

CDM

Closure Properties

KLAUS SUTNER

CARNEGIE MELLON UNIVERSITY

SPRING 2024



1 Closure Properties

2 Determinization

3 More Closure Properties

4 Exponential Blowup

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

We need to develop a theory of finite state machines to understand how these algorithms work.

Another way to think about finite state machines is 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 $\Sigma^* \rightarrow \mathbf{2}$, 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

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

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

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

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

Products generalize easily to nondeterministic machines. Say, we have two FAs 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

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

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 .

By our choice of acceptance condition we still obtain intersections:

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

Warning:

In general, products of FAs cannot be used to handle union and complement.

Exercise

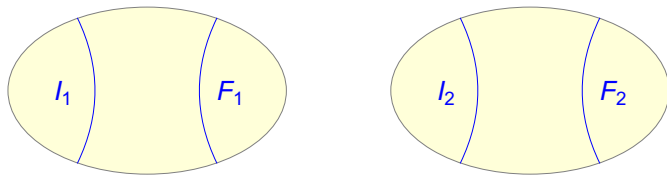
Find simple examples for the warning.

In the realm of nondeterministic machines there is a better construction for union: We can simply form the **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 declare the two machines to be one machine.



This construction is trivially linear time; alas, it wrecks determinism, the result is always nondeterministic.

There are two distinct 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.

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

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

We have already seen that Boolean operations can be handled nicely with deterministic machines. Alas, it is somewhat difficult to handle the other operations without resorting to nondeterministic machines.

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 $L_1 \text{ op } L_2$.

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

And, in many interesting cases, these algorithms for FSMs are in fact highly efficient. Alas, not always, in particular complementation causes major problems.

At any rate, we can now deal more intelligently with the Equivalence problem as long as we have deterministic machines.

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.

Definition

Given two languages $L_1, L_2 \subseteq \Sigma^*$ 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:

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^{k+1} &= L^k \cdot L \end{aligned}$$

The **Kleene star** of L is the language

$$L^* = L^0 \cup L^1 \cup L^2 \dots \cup L^n \cup \dots$$

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

Example

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

Example

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

Example

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

Example

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

First assume we have FAs \mathcal{A}_i for recognizable languages L_i . We want to construct a new machine \mathcal{A} for $L_1 \cdot L_2$. To this end, 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 .

There are two basic lines of attack:

- Use nondeterminism to “guess” when to split.
- Find a way to directly construct a DFA for concatenation (assuming the \mathcal{A}_i are DFAs).

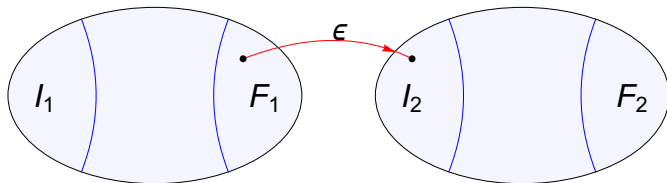
The construction is fairly simple if we push nondeterminism a little bit further. The idea is that we allow our machines to jump from one state to another **without** consuming any input. Technically, this is handled by so-called ϵ -transitions.

Definition

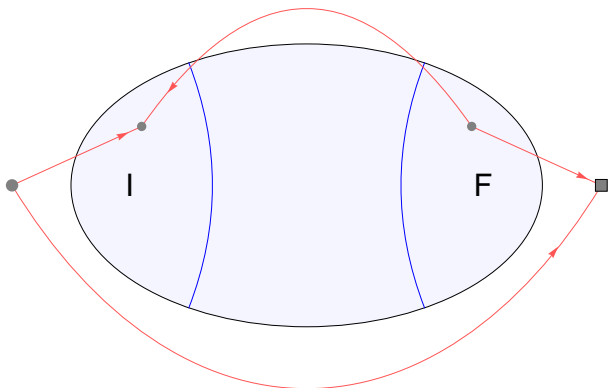
A **nondeterministic finite automaton with ϵ -moves (NFAE)** is defined like an NFA, except that the transition relation has the format $\tau \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

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.

Very well, but can we handle concatenation without nondeterminism? Here is a trick that sometimes helps to construct deterministic 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 leader, and one copy of 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.

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 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

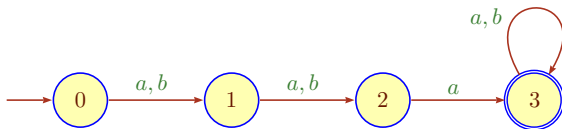
$$L^{\text{op}} = \{ x^{\text{op}} \mid x \in L \}$$

be the **reversal** of a language, $(x_1x_2 \dots x_{n-1}x_n)^{\text{op}} = x_nx_{n-1} \dots x_2x_1$.

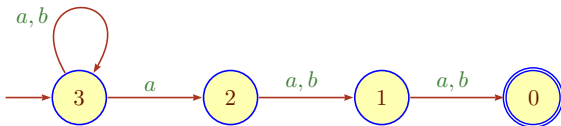
It turns out the L is recognizable iff L^{op} is recognizable.

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.

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 $L_{a,3}^{\text{op}} = \{x \mid x_{-3} = a\} = L_{a,-3}$ 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 $L_{a,-3}$.

1 Closure Properties

2 **Determinization**

3 More Closure Properties

4 Exponential Blowup

Our first order of business is to show that NFAs and NFAEs are no more powerful than DFAs in the sense that any recognizable language is already accepted by some DFA. Note, though, that the size of the machines may differ exponentially; from an algorithmic perspective the equivalence is quite problematic.

We will show how to convert an NFAE into an equivalent DFA:

Epsilon Elimination Convert an NFAE into an equivalent NFA.

Determinization Convert an NFA into an equivalent DFA.

In fact, we could push our nondeterminism even further by allowing transitions to be labeled 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 GFA for any finite language.

Still, they are no more powerful than NFAE (and, ultimately, than DFA). 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.

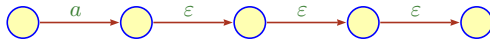
Warning: Some algorithms require NFAs or even DFAs. We cannot simply stay in the realm of GFA and ignore the rest.

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

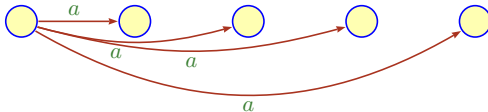
- introduce new ordinary transitions that have the same effect as chains of ϵ transitions, and
- remove all ϵ -transitions.

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. \square

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 $\text{pow}_f(\mathcal{A})$, a machine of size 2^n .

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

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 RealWorldTM 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 be 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 $\mathfrak{P}(Q)$:

$$\text{clos}\big(I, (\tau_a)_{a \in \Sigma}\big)$$

Here τ_a is the function $\mathfrak{P}(Q) \times \Sigma \rightarrow \mathfrak{P}(Q)$ 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.

```
act = S = {I}
while( act  $\neq \emptyset$  )
    P = pop(act)
    foreach a  $\in \Sigma$  do
        compute  $P' = \tau(P, a)$ 
        store  $P \xrightarrow{a} P'$ 
        if(  $P' \notin S$  ) then
            add  $P'$  to S and act
return S
```

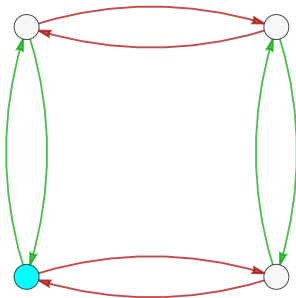
Upon completion, $S \subseteq \mathfrak{P}(Q)$ is the state set of the (accessible part of the) power automaton.

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

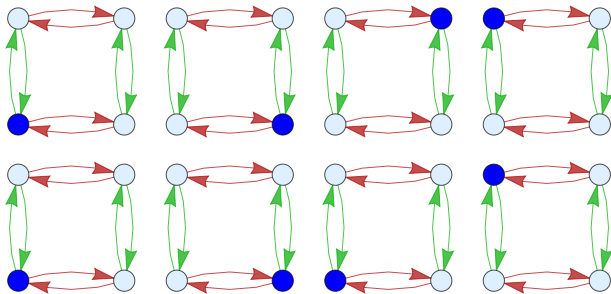
We cannot write down a transition table for the full power automaton $\text{pow}_f(\mathcal{A})$ (except when the size of \mathcal{A} is tiny), but we can generate parts of it on the fly.

With luck, the parts we need are manageable in size.

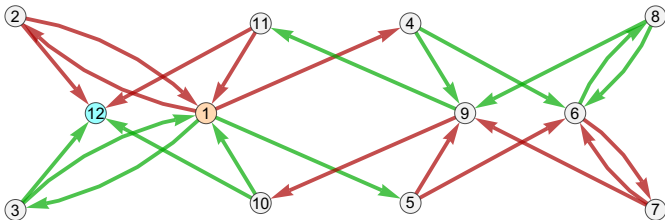
The language of all strings over $\{a, b\}$ with an even number of a s and b s.



There is a very simple 4-state DFA for this language based on keeping track of the parity of letters a and b .



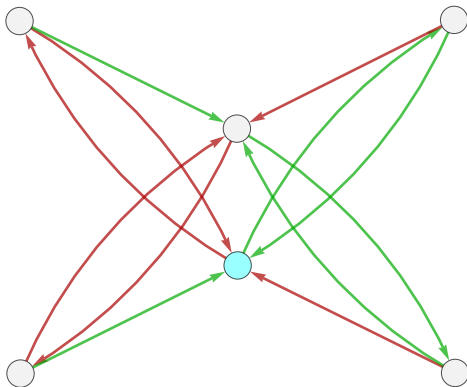
What would a correctness proof for the DFA look like?



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!

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

Thinking about this slightly more abstractly, consider the labeled digraph

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

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.

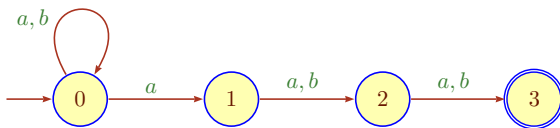
The only difference is that we are not given an adjacency list representation of \mathcal{G} : we compute edges on the fly. No problem at all.

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

Recall

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

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

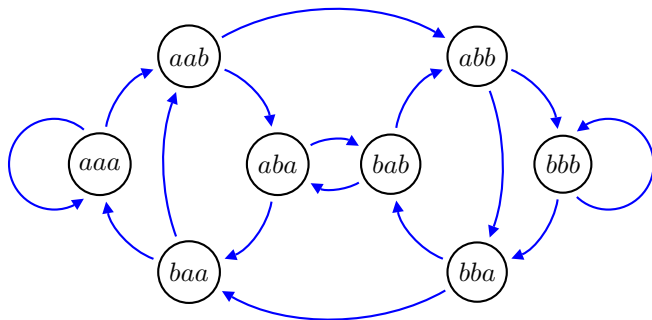
$$\{0\}, \{0, 1\}, \{0, 1, 2\}, \{0, 2\}, \{0, 1, 2, 3\}, \{0, 2, 3\}, \{0, 1, 3\}, \{0, 3\}$$

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
<i>a</i>	2	3	5	7	5	7	3	2
<i>b</i>	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).

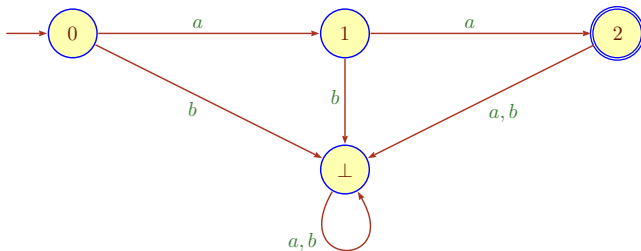


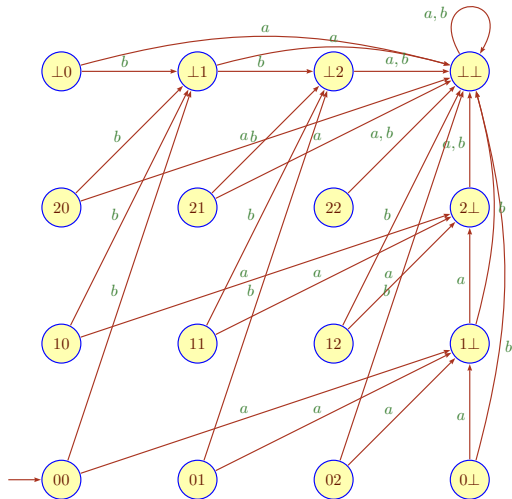
Exercise

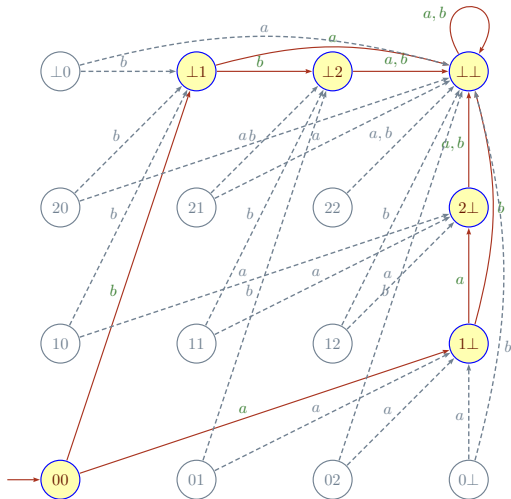
Explain this picture in terms of the Rabin-Scott construction.

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

\mathcal{A}_{aa} :







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.

Acceptance testing in a DFA is naturally fast, just a repeated table lookup.

```
// deterministic acceptance testing  
     $p = q_0$   
    while  $a = x.next()$  do  
         $p = \delta(p, a)$   
    return  $p \in F$ 
```

Of course, it might take some time to compute the lookup table δ in the first place, but once we have it, acceptance testing is very fast.

The key insight is that testing for nondeterministic machines is very, very similar: instead of single states p , we have sets of states $P \subseteq Q$.

```
// nondeterministic acceptance testing
```

```
   $P = I$ 
```

```
  while  $a = x.\text{next}()$  do
```

```
     $P = \{ q \mid \exists p \in P (p \xrightarrow{a} q) \}$ 
```

```
  return  $P \cap F \neq \emptyset$ 
```

Dealing with a set of states P rather than a single state p is slower, but only by a constant depending on the machine. And there are many hacks to make the computation reasonably fast in typical practical situations (e.g., it seems that in some applications $|P|$ is always small).

The total damage is still $O(|x|)$ and the constants are often quite reasonable.

- **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 Closure Properties

2 Determinization

3 **More Closure Properties**

4 Exponential Blowup

Definition

A **homomorphism** is a map $f : \Sigma^* \rightarrow \Gamma^*$ such that

$$f(x_1x_2 \dots x_n) = f(x_1)f(x_2) \dots f(x_n)$$

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

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

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

$$\text{image} \quad f(L) = \{ f(x) \mid x \in L \}$$

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

Lemma

Regular languages are closed under homomorphisms and inverse homomorphisms.

Proof.

Let $f : \Sigma^* \rightarrow \Gamma^*$ be a homomorphism. Suppose $L \subseteq \Sigma^*$ is recognized by \mathcal{A}

Say, we have a DFA \mathcal{A} for $K \subseteq \Gamma^*$. 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 $L \subseteq \Sigma^*$, 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

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

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

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

Lemma

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

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?

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:

$$\text{pref}(L) = \{ u \in \Sigma^* \mid \exists v (uv \in L) \}$$

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

Lemma

$\text{pref}(L)$, $\text{fact}(L)$ and $\text{suff}(L)$ 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 $f, g : \Gamma^* \rightarrow \Sigma^*$ by

$$\begin{aligned} f(\overline{a}) &= a & f(a) &= a \\ g(\overline{a}) &= a & g(a) &= \varepsilon \end{aligned}$$

Then

$$\text{pref}(L) = g(f^{-1}(L) \cap \overline{\Sigma}^* \Sigma^*)$$

Done by closure properties.

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

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 Δ^* is the set of all sequences of transitions.

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

Let $C_p = \Delta^*(p, a, q)\Delta^* \cup \Delta^*(q, a, p)\Delta^*$ 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

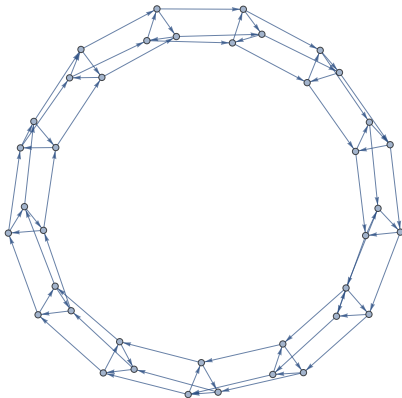
Worst case blow-up starting from machine(s) of size m , n and applying the corresponding operation (accessible part only).

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

The “mod-counter” language

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

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



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 $L = \Sigma^*$?

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

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

has $n = n_1 n_2 \dots n_s$ 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.

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

Problem: **DFA Intersection**
Instance: A list $\mathcal{A}_1, \dots, \mathcal{A}_n$ 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.

1 Closure Properties

2 Determinization

3 More Closure Properties

4 **Exponential Blowup**

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 RealWorldTM 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 k th symbol languages

$$L(a, k) = \{ x \in a, b^* \mid x_k = a \}$$

Proposition

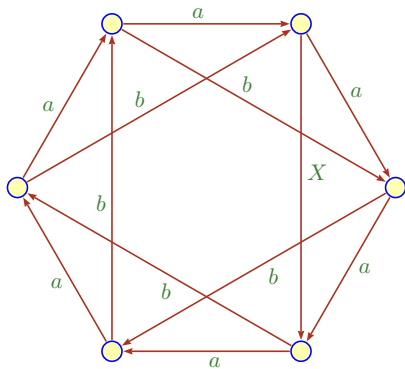
$L(a, -k)$ can be recognized by an NFA on $k + 1$ states, but the state complexity of this language is 2^k .

Applying determinization to the NFA produces a DFA of size 2^k , and there is no smaller DFA.

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

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

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



More generally, we are dealing with **circulant** machines on n states with diagram $C(n; 1, 2)$:

Vertices $\{0, 1, \dots, n - 1\}$

Edges $(v, v + 1 \bmod n)$ and $(v, v + 2 \bmod n)$

Let's say we label the stride-1 edges 1 by a and the stride-2 edges by b .

Then we flip the label of exactly one transition and determinize.

Exercise

Prove that full blowup occurs when we flip a stride-2 transition.

Exercise

What happens when we flip a stride-1 transition?

Exercise

How about circulants $C(n; 1, k)$?

Exercise

How about circulants $C(n; s_1, s_2)$?

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

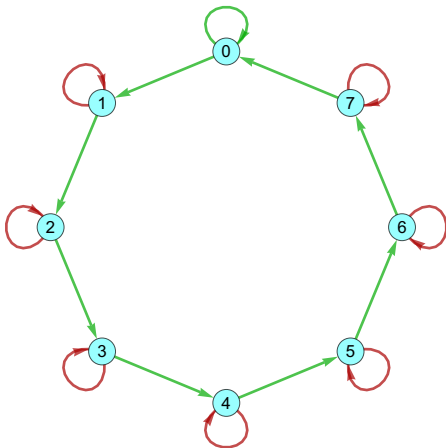
Then push a button $s \in \Sigma$ and move all the pebbles accordingly.

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.



We need some organizing principle for the pebble game. The atomic actions work like so:

δ_a kill 0

δ_b sticky rotate

Sticky rotate means that a pebble at 0 does not go away (and, in addition, spawns a pebble at 1).

Idea: If we can concoct a plain “rotate” operation, we are done.

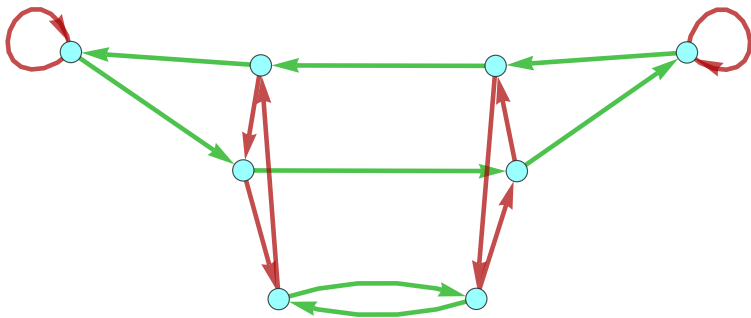
Case 1: $0 \notin P$ δ_b works

Case 2: $0 \in P$

If $n - 1 \in P$: δ_b works

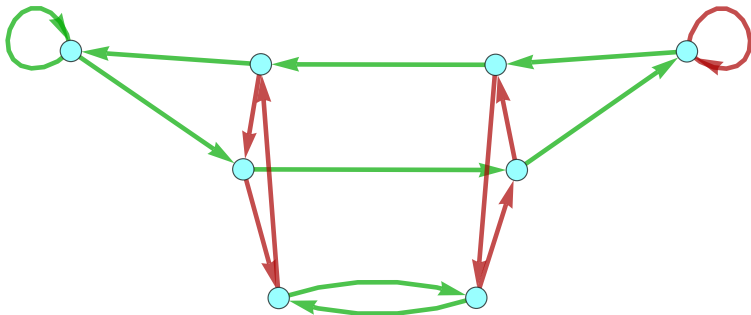
If $n - 1 \notin P$: Ponder deeply.

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

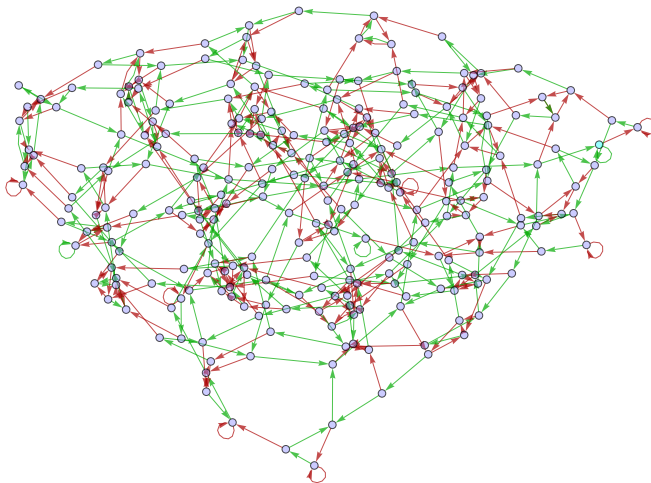


There are 2 loops and 2 3-cycles labeled a (red), a 2-cycle and a 6-cycle labeled b (green).

Now flip the label of one of the two loops.



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



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

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

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.