

CDM

# Constructing Finite State Machines

Klaus Sutner

Carnegie Mellon University

Fall 2009

# Outline

- 1 Closure Properties
- 2 More Closure Properties
- 3 Nondeterministic Machines
- 4 Autonomous Transitions
- 5 Machines and Complexity

## Divisibility Testing and Union

Recall that we can build Horner DFAs that test divisibility by any modulus  $m$ . As a preprocessing step for primality testing we would like to check, say,  $m = 3, 5, 7, 11$ .

We can do this by running the 4 DFAs sequentially, but it would be nice to have to read the input only once. In other words, we want a single DFA that tests all 4 moduli.

### Union

This really comes down to constructing a DFA  $M = M_1 \oplus M_2$  that accepts  $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$  for any two DFAs  $M_1$  and  $M_2$ .

## Products

Suppose we have two transition systems  $T_1 = \langle Q_1, \Sigma, \tau_1 \rangle$  and  $T_2 = \langle Q_2, \Sigma, \tau_2 \rangle$  over the same alphabet  $\Sigma$ .

Construct a new transition system  $T = T_1 \times T_2$ , the so-called **(Cartesian) product** as follows.

$$Q = Q_1 \times Q_2$$
$$\tau((p, q), a, (p', q')) \iff \tau(p, a, q) \wedge \tau(p', a, q')$$

We can think of the new transition system as running  $T_1$  and  $T_2$  in parallel. Note that the size of  $T$  is quadratic in the sizes of  $T_1$  and  $T_2$ .

## Product Machine

If  $T_1$  and  $T_2$  come from DFAs  $M_1$  and  $M_2$  we can use their acceptance conditions to define an acceptance condition for  $T$  and produce an automaton  $M$ .

The new initial state is  $(q_{01}, q_{02})$ .

As it turns out, we can get union, intersection and difference of  $\mathcal{L}(M_1)$  and  $\mathcal{L}(M_2)$  by selecting the final states the right way:

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

## It's a DFA

Note that the construction of  $T$  is a little be easier when the given transition systems are complete and deterministic: we get back a complete and deterministic system.

$$Q = Q_1 \times Q_2$$

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

Attaching an acceptance condition to  $T$  thus produces a DFA.

## Closure Properties

Note that we have established a closure property for regular languages:

### Lemma

*Regular languages are closed under union, intersection and complement. Thus they form a Boolean algebra.*

More importantly, we have effective closure: give two DFAs we can easily compute a new DFA for  $\mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ ,  $\mathcal{L}(M_1) \cup \mathcal{L}(M_2)$  and  $\mathcal{L}(M_1) - \mathcal{L}(M_2)$ .

This should sound very familiar by now: there is an analogous result for decidable and semi-decidable relations.

## Aside: Complement

The product construction is needed for differences  $L_1 - L_2$ .

If we are interested in a plain complement  $\Sigma^* - L$  we can trivially build a DFA:

Given a DFA  $M$  for  $L$ , keep the transition system and modify the acceptance condition by replacing  $F$  by its complement:

$$F' = Q - F.$$

in the given DFA.

### Dire Warning:

Determinism is essential here, we will see shortly that complementation for nondeterministic machines is much harder.

## Deciding Equivalence

There is another decision problem lurking in the dark:

Problem:       **Equivalence**  
Instance:       Two DFAs  $M_1$  and  $M_2$ .  
Question:       Are the two machines equivalent?

We can solve this problem now by a product machine construction:

### Lemma

$M_1$  and  $M_2$  are equivalent iff  $\mathcal{L}(M_1) - \mathcal{L}(M_2) = \emptyset$  and  $\mathcal{L}(M_2) - \mathcal{L}(M_1) = \emptyset$ .

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:	<b>Inclusion</b>
Instance:	Two DFAs $M_1$ and $M_2$ .
Question:	Is $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$ ?

which problem can be handled by

### Lemma

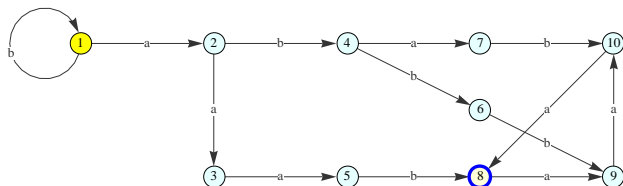
$\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$  iff  $\mathcal{L}(M_1) - \mathcal{L}(M_2) = \emptyset$ .

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.

## Aside: Complicated Intersections

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

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

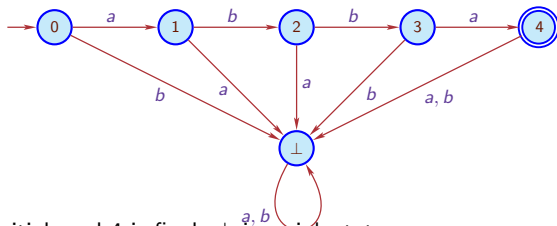


## Recognizing Words

Given a word  $w$ , it is trivial to construct a DFA  $M_w$  on  $|w| + 2$  states such that

$$\mathcal{L}(M_w) = \{w\}.$$

For example, for  $w = abba$  we get



State 0 is initial, and 4 is final.  $\perp$  is a sink state.

### Exercise

Show that  $|w| + 2$  is indeed the state complexity of  $\{w\}$ .

## Finite Languages

It follows from our closure properties that every finite language is also regular: we can build a DFA  $M$  for any finite set of words

$$\mathcal{L}(M) = \{w_1, w_2, \dots, w_s\}.$$

by forming the product of the  $M_{w_i}$

Alas, this does not really work: the size of this product machine grows exponentially.

But, there are several efficient algorithms to build machines for finite sets of words. In fact, there is a whole industry of such algorithms. Bear in mind: blind application of powerful methods sometimes leads to disaster.

## Sizes of Product Machines

More generally, suppose we have DFAs  $M_i$  of size  $n_i$ , respectively.

Then the product machine

$$M = M_1 \times M_2 \times \dots \times M_{s-1} \times M_s$$

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

- The product machine grows exponentially, but at least on occasion there are ways around this problem (e.g. finite languages).
- Are there cases where exponential blow-up cannot be avoided?
- If so, what can be done in general to improve efficiency?

## 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  $M_1, \dots, M_n$  of DFAs  
Question:       Is  $\bigcap \mathcal{L}(M_i)$  empty?

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

### Theorem

*The DFA Intersection Problem is PSPACE-hard.*

## Accessible Part

### Definition

A state  $p$  in a DFA is **accessible** if  $\delta(q_0, x) = p$  for some word  $x$ . The automaton is accessible if all its states are.

Thus a state is accessible it can be reached from the initial state by a sequence of transitions.

Now suppose we remove all the inaccessible states from a DFA  $M$ .

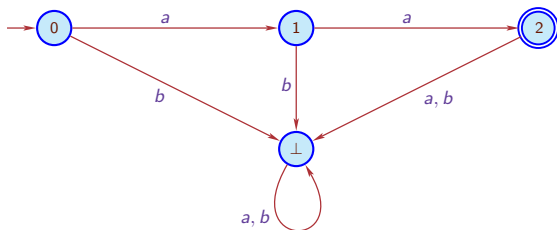
After adjusting  $Q$ ,  $\delta$  and  $F$  we obtain a new DFA  $M'$ , the so-called **accessible part** of  $M$ .

### Lemma

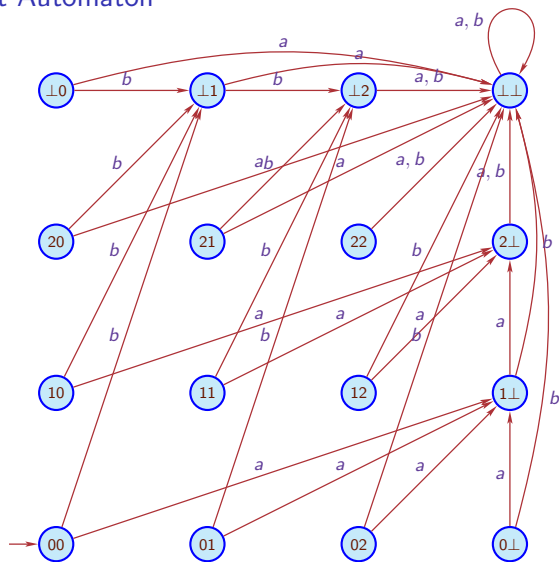
*The machines  $M$  and  $M'$  are equivalent.*

## Example

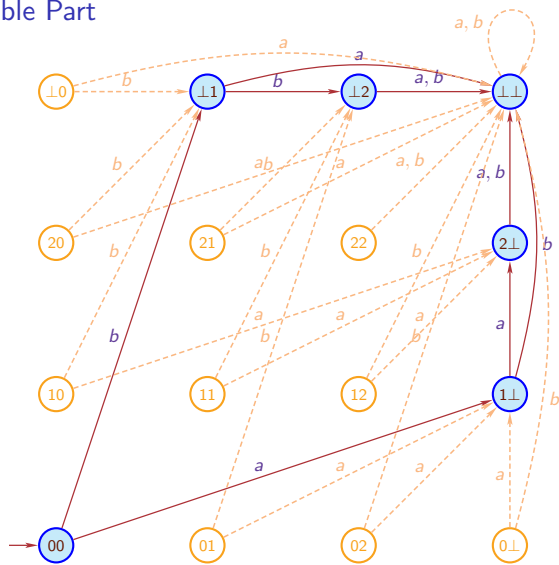
Consider the product automaton for  $M_{aa}$  and  $M_{bb}$ .



## Full Product Automaton



## The Accessible Part



## Constructing the Accessible Part

So suppose we are given a DFA  $M$  and we want to construct its accessible part  $M'$ .

Testing whether a state is accessible is really a path existence problem in the diagram of the machine. Routinely solved in linear time by standard graph exploration algorithms (DFS, BFS).

### Lemma

*The accessible part of a DFA can be constructed in linear time.*

This is important for machines constructed by other algorithms, not those built by hand – no one would ever write down inaccessible states (hopefully).

## Avoiding Inaccessibility

There are really two separate issues here.

One is to clean up machines by running an accessible part algorithm whenever necessary – this is easy.

Much more interesting is to avoid the construction of inaccessible machines in the first place: ideally any algorithm on FSMs should only produce accessible machines. E.g., we don't need the full product automaton, just its accessible part. It would be nice to have some general framework for this.

As we will see later, this kind of construction is very familiar in algebra.

- Closure Properties

## 2 More Closure Properties

- Nondeterministic Machines
- Autonomous Transitions
- Machines and Complexity

## More Operations

Regular languages are closed under many more operations, in particular

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

Alas, it is difficult to establish these properties within the framework of DFAs: the constructions of the corresponding machines become rather too complicated.

One elegant way to avoid these problems is to generalize our machine model to allow for nondeterminism, and show that the general machines still only accept regular languages.

First some examples.

## Concatenation and Kleene Star

### 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.

## Star Examples

### 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

$\{\epsilon, a, aa\} \{b, ba, baa\}^*$ : 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

## Concatenation is Hard

Suppose we have DFAs  $M_1$  and  $M_2$  for  $L_1$  and  $L_2$ .

Can we build a DFA for  $L_1 \cdot L_2$ ?

The problem is that given a word  $w$  we need to split it as  $w = xy$  and then feed  $x$  to  $M_1$  and  $y$  to  $M_2$ . But there are  $|w| + 1$  many ways to do the split, and we have a priori no idea where the break should be.

One can also think of this as a **guess and verify** problem: guess  $x$  and  $y$ , and then check that indeed  $M_1$  accepts  $x$ , and  $M_2$  accepts  $y$ .

Of course, there is a slight problem: DFAs don't know how to guess.

## Kleene Star and More Nondeterminism

The situation gets worse if we try to construct a DFA for the **Kleene star** of a language:

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

Not only do we not know where to split the string, we also don't know how many blocks there are.

Moreover, the number of blocks is unbounded (at least in general), and it is far from clear how this type of processing can be handled by a DFA.

## Brute Force

One way to avoid guessing is to systematically enumerate all possibilities.

For this and other constructions it is helpful to think of a finite state machine as **pebbled digraphs**.

- Initially a pebble is placed on the node representing the initial state.
- For every input symbol, the pebble is moved along the corresponding labeled edge.
- Acceptance corresponds to the pebble sitting on a final state when the input is finished.

Actually, there may be several pebbles (which is OK as long as we can think of every arrangement of pebbles as a state in a transition system, and the evolution is deterministic).

## Pebbling Automaton for Concatenation

We start with one copy of  $M_1$  (the master) and  $n$  copies of  $M_2$  (the slaves) where  $n$  is the state complexity of  $M_2$ .

- Place one pebble on the initial state of the master machine.
- Move this and other pebbles according to the input.
- Whenever the master pebble reaches a final state, place a new pebble on the initial state of a currently unpebbled slave automaton.
- If two slave automata have their pebble on the same state, remove one of them.

In the end accept if at least one of the slave pebbles sits on a final state.

## Exercises

### Exercise

*There are several gaps and inaccuracies in the outline above, fix them all.*

### Exercise

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

### Exercise

*Explain why the pebbling construction really defines a DFA.*

### Exercise

*Carry out a pebbling construction for Kleene star.*

## Reversal Closure

Here is yet another example of an operation that is difficult to capture within the confines of DFAs.

Let

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

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

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

How would we go about constructing a machine for  $L^{\text{op}}$ ?

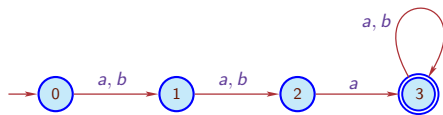
Well, flip all the transitions around. In spirit, that is the right answer, but, of course, the result will almost never be a DFA.

Pebbles don't seem to help much either.

## Example: Third Symbol

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

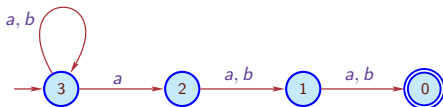
We omit the sink to keep the diagram simple.



## Third Symbol From the End

But  $L_{a,3}^{\text{OP}} = \{x \mid x_{-3} = a\}$  is hard for DFAs: we don't know how far from the end we are.

By flipping transitions (and the whole machine to get the initial state over on the left) we obtain a machine that looks like so:



Now 3 is the initial state and 0 is final.

The problem with this machine is that there are now potentially many computations for the same input.

- Closure Properties
- More Closure Properties
- ③ Nondeterministic Machines
  - Autonomous Transitions
  - Machines and Complexity

## Nondeterministic FSMs

The last few examples suggest that we need a conceptual adjustment: we should consider machines that are not deterministic.

### Definition

A **nondeterministic finite automaton (NFA)** is a structure

$$M = \langle Q, \Sigma, \tau; I, F \rangle$$

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

So in general there is no unique next state in an NFA: there may be no next state, or there may be many. But note that every DFA is automatically also an NFA, albeit a very special one.

We have already seen that this is useful in constructing a machine for  $L^{\text{op}}$ ; we will see in a moment that concatenation and Kleene star is also easier to handle with nondeterministic machines.

## Sources of Nondeterminism

Nondeterminism means that the number of runs of a machine on a particular input can be larger than one.

Note that nondeterminism can arise from two different sources:

- Transition nondeterminism: there are different transitions  $p \xrightarrow{a} q$  and  $p \xrightarrow{a} q'$ .
- Initial state nondeterminism: there are multiple initial states.

In other words, even if the transition relation is deterministic we obtain a nondeterministic machine by using multiple initial states.

## Traces, Runs and Labels

In order to define acceptance for a nondeterministic machine we use traces and runs, just as before for deterministic machines.

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

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

where  $p_i \in Q$ ,  $a_i \in \Sigma$  and  $\tau(p_{i-1}, a_i, p_i)$  for all  $i = 1, \dots, r$ .

$p_0$  is the **source** of the run and  $p_r$  its **target**. The length of  $\pi$  is  $r$ .

The corresponding **run** is the sequence  $p_0, p_1, \dots, p_r$  of states.

The corresponding **label** or **input** is the word  $a_1 a_2 \dots a_r$ .

## The Fatal Definition

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

### Definition

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

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

There is a hidden existential quantifier here.

Again: all that is needed for acceptance is one accepting trace.

## Example: Third Symbol from the End

Consider the input  $x = baaba$ . Here are the possible traces of  $M$  from above with this input (for emphasis we write the transitions with arrows). The last one leads from the initial state to the final state, so the machine accepts  $x$ .

$$\begin{array}{cccccccccccc}
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 \\
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 2 \\
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 2 & \xrightarrow{b} & 1 & \xrightarrow{a} & 0
 \end{array}$$

But  $x = babaa$  is not accepted, none of runs has the right source and target.

$$\begin{array}{cccccccccccc}
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 3 \\
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{a} & 2 \\
 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 3 & \xrightarrow{b} & 3 & \xrightarrow{a} & 2 & \xrightarrow{a} & 1
 \end{array}$$

## Reversal Closure

Let's give a somewhat formal proof that NFAs really solve the problem of closure under reversal assuming that we already know that NFAs recognize only regular languages.

### Theorem

*For any regular language  $L$ , the reversal  $L^{\text{op}}$  is also regular.*

*Proof.*

Let  $M = \langle Q, \Sigma, \tau, I, F \rangle$  be an NFA for some regular language  $L$ .

Then

$$M^{\text{op}} = \langle Q, \Sigma, \tau^{\text{op}}, F, I \rangle$$

is an NFA for  $L^{\text{op}}$  where

$$\tau^{\text{op}} = \{ (p, a, q) \mid \tau(q, a, p) \}$$

It follows that  $L^{\text{op}}$  is regular. □

## Acceptance Testing

Recall one of our central motivations for studying DFAs: acceptance testing is very fast.

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

In a DFA any input determines a unique run with initial state  $q_0$  and we can simply follow this run. But in an NFA there may be multiple runs starting at several initial states.

We need to follow all these runs. We could try to enumerate the corresponding paths in the transition diagram, but that's a bad idea: there might be exponentially many.

But note that the only important quantity that we need to determine is the set of states we could reach from  $I$  with input  $x$ .

So all we need to do is to maintain a set of states (which we will do using a simple iterative algorithm scanning the input symbol by symbol).

## Membership Testing

Here is a natural modification of the program that tests acceptance for a DFA.

```
P = I;
while( a = x.next() )    // next input symbol
    P = Tau[P,a];

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

The update step uses the map

$$\tau(P, a) = \{ q \in Q \mid \tau(p, a, q) \wedge p \in P \}$$

where  $P$  is a set of states. So we are thinking of  $\tau$  as a function

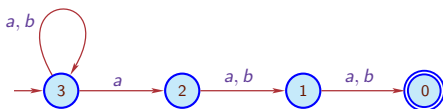
$$\tau : \text{pow}(Q) \times \Sigma \rightarrow \text{pow}(Q)$$

It's a free country, after all.

## Running Time

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

## Example: Third Symbol from the End



It is helpful to pre-compute the following table,  $\tau(P, a)$  is then just the union of some of the sets in the table. Note how the sets in the table are small (think about the 20th symbol from the end).

	0	1	2	3
<i>a</i>	$\emptyset$	$\{0\}$	$\{1\}$	$\{2, 3\}$
<i>b</i>	$\emptyset$	$\{0\}$	$\{1\}$	$\{3\}$

## So What?

- All we have so far is fairly elegant way to construct finite state machines for languages obtained from regular ones by concatenation, Kleene star and reversal. Alas, the new machines fail to be DFAs, so none of this establishes closure.
- We need to show that the acceptance languages of NFAs are again regular.
- Note that our constructions also produce NFAs when the original languages are given by NFAs, so things generalize nicely.
- Since NFAs are a priori more general than DFAs the question arises if there is an NFA that is not equivalent to any DFA. It turns out the answer is No, but there is a price to pay.

## Determinization

Conversion of a nondeterministic machine to a deterministic one appeared first in a seminal paper by Rabin and Scott titled “Finite Automata and Their Decision Problem.” In fact, nondeterministic machines were introduced there.

### Theorem (Rabin,Scott 1959)

*For every NFA there is an equivalent DFA.*

The idea is to use the Acceptance Testing algorithm for NFAs from above: compute the set of states the automaton could be in after scanning some input.

More precisely, instead of re-calculating the sets  $\tau(P, a)$  for every input we compute them once and for all.

Note that the map  $(P, a) \mapsto \tau(P, a)$  is perfectly deterministic, we really obtain a DFA this way.

## Proof of Rabin-Scott

Suppose  $M = \langle Q, \Sigma, \tau, I, F \rangle$  is an NFA. Let

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

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

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

□

The machine from the proof is the **full power automaton** of  $M$ , written  $\text{pow}_f(M)$ , a machine of size  $2^n$ .

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

Example:  $L_{a,-3}$ 

Applying this construction (accessible part only) to the NFA for  $L_{a,-3}$  from above we obtain a machine with 8 states

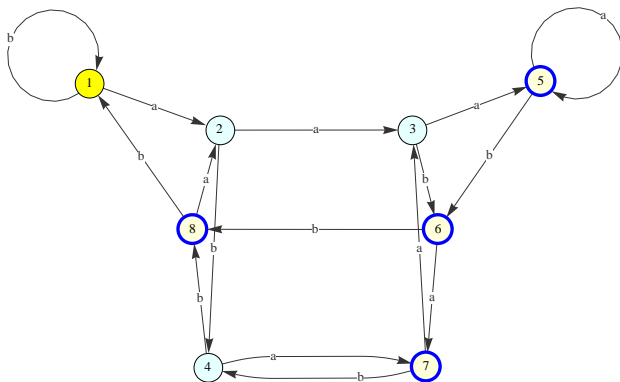
$$\{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 3\}, \{1, 2, 3, 4\}, \{1, 3, 4\}, \{1, 2, 4\}, \{1, 4\}$$

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, this construction only builds the accessible part (which happens to have size 8).

## Transition System



## 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.

- Closure Properties
- More Closure Properties
- Nondeterministic Machines
- 4 Autonomous Transitions
- Machines and Complexity

## Back To Concatenation

The weary reader will notice that we still have not shown how to handle concatenation (except for the pebbling argument).

Can we construct an NFA for  $L_1 \cdot L_2$ , assuming we have two NFAs (or DFAs, it does not matter much)?

- Assume that the state sets are disjoint and think of both machines together as one machine.
- We use  $l_1$  as initial states, and  $F_2$  as final states.
- A computations starts in  $l_1$  and continues until we get to  $F_1$ .
- Then we either continue in  $M_1$  or move to  $M_2$ .
- Thus, we need to add transitions from  $F_1$  to  $l_2$ .
- **Problem:** The transitions should not consume any input.

Exactly how should we do this?

## Epsilon Moves

Here is a trick: we generalize our machines even more, and then prove a result similar to Rabin-Scott that allows us to go back to ordinary NFAs.

If you recall the origin of our machines (braindamaged Turing machines) this is perfectly acceptable.

### Definition

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

A transition labeled  $\varepsilon$  does not consume an input symbol, think of it as an autonomous transition. Thus, an NFAE may perform several transitions without scanning a symbol.

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

## NFAE versus NFA

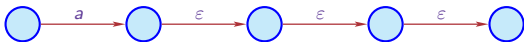
As we will show in a moment, languages accepted by NFAEs are again regular. To this end, it suffices to show how any NFAE can be converted into an equivalent NFA, a process called **epsilon elimination**.

The idea is simple: we remove all  $\epsilon$ -transitions and introduce new ordinary transitions that have the same effect.

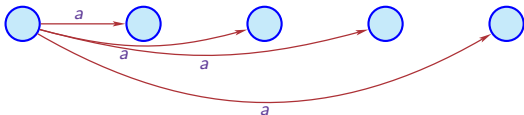
Since there may be chains of  $\epsilon$ -transitions this is in essence a transitive closure problem.

$\epsilon$ -Closure

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



by new transitions



## Epsilon Elimination

### Theorem

*For every NFAE there is an equivalent NFA.*

*Proof.*

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

Suppose  $M = \langle Q, \Sigma, \tau, I, F \rangle$  is an NFAE for  $L$ . Let

$$M' = \langle Q, \Sigma, \tau', I', F \rangle$$

where  $\tau'$  is obtained from  $\tau$  as on the last slide.

$I'$  is the  $\varepsilon$ -closure of  $I$ : all states reachable from  $I$  using only  $\varepsilon$ -transitions.  $\square$

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

## Concatenation Example

Over the alphabet  $\{a, b\}$ , consider

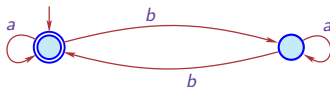
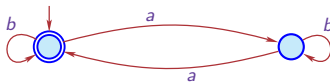
$E_a =$  even number of  $a$ 's

$E_b =$  even number of  $b$ 's

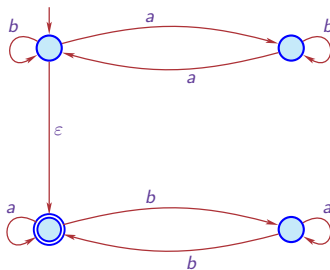
We already know how to construct two machines  $M_1$  and  $M_2$  for  $E_a$  and  $E_b$ .

The machines can then be combined into one machine for  $L = E_a E_b$ .

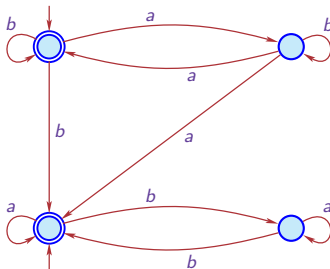
## The Two Machines



No problem. But it is not so clear what a machine for  $L$  would look like. In fact, it's not even clear what  $L$  is.

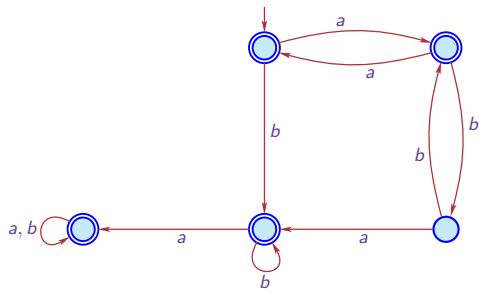
Using  $\epsilon$ -Moves

From the final state of the first machine, allow for autonomous transitions to the initial state of the second.

No  $\epsilon$ -Moves

We can eliminate the  $\epsilon$ -transition by adding appropriate honest transitions.

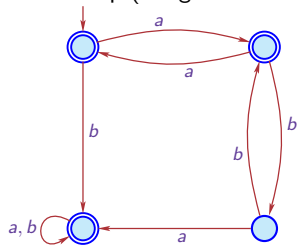
## No Nondeterminism



Eliminating nondeterminism using the Rabin-Scott construction.

## No Nondeterminism

A little cleanup (merge the two states on bottom left).



The language  $E = E_a E_b$  is a bit more complicated than one might think. For example, the last DFA shows that every odd-length string is in  $E$ .

## Growth Functions

A standard tool in the study of languages is a count of the number of words of a given length.

### Definition

Given a language  $L \subseteq \Sigma^*$  its **growth function** (or **census function**) is defined by  $\gamma_L : \mathbb{N} \rightarrow \mathbb{N}$ ,

$$\gamma_L(n) = |L \cap \Sigma^n|$$

For regular languages growth functions are particularly simple (they have rational generating functions).

Growth of  $E$ 

Here is the complement of  $E = E_a E_b$  up to words of length 8.

*ab*

*aaab, abbb*

*aaaaab, aaabbb, abbaab, abbbbbb*

*aaaaaaaaab, aaaaaabbb, aaabbaab, aaabbbbb, abbaaaab, abbaabbb, abbbbaab, abbbbbbb*

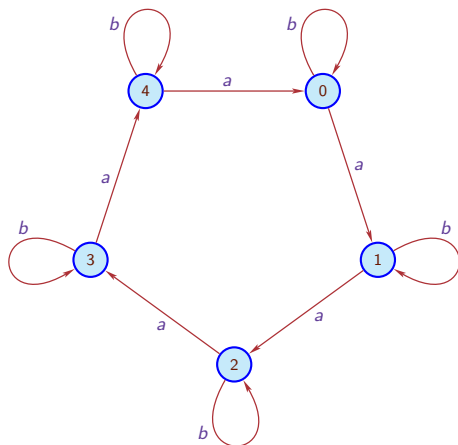
Sizes of the complement of  $E$  for words up to length 18.

$n$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\gamma(n)$	0	0	1	0	2	0	4	0	8	0	16	0	32	0	64	0	128

## Exercise

*Explain the growth rate of  $L$  (or, alternatively, of its complement).*

## Exercise



It is easy to generalize from counting modulo 2 to any other modulus. What would the last construction look like in the general case?

## Concatenation Closure

### Theorem

*For any two regular languages  $L_1$  and  $L_2$ , their concatenation product  $L_1L_2$  is again regular.*

*Proof.*

Suppose we have NFAEs  $M_1$  and  $M_2$  for  $L_1$  and  $L_2$ . We may safely assume that the state sets are disjoint, set  $Q = Q_1 \cup Q_2$ .

Add  $\varepsilon$ -transitions from  $F_1$  to  $I_2$ , choose  $I_1$  as the set of initial states and  $F_2$  as the set of final states. □

Note that the number of states increases linearly, but the number of transitions may increase quadratically. This indicates that state complexity is not a good measure for the size of a nondeterministic machine.

## Kleene Star

How do we establish closure with respect to the Kleene star operation?

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

Note that star is not a finitary operation, so it is not entirely clear how a machine for  $L^*$  could be constructed from a machine for  $L$ .

Of course, we can build machines for  $L^n$  for any  $n$ , and thus for

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

Alas, that's neither here nor there: these machines get bigger and bigger and certainly do not have a finite state machine as limit.

## Star Closure

### Theorem

*For any regular language  $L$ ,  $L^*$  is also regular.*

*Proof.*

Suppose  $M = \langle Q, \Sigma, \tau, I, F \rangle$  is an NFAE for  $L$ .

Let  $b$  and  $e$  be two new states and set

$$M' = \langle Q \cup \{b, e\}, \Sigma, \tau', \{b\}, \{e\} \rangle$$

where  $\tau'$  is  $\tau$  plus  $\varepsilon$ -transitions from  $b$  to  $I$ ,  $F$  to  $b$ , and  $I$  to  $e$ . □

This is more complicated than necessary, but adding begin/exit states keeps the number of new transitions linear. Of course,  $\varepsilon$ -elimination makes a mess.

- Closure Properties
- More Closure Properties
- Nondeterministic Machines
- Autonomous Transitions
- ⑤ Machines and Complexity

## Algorithmic Issues

So we have three increasingly complicated types of machines: DFAs, NFAs and NFAEs, that all accepted exactly the regular languages. There are two conversion algorithms:

- Elimination of  $\varepsilon$ -moves: conversion from NFAE to NFA.
- Elimination of nondeterminism: conversion from NFA to DFA.

The first one is comes down to computing transitive closure of the  $\varepsilon$ -transitions and can be handled efficiently using standard graph algorithms.

But nondeterminism is more difficult to get rid of: there may be an exponential blow-up in the state complexity of the deterministic machine.

## Exponential Blow-Up

From an algorithmic point of view,  $\varepsilon$ -elimination is no problem: we can compute all the  $\varepsilon$ -closures in time at most  $O(n^3)$ .

But the powerset construction is potentially exponential in the size of  $M$ .

Of course, it may happen that the accessible part is small, but sometimes (a large part of) the full powerset is accessible. Even worse, it can happen that this large power automaton is already reduced, so there is no way to get rid of these exponentially many states.

### Exercise

*Determine the running time of a reasonable implementation of the Rabin-Scott construction. Make sure to build only the accessible part.*

## Blow-Up Example 1

Recall the languages

$$L(a, k) = \{ x \mid x_k = a \}$$

### Proposition

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

*Proof.*

Applying the power automaton construction to the canonical NFA produces a DFA of size  $2^k$ , and one can show that this machine is reduced.

□

## Constructing a DFA by Hand

Of course, one can also build a DFA for  $L(a, -k)$  from scratch. Let's assume  $k = 3$ .

We need to remember the last 3 symbols of the input: if we've reached the end we have enough information to decide whether we should accept.

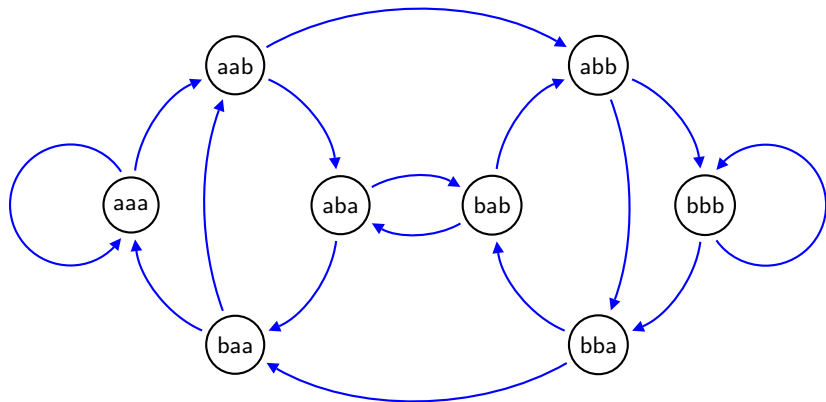
So we use  $Q = \{a, b\}^3$  and transitions

$$xyz \xrightarrow{s} yzs$$

Final states are  $\{aaa, aab, aba, abb\}$ .

Initial state is  $bbb$  (a clever hack, otherwise we would have to add some more states  $\varepsilon, a, b, aa, ab, \dots$ ).

## The Diagram



This is a so-called **de Bruijn graph** (of rank 3).

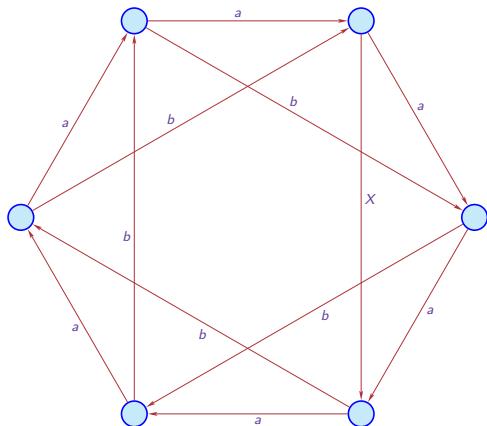
These graphs have lots of interesting properties.

## Blow-Up Example 2

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

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

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



## Tip of an Iceberg

The example generalizes to a whole group of circulant machines on  $n$  states with diagram  $C(n; 1, 2)$ .

Start with a labeling where the edges with stride 1 are labeled  $a$  and the edges with stride 2 are labeled  $b$ .

Then change exactly one of these edge labels: the resulting nondeterministic machines have power automata of size  $2^n$  and the power automata are already reduced.

### Exercise

*Full blow-up means that for any subset  $P \subseteq [n]$  there is some word  $x$  such that  $\delta([n], x) = P$ . Determine such a word  $x$ .*

### Exercise

*Prove that full blow-up occurs for all these NFA.*

## Decision Problems

- The paper by Rabin and Scott also introduced the study of the computational complexity of various decision problems associated with finite state machines.
- We have already seen some of these: Emptiness, Finiteness, Universality, Equality and Inclusion. For DFAs they are all easily solvable (linear or quadratic time).
- We can ask the same questions for other representations of regular languages, in particular NFAs and regular expressions (next lecture). Since the conversion NFA to DFA can be exponential it is not clear that there are good algorithms.

## Decision Problems for Regular Languages

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  $L = \Sigma^*$ ?

Emptiness and Finiteness are easily polynomial time for DFAs and NFAs.

Universality is polynomial time for DFAs but PSPACE-complete for NFAs.

## More Problems

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  $L_2$ ?

Equality and Inclusion are polynomial time for DFAs.

Both problems are PSPACE-complete for NFAs.

## Predicting Blow-Up

Many algorithms for example in the area of pattern matching naturally produce nondeterministic machines. Exponential blow-up makes it somewhat difficult to decide whether it is advantageous to compute the corresponding power automaton: the actual matching process is faster but the machine may be too large.

Likewise, it is not clear in general whether minimization is preferable: the cost of minimization is significant, the speed-up in acceptance testing essentially nil.

It would be nice if one could perform a simple, cheap test to determine what the size of the power automaton would if one were to go ahead with conversion. Unfortunately, the following problem is PSPACE-hard:

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

## Summary

- Deterministic finite state machines can be used to recognize some patterns (certain classes of input strings) very efficiently.
- In order to deal with more complicated patterns it is convenient to generalize to nondeterministic machines and even machines with autonomous transitions.
- Acceptance testing for these generalized machines is slower than for DFAs.
- All these generalized machines turn out to be equivalent to the original DFAs, albeit at a potentially exponential blow-up in size.
- Some decision problems are polynomial time solvable for deterministic and nondeterministic machines, but some problems for nondeterministic machines are PSPACE-hard (Equality, Universality).