

CDM

State Complexity

KLAUS SUTNER

CARNEGIE MELLON UNIVERSITY



1 Transformation Semigroups

2 State Complexity

3 Alternating Automata

We have seen that the class of regular languages has much stronger closure properties than one might suspect from the definition.

Even better, the proofs for these closure properties are all constructive: there are relatively straightforward algorithms to build the corresponding FSMs.

This raises the question: how effective can these FSM algorithms be? A good first step in this direction is to figure out the sizes of the machines.

But first a little algebra ...

Recall that any map $f : \Sigma \rightarrow \Gamma^*$ gives rise to uniquely determined monoid homomorphism $f : \Sigma^* \rightarrow \Gamma^*$

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

For example,

$$f(a) = 00 \quad f(b) = 01 \quad f(c) = 11$$

translates from $\{a, b, c\}$ to $\mathbf{2}^*$.

It is fairly clear that $f(L) \subseteq \mathbf{2}^*$ is regular whenever $L \subseteq \{a, b, c\}^*$ is. But the opposite direction is not so obvious.

Theorem

Regular languages are closed under homomorphisms and inverse homomorphisms.

Proof.

Let $f : \Sigma \rightarrow \Gamma^*$ be the homomorphism. Consider a NFA for $L \subseteq \Sigma^*$ and replace transitions $p \xrightarrow{a} q$ by $p \xrightarrow{f(a)} q$.

For the opposite direction suppose $\mathcal{A} = \langle Q, \Gamma, \gamma; q_0, F \rangle$ is a DFA for $L \subseteq \Gamma^*$. Let $\mathcal{B} = \langle Q, \Sigma, \delta; q_0, F \rangle$ be a DFA where $\delta(p, a) = \gamma(p, f(a))$. An easy induction on x shows $\gamma(p, x) = \delta(p, f(x))$.

A big part of automata theory can be rephrased in algebraic terms. This adds a bit of abstraction, but sometimes leads to very elegant proofs—remember, proofs are supposed to explain why the claim is true.

Batten Down the Hatches!

Suppose we have some set X and a collection F of endofunctions on X .

Definition

(X, F) is a **transformation semigroup** or **composition semigroup** if F is closed under composition, and a **transformation monoid** if, in addition, F contains a unit element.

If you prefer, you can think of the semigroup F as acting on X on the left in the natural way:

$$f \cdot x = f(x)$$

This is for standard composition; if we use diagrammatic composition (which is more natural in connection with finite state machines), we get a right action.

To see the connection to finite state machines, note that we can think of the transition function of a DFA as a Σ -indexed list of functions from states to states:

$$\begin{aligned}\delta_a &: Q \rightarrow Q \\ \delta_a(p) &= \delta(p, a)\end{aligned}$$

This turns the DFA into a Σ -algebra

$$\mathcal{A} = \langle Q; \delta_{a_1}, \dots, \delta_{a_k} \rangle$$

This may seem like a pointless exercise, but it naturally leads to another interesting perspective: algebra.

First off, nothing is lost: we can “iterate” these functions according to some input word $u = u_1 u_2 \dots u_n$:

$$\delta_u = \delta_{u_1} \circ \delta_{u_2} \circ \dots \delta_{u_{n-1}} \circ \delta_{u_n}$$

Acceptance then translates into: \mathcal{A} accepts a word u iff $\delta_u(q_0) \in F$.

Plus, we get some additional concepts more or less for free: a **subautomaton** of \mathcal{A} is another Σ -algebra $\mathcal{B} = \langle P; \gamma_{a_1}, \dots, \gamma_{a_k} \rangle$ such that $P \subseteq Q$ and $\gamma_a(p) = \delta_a(p)$.

Similarly, a **morphism** $\varphi : \mathcal{A} \rightarrow \mathcal{B}$ of Σ -algebras must be a map $\varphi : Q \rightarrow P$ such that

$$\varphi(\delta_a(p)) = \gamma_a(\varphi(p))$$

One may want to augment this by conditions about initial and final states.

It is also straightforward to define products of the form

$$\mathcal{A} \times \mathcal{B}$$

And we get congruences: an equivalence relation E on Q is a **congruence** if

$$p E q \text{ implies } \delta_a(p) E \delta_a(q)$$

To be sure, all these concepts can be developed without any appeal to algebra, given enough thought.

But the whole point here is that they pop up for free, courtesy of some general, universal ideas.

Exercise

Figure out exactly what morphism, product and congruence mean in this context.

The functions δ_a , $a \in \Sigma$, generate a transformation semigroup (monoid) T over Q , a subsemigroup of the full monoid of endofunctions $Q \rightarrow Q$.

Definition

T is called the **transformation semigroup (monoid)** of the DFA.

One way of writing down T is

$$T = \{ \delta_x : Q \rightarrow Q \mid x \in \Sigma^* \} = \langle \delta_a \mid a \in \Sigma \rangle$$

where $\delta_x(p) = \delta(p, x)$.

Analyzing this semigroup can help quite a bit in getting a better understanding of a DFA. And, there are powerful algebraic tools available that help in dealing with the monoid.

There is a natural 4-state DFA that accepts all strings over $\{a, b\}^*$ that contain an even number of a 's and an even number of b 's.

p	1	2	3	4
$\delta_a(p)$	2	1	4	3
$\delta_b(p)$	3	4	1	2

The initial state is 1 and $F = \{1\}$.

But note that

$$\delta_a \circ \delta_a = I$$

$$\delta_b \circ \delta_b = I$$

$$\delta_a \circ \delta_b = \delta_b \circ \delta_a$$

so that the transformation semigroup consists of $\{I, \delta_a, \delta_b, \delta_a \circ \delta_b\}$. Note that this is actually a monoid and even a group (Kleinsche Vierergruppe).

Moreover, from the equations it is easy to see that for any word x

$$\delta_x = I \iff \#_a x \text{ even}, \#_b x \text{ even}$$

Similarly we have

$$\delta_x = \delta_a \iff \#_a x \text{ odd}, \#_b x \text{ even}$$

and so on.

At the very least this very elegant and concise.

The usual de Bruijn automaton yields

p	1	2	3	4
$\delta_a(p)$	1	3	1	3
$\delta_b(p)$	2	4	2	4

This generates the semigroup (no monoid here)

$$(1, 3, 1, 3), (2, 4, 2, 4), (1, 1, 1, 1), (2, 2, 2, 2), (3, 3, 3, 3), (4, 4, 4, 4)$$

Question: What do the constant functions mean?

The distinction between semigroups and monoids here is a bit of a technical nuisance, but there is no easy way to get rid of it.

At any rate, note that we can turn any semigroup \mathcal{S} into a monoid \mathcal{S}^1 by simply adding a new element 1 and defining

$$x \cdot 1 = 1 \cdot x = x$$

for all x in \mathcal{S} .

Clearly, \mathcal{S} and \mathcal{S}^1 are essentially the same.

Also note that a transformation semigroup may be a monoid without containing the identity function.

The reason monoids are important here is because they provide a characterization of regular languages that is free of any combinatorial aspect. Always remember: algebra is the great simplifier.

Theorem

A language $L \subseteq \Sigma^$ is regular iff there is a finite monoid M , $M_0 \subseteq M$ and a monoid homomorphism $f : \Sigma^* \rightarrow M$ such that $L = f^{-1}(M_0)$.*

Proof.

If L is regular, let M be the transformation monoid of a DFA that recognizes L , and define $f(x) = \delta_x$ and $M_0 = \{g \in M \mid g(q_0) \in F\}$.

The opposite direction is more interesting: we construct a DFA

$$\mathcal{A} = \langle M, \Sigma, \delta; 1_M, M_0 \rangle$$

where $\delta(p, a) = p \cdot f(a)$. Then $\delta(p, x) = p \cdot f(x)$ and $\delta(q_0, x) = f(x)$. □

Message: anything goes as a state set, as long as the set is finite. For the implementer, the state set is always $[n]$, but that's not a good way to think about it.

Algebraic automata theory is a fascinating subject with lots of elegant results, but it requires work and there is no essential algorithmic payoff. So, we won't go there.

1 Transformation Semigroups

2 **State Complexity**

3 Alternating Automata

We already defined the **state complexity** of a FSM to be the number of states of the machine.

This is the standard measure of the size of a FSM and most results are phrased in terms of state complexity.

But note that this is a bit of an oversimplification: we really should be dealing with the **transition complexity**, the number of transitions, simply because this number corresponds more faithfully to the size of a FSM data structure.

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 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.

Given an NFAE \mathcal{A} of state complexity n , the first step in ε -elimination is to compute the ε -closures of all states; this takes at most $O(n^3)$ steps.

Introducing new transitions preserves state complexity, but can increase the transition complexity by a quadratic factor.

One interesting implementation idea for pattern matching is not to pre-process all of \mathcal{A} : instead one computes the closures on the fly and only when actually needed. This may be faster if the machine is large and only used a few times.

Alas, the powerset construction is potentially exponential in the size of \mathcal{A} , even when only the accessible part $\text{pow}(\mathcal{A})$ is constructed. The only general bound for the state complexity of $\text{pow}(\mathcal{A})$ is 2^n .

In practice, it happens quite often that the accessible part is small, but there are cases when the state complexity of the deterministic machine is close to 2^n . Even worse, it can happen that this large power automaton is already minimal, so there is no way to get rid of these exponentially many states.

Incidentally, determinization is quite fast as long as the resulting automaton is not too large.

Recall the k th symbol 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.

We have already seen the de Bruijn DFA for the language, a machine of size 2^k (at least for a binary alphabet).

It remains to show that this machine already has the smallest possible number of states.

Suppose \mathcal{A} is a DFA for $L_{0,-k}$ on less than 2^k states.

Consider all 2^k inputs $x \in \mathbf{2}^k$ and let

$$p_x = \delta(q_0, x)$$

Then $p_x = p_y$ for some $x \neq y$.

But then there is a word u such that $xu \in L_{0,-k}$ and $yu \in L_{0,-k}$.

Contradiction.



We know already that

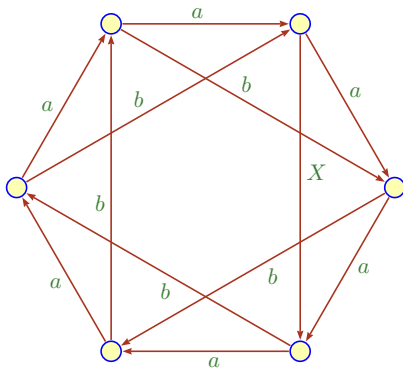
$$L \text{ is regular} \iff L^{\text{op}} \text{ is regular}$$

But there may be a price to pay: the state complexity of L^{op} may be exponential in the state complexity of L .

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

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

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

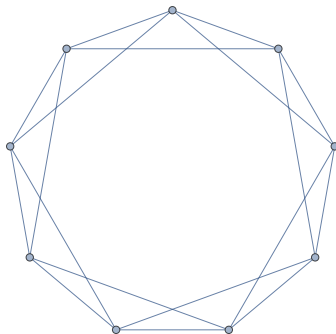


The example generalizes to a whole group of **circulant** machines on n states with diagram $C(n; 1, 2)$.

These machines are based on circulant graphs:

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

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



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 minimal.

Exercise

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

Exercise

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

In general, we have some nondeterministic automaton $\mathcal{A} = \langle Q, \Sigma, \tau; Q, Q \rangle$ and we want to show that $\text{pow}(\mathcal{A})$ has size 2^n (or some such).

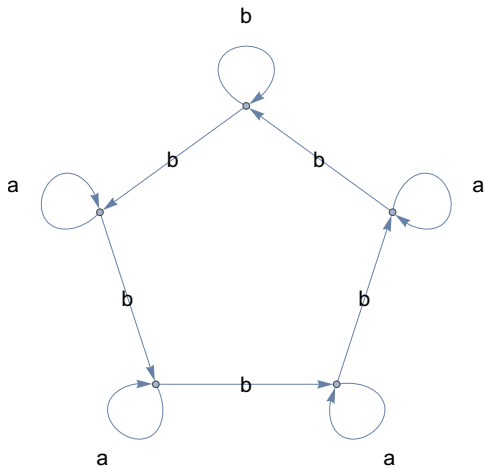
This boils down to the following. Let $\delta_a : \mathfrak{P}(Q) \rightarrow \mathfrak{P}(Q)$ be the function induced by the relation τ , as in the determinization algorithm:

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

By composition we get the transformation semigroup, functions δ_x for all $x \in \Sigma^*$. For each $P \subseteq Q$ we need to find a **witness** $x \in \Sigma^*$ such that

$$\delta_x(Q) = P$$

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



δ_a kill 0

δ_b sticky rotate

Note that $Q = \{0, 1, \dots, n-1\}$ is reachable from any $P \neq \emptyset$.

If we can concoct the operation “rotate” we are done.

Case 1: $0 \notin P$ δ_b works

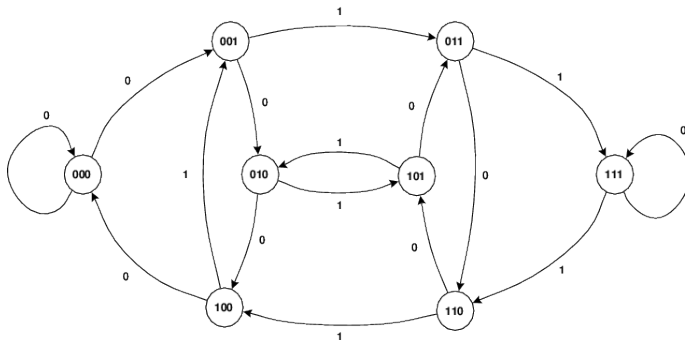
Case 2: $0 \in P$

If $n-1 \in P$: δ_b works

If $n-1 \notin P$: ????

Start with a binary de Bruijn semiautomaton where both δ_0 and δ_1 are permutations (so that the transformation semigroup is actually a subgroup of the symmetric group on Q). Now flip the label of the loop at 0 .

For $I = F = Q$, full blow-up occurs.



The loop case I can prove. But here is an open problem:

One can show that the number of permutation labelings in the binary de Bruijn graph of rank k is $2^{2^{k-1}}$. Flipping the label of an arbitrary edge will produce full blow-up in exactly half of the cases.

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

Verified experimentally up to $k = 5$ (on Blacklight at PSC, rest in peace).
There are 8,388,608 machines to check, ignoring symmetries.

Many finite state machine algorithms 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.

Alas, we cannot predict how big the deterministic machine will be: the following problem is PSPACE-complete.

Problem: **Power Automaton Size**

Instance: A nondeterministic machine \mathcal{A} , a bound B .

Question: Is the size of the power automaton of \mathcal{A} bounded by B ?

In general one would like to have estimates for the size of a machine constructed by a certain algorithm, as a function of the size of the input machines.

Upper bounds are usually easy to get, they are quite obvious from the constructions.

But lower bounds are tricky; one needs to construct particular inputs that make the algorithm perform poorly.

Note that this is just worst-case analysis, very little is currently known about average-case behavior.

	DFA	NFA
intersection	mn	mn
union	mn	$m + n$
concatenation	$(m - 1)2^n - 1$	$m + n$
Kleene star	$3 \cdot 2^{n-2}$	$n + 1$
reversal	2^n	n
complement	n	2^n

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

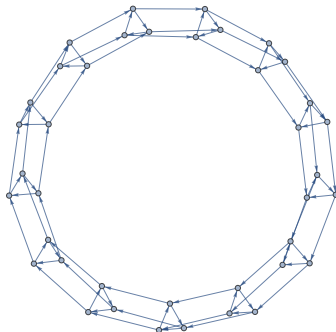
Note that we are only dealing the state complexity, not transition complexity.

Let

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

be the “mod-counter” language. Clearly $K_{a,m}$ has state complexity m .

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



Problem: **Emptiness Problem**
Instance: A regular language L .
Question: Is L empty?

Problem: **Finiteness Problem**
Instance: A regular language L .
Question: Is L finite?

Problem: **Universality Problem**
Instance: A regular language L .
Question: Is $L = \Sigma^*$?

Note that these problems are all rather ambiguous as stated: exactly how is the the input (a regular language) given?

As far as decidability is concerned there is no difference between DFAs and NFAs: we can simply convert the NFA.

But the determinization may be exponential, so efficiency becomes a problem.

- Emptiness and Finiteness are easily polynomial time for DFAs and NFAs.
- Universality is polynomial time for DFAs but PSPACE-complete for NFAs.

Problem: **Equality Problem**

Instance: Two regular languages L_1 and L_2 .

Question: Is L_1 equal to L_2 ?

Problem: **Inclusion Problem**

Instance: Two regular languages L_1 and L_2 .

Question: Is L_1 a subset of L_2 ?

- Equality and Inclusion are polynomial time for DFAs.
- Both problems are PSPACE-complete for NFAs.

1 Transformation Semigroups

2 State Complexity

3 **Alternating Automata**

A DFA is just the formalization of a perfectly practical algorithm: scan a string letter by letter, update your state via table lookup, decide acceptance on the basis of the last state. Runs in linear time and constant space with very good constants.

By contrast, an NFA is *prima facie* an abstraction: there may be exponentially many possible runs on a single input, and acceptance is determined by an existential quantification: is there a run

The reason this abstract model is still hugely important for practical algorithms is that acceptance testing for NFAs is still linear time, albeit with worse constants. On the other hand, other operations such as union are actually easier for NFAs.

Proposition

For any DFA \mathcal{A} and any input string x we can test in time linear in $|x|$ whether \mathcal{A} accepts x , with very small constants.

```
// deterministic acceptance testing
```

```
   $p = q_0$ 
```

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

```
     $p = \delta(p, a)$ 
```

```
  return  $p \in F$ 
```

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

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

```
// nondeterministic acceptance testing
```

```
   $P = I$ 
```

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

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

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

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

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

So here is a wild idea:

Question: Is there a useful notion of acceptance based on “for all runs such that such and such”?

One problem is whether these “universal” automata are more powerful than ordinary FSMs. As we will see, we still only get regular languages.

But this raises the question of how the state complexities compare: recall that nondeterministic FSMs can be exponentially smaller than their deterministic counterparts—one of the reasons they are attractive in practical pattern matching applications.

How would one formally define a type of FSM $\mathcal{A} = \langle Q, \Sigma, \delta; I, F \rangle$ where acceptance means all runs have a certain property?

The underlying transition system $\langle Q, \Sigma, \delta \rangle$ will be unaffected, it is still an edge-labeled digraph.

The acceptance condition now reads:

\mathcal{A} accepts x if all runs of \mathcal{A} on x starting at I end in F .

Let's call these machines \forall FA.

Read: for-all-FA. It's tempting to call them “universal FA”, but that collides with the standard use where universal means “accepting all inputs.”

By the same token, a NFA would be a \exists FA, a there-exists-FA.

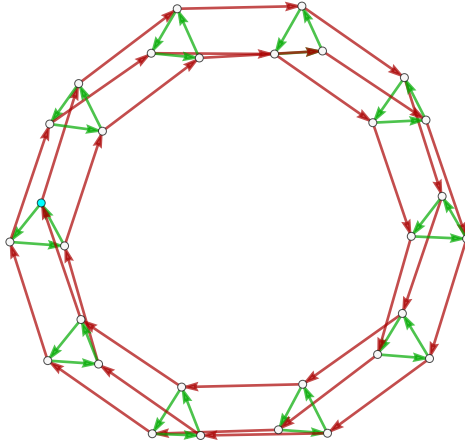
As an example consider the mod counter languages

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

with state complexity m . For the union $K_{0,m} \cup K_{1,n}$ we have a natural NFA of size $m + n$. However, for the intersection $K_{0,m} \cap K_{1,n}$ we only have a product machine that has size mn .

More importantly, note that nondeterminism does not seem to help with intersection: there is no obvious way to construct a smaller NFA for $K_{0,m} \cap K_{1,n}$.

This happens on a number of occasions: there are regular languages where nondeterminism seems utterly useless. The natural construction of the machine is automatically deterministic.



But we can build a \forall FAs of size just $m + n$: take the disjoint union and declare the acceptance condition to be universal.

What is really going on here?

Let's assume that Q_1 and Q_2 are disjoint. Starting at $\{q_{01}, q_{02}\}$ we update both components. So after a while we are in state

$$\{p, q\} \quad p \in Q_1, q \in Q_2$$

In the end we accept iff $p \in F_1$ and $q \in F_2$.

This is really no different from a product construction, we just don't spell out all the product states explicitly: a perfect example of a **succinct representation**.

Choosing clever representations is often critically important.

For example, acceptance testing for a \forall FSA is basically the same as for an NFA: we keep track of the set of states $\delta(I, x) \subseteq Q$ reachable under some input and simply change the notion of acceptance: this time we want $\delta(I, x) \subseteq F$.

Hence, if some word x crashes all possible computations so that $\delta(I, x) = \emptyset$, then x is accepted. This may sound weird, but it's perfectly fine.

Likewise we can modify the Rabin-Scott construction that builds an equivalent DFA: as before calculate the (reachable part of the full) powerset and adjust the notion of final state:

$$F' = \{ P \subseteq Q \mid P \subseteq F \}$$

There is almost no difference.

A mathematician is a person who can find analogies between theorems; a better mathematician is one who can see analogies between proofs and the best mathematician can notice analogies between theories. One can imagine that the ultimate mathematician is one who can see analogies between analogies.

S. Banach

We can think of the transitions in a NFA as being disjunctions:

$$\delta(p, a) = q_1 \vee q_2$$

We can arbitrarily pick q_1 or q_2 to continue. Similarly, in a \forall FA, we are dealing with conjunctions:

$$\delta(p, a) = q_1 \wedge q_2$$

meaning: We must continue both at q_1 and at q_2 . So how about

$$\delta(p, a) = (q_1 \vee q_2) \wedge (q_3 \vee q_4)$$

Or perhaps

$$\delta(p, a) = (q_1 \vee \neg q_2) \wedge q_3$$

Does this make any sense?

Think of threads: both \wedge and \vee correspond to launching multiple threads. The difference is only in how we interpret the results returned from each of the threads.

For \neg there is only one thread, and we flip the answer bit.

In other words, a “Boolean” automaton produces a computation tree very much like a plain NFA. But the acceptance condition is a bit more involved.

For historical reasons, these devices are called **alternating automata**.

In an **alternating finite automaton (AFA)** we admit transitions of the form

$$\delta(q, a) = \varphi(q_1, q_2, \dots, q_n)$$

where φ is an arbitrary Boolean formula over $Q = \{q_1, q_2, \dots, q_n\}$, even one containing negations.

How would such a machine compute? Initially we are in “state”

$$q_{01} \vee q_{02} \vee \dots \vee q_{0k}$$

the disjunction of all the initial states.

Suppose we are in state Φ , some Boolean formula over Q . Then under input a the next state is defined by substituting formulae for the variables:

$$\Phi[q_1 \mapsto \delta(q_1, a), \dots, q_n \mapsto \delta(q_n, a)]$$

The substitutions are supposed to be carried out in parallel, so each variable $q \in Q$ is replaced by $\delta(q, a)$, yielding a new Boolean formula. In the end we accept if

$$\Phi[F \mapsto 1, \overline{F} \mapsto 0] = 1$$

Meaning: replace all variables in F by true, and all variables in \overline{F} by false.

Exercise

Verify that for both NFA and \forall FA this definition behaves as expected.

The name “alternating automaton” may sound a bit strange.

The original paper by Chandra, Kozen and Stockmeyer that introduced these machines in 1981 showed that one can eliminate negation without reducing the class of languages.

One can then think of alternating between existential states (at least one spawned process must succeed) and universal states (all spawned processes must succeed).

In a moment, we will apply an analogous construction to Turing machines.

Theorem

Alternating automata accept only regular languages.

Proof.

Let $\text{Bool}(Q)$ be the collection of all Boolean formulae with variables in Q and $\text{Bool}_0(Q)$ a subset where one representative is chosen in each class of equivalent formulae (say, the length-lex first in DNF) and consider the corresponding normalization map $\nu : \text{Bool}(Q) \rightarrow \text{Bool}_0(Q)$.

We can build an equivalent DFA over the state set $\text{Bool}_0(Q)$ of state complexity at most 2^{2^n} .

- The initial state is $\nu(\bigvee_{q \in I} q)$.
- Transitions are $\Delta(\mathbf{p}, a) = \nu(\mathbf{p} \mapsto \delta(\mathbf{p}, a))$.
- The final states are $\{ \varphi \in \text{Bool}_0(Q) \mid \varphi[F \mapsto 1, \overline{F} \mapsto 0] = 1 \}$.

It is easy to see that the new, ordinary DFA is equivalent to the given, alternating one.



But note that the cost of eliminating alternation is potentially doubly exponential, significantly worse than for determinization (corresponding to logical-or only).

Because an AFA can be much, much smaller than the minimal DFA. In fact, the 2^{2^n} bound is tight: there are AFAs on n states where the minimal equivalent DFA is doubly exponential in n .

So we have a succinct representation for a regular language, but one that still behaves reasonably well under the usual algorithms. Avoiding the standard DFA representation is often critical for feasibility: in reality we cannot actually construct the full DFA in many cases. Laziness is a good idea in this context.

BTW, this is even true in pattern matching: DFAs should be avoided unless they are absolutely necessary (because the pattern contains a negation).