CDM
Closure Properties

Klaus Sutner

Carnegie Mellon Universality

15-closure 2017/12/15 23:19

1 Nondeterministic Machines

2 Determinization

3 Closure Properties

Where Are We?

We have a definition of regular languages in terms of deterministic finite automata (DFAs).

There are two killer apps for regular languages: pattern matching and decision algorithms.

In order to get a better understanding of regular languages, it turns out that other characterizations can be very useful, both from the theory perspective as well as for the construction of algorithms.

The Logic Link

As already mentioned, we are primarily interested in using automata to solve decision problems in logic.

To this end we have to construct machines for various languages, quite often over large alphabets of the form

$$\Gamma = \Sigma \times \Sigma \times \ldots \times \Sigma$$

This turns out to be the right environment for checking validity of logical formulae over certain structures. Note that these alphabets can get quite large.

Language Operations

The key to many of these algorithms is the fact that regular languages are closed under a great many more operations. To begin with, there is closure under Boolean operations:

- union
- intersection
- complement

This is useful for pattern matching but also for decision algorithms (one can deal with propositional logic; in a while, we will see how to handle quantifiers).

More Operations

Here are some more operations on languages that do not affect regularity:

- 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 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.

Nondeterministic FSMs

Here is a straightforward generalization of DFAs that allows for nondeterministic behavior. Recall that transition systems may well be nondeterministic.

Definition

A nondeterministic finite automaton (NFA) is a structure

$$\mathcal{A} = \langle\, Q, \Sigma, \tau; I, F \,\rangle$$

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. Of course, we can think of a DFA as a special type of NFA.

Some authors insist that $I = \{q_0\}$. This makes no sense.

Traces and Runs

It is straightforward to lift the definition of acceptance from DFAs to NFAs (it all comes down to path existence, anyway).

Recall that in any transition system $\langle Q, \Sigma, \tau \rangle$ a run 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 $i = 1, \ldots, r$. p_0 is the source of the run and p_r its target. The length of π is r.

The corresponding trace or label is the word $a_1 a_2 \ldots a_r$.

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

But note that now there may be exponentially many runs with the same label. In particular, some of the runs 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 run.

Sources of Nondeterminism

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 allowing multiple initial states. Intuitively, this second type of nondeterminism is less wild.

Autonomous Transitions Epsilon Moves

While we are at it: there is yet another natural generalization beyond just nondeterminism: autonomous transitions, aka epsilon moves. These are transitions where no symbol is read, only the state changes. This is perfectly fine considering our Turing machines ancestors.

Definition

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

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.

We will encounter several occasions where it is convenient to “enlarge” the alphabet Σ by adding the empty word ε:

$$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$$

Of course, ε is not a new alphabet symbol. What’s really going on?

Σ freely generates the monoid $\Sigma^\star$, and ε is the unit element of this monoid. We can add the unit element to the generators without changing the monoid.

We could even add arbitrary words and allow super-transitions like

$$p \xrightarrow{aba} q$$

Exercise

Explain why this makes no difference as far as languages are concerned.

Reversal Closure

Here is a perfect example of an operation that preserves regularity, but is difficult to capture within the confines of DFAs.

Let

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

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

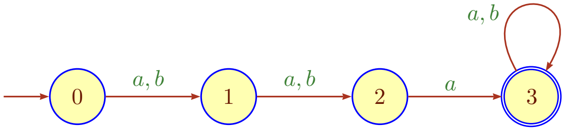
The direction in which we read a string should be of supreme irrelevance, so for regular languages to form a reasonable class they should be closed under reversal.

Suppose L is regular. How would we go about constructing a machine for L^{op}?

Example: Third Symbol

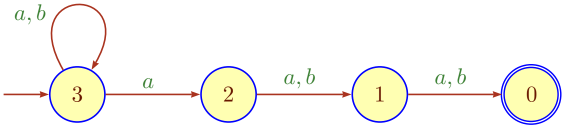
It is very easy to build a DFA for $L_{a,3} = \{ x \mid x_3 = a \}$.

We omit the sink to keep the diagram simple.



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

By flipping transitions and interchanging initial and final states we obtain a machine that looks like so:



It is clear that the new machine accepts $L_{a,-3}$.

Of course, it's no longer a DFA.

- Nondeterministic Machines

2. Determinization

- Closure Properties

Conversion to DFA

Our first order of business is to show that NFAs and NFAEs are no more powerful than DFAs in the sense that they only accept regular languages. Note, though, that the size of the machines may change in the conversion process, so one needs to be a bit careful.

The transformation is effective: the key algorithms are

Epsilon Elimination Convert an NFAE into an equivalent NFA.

Determinization Convert an NFA into an equivalent DFA.

Terminology

For the time being, we will refer to DFAs, NFAs and NFAEs simply as finite automata.

Strictly speaking, all three types are distinct, but there are natural inclusions

$$\mathrm{DFA} \subseteq \mathrm{NFA} \subseteq \mathrm{NFAE}$$

The heart of the OO fanatic now beats faster . . .

NFAE to NFA

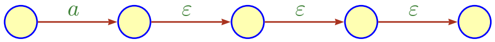
Epsilon elimination is quite straightforward and can easily be handled in polynomial time:

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

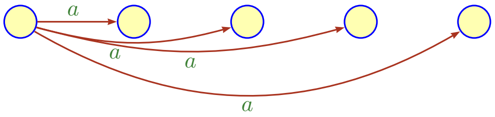
Since there may be chains of ε-transitions this is in essence a transitive closure problem. Hence part I of the algorithm can be handled with the usual graph techniques.

ε-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 $\mathcal{A} = \langle Q, \Sigma, \tau; I, F \rangle$ 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. $\Box$

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.

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 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.

General Abstract Nonsense to the Rescue

The transition relation in an NFA has the form

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

By GAN we can think of it as a function:

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

and this function naturally extends to

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

The latter function can be interpreted as the transition function of a DFA on $\mathfrak{P}(Q)$. Done.

;-)

Proof of Rabin-Scott

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. $\square$

The machine from the proof is the full power automaton of $\mathcal{A}$, written $\text{pow}_{\text{f}}(\mathcal{A})$, 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.

Accessible Part

This is as good a place as any to talk about "useless" states: states that cannot appear in any accepting computation and that can therefore be eliminated.

Definition

A state p in a finite automaton $\mathcal{A}$ is accessible if there is a run with source an initial state and target p. The automaton is accessible if all its states are.

Now suppose we remove all the inaccessible states from a automaton $\mathcal{A}$ (meaning: adjust the transition system and the set of final states). We obtain a new automaton $\mathcal{A}'$, the so-called accessible part of $\mathcal{A}$.

Lemma

The machines $\mathcal{A}$ and $\mathcal{A}'$ are equivalent.

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. 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.

Keeping Trim

Of course, in implementations we would like to keep machines small, so making them accessible or trim is a good idea. There are really two separate issues here.

- First, we may need to clean up machines by running an accessible or trim part algorithm whenever necessary–this is easy.
- Much more interesting is to avoid the construction of inaccessible states of a machine in the first place: ideally any algorithm should only produce accessible machines.

While accessibility is easy to guarantee, coaccessibility is not: while constructing a machine we do not usually know the set of final states ahead of time. So, there may by need to eliminate non-coaccessible states.

Smart Power Automata

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 $\mathfrak{P}(Q)$:

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

Here τ_a is the function $\mathfrak{P}(Q) \times \Sigma \to \mathfrak{P}(Q)$ defined by

$$\tau_a(P) = \{\, q \in Q \mid \exists\, p \in P\, (p \xrightarrow{a} q) \,\}$$

This produces the accessible part only, and, with luck, is much smaller than the full power automaton.

WTF?

Given a collection $\boldsymbol{f} = (f_1, \ldots, f_k)$, of endofunctions $f_i : A \to A$ on a set A and a subset $B \subseteq A$, the closure of B under $\boldsymbol{f}$ is defined by

$$\text{clos}(\, B, \boldsymbol{f}\,) := \bigcap \{\, X \subseteq A \mid B \subseteq X, f_i(X) \subseteq X, i \in [k]\,\}$$

In other words, the closure is the least subset of A that contains B and all the images of B under the f_i.

As written, the definition is impredicative, but we can easily turn this into an efficient algorithm (at least in the finite case).

Virtual Graphs

Think of

$$\mathcal{G} = \langle\, A; f_1, f_2, \ldots, f_k \,\rangle$$

as a labeled digraph with edges $p \xrightarrow{i} q$ for $f_i(p) = q$, the virtual graph (or ambient graph) where we live.

We need to compute the reachable part of B 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.

Power Automaton Algorithm

Here is a slightly more hacky version of this construction.

```
active = QQ = {I};

while( active != empty )
     P = active.extract();
     foreach a in Sigma do
         compute  R = tau(P,a)
         keep track of transition P to R
         if( R notin QQ ) then
               add R to QQ and active
```

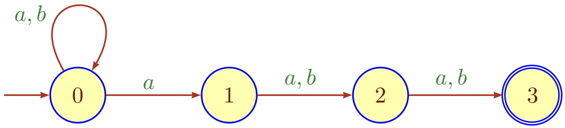
Upon completion, $QQ \subseteq \mathfrak{P}(Q)$ is the state set of the accessible part of the full power automaton. We write $\mathsf{pow}(\mathcal{A})$ for this machine.

Example: $L_{a,-3}$

Recall

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

For negative k this means: $-k$th symbol from the end. It is trivial to construct an NFA for $L_{a,-3}$:


$$\begin{tikzcd} \arrow[r] & 0 \arrow[loop above, "a,b"] \arrow[r, "a"] & 1 \arrow[r, "a,b"] & 2 \arrow[r, "a,b"] & \boxed{3} \end{tikzcd}$$

Rabin-Scott

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

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

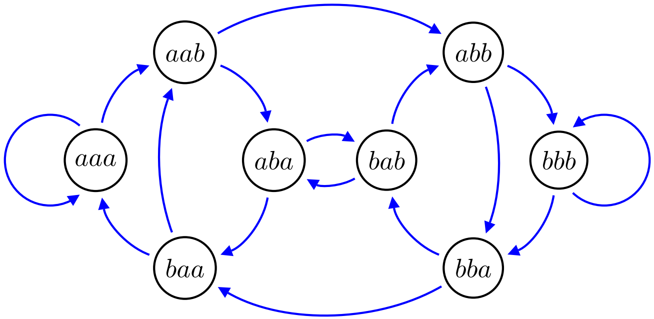
where 1 is initial and 5, 6, 7, and 8 are final. The transitions are given by

	1	2	3	4	5	6	7	8
a	2	3	5	7	5	7	3	2
b	1	4	6	8	6	8	4	1

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

The Diagram

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.

Acceptance Testing

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.

Membership Testing

Here is a natural modification of the DFA acceptance testing program.

```
P = I;
while(  a = x.next() )    // next input symbol
      P = tau_a(P);

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

The update step uses the same maps $\tau_a : \mathfrak{P}(Q) \to \mathfrak{P}(Q)$ as in the Rabin-Scott construction.

Exercise

Think of dirty tricks like hashing to speed things up.

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.

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.

- Nondeterministic Machines

- Determinization

3. Closure Properties

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 Σ.

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

$$
\begin{aligned}
&Q = Q_1 \times Q_2 \\
&\tau((p,q),a,(p',q')) \iff \tau_1(p,a,p') \wedge \tau_2(q,a,q')
\end{aligned}
$$

It is often helpful to think of the new transition system as running T_1 and T_2 in parallel.

Alas, the size of T is quadratic in the sizes of T_1 and T_2. This causes problems if a product machine construction is used repeatedly.

Product Machine

To get a machine we need to define an acceptance condition.

The new initial state set is $I_1 \times I_2$.

By selecting final states in the product, we can get union and intersection $\mathcal{L}(\mathcal{A}_1)$ and $\mathcal{L}(\mathcal{A}_2)$:

union	$F = F_1 \times Q_2 \cup Q_1 \times F_2$
intersection	$F = F_1 \times F_2$

It's a DFA

In the case where T_1 and T_2 come from DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ the product is again a DFA: the new transition function $\delta = \delta_1 \times \delta_2$ looks like

$$\begin{aligned} &Q = Q_1 \times Q_2 \\ &\delta((p,q),a) = (\delta_1(p,a), \delta_2(q,a)) \end{aligned}$$

More importantly, we can also construct a machine for the complement $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2)$.

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

Absolute Complements

In the special case where $\mathcal{A}_1$ is universal we get the plain complement $\Sigma^\star - \mathcal{L}(\mathcal{A}_2)$. Of course, we do not need a product construction here, all we need to do is switch final and non-final states in the given DFA.

$$F' = Q - F.$$

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

Exercise

Construct a counterexample that shows that the switch-states construction in general fails to produce complements in NFAs.

And back to closures . . .

For the umpteenth time: a real algorithm for product machines should **not** construct the full Cartesian product.

Instead, one should compute closures of the appropriate initial states under the transition function/relation of the product machine.

For example, for a product DFA construction we need

$$\mathsf{clos}(\,(q_{01}, q_{02}), \delta_1 \times \delta_2)$$

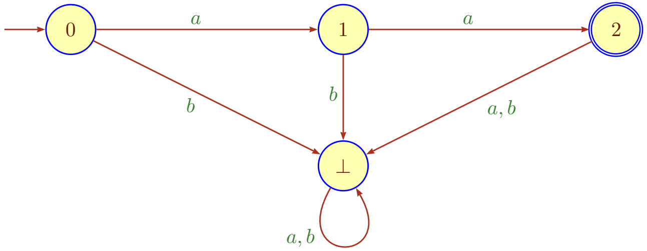
Exercise

Figure out in detail how to do these constructions producing the accessible part only.

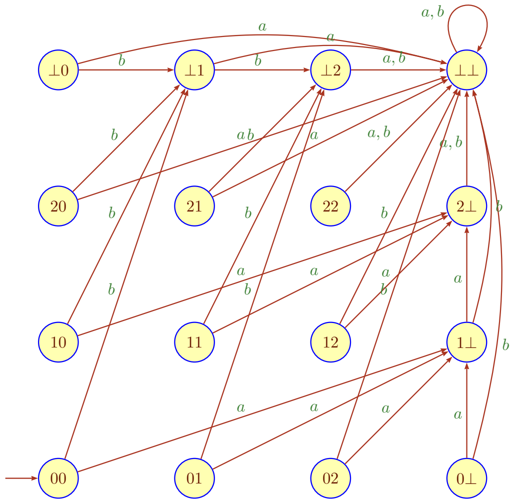
Example

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

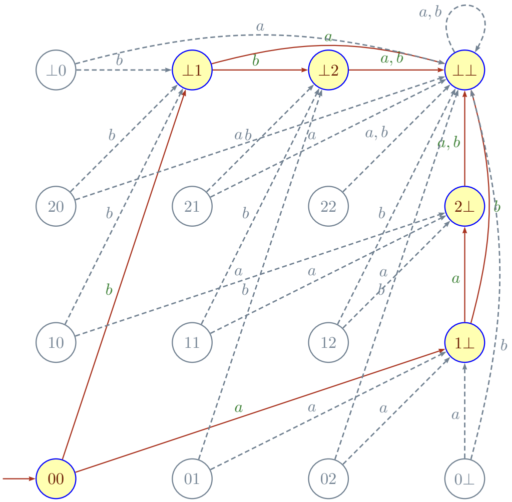
$\mathcal{A}_{aa}$:



Full Product Automaton



The Accessible Part



Closure Properties of Reg_Σ

At any rate, we have established a critical closure property for regular languages:

Lemma

Regular languages form a Boolean algebra: they are closed under union, intersection and complement. Moreover, the closure is effective and even polynomial time for DFAs.

Effective here means that, given two machines $\mathcal{A}_1$ and $\mathcal{A}_2$, we can compute a new machine for $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$, $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ and $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2)$. If the machines are DFAs all operations are polynomial time (but complement may blow up for NFAs).

We will have more to say about the complexity of the corresponding algorithms; this is critical for applications.

Déjà Vu All Over Again

The last lemma should sound very familiar by now: there are analogous results for semi-decidable sets, decidable set and primitive recursive sets.

Of course, for semi-decidable ones we have to omit complements.

In all cases, we can effectively construct machines/programs that recognize the sets obtained by Boolean operations. The real challenge is state complexity: in many applications one needs to deal with very large machines. The corresponding algorithm may require a lot of ingenuity.

Deciding Equivalence

We can now deal with the Equivalence problem from last time.

Problem:	**Equivalence**
Instance:	Two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$.
Question:	Are the two machines equivalent?

Lemma

$\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent iff $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2) = \emptyset$ and $\mathcal{L}(\mathcal{A}_2) - \mathcal{L}(\mathcal{A}_1) = \emptyset$.

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

Deciding Inclusion

Observe that we actually are solving two instances of a closely related problem here:

Problem:	**Inclusion**
Instance:	Two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$.
Question:	Is $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$?

which problem can be handled by

Lemma

$\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ *iff* $\mathcal{L}(\mathcal{A}_1) - \mathcal{L}(\mathcal{A}_2) = \emptyset$*.*

Note that for any class of languages Equivalence is decidable when Inclusion is so decidable. However, the converse may be false – but it's not so easy to come up with an example.

Concatenation and Kleene Star

Definition

Given two languages $L_1, L_2 \subseteq \Sigma^\star$ their concatenation (or product) is defined by

$$L_1 \cdot L_2 = \{\, xy \mid x \in L_1, y \in L_2 \,\}.$$

Let L be a language. The powers of L are the languages obtained by repeated concatenation:

$$L^0 = \{\varepsilon\}$$
$$L^{k+1} = L^k \cdot L$$

The Kleene star of L is the language

$$L^\star = L^0 \cup L^1 \cup L^2 \ldots \cup L^n \cup \ldots$$

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

Star Examples

Example

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

Example

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

Example

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

Example

$\{0, 1\}\{0, 1\}^\star$: all numbers in binary, with leading 0's

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

Concatenation is Easy

Suppose we have two NFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ for L_1 and L_2. To build a machine for $L_1 \cdot L_2$ it is easiest to go with an NFAE $\mathcal{A}$:

Let $Q = Q_1 \cup Q_2$, keep all the old transitions and add ε-transitions from F_1 to I_2. I_1 are the initial states and F_2 the final states in $\mathcal{A}$.

It is clear that $\mathcal{L}(\mathcal{A}) = L_1 \cdot L_2$. But note that this construction may introduce quadratically many transitions.

Exercise

Find a way to keep the number of new transitions linear.

Concatenation is Hard

Now suppose we have two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ for L_1 and L_2.

We can get a DFA from the previous machine, but can we build a DFA for $L_1 \cdot L_2$ directly?

The problem is that given a word w we need to split it as $w = xy$ and then feed x to $\mathcal{A}_1$ and y to $\mathcal{A}_2$. 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 $\mathcal{A}_1$ accepts x, and $\mathcal{A}_2$ 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:

$$L^\star = L^0 \cup L^1 \cup L^2 \ldots \cup L^n \cup \ldots$$

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.

Of course, we can simply start with an NFA or even and NFAE, and then convert back to a DFA.

NFAEs

If we are willing to deal with ε-transitions closure under concatenation and Kleene star is not hard at all.

Exercise

Given two NFAs, construct an NFAE for the concatenation of the two machines.

Exercise

Given a NFA, construct an NFAE for the Kleene star of the machine.

Pebbles

Another way of thinking about this is to place pebbles on the states.

- Initially, each state in I has a pebble.
- Under input a, a pebble on p multiplies and moves to all q such that $p \xrightarrow{a} q$.
- Multiple pebbles on a state are condensed into a single one.
- We accept whenever a pebble appears in F.

Pebbles are very helpful in particular in the direct construction of DFAs: the movement of the set of all pebbles is perfectly deterministic.

Pebble Automaton for Concatenation

We start with one copy of DFA $\mathcal{A}_1$, the master, and one copy of DFA $\mathcal{A}_2$, the slave.

- Place one pebble on the initial state of the master machine.
- Move this and all other pebbles along transitions according to standard rules.
- Whenever the master pebble reaches a final state, place a new pebble on the initial state of the slave automaton.
- The composite machine accepts if a pebble sits on final state in the slave machine.

So the number of states is bounded by $|\mathcal{A}_1|2^{|\mathcal{A}_2|}$: the $\mathcal{A}_1$ part is deterministic but the $\mathcal{A}_2$ part is not.

Quoi?

First, the machine just described is deterministic: we place and remove a collection of pebbles according to an entirely deterministic rule.

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.

Now let n_i be the state complexity of $\mathcal{A}_i$. Then the number of states is at most

$$n_1 \, 2^{n_2}$$

Of course, the accessible part may well be smaller.

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.

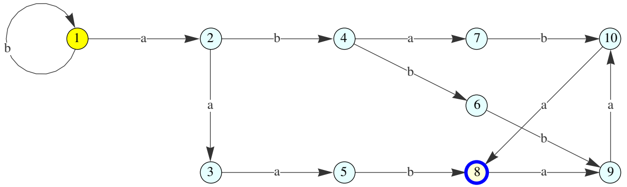
Exercise

Carry out a pebbling construction for Kleene star.

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



Sizes of Product Machines

More generally, suppose we have DFAs $\mathcal{A}_i$ of size n_i, respectively.

Then the full product machine

$$\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_{s-1} \times \mathcal{A}_s$$

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

- The full product machine grows exponentially, but its accessible part may be much smaller.
- Alas, there are cases where exponential blow-up cannot be avoided.

Bad News: DFA Intersection

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

Problem:	**DFA Intersection**
Instance:	A list $\mathcal{A}_1, \ldots, \mathcal{A}_n$ of DFAs
Question:	Is $\bigcap \mathcal{L}(\mathcal{A}_i)$ empty?

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

Theorem

The DFA Intersection Problem is PSPACE*-hard.*