink[v] := min(d[w], lowlink[v])
    end if
  end for
  if lowlink[v] = d[v] then
    repeat
      pop x from top of STACK
      if x ∈ F then
        terminate with "Yes"
      end if
    until x = v
  end if
end proc
```

Finally: the algorithm

```
proc SCC_SEARCH(v)
    add v to Visited
    d[v] := time
    time := time + 1
    lowlink[v] := d[v]
    push v on STACK
    for all w ∈ succ(v) do
        if w ∉ Visited then
            SCC_SEARCH(w)
            lowlink[v] := min(lowlink[v], lowlink[w])
        else if d[w] < d[v] and w is on STACK then
            lowlink[v] := min(d[w], lowlink[v])
        end if
    end for
    if lowlink[v] = d[v] then
        repeat
            pop x from top of STACK
            if x ∈ F then
                terminate with “Yes”
            end if
        until x = v
    end if
end proc
```



Tarjan's SCC algorithm – Analysis

- Running time
  - linear in the size of the graph
- Memory
  - STACK – sequential, ignore
  - $Visited – O(S \times n)$
  - $lowlink – \log n \times n$
  - n is not known a priori
    - assume n is at least $\geq 2^{32}$
- Counterexamples
  - can be extracted from the STACK
  - even more – get multiple counterexamples
- If we sacrifice some of generality, can we do better?

Take 2 – Two Sweeps

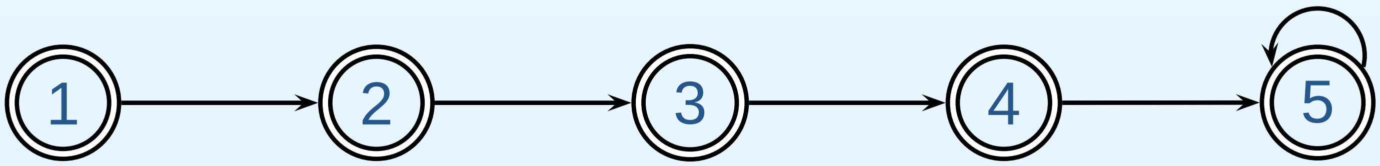
- Don’t look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states

Take 2 – Two Sweeps

- Don't look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)

Take 2 – Two Sweeps

- Don’t look for maximal SCCs
- Find a reachable accepting state that is on a cycle
- Idea: use two sweeps
  - sweep one: find all accepting states
  - sweep two: look for cycles *from* accepting states
- Problem?
  - no longer a linear algorithm (revisit the states multiple times)

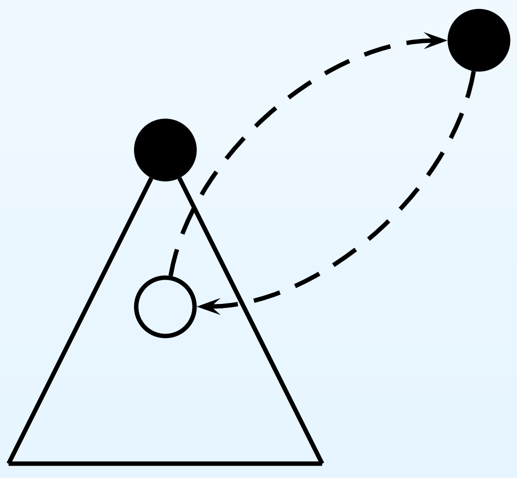


Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v

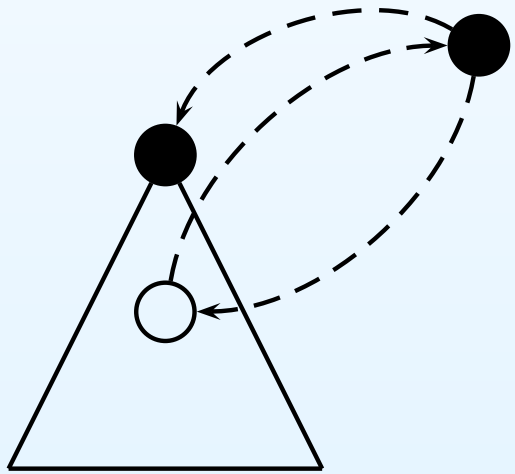
Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v



Fixing non-linearity

- Graph Theoretic Result: let v and u be two nodes, such that
  - $f[v] < f[u]$
  - v is not on a cycle
  - then, no cycle containing u contains nodes reachable from v



Take 3 – Double DFS

```
proc DFS1(v)
    add v to Visited
    for all w ∈ succ(v) do
        if w ∉ Visited then
            DFS1(w)
        end if
    end for
    if v ∈ F then
        add v to Q
    end if
end proc
```

```
proc DFS2(v, f)
    add v to Visited
    for all w ∈ succ(v) do
        if v = f then
            terminate with “Yes”
        else if w ∉ Visited then
            DFS2(w, f)
        end if
    end for
end proc
```

```
proc SWEEP2(Q)
    while Q ≠ [] do
        f := dequeue(Q)
        DFS2(f, f)
    end while
    terminate with “No”
end proc
```

```
proc DDFS(v)
    Q = ∅
    Visited = ∅
    DFS1(v)
    Visited = ∅
    SWEEP2(Q)
end proc
```

Double DFS – Analysis

- Running time
  - linear! (single $Visited$ table for different final states, so no state is processed twice)
- Memory requirements
  - $O(n \times S)$
- Problem
  - where is the counterexample?!

Take 4 – Nested DFS

- Idea
  - when an accepting state is finished
    - stop first sweep
  - start second sweep
    - if cycle is found, we are done
  - otherwise, restart the first sweep
- As good as double DFS, but
  - does not need to *always* explore the full graph
  - counterexample is readily available
    - a path to an accepting state is on the stack of the first sweep
    - a cycle is on the stack of the second

A Few More Tweaks

- No need for two *Visited* hashtables
  - empty hashtable wastes space
  - merge into one by adding one more bit to each node
    - $(v, 0) \in Visited$ iff v was seen by the first sweep
    - $(v, 1) \in Visited$ iff v was seen by the second sweep
- Early termination condition
  - nested DFS can be terminated as soon as it finds a node that is on the stack of the first DFS

Nested DFS

```
proc DFS1(v)
    add (v, 0) to Visited
    for all w ∈ succ(v) do
        if (w, 0) ∉ Visited then
            DFS1(w)
        end if
    end for
    if v ∈ F then
        DFS2(v, v)
    end if
end proc
```

```
proc DFS2(v, f)
    add (v, 1) to Visited
    for all w ∈ succ(v) do
        if v = f then
            terminate with “Yes”
        else if (w, 1) ∉ Visited then
            DFS2(w, f)
        end if
    end for
end proc
```

On-the-fly Model-Checking

- Typical problem consists of
  - description of several process $P_1, P_2, \ldots$
  - property φ in LTL
- Before applying DFS algorithm
  - construct graph for $P = \Pi_{i=1}^{n} P_i$
  - construct Büchi automaton $A_{\neg\varphi}$ for $\neg\varphi$
  - construct Büchi automaton for $P \cap A_{\neg\varphi}$

On-the-fly Model-Checking

- Typical problem consists of
  - description of several process $P_1, P_2, \ldots$
  - property φ in LTL
- Before applying DFS algorithm
  - construct graph for $P = \Pi_{i=1}^{n} P_i$
  - construct Büchi automaton $A_{\neg\varphi}$ for $\neg\varphi$
  - construct Büchi automaton for $P \cap A_{\neg\varphi}$
- But,
  - all constructions can be done in DFS order
  - combine everything with the search
  - result: on-the-fly algorithm, only the necessary part of the graph is built

Symbolic LTL Model-Checking

- LTL Model-Checking = Finding a reachable cycle
- Represent the graph symbolically
  - and use symbolic techniques to search
- There exists an infinite path from s, iff $s \models EG$ true
  - the graph is finite
    - infinite $\Rightarrow$ cyclic!
  - exists a cycle containing an accepting state a iff a occurs infinitely often
    - use fairness to capture accepting states
- LTL Model-Checking = EG true under fairness!