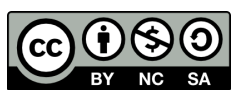
CDM

Closure Properties

Klaus Sutner

Carnegie Mellon University

Spring 2021



1 Nondeterministic Machines

2 Determinization

3 More Closure Properties

Total Recall: Closure

Regular languages are closed with respect to a number of operations. In fact, it is sort of difficult to turn a regular language into a non-regular one, using some "reasonable" operation.

- Union, intersection, complement.
- Concatentation, Kleene star.
- Reversal.
- Homomorphisms, inverse homomorphisms, regular substitutions.

Theorem (Kleene 1956)

Every regular language over Σ can be constructed from $\emptyset$ and $\{a\}$, $a \in \Sigma$, using only the operations union, concatenation and Kleene star.

Hence we denote any regular language by an algebraic expression:

Definition

A regular expression is a term constructed as follows:

- Basic expressions: $\emptyset$, a for $a \in \Sigma$.
- Operators: $(E_1 + E_2)$, $(E_1 E_2)$, $(E^\star)$.

Theorem

There are higly efficient algorithms to convert a regular expression into a corresponding finite state machine.

This result is critical for applications; without it, pattern matching would be a nightmare: one would have to input a FSM rather than the expression.

Theorem

There are algorithms to convert a finite state machine into a corresponding regular expression.

Unfortunately, the expressions are typically exponentially large, so this result is mostly of academic interest.

Develop a toolbox of algorithms that manipulate finite state machines. Try to make these algorithms as efficient as possible; prove hardness results if no efficient solution exists.

For example, we already know how to solve Recognition and Emptiness in linear time. The algorithms provide constructive proofs for the all the closure properties claimed above.

Unsurprisingly, the algorithms for deterministic versus nondeterministic machines are often quite different. As we will see, sometimes PDFA are easier to deal with, sometimes NFA are.

Here is one important idea: we want use parallel composition. Say, we have two FAs over Σ: $\mathcal{A}_i = \langle Q_i, \Sigma, \tau_i; I_i, F_i \rangle$. To run the machines in parallel we define a new machine as follows:

Definition (Cartesian Product Automaton)

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

where $\tau = \tau_1 \times \tau_2$ is defined by

$$\big((p,q), a, (p',q')\big) \in \tau \Leftrightarrow (p, a, p') \in \tau_1, (q, a, q') \in \tau_2$$

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

By our choice of acceptance condition we have

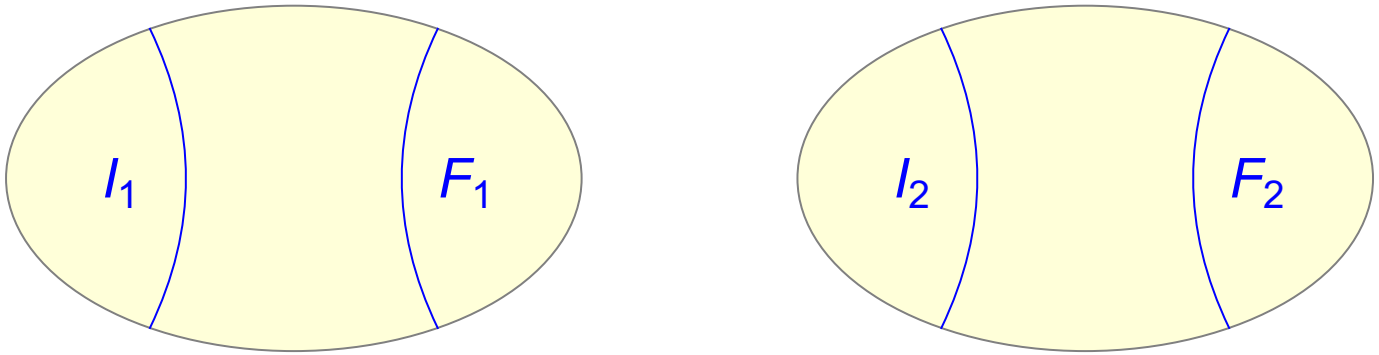
$$\mathcal{L}(\mathcal{A}_1 \times \mathcal{A}_2) = L_1 \cap L_2$$

For union we can resort to a sledgehammer construction, disjoint union or sum. We may safely assume that the state sets are disjoint.

Definition (Sum)

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

In other words, we simply declare the two machines to be one machine.



This is trivially linear time, but it wrecks DFA, the result is always nondeterministic.

There are two disctinct source of nondeterminism:

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

- Initial state nondeterminism:
  there are multiple initial states.

Transition-deterministic automata with multiple initial states are called multi-entry automata.

For DFA, the product construction produces another DFA.

The new initial state is (q_{01}, q_{02}) and the new transition function is $\delta = \delta_1 \times \delta_2$, defined by

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

In this case, we can get all Boolean operations out of a product construction:

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

But complement fails catastrophically for nondeterministic machines.

We can now deal more intelligently with the Equivalence problem from last time, at least in the case where the machines are DFAs.

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.

Deciding Inclusion

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.

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.

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

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

and feed the first part to $\mathcal{A}_1$ and the second to $\mathcal{A}_2$.

The key 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.

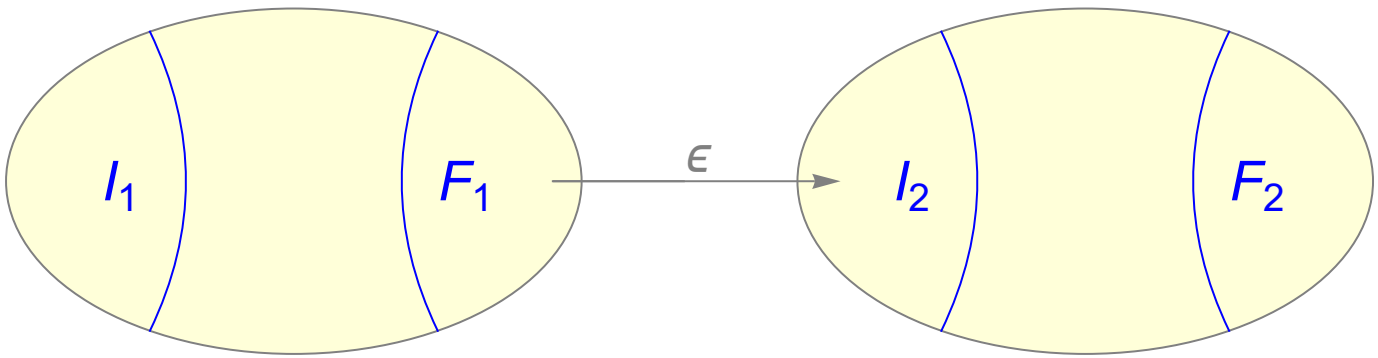
This is a place where nondeterminism is critical: we just "guess" when to split (i.e., jump form the first machine to the second).

Alas, there is a problem: our transitions are associated with reading a symbol, here we just want to jump.

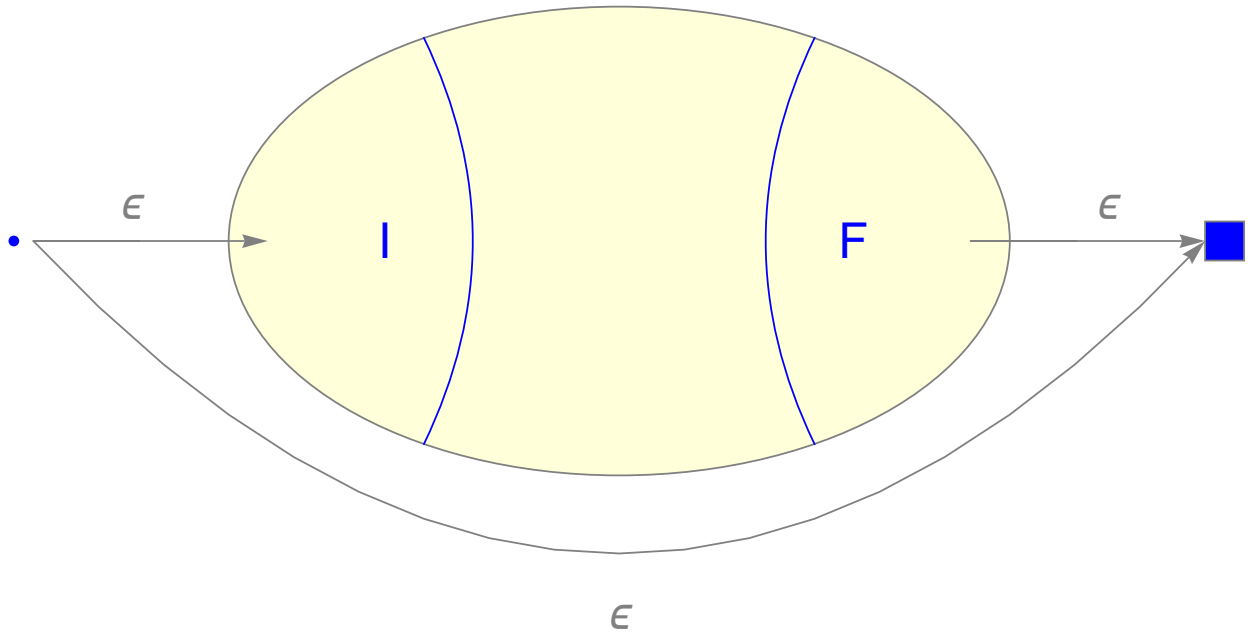
No problem, we just add a new kind of transition with label empty word.

Definition

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



Place an ε-transition between all states in F_1 and I_2 (potentially quadratically many).



ε-transitions also dispatch Kleene star. For example, we could add a new initial state, a new final state and transitions as indicated.

While we are at it, we could also allow transitions to be labled by arbitrary words over Σ. These are called generalized finite automata (GFA):

$$p \xrightarrow{aba} q$$

GFA are convenient to write down, but are no more powerful than just NFAE (which, ultimately, turn out to be no more powerful than NFA). We can just split the word transitions into a sequence of plain transitions:

$$p \xrightarrow{aba} q \rightsquigarrow p \xrightarrow{a} p_1, p_1 \xrightarrow{b} p_2, p_2 \xrightarrow{a} q$$

So we have the following hierarchy of FSM:

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

This is a feature, not a bug: one often uses different types of machines for different purposes, whichever kind works best under the circumstances.

Here is another example of an operation that preserves recognizability, but is difficult to capture within the confines of deterministic machines. For nondeterministic machines, on the other hand, it is entirely trivial.

Let

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

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

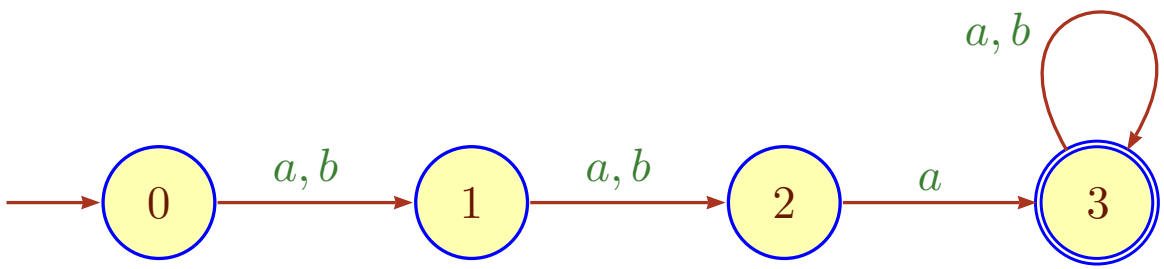
It turns out the L is recognizability iff L^{op} is recognizable.

This result is actually quite important: the direction in which we read a string should be of supreme irrelevance. We really want a language to be recognizable no matter whether we read left-to-right or right-to-left.

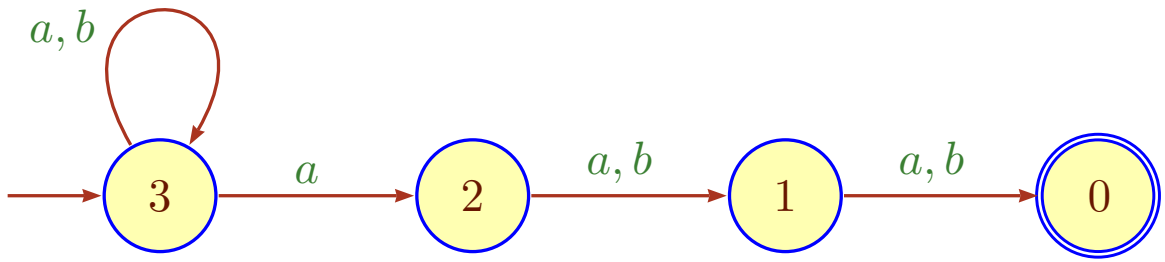
Example: Third Symbol

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 $L_{a,-3}$.

We will show in a momemt that GFA are in fact equivalent to DFA in the sense that every GFA can be converted into a DFA that accepts the same language.

Just as a little mental exercise, for once let's try to argue directly in terms of DFA, without the use of any helpful theorems.

Say, we want to build a DFA for the concatenation of two languages, given DFA for the languages.

Here is a psychological trick that sometimes helps to construct deterministic machines. 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 the DFA $\mathcal{A}_1$, the leader, and the DFA $\mathcal{A}_2$, the follower.

- Place one pebble on the initial state of the leader machine.
- Move the pebbles according to our standard rules.
- Whenever the leader pebble reaches a final state, place a new pebble on the initial state of the follower automaton.
- The composite machine accepts if a pebble sits on final state in the follower 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: there are multiple pebbles floating around in $\mathcal{A}_2$.

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.

1 Nondeterministic Machines

2 **Determinization**

3 More Closure Properties

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.

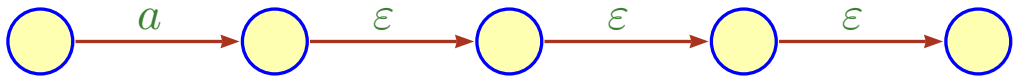
Determinization Convert an NFA into an equivalent DFA.

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

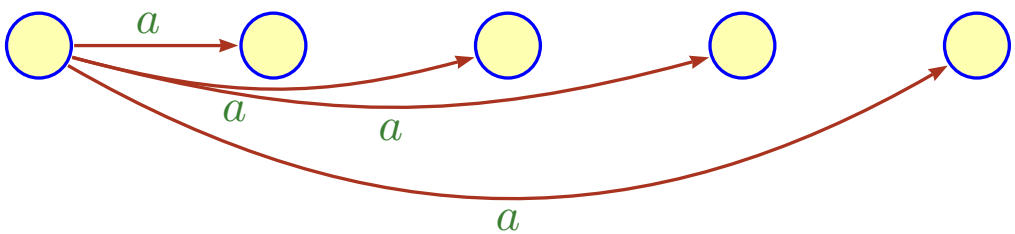
- introduce new ordinary transitions that have the same effect as chains of ε transitions, and
- remove all ε-transitions.

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.

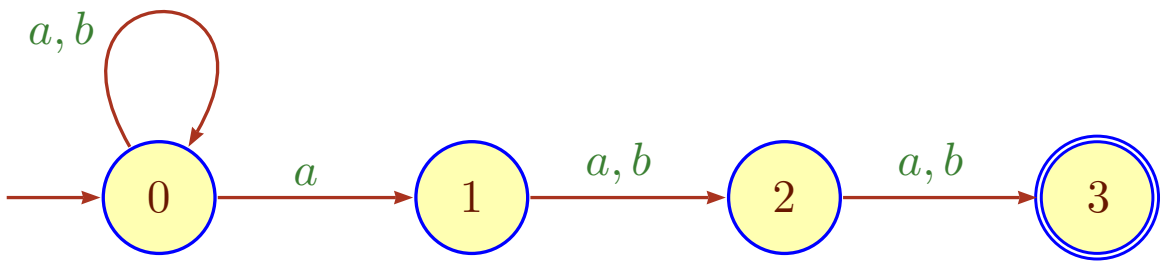
This is very important when the ambient graph is huge: we may only need to touch a small part.

Example: $L_{a,-3}$

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

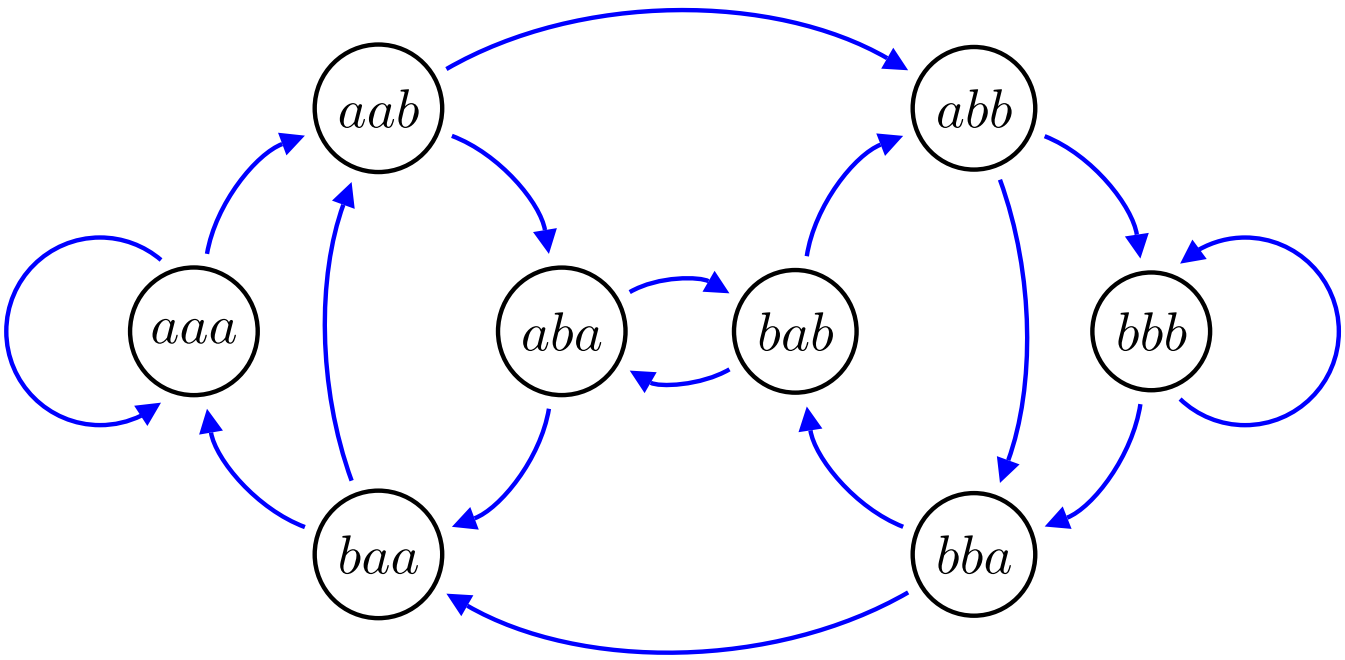
$$\{0\}, \{0,1\}, \{0,1,2\}, \{0,2\}, \{0,1,2,3\}, \{0,2,3\}, \{0,1,3\}, \{0,3\}$$

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).

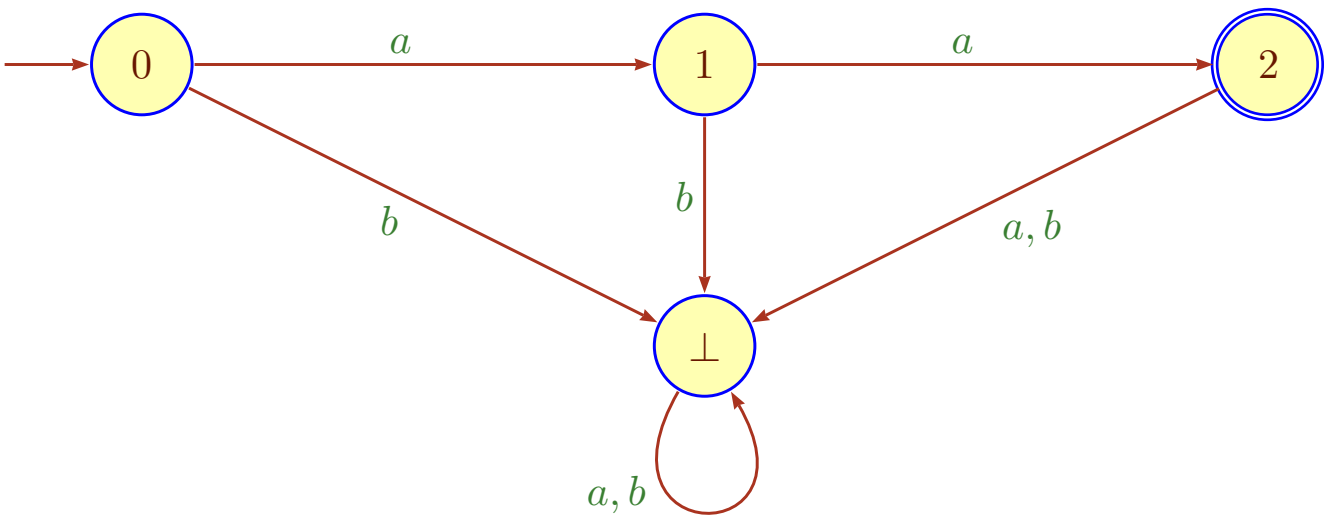


Exercise

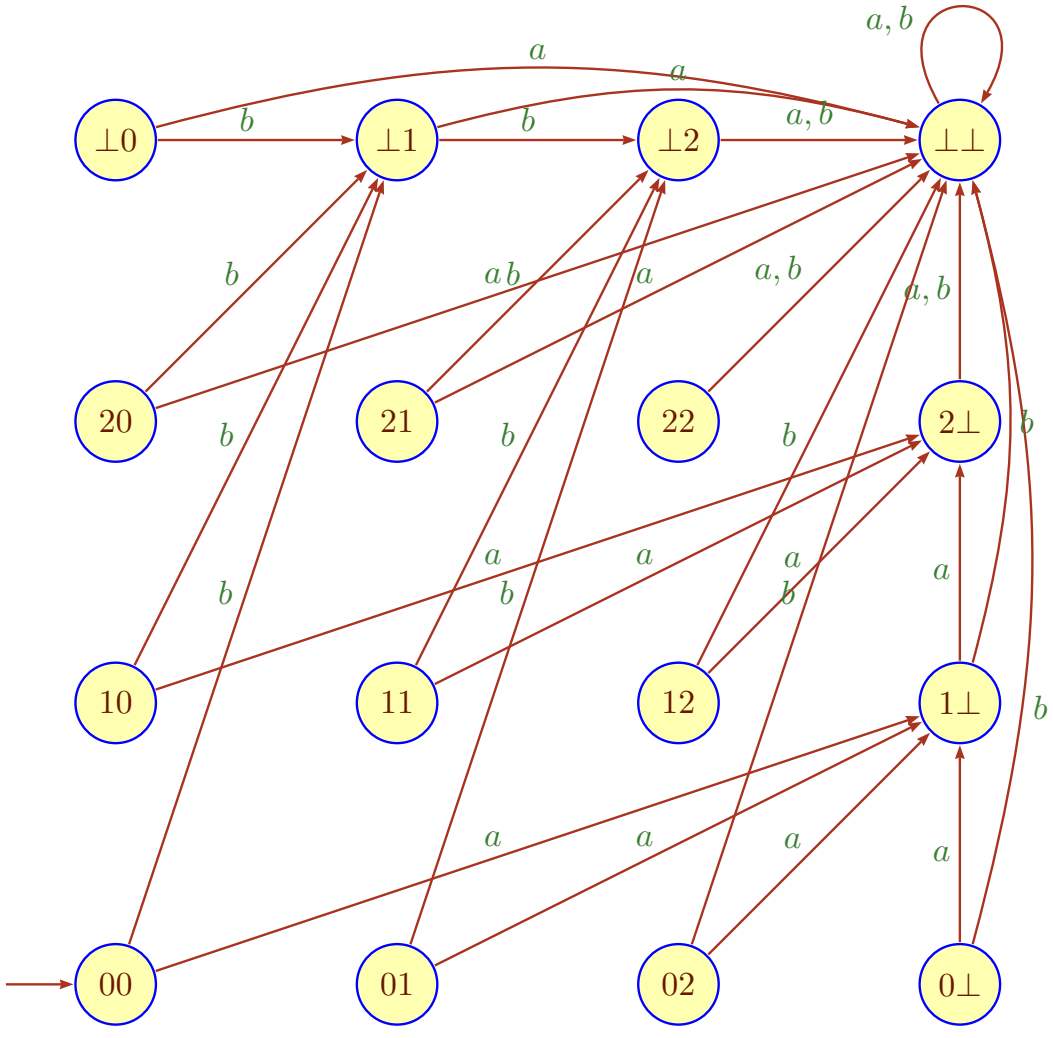
Explain this picture in terms of the Rabin-Scott construction.

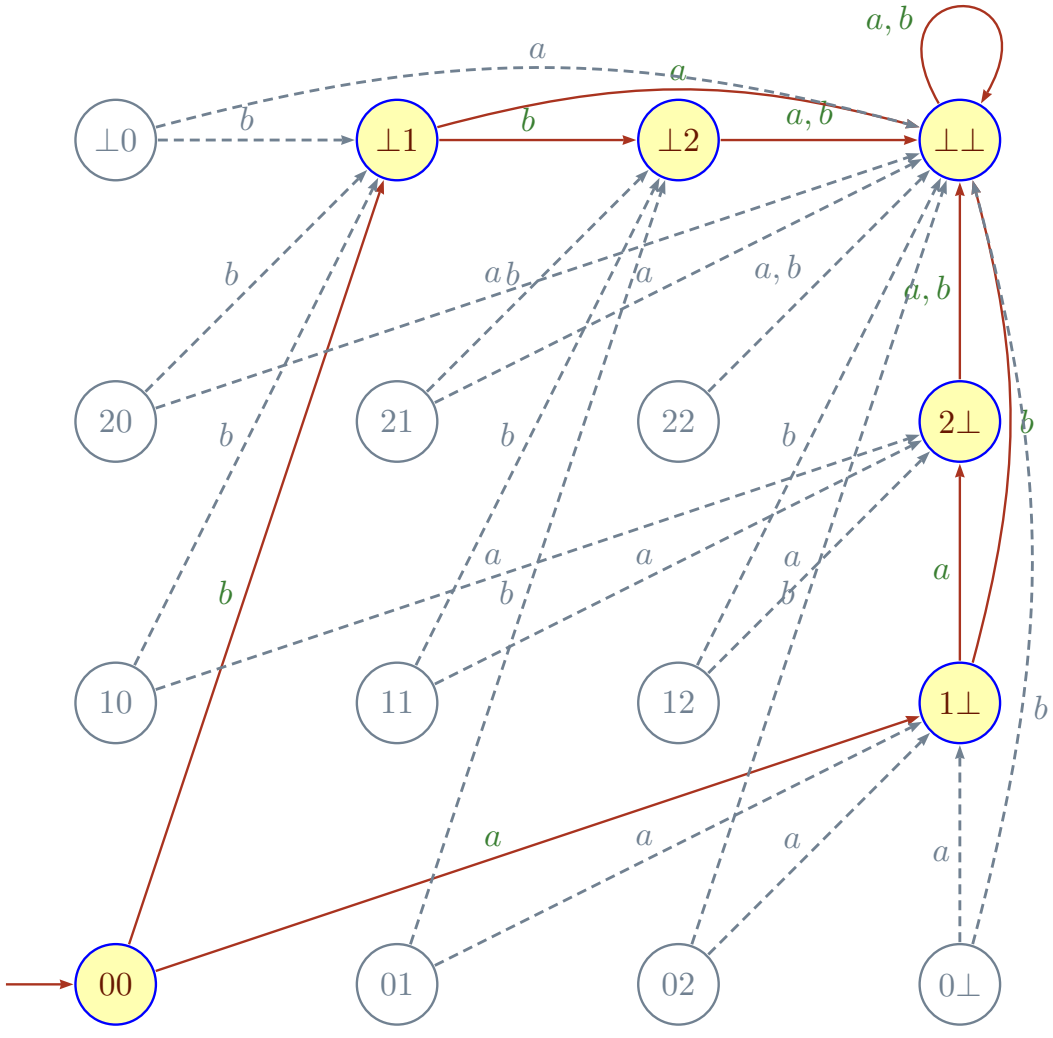
Consider the product automaton for DFAs $\mathcal{A}_{aa}$ and $\mathcal{A}_{bb}$, accepting aa and bb, respectively.

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



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
a
a
a
a
b
b
b
b
10
11
12
1⊥
b
a
a
a
a
a
00
01
02
0⊥

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.

1 Nondeterministic Machines

2 Determinization

3 More Closure Properties

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.

Let $f : \Sigma^\star \to \Gamma^\star$ be a homomorphism.

Say, we have a DFA $\mathcal{A}$ for $K \subseteq \Gamma^\star$. 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 Σ that accepts $f^{-1}(K)$.

For the opposite direction, given a regular expression α for $L \subseteq \Sigma^\star$, replace all letters a by $f(a)$. This produces a regular expression for $f(L)$.

□

We can push the last result a little further: we could consider regular substitutions, maps obtained from a lookup table

$$f(a) = K_a \subseteq \Gamma^\star$$

where K_a is a whole regular language, rather than just a single word. As before, $f(x_1x_2\ldots x_n) = f(x_1)f(x_2)\ldots f(x_n) \subseteq \Gamma^\star$ and we set

$$f(L) = \bigcup_{x\in L} f(x)$$

Lemma

Regular languages are closed under regular substitutions and inverse regular substitutions.

In our proof sketch, we used regular expressions. We can convert back and forth between expressions and machines, but one direction is not really feasible.

How about a machine-based proof?

No problem. Say, we have a DFA $\mathcal{A}$ for a language $L \subseteq \Gamma^\star$ and a homomorphism $f : \Sigma^\star \to \Gamma^\star$.

Keep the state set, initial and final states; switch the alphabet to Σ and modify the transition function:

$$\delta'(p, a) = \delta(p, f(a))$$

Exercise

How would this work for regular substitutions?

For a word $x = uvw$, u is prefix of x, v is factor or infix of x and w is suffix of x.

We can lift these concepts to languages:

$$\mathsf{pref}(L) = \{\, u \in \Sigma^\star \mid \exists\, v\, (uv \in L) \,\}$$

and similarly for $\mathsf{fact}(L)$ and $\mathsf{suff}(L)$.

Lemma

$\mathsf{pref}(L)$, $\mathsf{fact}(L)$ *and* $\mathsf{suff}(L)$ *are regular whenever* L *is.*

Proof. We may assume that $\mathcal{A}$ is a trim automaton for L.

Set $F = Q$, $I = F = Q$ and $I = Q$, respectively.

□

For any alphabet Σ define $\overline{\Sigma}$ to be a copy of Σ with elements $\overline{a}$ for $a \in \Sigma$; set $\Gamma = \Sigma \cup \overline{\Sigma}$.

Define homomorphisms $f, g : \Gamma^\star \to \Sigma^\star$ by

$$f(\overline{a}) = a \qquad f(a) = a$$
$$g(\overline{a}) = a \qquad g(a) = \varepsilon$$

Then

$$\mathsf{pref}(L) = g\Big(f^{-1}(L) \cap \overline{\Sigma}^\star \Sigma^\star\Big)$$

Done by closure properties.

Suppose $\mathcal{A}$ is a DFA accepting $L \subseteq \Sigma^\star$.

Claim: Let K be the words x in L such that the computation of $\mathcal{A}$ on x uses every state at least once. Then K is regular.

Sketch of proof.

Consider the transitions $\Delta \subseteq Q \times \Sigma \times Q$ as a new alphabet, so $\Delta^\star$ is the set of all sequences of transitions.

Let $C = \{ W \in \Delta^\star \mid W = \ldots (p, a, q)(q', b, r) \ldots, q \neq q' \}$ Then $\Delta^\star - C$ represents all computations of $\mathcal{A}$. Similarly we can filter out accepting computations.

Let $C_p = \Delta^\star(p, a, q)\Delta^\star \cup \Delta^\star(q, a, p)\Delta^\star$ be the computations using state p.

By intersecting with all the C_p we get computations we want.

Lastly apply the homomorphism $(p, a, q) \mapsto a$.

□

	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).

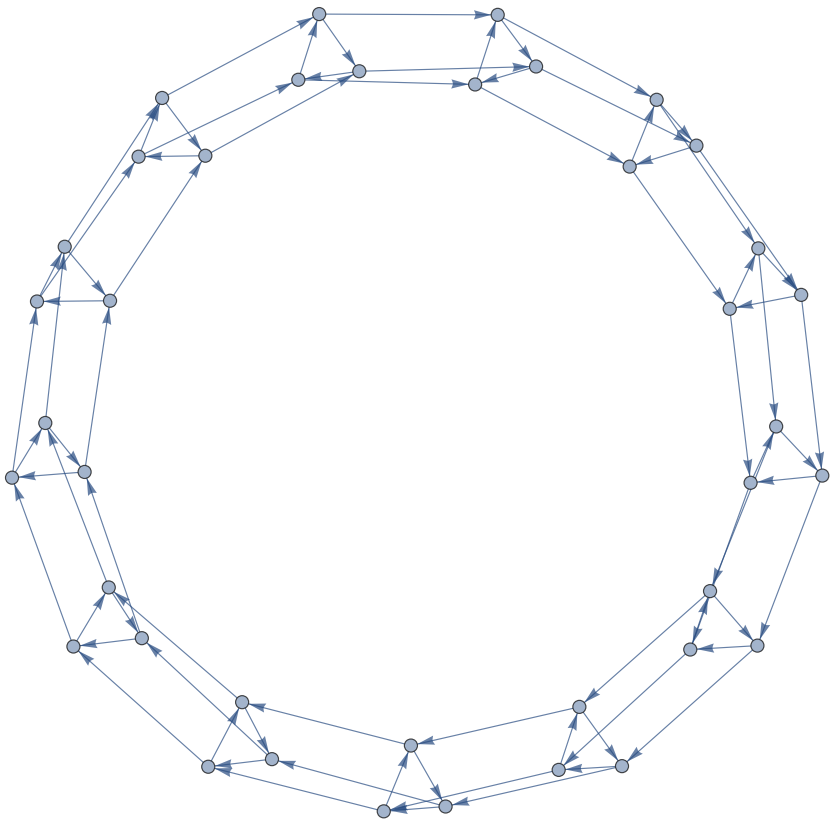
Note that we are only dealing the state complexity, not transition complexity (which is arguably a better measure for NFAs).

Example: Intersection

The "mod-counter" language

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

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



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.*