Finite State Machine Algorithms

Klaus Sutner
Carnegie Mellon University
Fall 2011

Outline:

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

Closure Properties:

Two Kinds of Algorithms

There are lots of highly efficient algorithms (and some not so efficient ones) operating on finite state machines. The fall roughly into two categories:

- String Related
  Test whether a finite state machine accepts a particular string; construction of these machines also fits here. These are used in many popular tools such as grep.

- Language Related
  Construct machines for various languages, test whether the language and/or machines have certain properties. These are used in decision algorithms e.g. in model checking and also play a role in complexity theory.

We will focus on the second type.

Closure Properties:

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 \( L(M_1) \cup L(M_2) \) for any two DFAs \( M_1 \) and \( M_2 \).

Closure Properties:

Products

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

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

\[
\begin{align*}
Q &= Q_1 \times Q_2 \\
\tau((p, q), a, (p', q')) &= \tau_1(p, a, p') \land \tau_2(q, a, q')
\end{align*}
\]

It is often helpful to 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 \( T_2 \).

Closure Properties:

Product Machine

To get a machine we need to define an acceptance condition. In the case where \( T_1 \) and \( T_2 \) come from DFAs \( M_1 \) and \( M_2 \) we can make the following choices:

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

By selecting final states in the product, we can get union, intersection and difference of \( L(M_1) \) and \( L(M_2) \):

- **Union**: \( F = F_1 \times Q_2 \cup Q_1 \times F_2 \)
- **Intersection**: \( F = F_1 \times F_2 \)
- **Difference**: \( F = F_1 \times (Q_2 - F_2) \)
Closure Properties

It’s a DFA

Note that the construction of the product system $T$ is a little bit 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 another DFA. So we can handle union, intersection and difference of DFAs easily.

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 $L(M_1) - L(M_2) = \emptyset$ and $L(M_2) - L(M_1) = \emptyset$.

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

Recognizing Words

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

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

For example, for $w = abba$ we get

State 0 is initial, and 4 is final. ⊥ is a sink state.

Exercise

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

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 $L(M) = \{w_1, w_2, \ldots, w_s\}$ 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 $M = M_1 \times M_2 \times \cdots M_{s-1} \times M_s$ has $n = n_1 n_2 \cdots n_s$ 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?

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

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

<table>
<thead>
<tr>
<th>Problem:</th>
<th>DFA Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance:</td>
<td>A list $M_1, \ldots, M_n$ of DFAs</td>
</tr>
<tr>
<td>Question:</td>
<td>Is $\bigcap L(M_i)$ empty?</td>
</tr>
</tbody>
</table>

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

The DFA Intersection Problem is PSPACE-hard.
Closure Properties:

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).

Coaccessible/Trim Part

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. 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 a DFA is equivalent to the whole machine. Moreover, we can construct the coaccessible and trim part of a DFA in linear time using standard graph algorithms.

Note, though, that the coaccessible part of a DFA may not be a DFA: the machine may become incomplete.

Avoiding Inaccessibility

There are really two separate issues here.

One is to clean up machines by running an algorithm that constructs the accessible/coaccessible/trim part whenever necessary – this is easy.

A much more interesting and challenging task is to avoid the construction of states that fail to be accessible/coaccessible in the first place.

Maintaining accessibility is not too difficult if one constructs the machine in stages starting at the initial state (this is really the problem of constructing the reachable part of an implicitly given graph). Coaccessibility is a bit more complicated since one does not know ahead of time which final states will turn up.

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:
\[
L^0 = \{ \varepsilon \},
L^{n+1} = L^n \cdot L
\]

The **Kleene star** of \( L \) is the language
\[
L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots
\]

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

**Star Examples**
- **Brute Force**
- **Concatenation and Kleene Star**
- **Pebbling Automaton for Concatenation**

**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:
\[
L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots
\]

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).

We start with one copy of \( M_1 \) (the master) and \( n_2 \) copies of \( M_2 \) (the slaves) where \( n_2 \) 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.
Note that the size of this machine is \( n_1 (n_2 + 1)^{n_2} \).
Better Pebbling

We can improve things significantly by using only one copy of $M_2$ and placing all pebbles into that one copy.

Of course, there is at most one pebble on each state.

This reduces the number of states to $n_12^{n_2}$

Note that the difference to the last construction is that we do not order the pebbles (sets versus lists).

Exercises

Exercise

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

Exercise

Carry out this construction for the languages $E_a = \text{even number of } a \text{'s}$ and $E_b = \text{even number of } b \text{'s}$ 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

$L^{op} = \{ x^{op} | x \in L \}$

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

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 $L^{op}$?

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 $L_{a,3} = \{ x | x_3 = a \}$.

We omit the sink to keep the diagram simple.

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.
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

\[ M = (Q, \Sigma, \tau; I, F) \]

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

Sources of Nondeterminism

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.

Nondeterministic Machines: 37

Traces, Runs and Labels

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.

Nondeterministic Machines: 42

The Fatal Definition

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 \((Q, \Sigma, \tau)\) a trace is an alternating sequence

\[ \pi = p_0, a_1, p_1, \ldots, a_r, p_r \]

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

The corresponding run is the sequence \(p_0, p_1, \ldots, p_r\) of states.

The corresponding label or input is the word \(a_1a_2\ldots a_r\).

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 \(x\).

\[
\begin{array}{ccccccc}
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 3 \\
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 2 \\
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 1
\end{array}
\]

But \(x = baaba\) is not accepted, none of runs has the right source and target.

\[
\begin{array}{ccccccc}
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 3 \\
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 2 \\
3 & \Downarrow & 3 & \Downarrow & 3 & \Downarrow & 1
\end{array}
\]

Reversal Closure

Theorem

For any regular language \(L\), the reversal \(L^\text{op}\) is accepted by an NFA.

Proof.

Let \(M = (Q, \Sigma, \tau; I, F)\) be an NFA for some regular language \(L\). Then

\[ M^\text{op} = (Q, \Sigma, \tau^\text{op}; F, I) \]

where

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

is an NFA that accepts \(L^\text{op}\).

We will show in a moment that NFAs accept only regular languages, so the last result shows that regular languages are closed under reversal.
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 \( x \) 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).

It is helpful to pre-compute the following table, some of the sets in the table. Note how the sets in the table are small (think about the 20th symbol from the end).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>{0}</td>
<td>{1}</td>
<td>{2,3}</td>
<td></td>
</tr>
<tr>
<td>( b )</td>
<td>{0}</td>
<td>{1}</td>
<td></td>
<td>{3}</td>
</tr>
</tbody>
</table>

Running Time

The transition relation in a NFA has the form

\[ \tau : Q \times \Sigma \rightarrow \mathcal{P}(Q) \]

By GAN we can think of it as a function:

\[ \tau^* : Q \times \Sigma \rightarrow \mathcal{P}(Q) \]

and this function naturally extends to

\[ \tau^* : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q) \]

One might ask WTF, but this is all we need for an acceptance testing algorithm.

Acceptance Testing Algorithm

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

\[
\begin{aligned}
P &= I; \\
\text{while}( a = x.next() ) & \quad // \text{next input symbol} \\
P &= \tau(P, a); \\
\text{return} \quad ( P \text{ intersect } F \neq \text{ empty} );
\end{aligned}
\]

The update step uses the map

\[ \tau(P, a) = \{ q \in Q \mid \tau(p, a, q) \land p \in P \} \]

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

\[ \tau : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q) \]

It’s a free country, after all.

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).

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. Also, 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.
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 recalculating 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.

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

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

where

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

and

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

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

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

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

$$\begin{align*}
\text{a:} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \\
\text{b:} & \quad 1 \quad 4 \quad 6 \quad 8 \quad 6 \quad 8 \quad 4 \quad 1
\end{align*}$$

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

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.
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 $I_1$ as initial states, and $F_2$ as final states.
- A computation starts in $I_1$ 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 $I_2$.

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 $\varepsilon$-transitions and introduce new ordinary transitions that have the same effect.

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

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.

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 $L = E_aE_b$. 

\[ E_a = \text{even number of } a \'s \]
\[ E_b = \text{even number of } b \'s \]
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.

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

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

Eliminating nondeterminism using the Rabin-Scott construction.

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

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

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 $\gamma_L : \mathbb{N} \to \mathbb{N}$:

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

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.

$ab$
$aaab, abbb$
$aaaaab, aaabbb, abbaab, abbbbb$
$aaaaaaab, aaaaabbb, aaabbaab, aaabbbbb, abbaaaab, abbaabbb, abbbbaab, abbbbbbb$

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\gamma(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>64</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>128</td>
</tr>
</tbody>
</table>

Exercise

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

Kleene Star

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

$L^* = L_0 \cup L_1 \cup L_2 \cup \ldots \cup L_n \cup \ldots$

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 $L_0 \cup L_1 \cup L_2 \cup \ldots \cup L_n$.

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 $M = (Q, \Sigma, \tau, I, F)$ is an NFAE for $L$.

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

$M' = (Q \cup \{b, e\}, \Sigma, \tau', \{b\}, \{e\})$

where $\tau'$ is $\tau$ plus $\epsilon$-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, $\epsilon$-elimination makes a mess.
Algorithmic Issues

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

- Elimination of ε-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 ε-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, ε-elimination is no problem: we can compute all the ε-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.

Blow-Up Example 2

Here is a 6-state NFA based on a circulant graph. Assume \( I = \emptyset = Q \).

If \( X = b \) than the power automaton has size 1.

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

The Diagram

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

These graphs have lots of interesting properties.
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 for Regular Languages

- **Emptiness Problem**
  - Instance: A regular language \( L \).
  - Question: Is \( L \) empty?
- **Finiteness Problem**
  - Instance: A regular language \( L \).
  - Question: Is \( L \) finite?
- **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.

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:

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

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

- **Emptiness Problem**
  - Instance: Two regular languages \( L_1 \) and \( L_2 \).
  - Question: Is \( L_1 \) a subset of \( L_2 \)?
- **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.

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).