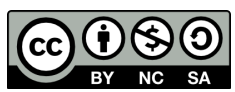
CDM

Closure Properties

Klaus Sutner

Carnegie Mellon University

Spring 2021



We have a definition of recognizable languages in terms of deterministic finite automata (DFAs).

There are two killer apps for recognizable languages:

- pattern matching
- logical decision procedures

In order to get a better understanding of recognizable languages, it turns out that other characterizations can be very useful, both from the theory perspective as well as for the construction of algorithms.

We can think of finite state machines as a particularly weak model of computation. It is then natural to ask basic questions about the model:

- Is there closure under sequential composition?
- Is there closure under parallel composition?

Alas, we only have acceptors so far giving maps $\Sigma^\star \to \mathbf{2}$, so sequential composition makes no sense (we need transducers for that).

But parallel composition we can handle: we want to combine two machines into a single one and run them in parallel. Intuitively, combining two finite state machines should produce another finite state machine: we only need to keep track of pairs of states.

Suppose we have two DFAs over Σ: $\mathcal{A}_i = \langle Q_i, \Sigma, \delta_i; q_{0i}, F_i \rangle$. To run the machines in parallel we define a new DFA as follows:

Definition (Cartesian Product Automaton)

$$\mathcal{A}_1 \times \mathcal{A}_2 = \langle Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2; (q_{01}, q_{02}), F_1 \times F_2 \rangle$$

where $\delta = \delta_1 \times \delta_2$ is defined by

$$\delta((p, q), a) = \big(\delta_1(p, a), \delta_2(q, a)\big)$$

So the computation of $\mathcal{A}_1 \times \mathcal{A}_2$ on input x combines the two computations of both machines on the same input.

Note $|\mathcal{A}_1 \times \mathcal{A}_2| = |\mathcal{A}_1||\mathcal{A}_2|$, a potential problem.

By our choice of acceptance condition we have

$$\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = L_1 \cap L_2$$

By changing the final states in the product, we can also get union and complement:

union	$F = F_1 \times Q_2 \cup Q_1 \times F_2$
intersection	$F = F_1 \times F_2$
difference	$F = F_1 \times (Q_2 - F_2)$

Here are some operations on languages that do not affect recognizability:

- Boolean (union, intersection, complement)
- concatenation, Kleene star
- reversal
- homomorphisms, inverse homomorphisms

Alas, it is difficult to establish some of these properties within the framework of DFAs: the constructions of the corresponding machines become rather complicated. We will exploit nondeterminism later for this purpose.

We can now deal more intelligently with the Equivalence problem from last time.

Problem:	**Equivalence**
Instance:	Two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$.
Question:	Are the two machines equivalent?

Lemma

$\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent iff $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2) = \emptyset$ and $\mathcal{L}(\mathcal{A}_2) - \mathcal{L}(\mathcal{A}_1) = \emptyset$.

Note that the lemma yields a quadratic time algorithm. We will see a better method later.

Observe that we actually are solving two instances of a closely related problem here:

Problem:	**Inclusion**
Instance:	Two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$.
Question:	Is $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$?

which problem can be handled by

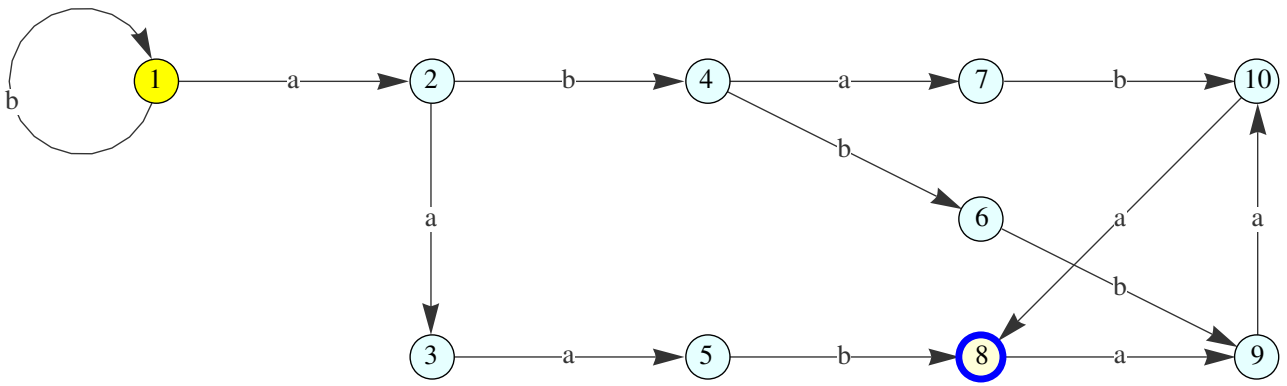
Lemma

$\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ *iff* $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2) = \emptyset$.

Note that for any class of languages Equivalence is decidable when Inclusion is so decidable. However, the converse may be false – but it's not so easy to come up with an example.

Product constructions are useful even for relatively simple languages, it can be quite difficult to build automata for, say, the intersection of two recognizable languages directly by hand.

Example: build a DFA for the language of all words that contain the scattered subword (not factor) ab 3 times, and a multiple-of-3 number of a's. Building the two component machines and taking their product we get the PDFA



Definition

Given two languages $L_1, L_2 \subseteq \Sigma^\star$ their concatenation (or product) is defined by

$$L_1 \cdot L_2 = \{\, xy \mid x \in L_1, y \in L_2 \,\}.$$

Let L be a language. The powers of L are the languages obtained by repeated concatenation:

$$L^0 = \{\varepsilon\}$$
$$L^{k+1} = L^k \cdot L$$

The Kleene star of L is the language

$$L^\star = L^0 \cup L^1 \cup L^2 \ldots \cup L^n \cup \ldots$$

Kleene star corresponds roughly to a while-loop or iteration.

Example

$\{a, b\}^\star$: all words over $\{a, b\}$

Example

$\{a, b\}^\star\{a\}\{a, b\}^\star\{a\}\{a, b\}^\star$: all words over $\{a, b\}$ containing at least two a's

Example

$\{\varepsilon, a, aa\}\{b, ba, baa\}^\star$: all words over $\{a, b\}$ not containing a subword aaa

Example

$\{0, 1\}\{0, 1\}^\star$: all numbers in binary, with leading 0's
$\{1\}\{0, 1\}^\star \cup \{0\}$: all numbers in binary, no leading 0's

Concatenation Closure

Given DFAs $\mathcal{A}_i$ for recognizable languages L_i, we want to construct a new DFA $\mathcal{A}$ for $L_1 \cdot L_2$. We need to split the string $x = uv$:

$$x = \underbrace{x_1 x_2 \dots x_k}_{u \in L_1} \underbrace{x_{k+1} \dots x_n}_{v \in L_2}$$

The problem is that we don't know where to split: in general, there are multiple prefixes u in L_1, but not all corresponding suffixes v are in L_2.

In our nondeterminism infused world it is tempting to "guess" when to split, but DFAs cannot guess.

Pebbles

Here is a trick that sometimes helps to construct machines in particular with lower bound arguments for state complexity. Assume we have some transition system (not necessarily deterministic).

- Initially, we place a few pebble on some states (typically initial states).
- Under input a, a pebble on p multiplies and moves to all q such that $p \xrightarrow{a} q$. If there are no transitions with source p, the pebble dies.
- Multiple pebbles on the same state coalesce into a single one.
- We accept whenever a pebble appears in F.

Note: The movement of the set of all pebbles is perfectly deterministic.

So even when the given transition system is nondeterministic, this method produces a deterministic machine.

We start with one copy of DFA $\mathcal{A}_1$, the master, and one copy of DFA $\mathcal{A}_2$, the slave.

- Place one pebble on the initial state of the master machine.
- Move the pebbles according to our standard rules.
- Whenever the master pebble reaches a final state, place a new pebble on the initial state of the slave automaton.
- The composite machine accepts if a pebble sits on final state in the slave machine.

Another way of thinking about the same construction is to have $|\mathcal{A}_2|$ many copies of the second DFA, each with just one pebble.

The number of states in the new DFA is bounded by

$$|\mathcal{A}_1|2^{|\mathcal{A}_2|}$$

since the $\mathcal{A}_1$ part is deterministic but the $\mathcal{A}_2$ part is not.

The states are of the form (p, P) where $p \in Q_1$ and $P \subseteq Q_2$, corresponding to a complete record of the positions of all the pebbles.

Of course, the accessible part may well be smaller. Alas, in general the bound is essentially tight.

Here is a straightforward generalization of DFAs that allows for nondeterministic behavior. Recall that transition systems may well be nondeterministic.

Definition (NFA)

A nondeterministic finite automaton (NFA) is a structure

$$\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$$

where $\langle Q, \Sigma, \tau \rangle$ is a transition system and the acceptance condition is given by $I, F \subseteq Q$, the initial and final states, respectively.

So in general there is no unique next state in an NFA: there may be no next state, or there may be many. Of course, we can think of a DFA as a special type of NFA.

Some authors insist that $I = \{q_0\}$. This makes no sense.

It is straightforward to lift the definition of acceptance from DFAs to NFAs (it all comes down to path existence, anyway).

Recall that in any transition system $\langle Q, \Sigma, \tau \rangle$ a run is an alternating sequence

$$\pi = p_0, a_1, p_1, \ldots, a_r, p_r$$

where $p_i \in Q$, $a_i \in \Sigma$ and $\tau(p_{i-1}, a_i, p_i)$ for all $i = 1, \ldots, r$. p_0 is the source of the run and p_r its target. The length of π is r.

The corresponding trace or label is the word $a_1 a_2 \ldots a_r$.

The acceptance condition is essentially the same as for DFAs, except that initial states are no longer unique (and even if they were, there could be multiple traces).

Definition

An NFA $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ accepts a word $w \in \Sigma^\star$ if there is a run of $\mathcal{A}$ with label w, source in I and target in F. We write $\mathcal{L}(\mathcal{A})$ for the acceptance language of $\mathcal{A}$.

But note that now there may be exponentially many runs with the same label. In particular, some of the runs starting in I may end up in F, others may not.

There is a hidden existential quantifier here.

Again: all that is needed for acceptance is one accepting run, there may be many runs that fail to lead to acceptance.

Note that nondeterminism can arise from two different sources:

- Transition nondeterminism:
  there are different transitions $p \xrightarrow{a} q$ and $p \xrightarrow{a} q'$.

- Initial state nondeterminism:
  there are multiple initial states.

In other words, even if the transition relation is deterministic we obtain a nondeterministic machine by allowing multiple initial states. Intuitively, this second type of nondeterminism is less wild.

While we are at it: there is yet another natural generalization beyond just nondeterminism: autonomous transitions, aka epsilon moves. These are transitions where no symbol is read, only the state changes.

Definition

A nondeterministic finite automaton with ε-moves (NFAE) is defined like an NFA, except that the transition relation has the format $\tau \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.

Thus, an NFAE may perform several transitions without scanning a symbol.

Hence a trace may now be longer than the corresponding input word. Other than that, the acceptance condition is the same as for NFAs: there has to be run from an initial state to a final state.

We will encounter several occasions where it is convenient to "enlarge" the alphabet Σ by adding the empty word ε:

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$$

Of course, ε is not a new alphabet symbol. What's really going on?

Σ freely generates the monoid $\Sigma^\star$, and ε is the unit element of this monoid. We can add the unit element to the generators without changing the monoid.

We could even consider generalized finite state machines: allow super-transitions like

$$p \xrightarrow{aba} q$$

Exercise

Explain why this makes no difference as far as languages are concerned.

Here is a perfect example of an operation that preserves recognizability, but is difficult to capture within the confines of DFAs.

Let

$$L^{\mathrm{op}} = \{\, x^{\mathrm{op}} \mid x \in L \,\}$$

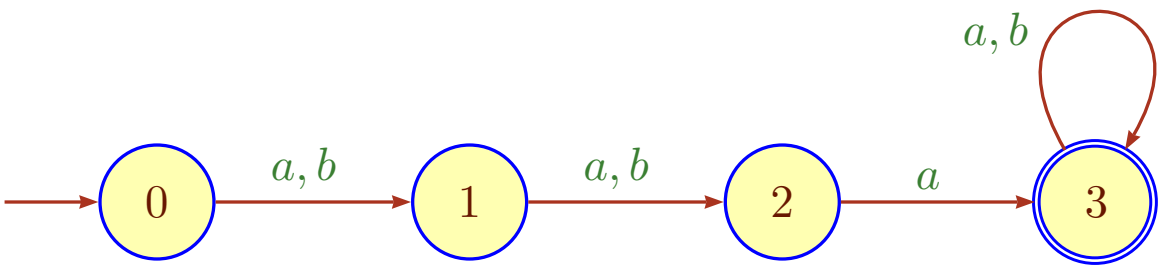
be the reversal of a language, $(x_1 x_2 \ldots x_{n-1} x_n)^{\mathrm{op}} = x_n x_{n-1} \ldots x_2 x_1$.

The direction in which we read a string should be of supreme irrelevance, so for recognizable languages to form a reasonable class they should be closed under reversal.

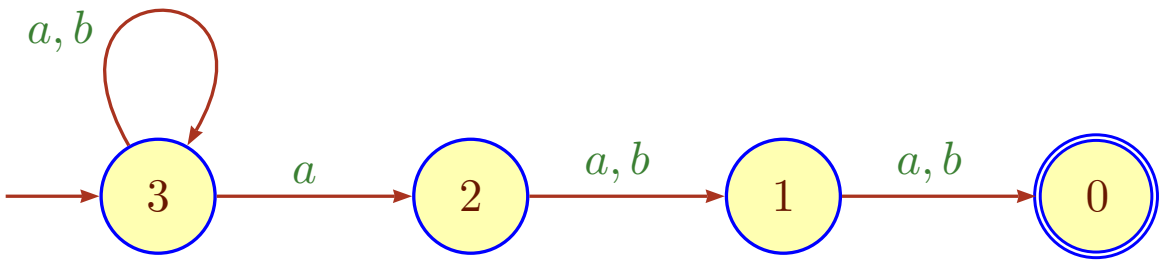
Suppose L is recognizable. How would we go about constructing a machine for L^{op}?

It is very easy to build a DFA for $L_{a,3} = \{ x \mid x_3 = a \}$.

We omit the sink to keep the diagram simple.



But $L^{\mathrm{op}}_{a,3} = \{\, x \mid x_{-3} = a \,\} = L_{a,-3}$ is somewhat hard for DFAs: we don't know how far from the end we are. Here is a perfectly legitimate NFA for this language: we flip transitions and interchange initial and final states.



It is clear that the new machine accepts $L_{a,-3}$.

The Problem: Is the acceptance language of an NFA still recognizable?

Our first order of business is to show that NFAs and NFAEs are no more powerful than DFAs in the sense that they only accept recognizable languages. Note, though, that the size of the machines may change in the conversion process, so one needs to be a bit careful.

The transformation is effective: the key algorithms are

Epsilon Elimination Convert an NFAE into an equivalent NFA.

Determinization Convert an NFA into an equivalent DFA.

As already mentioned, terminology for finite state machines is not settled, we will use the following hierarchy:

$$\text{DFA} \subseteq \text{PDFA} \subseteq \text{NFA} \subseteq \text{NFAE} \subseteq \text{GFA}$$

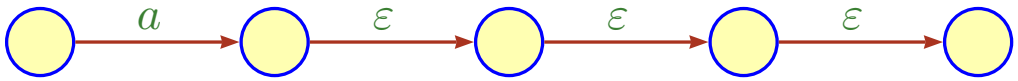
The heart of the OO fanatic beats faster . . .

Epsilon elimination is quite straightforward and can easily be handled in polynomial time:

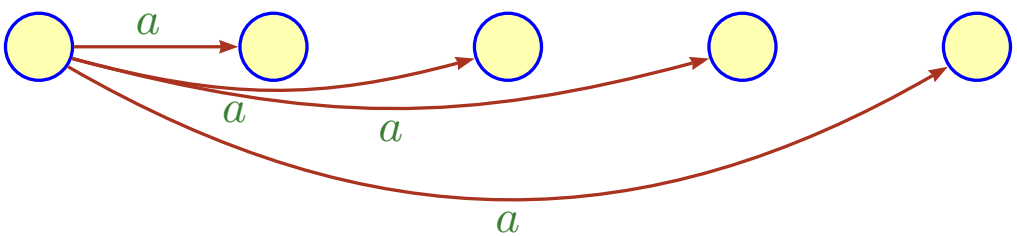
- introduce new ordinary transitions that have the same effect as chains of ε transitions, and
- remove all ε-transitions.

Since there may be chains of ε-transitions this is in essence a transitive closure problem. Hence part I of the algorithm can be handled with the usual graph techniques.

A transitive closure problem: we have to replace chains of transitions



by new transitions



Theorem

For every NFAE there is an equivalent NFA.

Proof. This requires no new states, only a change in transitions.

Suppose $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ is an NFAE for L. Let

$$\mathcal{A}' = \langle Q, \Sigma, \tau'; I', F \rangle$$

where τ' is obtained from τ as on the last slide.

I' is the ε-closure of I: all states reachable from I using only ε-transitions. □

Again, there may be quadratic blow-up in the number of transitions and it may well be worth the effort to try to construct the NFAE in such a way that this blow-up does not occur.

In the realm of finite state machines, nondeterministic machines are no more powerful than deterministic ones (this is also true for register/Turing machines, but fails for pushdown automata).

Theorem (Rabin, Scott 1959)

For every NFA there is an equivalent DFA.

The idea is to keep track of the set of possible states the NFA could be in. This produces a DFA whose states are sets of states of the original machine.

$$\tau \subseteq Q \times \Sigma \times Q$$

$$\tau : Q \times \Sigma \times Q \longrightarrow \mathbf{2}$$

$$\tau : Q \times \Sigma \longrightarrow (Q \longrightarrow \mathbf{2})$$

$$\tau : Q \times \Sigma \longrightarrow \mathfrak{P}(Q)$$

$$\tau : \mathfrak{P}(Q) \times \Sigma \longrightarrow \mathfrak{P}(Q)$$

The latter function can be interpreted as the transition function of a DFA on $\mathfrak{P}(Q)$. Done.

;-)

Suppose $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ is an NFA. Let

$$\mathcal{A}' = \langle \mathfrak{P}(Q), \Sigma, \delta; I, F' \rangle$$

where $\delta(P, a) = \{\, q \in Q \mid \exists\, p \in P \;\tau(p, a, q) \,\}$

$F' = \{\, P \subseteq Q \mid P \cap F \neq \emptyset \,\}$

It is straightforward to check by induction that $\mathcal{A}$ and $\mathcal{A}'$ are equivalent. □

The machine from the proof is the full power automaton of $\mathcal{A}$, written $\mathrm{pow}_{\mathrm{f}}(\mathcal{A})$, a machine of size 2^n.

Of course, for equivalence only the accessible part $\mathrm{pow}(\mathcal{A})$, the power automaton of $\mathcal{A}$, is required.

Accessible Part

This is as good a place as any to talk about "useless" states: states that cannot appear in any accepting computation and that can therefore be eliminated.

Definition

A state p in a finite automaton $\mathcal{A}$ is accessible if there is a run with source an initial state and target p. The automaton is accessible if all its states are.

Now suppose we remove all the inaccessible states from a automaton $\mathcal{A}$ (meaning: adjust the transition system and the set of final states). We obtain a new automaton $\mathcal{A}'$, the so-called accessible part of $\mathcal{A}$.

Lemma

The machines $\mathcal{A}$ and $\mathcal{A}'$ are equivalent.

There is a dual notion of coaccessibility: a state p is coaccessible if there is at least one run from p to a final state. Likewise, an automaton is coaccessible if all its states are.

An automaton is trim if it is accessible and coaccessible.

It is easy to see that the trim part of an automaton is equivalent to the whole machine. Moreover, we can construct the coaccessible and trim part in linear time using standard graph algorithms.

Warning: Note that the coaccessible part of a DFA may not be a DFA: the machine may become incomplete and we wind up with a partial DFA. The accessible part of a DFA always is a DFA, though.

In the RealWorld$^{\text{TM}}$ we would avoid the full power set at all costs: instead of building a DFA over $\text{pow}(Q)$ we would only construct the accessible part—which may be exponentially smaller. There are really two separate issues here.

- First, we may need to clean up machines by running an accessible (or trim) part algorithm whenever necessary–this is easy.
- Much more interesting is to avoid the construction of inaccessible states of a machine in the first place: ideally any algorithm should only produce accessible machines.

While accessibility is easy to guarantee, coaccessibility is not: while constructing a machine we do not usually know the set of final states ahead of time. So, there may by need to eliminate non-coaccessible states.

The right way to construct the Rabin-Scott automaton for $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ is to take a closure in the ambient set $\mathfrak{P}(Q)$:

$$\mathsf{clos}\Big(I,\ (\tau_a)_{a \in \Sigma} \Big)$$

Here τ_a is the function $\mathfrak{P}(Q) \times \Sigma \to \mathfrak{P}(Q)$ defined by

$$\tau_a(P) = \{\, q \in Q \mid \exists\, p \in P\, (p \xrightarrow{a} q) \,\}$$

This produces the accessible part only, and, with luck, is much smaller than the full power automaton.

Think of the labeled digraph

$$\mathcal{G} = \langle \mathfrak{P}(Q); \tau_1, \tau_2, \ldots, \tau_k \rangle$$

with edges $p \xrightarrow{a} q$ for $\tau_a(p) = q$, the virtual graph or ambient graph where we live. The graph is exponential in size, but we don't need to construct it explicitly.

We only need to compute the reachable part of $I \in \mathfrak{P}(Q)$ in this graph $\mathcal{G}$. This can be done using standard algorithms such as Depth-First-Search or Breadth-First-Search.

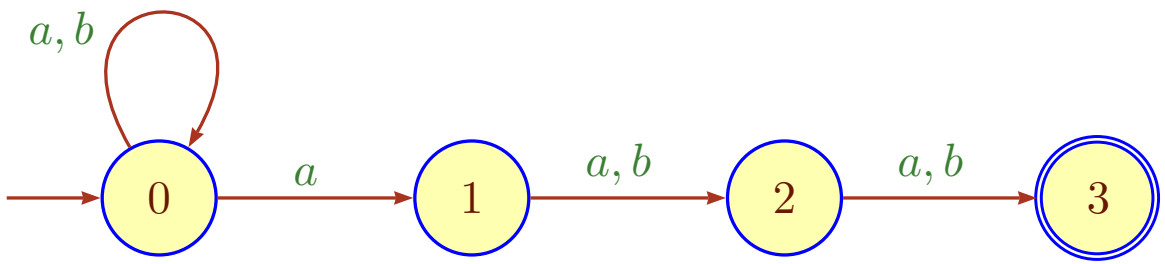
The only difference is that we are not given an adjacency list representation of $\mathcal{G}$: we compute edges on the fly. No problem at all.

This is very important when the ambient graph is huge: we may only need to touch a small part.

Recall

$$L_{a,k} = \{\, x \in \{a,b\}^\star \mid x_k = a \,\}.$$

For negative k this means: $-k$th symbol from the end. It is trivial to construct an NFA for $L_{a,-3}$:



Applying the Rabin-Scott construction we obtain a machine with 8 states

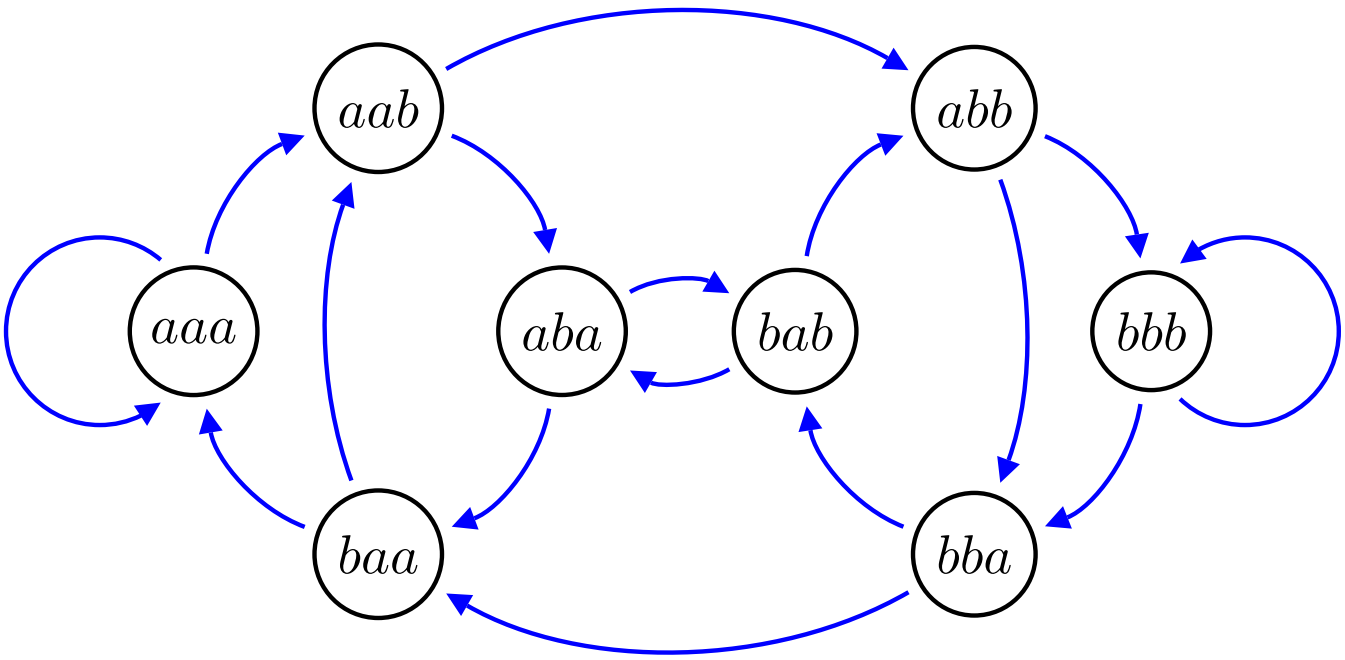
$$\{0\}, \{0,1\}, \{0,1,2\}, \{0,2\}, \{0,1,2,3\}, \{0,2,3\}, \{0,1,3\}, \{0,3\}$$

where 1 is initial and 5, 6, 7, and 8 are final. The transitions are given by

	1	2	3	4	5	6	7	8
a	2	3	5	7	5	7	3	2
b	1	4	6	8	6	8	4	1

Note that the full power set has size 16, our construction only builds the accessible part (which happens to have size 8).

Here is the corresponding diagram, rendered in a particularly brilliant way. This is a so-called de Bruijn graph (binary, rank 3).

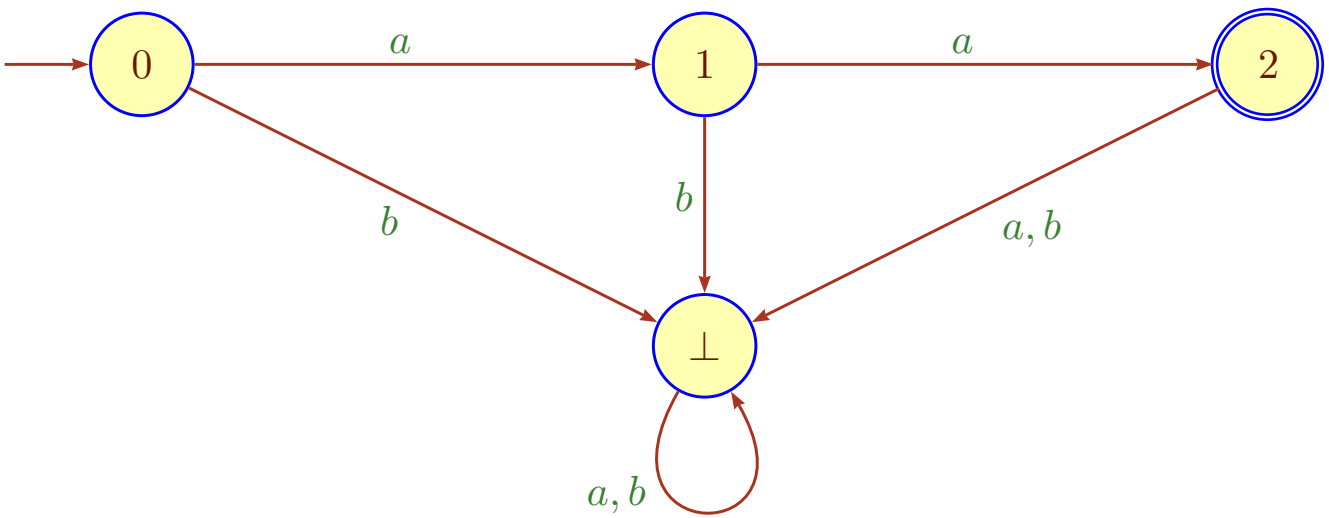


Exercise

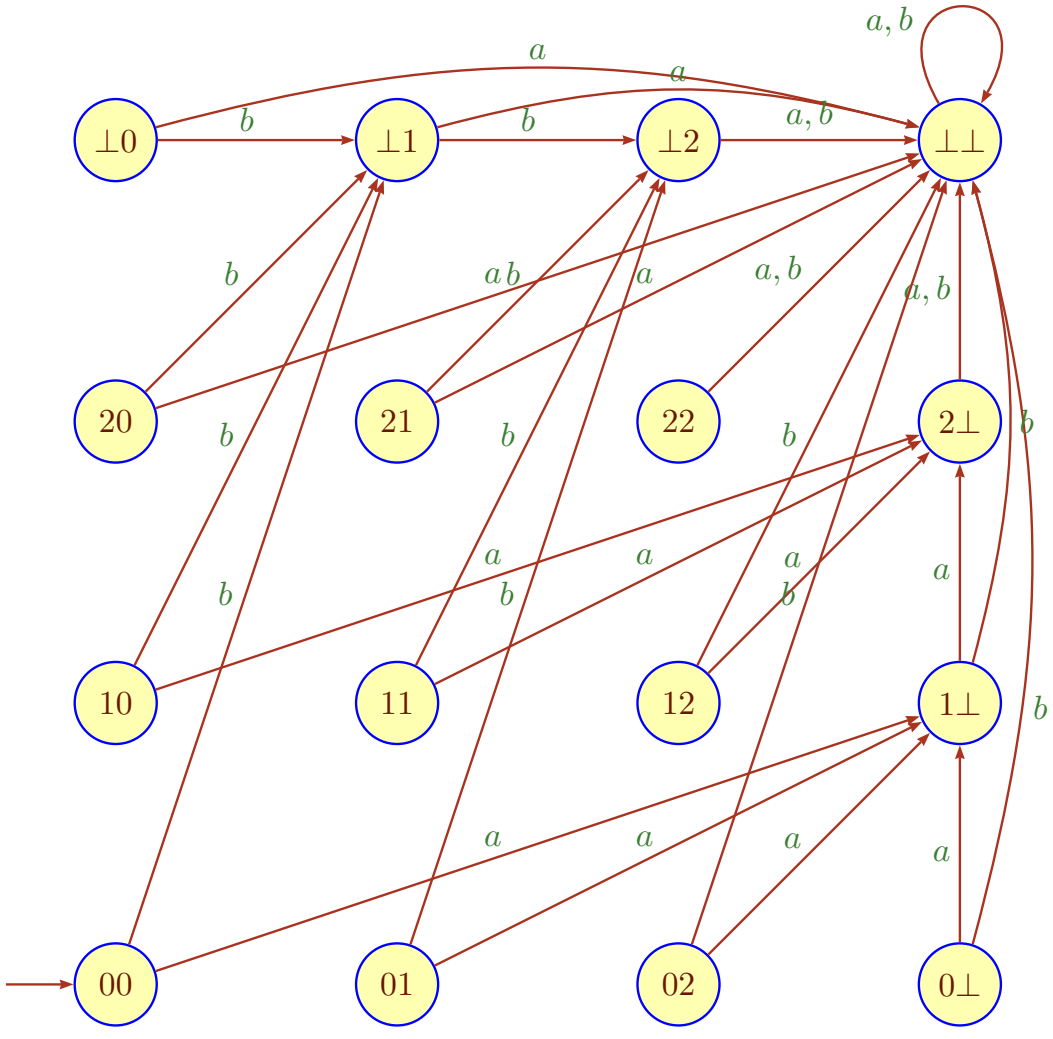
Explain this picture in terms of the Rabin-Scott construction.

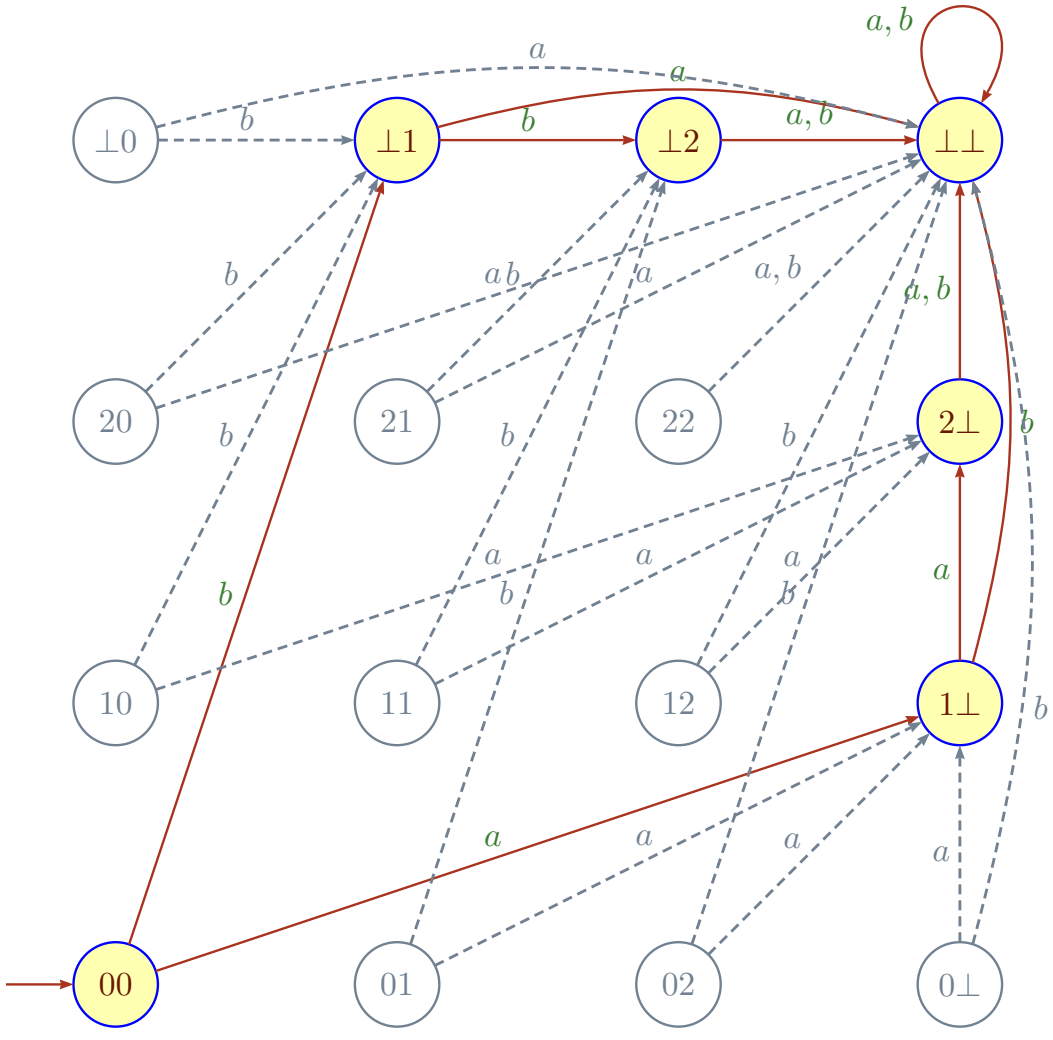
Consider the product automaton for DFAs $\mathcal{A}_{aa}$ and $\mathcal{A}_{bb}$, accepting aa and bb, respectively.

$\mathcal{A}_{aa}$:



Full Product Automaton




a, b
a
a
b
b
a, b
⊥0
⊥1
⊥2
⊥⊥
b
a b
a
a, b
a, b
20
21
22
2⊥
b
b
b
b
b
a
a
a
a
b
b
b
10
11
12
1⊥
b
a
a
a
a
a
00
01
02
0⊥

Recall one of the key applications of FSMs: acceptance testing is very fast and can be used to deal with pattern matching problems.

How much of a computational hit do we take when we switch to nondeterministic machines?

We can use the same approach as in determinization: instead of computing all possible sets of states reachable from I, we only compute the ones that actually occur along a particular trace given by some input word.

Here is a natural modification of the DFA acceptance testing program.

```
P = I;                          // set of ints
while(  a = x.next() )          // next input symbol
      P = tau_a(P);

return ( P intersect F != empty );
```

The update step uses the maps $\tau_a : \mathfrak{P}(Q) \to \mathfrak{P}(Q)$ from above.

Exercise

Think of dirty tricks like hashing to speed things up.

- The loop executes $|x|$ times, just as with DFAs.
- Unfortunately, the loop body is no longer constant time: we have to update a set of states $P \subseteq Q$.
- This can certainly be done in $O(|Q|^2)$ steps though smart data structures may sometimes give better performance.
- Actually, it seems that in practice (i.e. in NFAs that appear naturally in some application such as pattern matching) one often deals with overhead that is linear in $|Q|$ rather than quadratic.
- At any rate, we can check acceptance in an NFA in $O(|x||Q|^2)$ steps. For fixed machines this is still linear in x, but the hidden constant may be significantly worse than in a DFA.

A Better Mousetrap?

Acceptance testing is slower, nondeterministic machines are not simply all-round superior to DFAs.

- Advantages:
  Easier to construct and manipulate.
  Sometimes exponentially smaller.
  Sometimes algorithms much easier.

- Drawbacks:
  Acceptance testing slower.
  Sometimes algorithms more complicated.

Which type of machine to choose in a particular application can be a hard question, there is no easy general answer.

Suppose we have two NFAs over Σ: $\mathcal{A}_i = \langle Q_i, \tau_i; I_i, F_i \rangle$. We may safely assume that the state sets are disjoint. There are two simple operations the combine the computations of both machines:

- Sum Automaton

$$\mathcal{A}_1 + \mathcal{A}_2 = \langle Q_1 \cup Q_2, \tau_1 \cup \tau_2; I_1 \cup I_2, F_1 \cup F_2 \rangle.$$

- Cartesian Product Automaton

$$\mathcal{A}_1 \times \mathcal{A}_2 = \langle Q_1 \times Q_2, \tau_1 \times \tau_2; I_1 \times I_2, F_1 \times F_2 \rangle$$

In the product construction

$$((p,q),a,(p',q')) \in \tau_1 \times \tau_2 \iff \tau_1(p,a,p') \wedge \tau_2(q,a,q')$$

Clearly, the computations of $\mathcal{A}_1 + \mathcal{A}_2$ are exactly the union of the computations of $\mathcal{A}_1$ and $\mathcal{A}_2$.

The size of the sum automaton is linear in the size of the components.

The computations of $\mathcal{A}_1 \times \mathcal{A}_2$ are the computations of $\mathcal{A}_1$ combined with the computations of $\mathcal{A}_2$ provided both have the same label: essentially, we are running both machines in parallel.

A real implementation will only construct the accessible part, but still, the size of $\mathcal{A}_1 \times \mathcal{A}_2$ is potentially quadratic in the sizes of $\mathcal{A}_1$ and $\mathcal{A}_2$. This causes problems if a product machine construction is used repeatedly.

By our choice of acceptance condition we have

$$\mathcal{L}(\mathcal{A}_1 + \mathcal{A}_2) = L_1 \cup L_2$$
$$\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = L_1 \cap L_2$$

By changing the final states in the product, we can also get union:

union	$F = F_1 \times Q_2 \cup Q_1 \times F_2$
intersection	$F = F_1 \times F_2$

Why bother with a quadratic product for union when we can get it cheaper from a linear sum?

It is tempting to try to use

$$F = F_1 \times (Q_2 - F_2)$$

to obtain complements $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2)$.

This works for DFAs, but not for NFAs. Determinism is essential here, we will see shortly that complementation for nondeterministic machines is much harder.

Exercise

Construct a counterexample that shows that the switch-final-states construction in general fails to produce complements in NFAs, even in trim ones.

Suppose we have two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ for L_1 and L_2. To build a machine for $L_1 \cdot L_2$ it is easiest to go with an NFAE $\mathcal{A}$:

Let $Q = Q_1 \cup Q_2$ disjoint, keep all the old transitions and add ε-transitions from F_1 to I_2. I_1 are the initial states and F_2 the final states in $\mathcal{A}$.

It is clear that $\mathcal{L}(\mathcal{A}) = L_1 \cdot L_2$. But note that this construction may introduce quadratically many transitions.

Exercise

Find a way to keep the number of new transitions linear.

Definition

A homomorphism is a map $f : \Sigma^\star \to \Gamma^\star$ such that

$$f(x_1 x_2 \dots x_n) = f(x_1) f(x_2) \dots f(x_n)$$

where $x_i \in \Sigma$. In particular $f(\varepsilon) = \varepsilon$.

Note that a homomorphism can be represented by a finite table: we only need $f(a) \in \Gamma^\star$ for all $a \in \Sigma$.

Given a homomorphism $f : \Sigma^\star \to \Gamma^\star$ and languages $L \subseteq \Sigma^\star$ and $K \subseteq \Gamma^\star$ we are interested in the languages

image	$f(L) = \{ f(x) \mid x \in L \}$
inverse image	$f^{-1}(K) = \{ x \mid f(x) \in K \}$

Lemma

Regular languages are closed under homomorphisms and inverse homomorphisms.

Proof.

Say, we have a DFA $\mathcal{A}$ for L.

Replace the labels of the transitions as follows

$$p \xrightarrow{a} q \quad \rightsquigarrow \quad p \xrightarrow{f(a)} q$$

This produces a GFA over Γ which can easily be converted into an NFAE.

□

Exercise

Give a proof of closure under inverse homomorphisms.

Exercise

Carry out the concatenation pebble construction for the languages E_a = even number of a's and E_b = even number of b's and run some examples.

Exercise

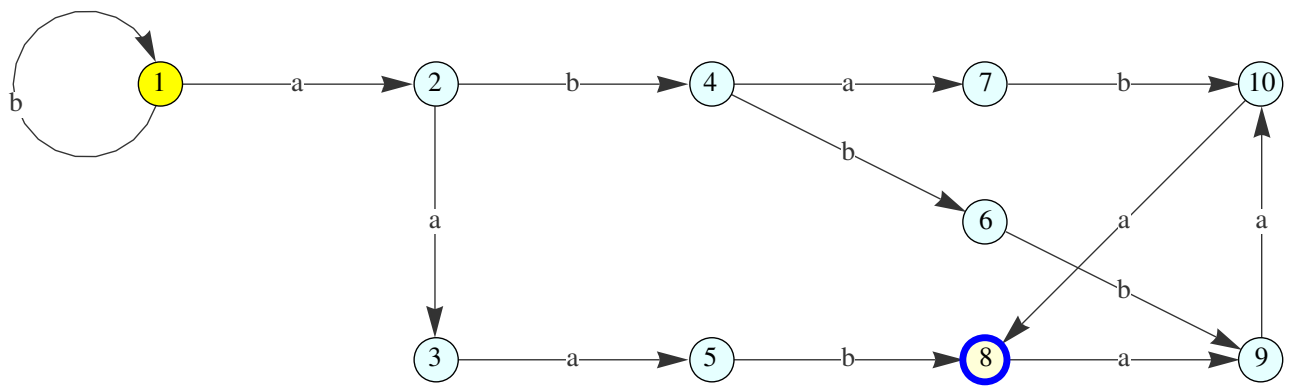
Carry out a pebbling construction for Kleene star.

Exercise

How would a pebbling construction work when the given machine(s) are NFAEs?

Product constructions are important even for relatively simple languages, it can be quite difficult to build automata for, say, the intersection of two recognizable languages directly by hand.

Here is an example: build a DFA for the language of all words that contain the scattered subword (not factor) ab 3 times, and a multiple-of-3 number of a's. Building the two component machines and taking their product we get



	DFA	NFA
intersection	mn	mn
union	mn	$m+n$
concatenation	$(m-1)2^n-1$	$m+n$
Kleene star	$3\cdot 2^{n-2}$	$n+1$
reversal	2^n	n
complement	n	2^n

Worst case blow-up starting from machine(s) of size m, n and applying the corresponding operation (accessible part only).

Note that we are only dealing the state complexity, not transition complexity (which is arguably a better measure for NFAs).

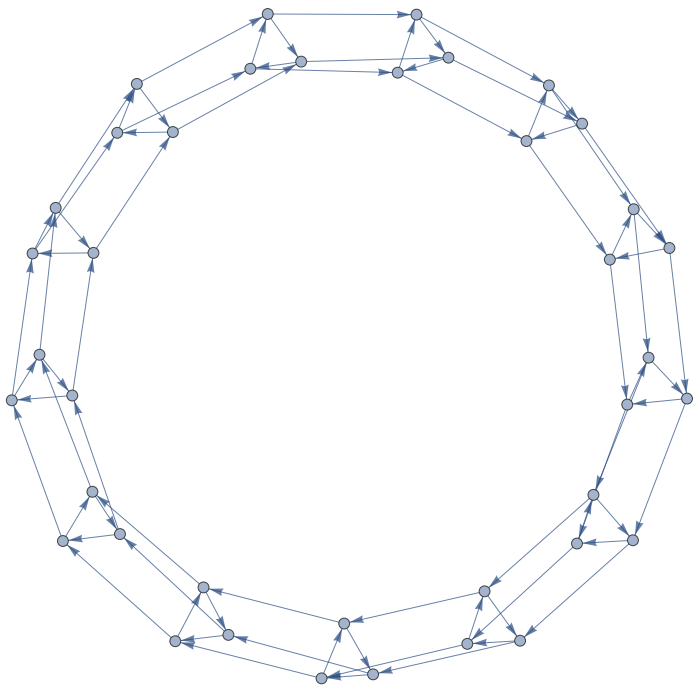
Example: Intersection

Let

$$K_{a,m} = \{\, x \in \mathbf{2}^\star \mid \#_a x = 0 \pmod m \,\}$$

be the "mod-counter" language. Clearly $K_{a,m}$ has state complexity m.

The intersection of $K_{0,m}$ and $K_{1,n}$ has state complexity mn.



- The seminal 1959 paper by Rabin and Scott also introduced the study of the computational complexity of various decision problems associated with finite state machines.

- We have already seen some of these: Emptiness, Finiteness, Universality, Equality and Inclusion. For DFAs they are all easily solvable (linear or quadratic time).

- For NFAs the situation is more complicated (NFAEs are not relevant here since they can be converted to NFAs in cubic time).

Again: Decision Problems

Problem:	**Emptiness Problem**
Instance:	A regular language L.
Question:	Is L empty?

Problem:	**Finiteness Problem**
Instance:	A regular language L.
Question:	Is L finite?

Problem:	**Universality Problem**
Instance:	A regular language L.
Question:	Is $L = \Sigma^\star$?

For DFAs these problems are all easily handled in linear time using depth-first-search.

As far as decidability is concerned there is no difference between DFAs and NFAs: we can simply convert the NFA.

But the determinization may be exponential, so efficiency becomes a problem.

- Emptiness and Finiteness are easily polynomial time for NFAs.
- Universality is PSPACE-complete for NFAs.

Problem:	**Equality Problem**
Instance:	Two regular languages L_1 and L_2.
Question:	Is L_1 equal to L_2?

Problem:	**Inclusion Problem**
Instance:	Two regular languages L_1 and L_2.
Question:	Is L_1 a subset of L_2?

- Inclusion is PSPACE-complete for NFAs.
- Equality is PSPACE-complete for NFAs.

Suppose we have a list of m DFAs $\mathcal{A}_i$ of size n_i, respectively.

Then the full product machine

$$\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_{m-1} \times \mathcal{A}_m$$

has $n = n_1 n_2 \ldots n_s$ states.

- The full product machine grows exponentially, but its accessible part may be much smaller.
- Alas, there are cases where exponential blow-up cannot be avoided.

Bad News: DFA Intersection

Here is the Emptiness Problem for a list of DFAs rather than just a single machine:

Problem:	**DFA Intersection**
Instance:	A list $\mathcal{A}_1, \ldots, \mathcal{A}_n$ of DFAs
Question:	Is $\bigcap \mathcal{L}(\mathcal{A}_i)$ empty?

This is easily decidable: we can check Emptiness on the product machine $\mathcal{A} = \prod \mathcal{A}_i$. The Emptiness algorithm is linear, but it is linear in the size of $\mathcal{A}$, which is itself exponential. And, there is no universal fix for this:

Theorem

The DFA Intersection Problem is PSPACE*-hard.*