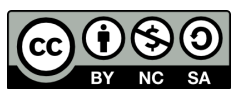
CDM

Closure Properties

Klaus Sutner

Carnegie Mellon University

Spring 2025



We have a definition of recognizable languages in terms of finite state machines (deterministic or nondeterministic).

There are two killer apps for recognizable languages:

- pattern matching
- logical decision procedures

For either application we need to develop the basic theory of finite state machines and recognizable languages.

How should we go about this?

We can think about finite state machines as a particularly weak model of computation. In this context it is 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, a future topic).

But parallel composition we can handle right now: 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.

For simplicity, 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 if the construction is used repeatedly.

By our choice of acceptance condition we have

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

By adjusting the final states, 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	$F = F_1 \times (Q_2 - F_2)$

Products generalize easily to nondeterministic machines. Say, we have two NFAs over Σ: $\mathcal{A}_i = \langle Q_i, \Sigma, \tau_i; I_i, F_i \rangle$.

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

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

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.

We still get intersections in an NFA product:

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

BUT:
In general, products of NFAs cannot be used to handle union and complement. Make sure to construct some small counterexamples.

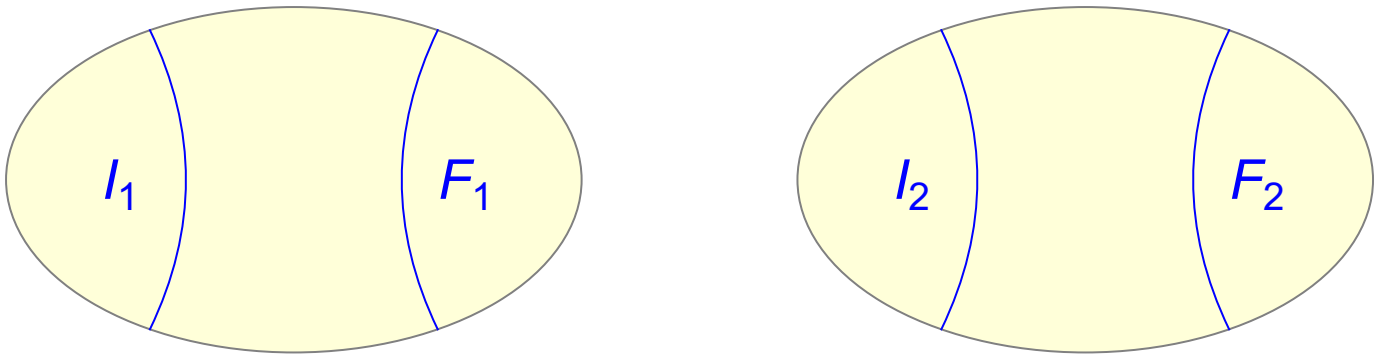
Hence we do not yet have closure under Boolean operations for recognizable languages: these are defined in terms of NFAs, not DFAs.

One can actually handle unions for NFAs very easily: We can form the disjoint union or sum. Assume that the state sets are disjoint and define

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 declare the two machines to be one machine.
Basta.



This construction is trivially linear time.

Alas, even if the given machines are DFAs the result is always an NFA.

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 (a milder form on nondeterminism).

We will show that the following operations do not affect recognizability:

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

So far we can handle union and intersection.

For complement we will need to lean heavily on deterministic machines.

For the rest, nondeterminism is extremely useful.

Effective Closure

All our arguments concerning closure properties are of the form:

> Given FAs $\mathcal{A}_i$ for recognizable languages L_i.
> One can effectively construct a new FA $\mathcal{A}$ for $L_1 \,\mathrm{op}\, L_2$.

In other words, we have effective closure: there are algorithms that compute the appropriate machines.

In many interesting cases, these algorithms are in fact highly efficient.

Alas, not always, in particular complementation causes major problems.

Deciding Equivalence

By effective closure, we can deal e.g. with the Equivalence problem for 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 $\mathcal{L}(\mathcal{A}_2) - \mathcal{L}(\mathcal{A}_1) = \emptyset$.

From the product construction, we get a quadratic time algorithm.

We will see a better method later.

In fact, we 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* $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2) = \emptyset$.

The opposite direction is false: there are classes of languages where equality is decidable, but inclusion is not.

There is a famous and difficult theorem by Sénizergues from 1997 that shows decidability of equality.

But inclusion is hard: DCLFs are effectively closed under complements and

$$\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2) \iff \mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)} = \emptyset$$

Alas, intersection emptiness is not decidable for DCLFs (one can reduce Post's Correspondence Problem).

Lemma

Equivalence of finite state machines is primitive recursive.

Sketch of proof.

The most ham-fisted approach would be to pick a "sufficently large bound" β and then check

$$\forall\, x \in \Sigma^{\leq \beta} \left(\mathcal{A}_1(x) = \mathcal{A}_2(x) \right)$$

What should β be? Suppose x is minimal such that the machines disagree on x, $m = |x|$. For $k \leq 4$, let

$$P_i(k) = \tau_i\left(I_i, x_1 \ldots x_k\right) \subseteq Q_i$$

If m is large, there must be $1 \leq k < \ell \leq m$ such that $P_i(k) = P_i(\ell)$ for $i = 1, 2$. Hence we can shorten x by removing the factor $x_{k+1}, \ldots, x_\ell$, contradiction.

From this we can extract the bound.

□

So the bound is $\beta = 2^{n_1}\, 2^{n_2}$. Even worse, we have to check exponential in β many strings. Clearly not feasible, though easily primitive recursive.

The real challenge is to find an efficient algorithm for equivalence testing in general, or to show that none can exist for NFAs.

In practice, this is done by establishing a computational hardness result.

Again, here are some language-related operations, listed roughly in increasing order of algorithmic difficulty.

- reversal
- concatenation, Kleene star
- homomorphisms, inverse homomorphisms

We will establish effective closure for all of these.

Reversal Closure

The reversal of a language is defined by

$$(x_1 x_2 \ldots x_{n-1} x_n)^{\text{op}} = x_n x_{n-1} \ldots x_2 x_1$$

$$L^{\text{op}} = \{ x^{\text{op}} \mid x \in L \}$$

Then L is recognizable 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

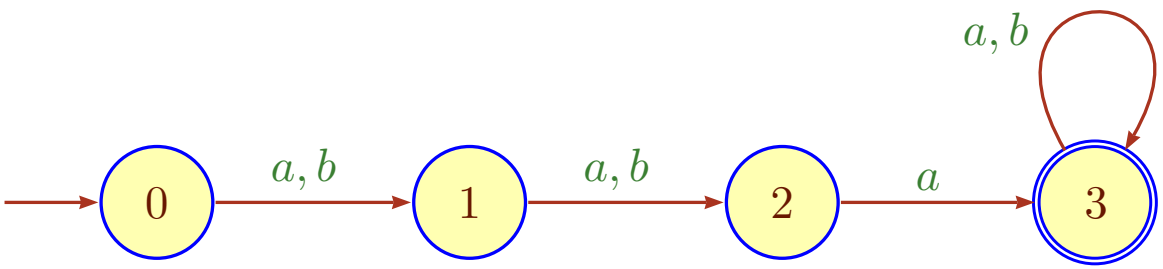
Define position languages by

$$\mathsf{Pos}_{a,k} = \{\, x \in \Sigma^\star \mid x_k = a \,\}$$

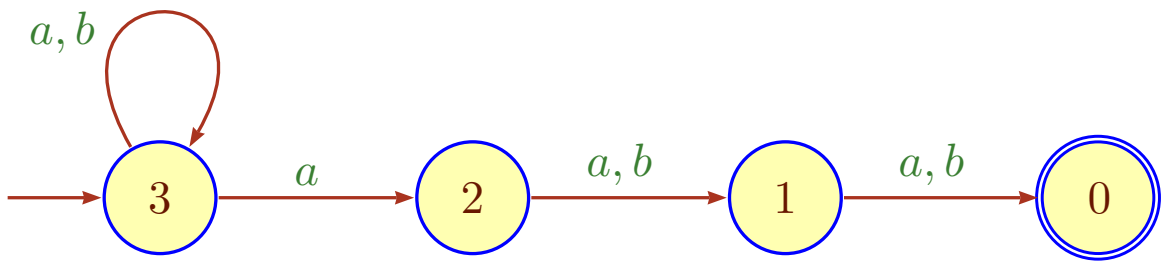
A negative index means: from the end.

Hence $\mathsf{Pos}^{\mathrm{op}}_{a,k} = \mathsf{Pos}_{a,-k}$.

It is staightfoward to build a PDFA for $\mathsf{Pos}_{a,3}$.



But for $\mathrm{Pos}_{a,-3}$ the natural machine is obtained by reversing all the arrows flipping initial and final states, hence nondeterministic.



It is not immediately clear how to build a PDFA for this language.

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 two NFAs $\mathcal{A}_i$ for recognizable languages L_i, we want to construct a new machine $\mathcal{A}$ for $L_1 \cdot L_2$.

So we need to split the string $x = u\,v$ and then send u to $\mathcal{A}_1$ and v to $\mathcal{A}_2$.

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

The problem is that we don't know where to split.

The natural answer would be to use nondeterminism to guess the right split.

But there is another problem: how do we jump to the second machine?

Here is a clever trick: we allow our machines to jump from one state to another **without** consuming any input. Technically, this is handled by so-called ε-transitions or ε-moves.

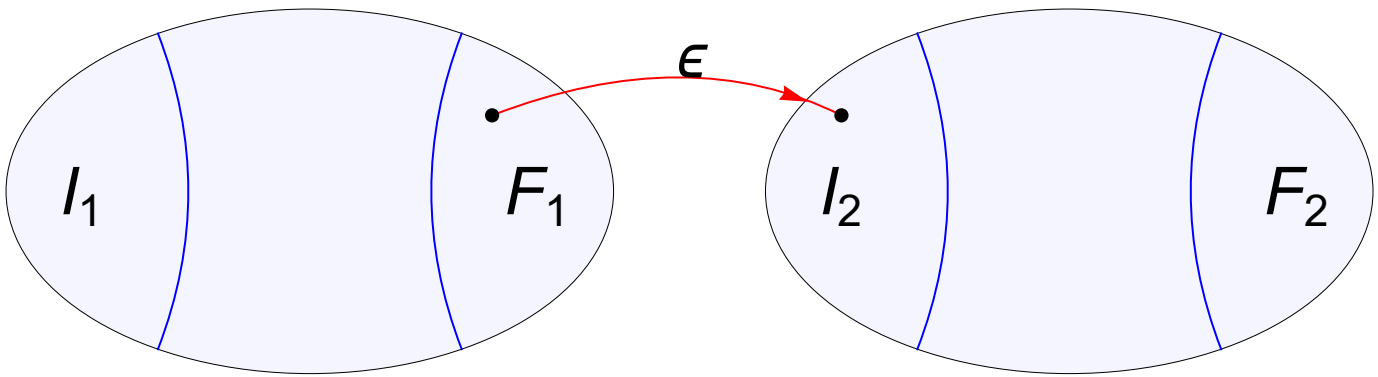
Definition

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

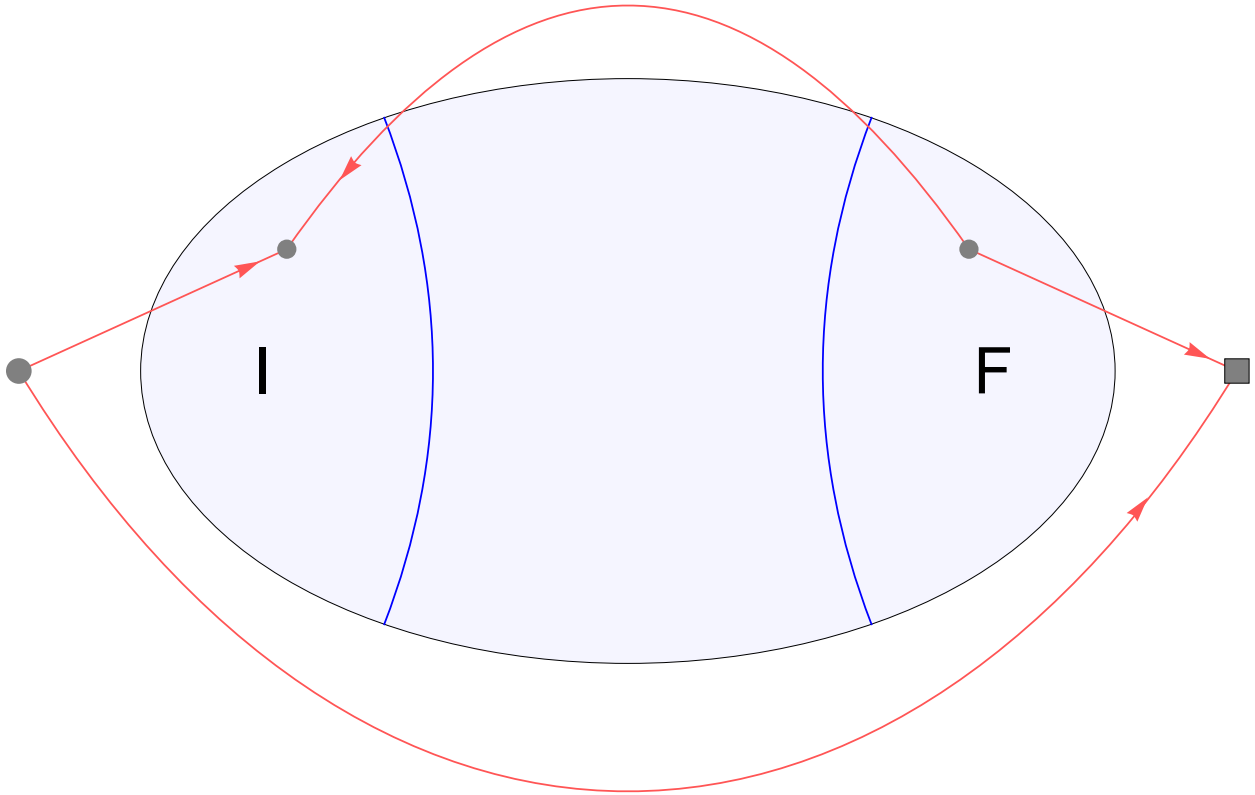
Note that an NFAE may have runs that are longer than the input.

We will see shortly how to convert an NFAE into an equivalent NFA and even in polynomial time, so this is perfectly fine.

Once we have ε-transitions, the construction for concatenation is fairly simple.



Place an ε-transition between all states in F_1 and I_2.
Note that there are 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're at, we can generalize further by allowing transitions to be labled by arbitrary words over Σ. These devices are called generalized finite automata (GFA):

$$p \xrightarrow{aba} q$$

GFA are very expressive.

E.g., it is trivial to write down a two-state GFA for any finite language: p is initial, q is final and there is a transition $p \xrightarrow{w} q$ for each word w.

So we have the following hierarchy of finite state machines:

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

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

Warning:

Many algorithms require NFAs or even (P)DFAs.
GFAs are concise, but often need to be converted back to NFA.

We will discuss a number of conversion algorithms, so we need to have a way to express the size of a finite state machine.

Definition

The state complexity of a FSM is its number of states.
The transition complexity of a FSM is its number of its transitions.

In symbols: $\mathrm{scp}(\mathcal{A})$ and $\mathrm{tcp}(\mathcal{A})$

The transition complexity corresponds nicely to the actual size of a FSM as a data structure, but most results in the literature are phrased in terms of the state complexity.

The next project is to show the following:

Theorem

For every GFA, we can effectively construct an equivalent NFA.

First off, we can easily convert a GFA into an NFA by transition-splitting:

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

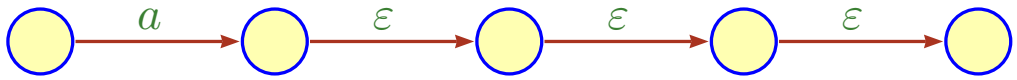
This increases the states/transition complexity only linearly.

Next we have to eliminate ε-moves. Epsilon elimination is quite straightforward and can easily be handled in polynomial time:

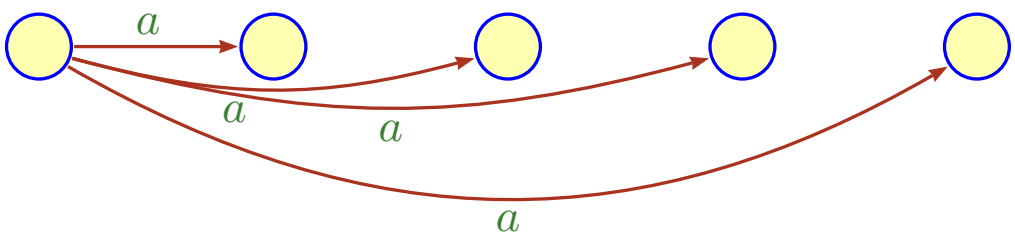
- introduce new letter transitions that have the same effect as chains of ε transitions, and
- remove all ε-transitions.

Since there may be chains of ε-transitions this is in essence a transitive closure problem and 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. □

This time there is potentially a quadratic blow-up in the number of transitions.

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 and let $L \subseteq \Sigma^\star$ be recognized by $\mathcal{A}$.

Relabel the transitions in $\mathcal{A}$ as follows

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

This produces a GFA over Σ that accepts $f(L)$.

For the opposite direction, suppose $\mathcal{A} = \langle Q, \Gamma, \tau; I, F \rangle$ is an NFA for $K \subseteq \Gamma^\star$.

Construct a new machine $\mathcal{A}'$ over Q and Σ by

$$p \xrightarrow{a} q \text{ in } \mathcal{A}' \qquad \Longleftrightarrow \qquad p \xrightarrow{f(a)} q \text{ in } \mathcal{A}$$

This produces an NFA over Σ that accepts $f^{-1}(K)$.

□

Example

$\Sigma = \{a, b, c\} \qquad \Gamma = \{0, 1\}$

$f(a) = 00 \quad f(b) = 01 \quad f(c) = 10$

$L =$ even number of as, no c

$K =$ even number of 0s

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.

The next and critical step is to eliminate nondeterminism[†].

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 superstates: sets of states of the original machine.

[†] This also works for Turing machines, but not for pushdown automata.

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

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

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

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

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

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 show 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 $2^{\mathrm{scp}(\mathcal{A})}$.

However, for equivalence only the accessible part $\mathrm{pow}(\mathcal{A})$, the power automaton of $\mathcal{A}$, is required. With a little luck, it will be much smaller.

The right way to determize $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ is to take a closure in the ambient set $\mathfrak{P}(Q)$, starting with the initial superstate I:

$$\mathsf{clos}\Big(I;\ (\tau_a)_{a \in \Sigma} \Big) \subseteq \mathfrak{P}(Q)$$

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.

Here is a more algorithmic version of this construction.

$\texttt{act} = S = \{I\}$

while($\texttt{act} \neq \emptyset$)
    $P = \text{pop}(\texttt{act})$
    foreach $a \in \Sigma$ **do**
        compute $P' = \tau_a(P)$
        store $P \xrightarrow{a} P'$
        if($P' \notin S$) **then**
            add P' to S and act
return S

Upon completion, $S \subseteq \mathfrak{P}(Q)$ is the state set of the accessible part of the full power automaton.

The determinization algorithm is very similar to nondeterministic acceptance testing: instead of following the superstates for one particular input word, it constructs all possible superstates.

If we are only interested in acceptance testing for a few words, there is no need to determize. But if we need to, say, compute complements, then we may have to build the whole DFA.

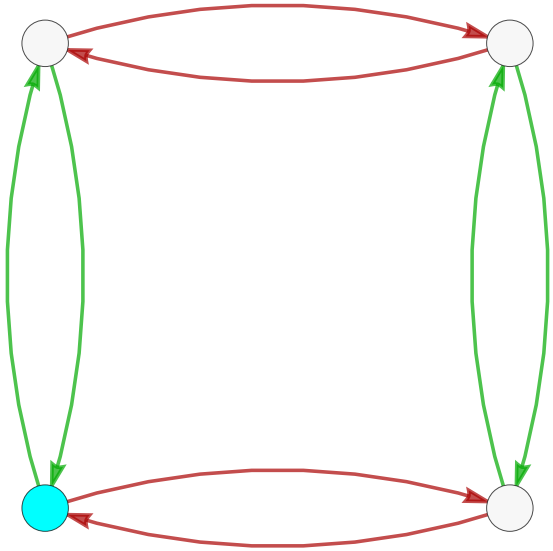
We can think of an NFA $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ as a very compact description of the DFA pow($\mathcal{A}$).

The DFA lives in the huge ambient space $\mathfrak{P}(Q)$ that we cannot even write down (except when the size of $\mathcal{A}$ is tiny).

But for equivalence we don't need the whole space, just a potentially much smaller fragment. Moreover, we can generate this fragment by using graph algorithms: essentially, we run DFS/BFS on a virtual graph.

For this to work, we do not need adjacency lists or matrices, it is enough to be able to generate a list of out-edges on the fly.

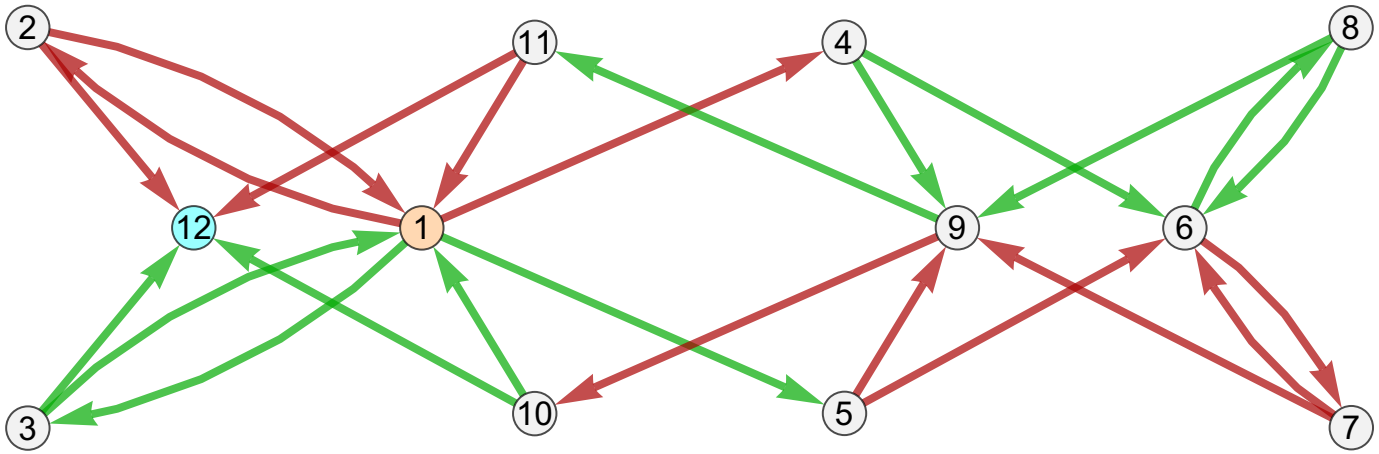
Example: Even/Even



A simple 4-state DFA for the language all strings over $\{a, b\}$ with an even number of as and bs.

The cyan state is initial and final.

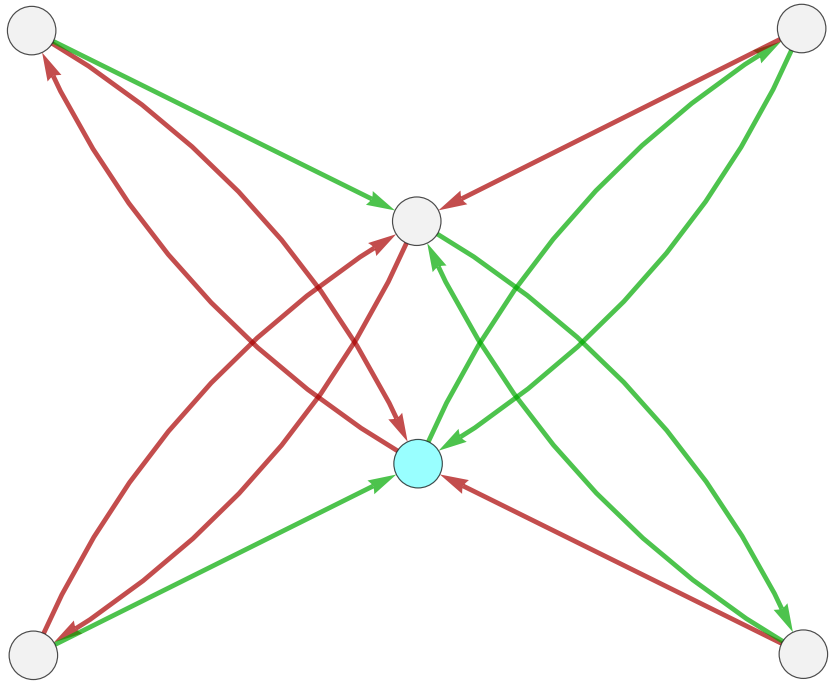
NFA for Even/Even



This is an NFA for the even/even language generated by an algorithm that converts a regular expression to a machine.

1 is an initial state, and 12 is both initial and final.

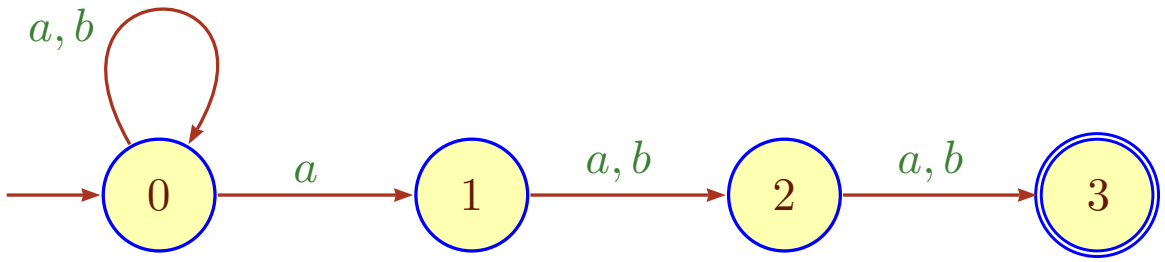
Other than 12, all states are nondeterministic.



The power automaton for the last NFA has only 6 states!

$$\{\{1,12\},\{2,4\},\{3,5\},\{6,9\},\{7,10\},\{8,11\}\}$$

There is hope, after all.



What happens if we determinize this machine?

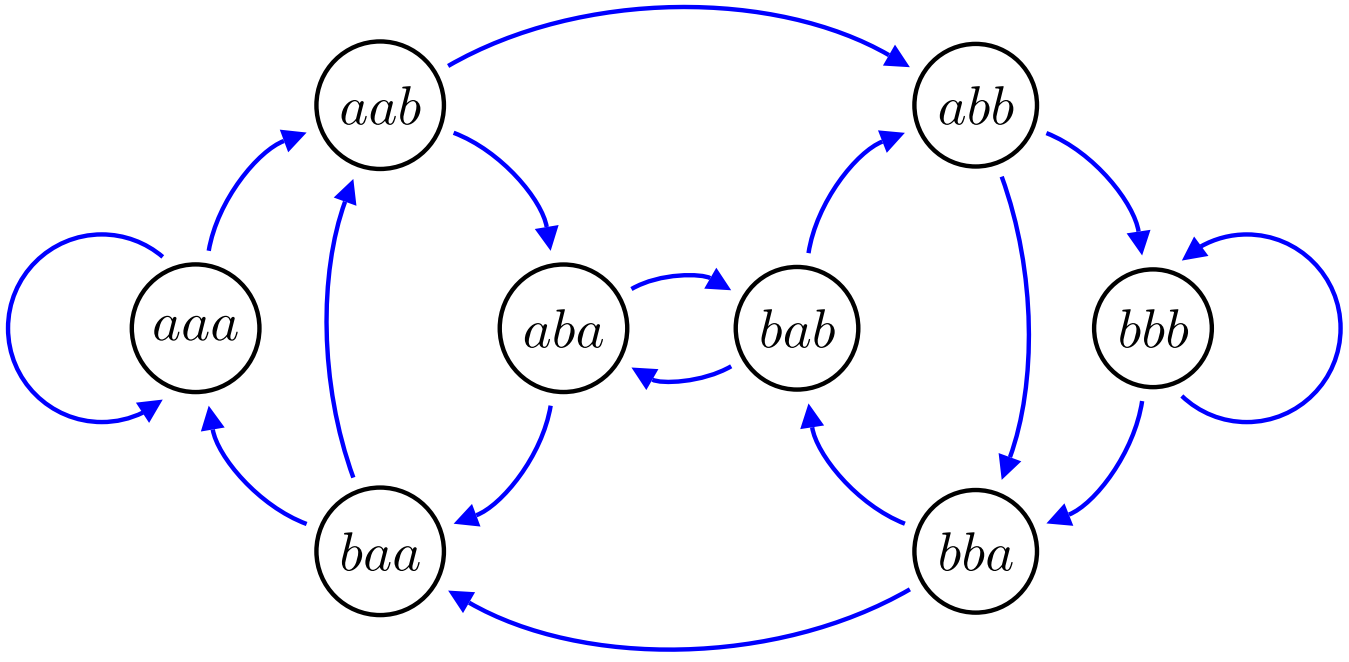
Applying the Rabin-Scott construction we obtain a machine with 8 states

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

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

The full power set has size 16, our construction builds the accessible part of size 8.

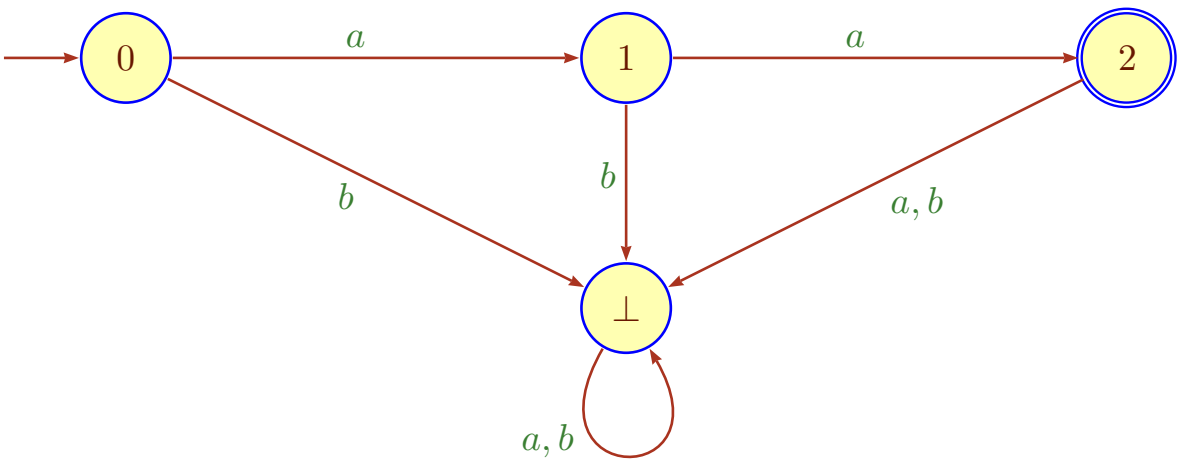


Here is the corresponding diagram, rendered in a particularly brilliant way.
This is a so-called de Bruijn graph (binary, rank 3).

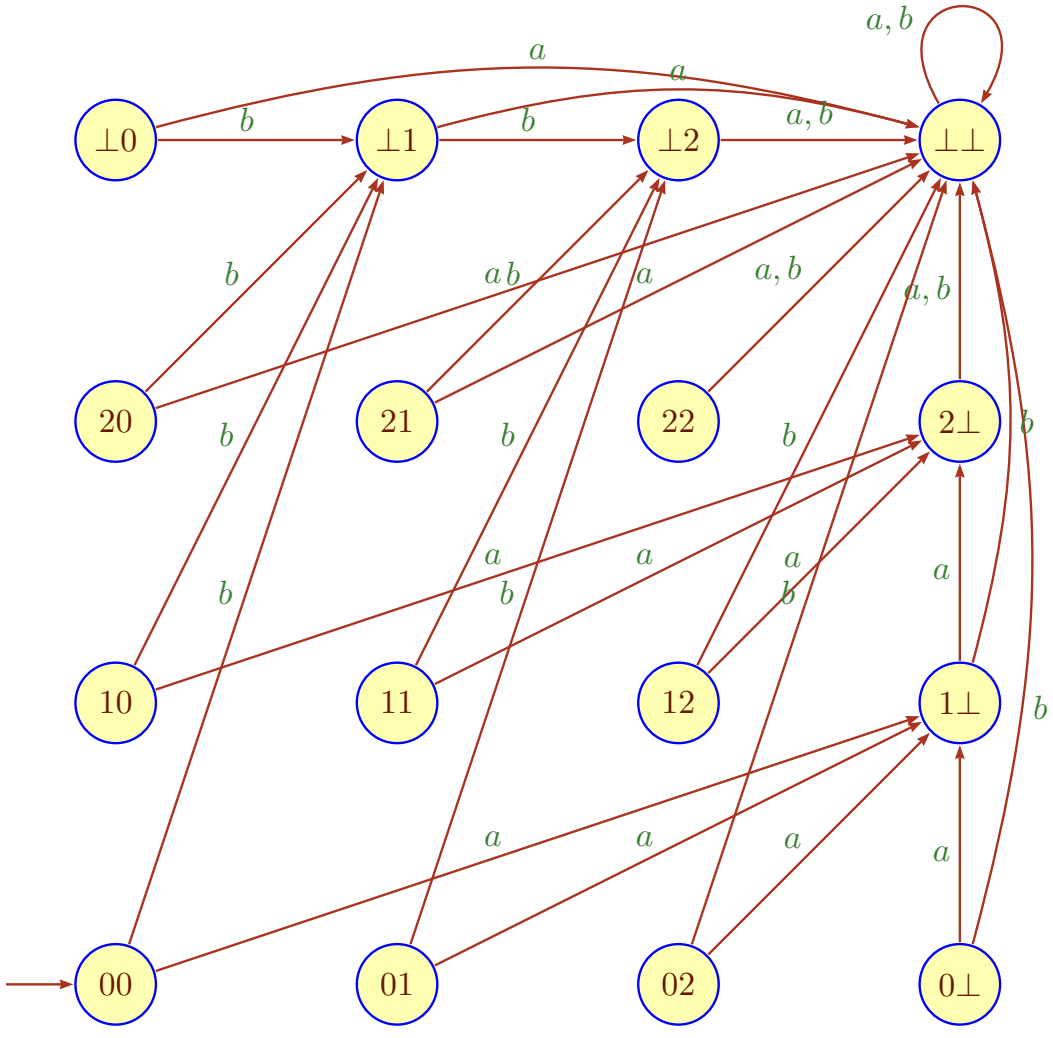
While we are at it, constructing only accessible parts is important, always.

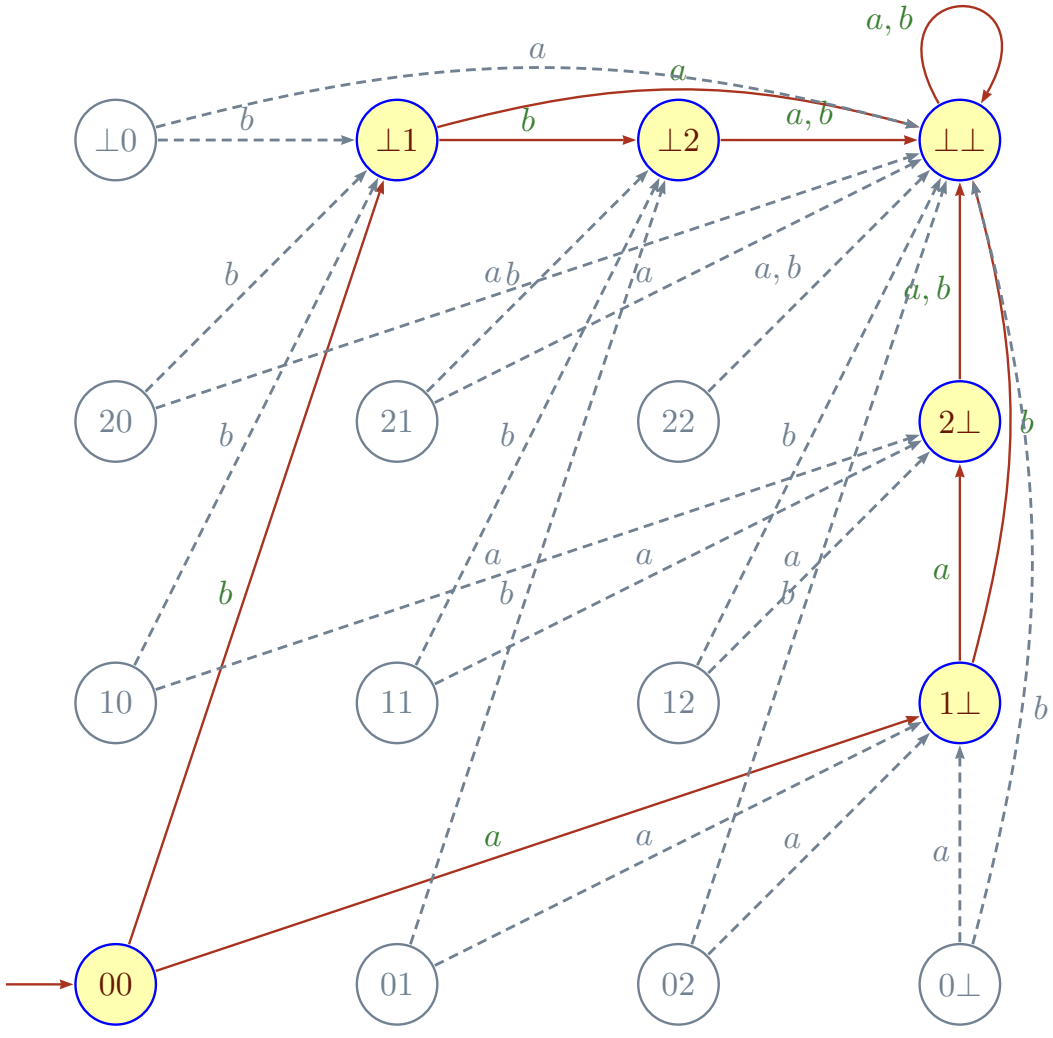
E.g., consider the product automaton for DFAs $\mathcal{A}_{aa}$ and $\mathcal{A}_{bb}$, accepting aa and bb, respectively.

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



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⊥

So are nondeterministic machines better than deterministic ones?

- **Advantages:**

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

- **Drawbacks:**

  Acceptance testing somewhat 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.

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

Recall that our vanilla acceptance only depends on the target state of a run, not the full run itself.

We could try different kinds of acceptance conditions.
For simplicity, assume that $\mathcal{A}$ is PDFA.

> $\mathcal{A}$ accepts x iff
> it has a vanilla accepting run that uses every state at least once.

Or we could insist that some state appears 42 times.
Or if state p appears, then state q must not appear.

Claim: This sort of condition still produces only recognizable languages.

Construct a new PDFA $\mathcal{A}'$

$$Q' = Q \times \mathfrak{P}(Q)$$

$$q'_0 = (q_0, \{q_0\})$$

$$F' = F \times \{Q\}$$

$$\delta'\big((p, R), a)\big) = \big(\delta(p, a), R \cup \{\delta(p, a)\}\big)$$

In other words, $\mathcal{A}'$ runs $\mathcal{A}$ and keeps track of all encountered states.

Yet another example where the state complexity may blow up exponentially.

	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).

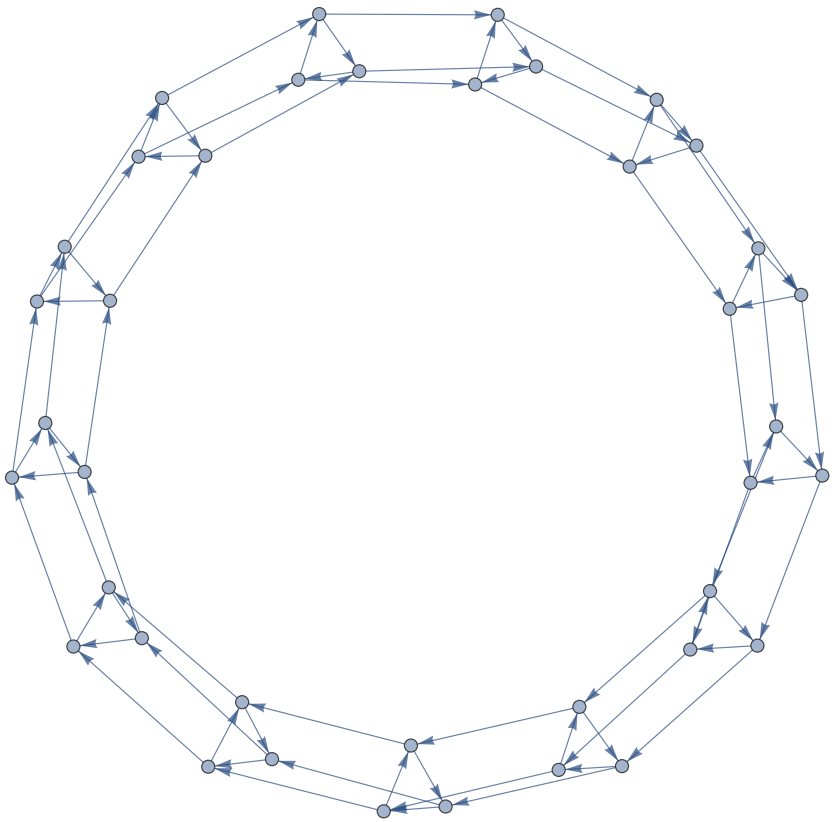
Note that we are only dealing with state complexity, not transition complexity (which is arguably a better measure for NFAs).

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:

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.*

The example for the even/even language shows that a power automaton may well be smaller than the original NFA.

Just to be clear: this phenomenon is a bit rare. It is still true that for RealWorld$^{\text{TM}}$ machines the blowup is often small, something like polynomial in the size of the NFA.

Unfortunately, full or nearly full blowup during determinization does occur, and there is simply no way around it.

Recall the kth symbol languages

$$\mathsf{Pos}_{a,k} = \{\, x \in a, b^\star \mid x_k = a \,\}$$

Proposition

$\mathsf{Pos}_{a,k}$ *can be recognized by an NFA on* $k+1$ *states.*
Any DFA has size at least 2^k*.*

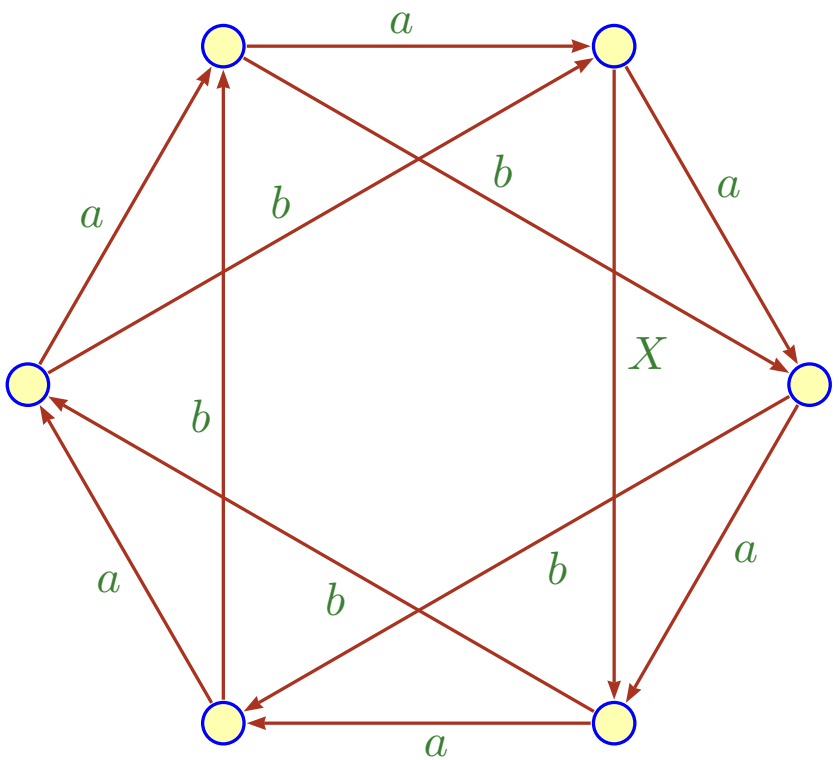
Applying determinization to the NFA produces a DFA of size 2^k.
The hard part is to show that there is no smaller DFA (later).

Blowup Example 2

Here is a 6-state NFA based on a circulant graph. Assume $I = Q$.

If $X = b$ than the power automaton has size 1.

However, for $X = a$ the power automaton has maximal size 2^6.



Think of placing a pebble on each state of the automaton.

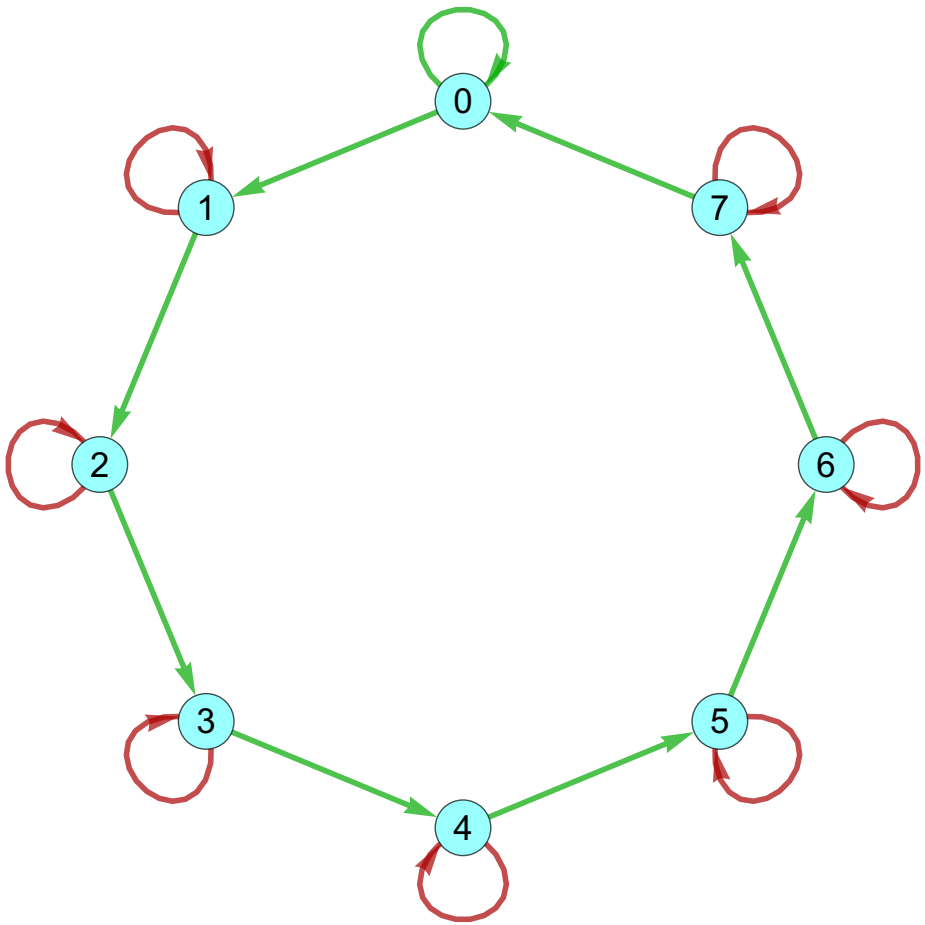
Then push a button $s \in \Sigma$ and move all the pebbles accordingly.

Lather, rinse and repeat, until the target configuration P of pebbles is reached.

To demonstrate full blowup, we have to explain how to accomplish this for all $P \subseteq Q$.

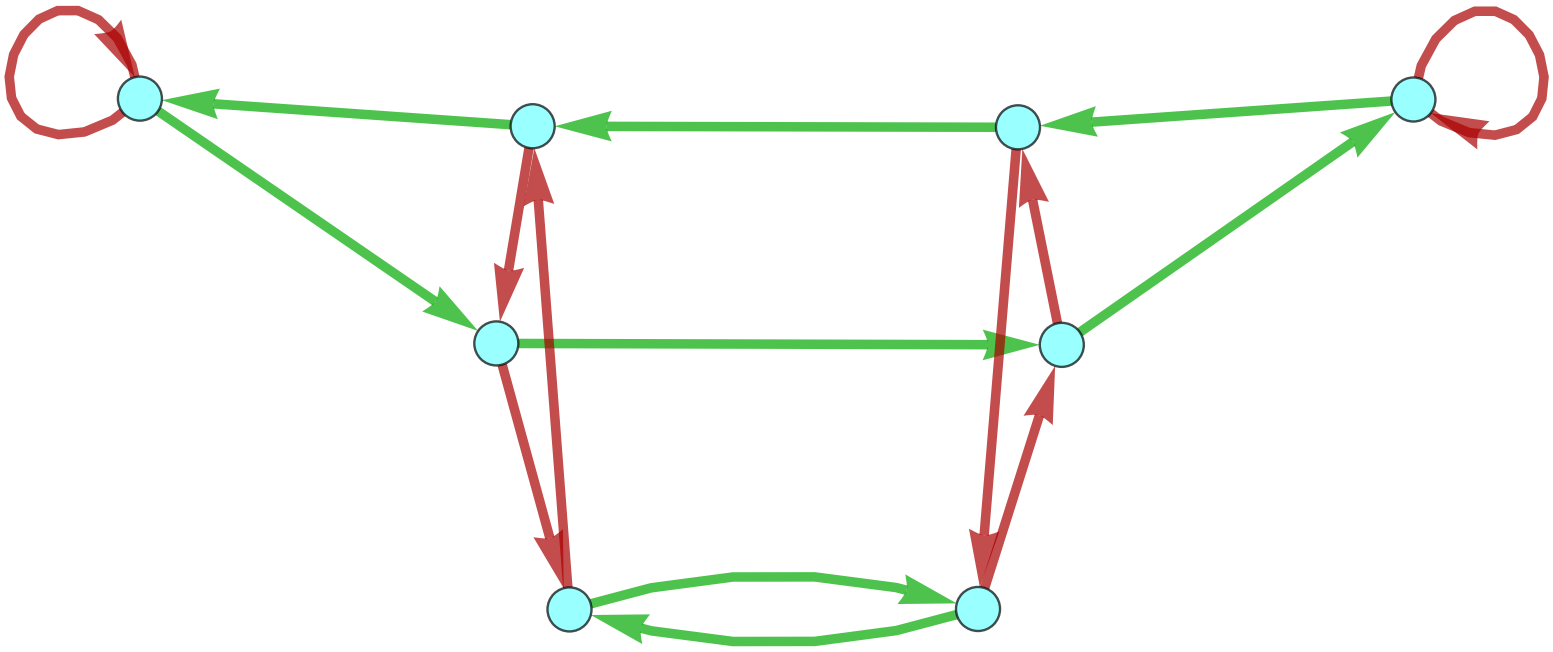
Slightly more complicated is a situation when we can only handle "almost all" P, a few special pebble configurations fail to be reachable.

Consider $C(n; 0, 1)$, label all loops a and all stride 1 edges b.
Then switch the label of the loop at 0.



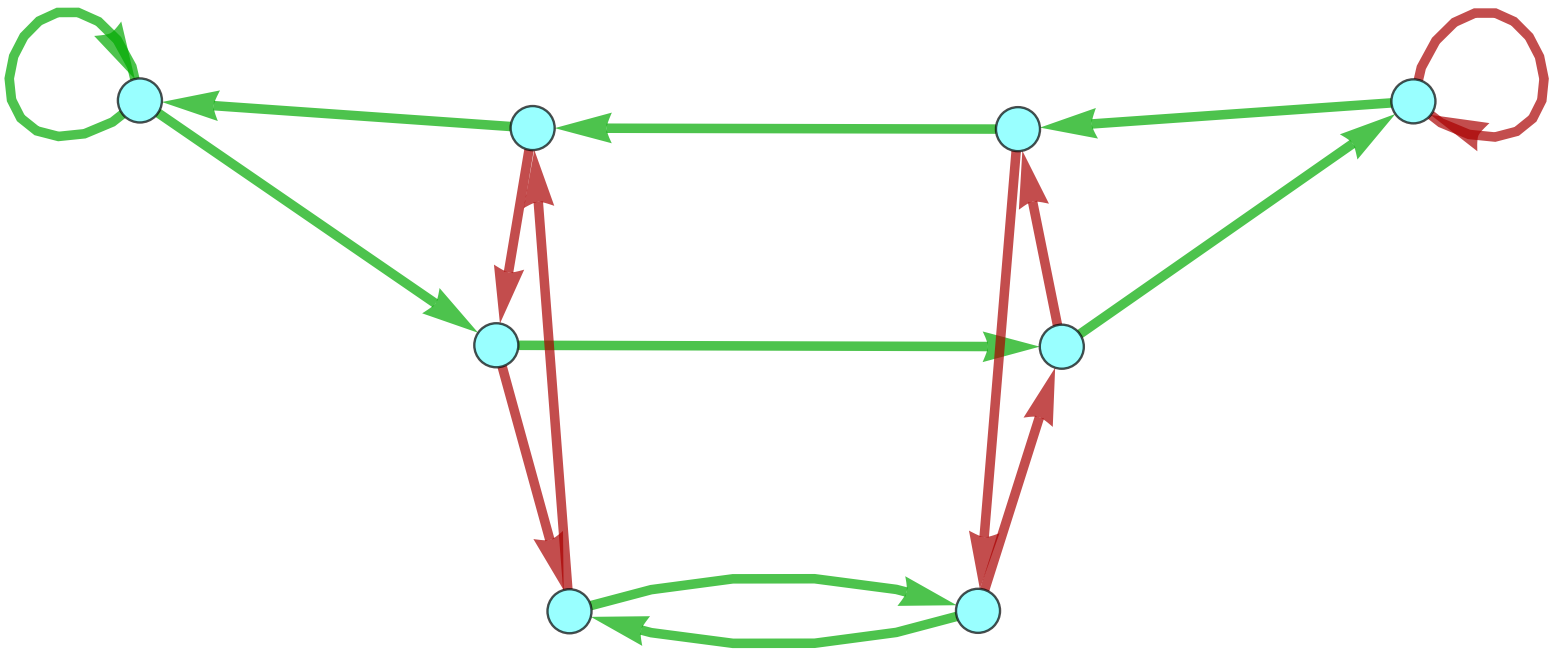
Blowup Example 3

Start with a binary de Bruijn automaton where both δ_a and δ_a are permutations. An example for rank 3:



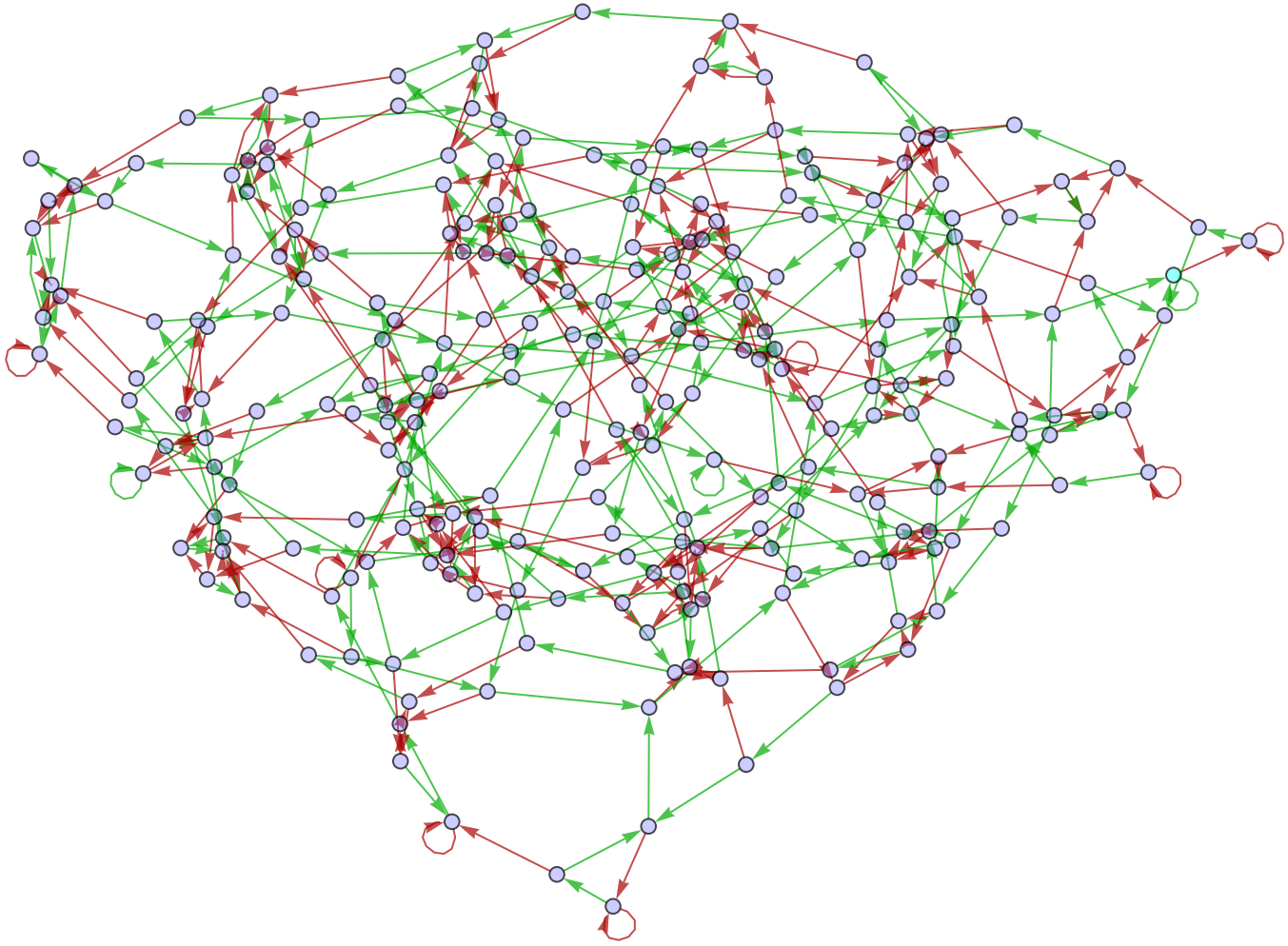
There are 2 loops and 2 3-cycles labeled a (red), a 2-cycle and a 6-cycle labeled b (green).

Now flip the label of one of the two loops.



We get an NFA that it almost deterministic. What happens if we perform determinization?

Full Blowup



Flipping the label at a loop produces full blowup for any $\emptyset \neq I \subseteq Q$.

The last claim I can prove. But this is just the tip of the iceberg.

One can show that the number of permutation labelings in the binary de Bruijn graph of rank k is $2^{2^{k-1}}$.

Conjecture:
Flipping the label of an arbitrary edge in a permutation labeling will produce full blowup in exactly half of the cases.

So the total number of cases with full blowup should be

$$\text{full blowup:} \qquad 2^k \, 2^{2^{k-1}}$$

This has been verified experimentally up to $k = 5$ (on Blacklight at PSC, rest in peace). There are $4,194,304$ machines to check, ignoring symmetries. Half of them blow up to size $2^{32} = 4,294,967,296$.

Since exponential blowup does occur, it would be very nice if we could run a quick precomputation that checks for a given NFA whether determinization on that machine would indeed blow up (so that we don't have to bother trying).

More precisely, we would like a fast algorithm for the following problem.

Problem:	**Power Automaton Size**
Instance:	A nondeterministic machine $\mathcal{A}$, a bound B.
Question:	Is the size of the power automaton of $\mathcal{A}$ bounded by B?

Theorem (KS 2003)

Power Automaton Size is PSPACE*-complete.*

Thus, essentially the only way to determine the size of the power automaton is to actually construct it, there are no computational shortcuts.