A Subset of the URM Language; FA and NFA

0.0.1 Definition. If M is a FA, then its $L(M)$ is called the **regular set** associated with M, or even the **regular language** *recognised/accepted* (**decided**, actually) by M. □

This Note continues from where Note #9 left but we will present first a few more simple examples of automata* that decide /recognise some given set of strings over some alphabet.

*Plural of automaton.

0.1. Examples

0.1.1 Example. We want to specify (to "program"!) an automaton M over $\Sigma = \{0, 1\}$, such that $L(M) = \{0^n 1 : n \geq 0\}$.

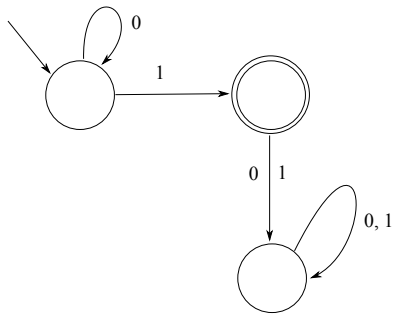
We recall that, for any string x, $x^0 =^{Def} \lambda$, while

$$x^{n+1} \overset{Def}{=} x^n * x \overset{\text{induction!}}{=} \overbrace{x * x * \ldots * x}^{n+1 \text{ copies of } x}$$

where I denoted concatenation by $*$. Thus the strings in $\{0^n 1 : n \geq 0\}$ are

$$1, 01, 001, 0001, 00001, \ldots \tag{1}$$

We readily see that the following automaton's **only accepting paths** will follow zero or more times the "loop" labeled 0 (attached to the start state), and then the edge labeled 1 to end up with an accepting state.



The state at the very bottom is a trap state. What is the need for it?

Well, the FA must be fully specified, so I am obliged to say what the accepting state does when it sees *one or the other legal input*.

And remember: Accepting states do NOT stop the machine! Any state stops the machine IFF it has just scanned *eof*.

A new thing we learnt in the above example is that in depicting an automaton as a graph we do *not* necessarily need to *name* the states!

As in all mathematical arguments, we will of course assign names to objects (in particular to states) **if we need to refer to them** in the course of the argument —it is convenient to refer to them by name!

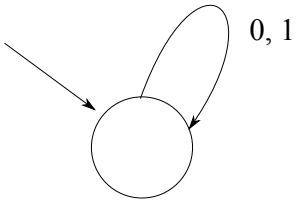
The reader should also note the use of *two shorthand notations in labeling*:

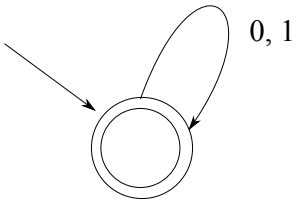
One, we used two labels on the vertical down-pointing edge.

This abbreviates the use of *two* edges going from the accepting to the trap state, one labeled 0, the other 1.

We could also have used the label "0, 1" both at the left or right of the arrow, "," serving as a separator. This latter notational convention was used in labeling the loop attached to the trap state. □

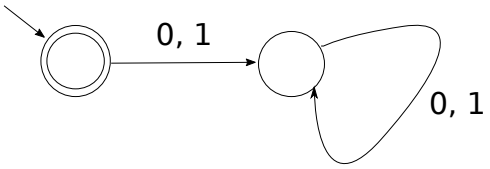
0.1.2 Example. The two FAs below, each over the input alphabet $\{0, 1\}$, accept the languages $\emptyset$ (the top one) and $\{0, 1\}^*$ (the bottom one).





□

0.1.3 Example. The FA below over the input alphabet $\{0, 1\}$ accepts the language $\{\lambda\}$.



Indeed, we saw in Notes #9 that making the start (initial) state also accepting we do accept λ. Moreover, the FA above accepts nothing else since any input symbol leads to the rejecting *trap* state. □

0.2. Some Closure Properties of Regular Languages

0.2.1 Theorem. *The set of all regular languages over an alphabet Σ is closed under complement. That is, if $L \subseteq \Sigma^*$ is regular, then so is $\overline{L} =^{Def} \Sigma^* - L$.*

Proof. Let $L = L(M)$ for some FA M over input alphabet Σ and state alphabet Q. Moreover, let $F \subseteq Q$ be the set of accepting states of M.

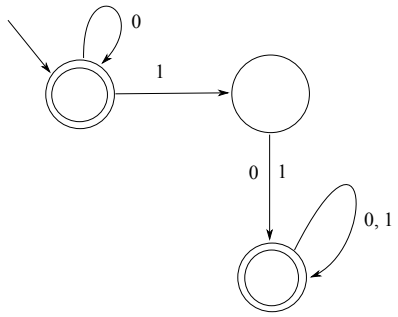
We need a FA that recognises/decides $\overline{L}$.

Trivially, *we want to swap the "yes" (accepting state) and "no" (rejecting state) behaviour of M*, changing nothing else.

Thus, $\overline{L} = L(\widetilde{M})$, where the FA $\widetilde{M}$ is the same as M, except that $\widetilde{M}$'s set of accepting states is $Q - F$. □

What makes the above proof tick is that FA are "**total**": Every input string will be scanned all the way to its *eof*. Only the yes/no decision changes.

0.2.2 Example. The automaton that accepts the complement of the language in Example 0.1.1 is found without comment below, just following the construction of the $L(M)$ complement for some FA M, given above.



□

0.2.3 Theorem. *The set of all regular languages over an alphabet Σ is closed under union. That is, if $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ are regular, then so is $L \cup L'$.*

Proof. This proof will *wait* until after the introduction of NFA which make the proof much easier! □

0.2.4 Corollary. *The set of all regular languages over an alphabet Σ is closed under intersection. That is, if $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma^*$ are regular, then so is $L \cap L'$.*

Proof. $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$. □

0.3. Proving Negative Results for FA; Pumping Lemma

Lecture #20, Nov. 25

Is there a FA M such that $L(M) = \{0^n 1^n : n \geq 0\}$?

How can we tell?

Surely, not by trying each FA (*infinitely many*) out there as a possible fit for this language!

The following theorem, known as the *pumping lemma* can be used to prove "negative" results such as: There is no FA M such as $L(M) = \{0^n 1^n : n \geq 0\}$. In short, the language $\{0^n 1^n : n \geq 0\}$ is **not regular**.

0.3.1 Theorem. (Pumping Lemma) *If the language S is regular, i.e., $S = L(M)$ for some FA M, then there is a constant C that we will refer to as* a pumping constant *such that for any string $x \in S$, if $|x| \geq C$, then we can decompose it as $x = uvw$ so that*

(1) $v \neq \lambda$

(2) $uv^i w \in S$, *for all* $i \geq 0$

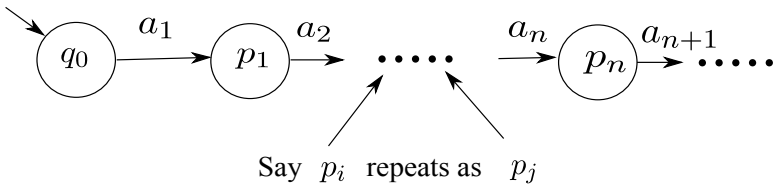
and

(3) $|uv| \leq C$.

A pumping constant is *not* uniquely determined by S.

Proof. So, let $S = L(M)$ for some FA M of n states. We will show that if we take $C = n$[†] this will work.

Let then $x = a_1 a_2 \cdots a_n \cdots a_m$ be a string of S. As chosen, it satisfies $|x| \geq C$. An accepting computation path of M with input x looks like this:



where $p_1, p_2, \ldots$ denotes a (notationally) convenient renaming[‡] of the states visited after q_0 in the computation.

[†]You see why C is not unique, since for any S that is an $L(M)$ we can have infinitely many different M that accept S. Can we not?

[‡]Why rename? What is wrong with $q_1, q_2, \ldots$? Well, the set Q is *given* as something like $\{q_0, q_1, q_2, q_3, \ldots\}$ using some arbitrary fixed enumeration order without repetition for its members. Now, it would be wrong to expect that the arbitrary input x caused the FA to walk precisely along q_1, q_2, q_3, etc., *after* it saw the first symbol of x.

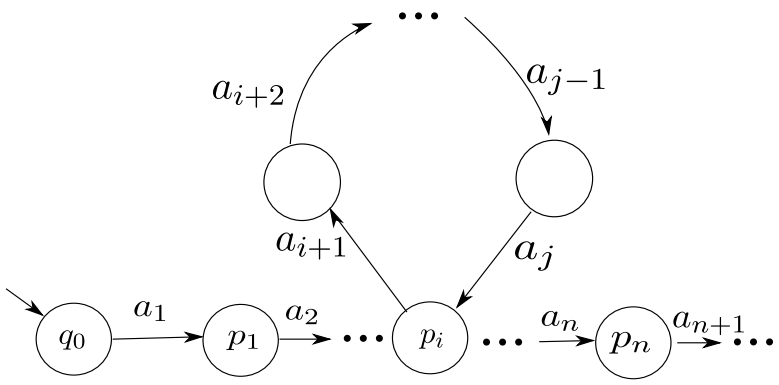
In the sequence

$$q_0, p_1, p_2, \ldots, p_n$$

we have named $n+1$ states, while we only have n states in the FA's "Q".

Thus, *at least two names "p_i and p_j" refer to the same state* "q_r" —as we say, two states repeat.

We may redraw the computation above as follows taking *without loss of generality* that $p_i = p_j$ indicating the repeating state $p_i = p_j$:



We can now partition x into u, v and w parts from the picture above: We set

$$u = a_1 a_2 \ldots a_i$$

$$v = a_{i+1} a_{i+2} \ldots a_j$$

and

$$w = a_{j+1} a_{j+2} \ldots a_m$$

Note that

(1) $v \neq \lambda$, since *there is at least one edge (labelled a_{i+1})* emanating from p_i on the sub-path that connects this state to the (identical) state p_j. *In short a_{i+1} is part of v.*

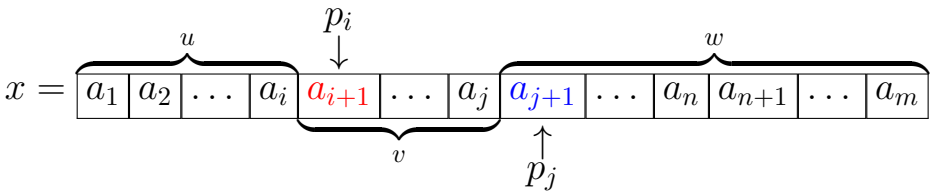
(2) We may utilize the loop v zero or more times (along with u in the front and w at the tail) to always obtain a NEW accepting path. Thus, all of uv^iw belong to $L(M)$ —i.e., S.

(3) *Since $|uv| = j \leq n$, we have also verified that $|uv| \leq C$.* □

The repeating pair p_i, p_j may occur anywhere between q_0 and p_n. A different "graphical proof" is common in the literature.

Let $x = a_1 a_2 \dots a_m$ as above.

Below we show the *original* x as an input stream array $x = a_1 \dots a_{i+1} \dots a_j a_{j+1} \dots a_n \dots a_m$, where the repeating $p_i = p_j$ is shown.



Observe:

1. By determinism, the subcomputation that starts at symbol a_{j+1} (blue) —while in state p_j ($= p_i$)— will end at the *eof* after consuming the string w and will be uniquely at (accepting) state p_m.

2. After consuming the prefix $a_1 \dots a_i$ of x the FA is uniquely at state p_i.

3. By determinism, the subcomputation that starts at symbol a_{i+1} (red) in state p_i, will consume v and end at p_j —uniquely, today, tomorrow and in 10^{350000} years from now— ready to process a_{j+1} (blue).

Thus, all of $uv, uvvw, uvvvw, \dots, uv^n w, \dots$ are in $L(M)$. Of course, so is $x = uvw$ (given).

0.3.2 Example. The language over $\{0,1\}$ given as $L = \{0^n1^n : n \geq 0\}$ is not regular.

Suppose it is. Then the pumping lemma holds for L, so let C be an appropriate pumping constant and consider the string $x = 0^C1^C$ of L. We can then decompose x as uvw with $|uv| \leq C$ so that we can "pump" $v \neq \lambda$ as much as we like and the obtained uv^iw will all be in L.

We will prove the statement in red false, so we cannot pump; but then L cannot be regular![§]

"*The red statement*" is false due to the observations:

1. By $|uv| \leq C$, uv (and hence v) lie entirely in the 0^C-part of the chosen $x = 0^C1^C$.
2. So, if we *pump down* —or above with $i \geq 2$ (i.e., use v^0 or $v^i, i \geq 2$) we obtain $uw \in L$ or $uv^iw \in L, i \geq 2$.

But $uw = 0^K1^C$ where $K < C$ since $|v| \geq 1$. But such unbalanced 0-1 strings cannot be in L, by specification, so we contradicted the pumping lemma. □

[§]All sufficiently long strings of regular languages can be pumped by 0.3.1 and stay in L.

All proofs by Pumping Lemma 0.3.1 are by contradiction and they prove *non acceptability* by *any* FA (or, equivalently, NFA to be introduced in Section 0.4.1).

0.3.3 Example. *We introduced FA as special URMs that cannot write.*

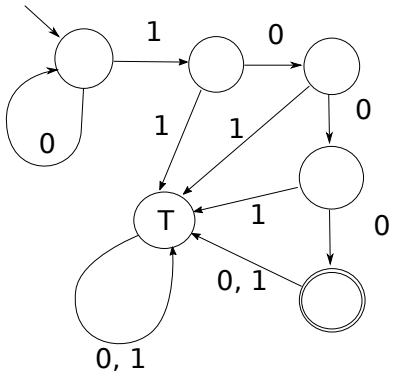
Is it then an immediate conclusion that they cannot compute functions?

Not at all! Such a general conclusion is false!

For example, we can agree that by "*compute* $f(x)$" we mean " decide the graph $y = f(x)$".

For example, we can "compute" $\lambda x.3$ by accepting all strings, but no others, of the form 0^n1000 over the alphabet $\{0, 1\}$.

That is, we use 1 as a separator between input $n \geq 0$ (depicted as 0^n) and output 3 (depicted as 000), then the following FA **decides** (*accepts/recognises*) the language $L = \{0^n1000 : n \geq 0\}$.



□

0.3.4 Example. FA cannot compute $\lambda x.x+1$.

"*Surely*", you say, "*how can they add 1 if they cannot do arithmetic or write anything at all?*"

Wrong reason!

Again, how about deciding the *"graph"-language* over $A = \{0, 1\}$, given by $T = \{0^n 10^{n+1} : n \geq 0\}$?

Here "0^n" represents input n, "0^{n+1}" *represents output* $n+1$ and 1 is a separator as in the previous example.

▶ Alas, no FA can do this.

Say T is FA-decidable, and let C be an appropriate pumping constant. Choose $x = 0^C 10^{C+1}$. Splitting x as uvw with $|uv| \leq C$ *we see that* 1 *is to the right of* v.
v is all zeros.
Thus, uw (using v^0) is not in T since the relation between the 0s to the left and those to the right of 1 is destroyed. This contradicts the assumption that T is FA-decidable. □

0.3.5 Exercise. Indeed FA cannot even compute the identity function, $\lambda x.x$, as it should be clear from the proof in 0.3.2. Adapt that proof to show the graph language for $\lambda x.x$, namely, $\{0^n 10^n : n \geq 0\}$ is not regular. □

0.3.6 Example. The set over the alphabet $\{0\}$ given by $P = \{0^q : q \text{ is a prime number}\}$ is not FA-decidable.

Assume the contrary, and let C be an appropriate pumping constant. Let $Q \geq C$ be prime.

We show that considering the string $x = 0^Q$ will lead us to a contradiction. Well, as x is longer than C, let us write —according to 0.3.1— $x = uvw$. By PL, we must have that all numbers $|u| + i|v| + |w|$, for $i \geq 0$ *are prime*. These numbers have the form

$$ai + b \tag{1}$$

where $a = |v| \geq 1$ and $b = |u| + |w|$. Can REALLY *all these numbers in (1)* (for all i) *be prime*?

Here is why not, and hence our contradiction. We consider cases:

- Case where $b = 0$. This is impossible, since the numbers in (1) now have the form ai. But, e.g., $a4$ is not prime.
- Case where $b > 0$. We have Subcases!
  - Subcase $b > 1$. Then taking $i = b$, one of the numbers of the form (1) is $(a+1)b$. But $(a+1)b$ is not prime (recall that $a + 1 \geq 2$ since $a \geq 1$).
  - Subcase $b = 1$. Then take $i = 2 + a$ to obtain the number (of type (1)) $a(2+a)+1 = a^2+2a+1 = (a+1)^2$. But this is not prime! □

The preceding shows that *we can have a set that is sufficiently complex and thus fails to be FA-decidable even over a single-symbol alphabet*. Here is another such case.

0.3.7 Example. Consider $Q = \{0^{n^2} : n \geq 0\}$ over the alphabet $A = \{0\}$. It will *not* come as a surprise that Q is not FA-decidable.

For suppose it is. Then, if C is an appropriate pumping constant, consider $x = 0^{C^2}$.

- Clearly, $x \in Q$ and is long enough.

So, split it as $x = uvw$ with $|uv| \leq C$ and $v \neq \lambda$.

Now, by 0.3.1,

$$uvvw \in Q \tag{1}$$

But

$$C^2 = |uvw| < |uvvw| \leq |uvw| + |uv| \leq C^2 + C$$
$$< C^2 + 2C + 1 = (C+1)^2$$

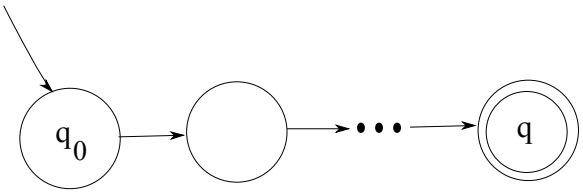
Thus, the number $|uvvw|$ is NOT a perfect square *being between two successive ones*.

But this will not do, because by (1), for some n, we have $uvvw = 0^{n^2}$ and thus $|uvvw| = n^2$ —a perfect square after all! □

0.4. Nondeterministic Finite Automata

The FA formalism provides us with tools to finitely define certain languages:

Such a language —defined as an $L(M)$ over some alphabet A, for some FA M— contains a string x iff there is an *accepting path* —within the FA— *whose labels from left to right form* x.



The computation above, that is, the path labeled x within the FA, is uniquely determined by x since the automaton is deterministic.

Much is to be gained in *theoretical and practical flexibility* if we **relax both** "deterministic" requirements NFA 1) and NFA 2) below

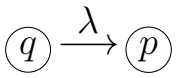
NFA 1) Every state is defined on all inputs from the input alphabet (totaleness)

NFA 2) No state has two different responses (i.e., does not send the process *to either of two different* states) *for the same input*.

AND moreover we profit —*theoretically and practically*— from *ADDING* the feature

NFA 3) The automaton can have *empty-moves*, that is, *λ-moves*, *meaning* it can go from state q to state p *WITHOUT CONSUMING ANY INPUT*.

An *empty move* from q to p is depicted in the flow diagram as:



0.4.1 Definition. (NFA) A so relaxed FA —that is also *augmented* by the feature "NFA 3)" above— is called *Non Deterministic Finite Automaton*, in short, *NFA*.

An NFA M accepts a string x iff there is a *path from its start state* (generically depicted as) "q_0" *to some accepting state* p whose edge-labels concatenated from q_0 toward p in order *form the string* x.

Of course empty moves do not contribute to the path name!

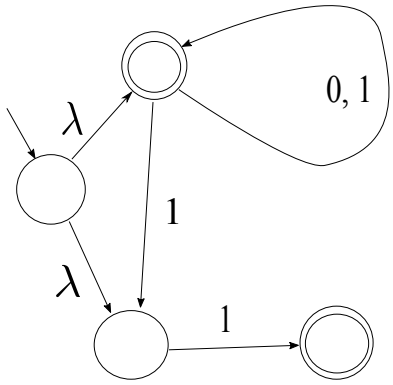
IMPORTANT! *Every FA is also an NFA* —but NOT vice versa— since the enhancements in *NFA 1) – NFA 3)* above are NOT compulsory! □

0.4.2 Example. The displayed flow diagram below, over the alphabet $\{0, 1\}$, incorporates all the liberties in notation and conventions introduced in Definition 0.4.1 and the items NFA 1) – NFA 3) preceding the definition.

We have two λ moves, and the string "1" can be accepted in two distinct ways: One is to follow the top λ move, and then go once around the loop, consuming input 1. The other is to follow the bottom λ move, and then follow the transition labeled 1 to the accepting state at the bottom (reading 1 in the process).

Folklore jargon —not based on science or theory— will have us speak of *guessing* when we describe what the diagram does with an input.

For example, to accept the input 00 one would say that *the NFA guesses* that it should follow the upper λ, and then it would go twice around the top loop, on input 0 in each case.



This diagram is an example of a *nondeterministic finite automaton*, or NFA;

- it has λ moves,
- its transition relation —as depicted by the arrows— is not a function (e.g., the top accepting state has *two distinct responses on input 1*),
- nor is it total.

  For example, the bottom accepting state is not defined on any input; nor is the start state: *λ is not an input!* □

Lecture #21, Nov.30

Returning to the issue of *guessing*, we emphasize that this use of this term is an unfortunate habit in the literature.

Nobody and Nothing guesses Anything!

A NFA simply provides the *mathematical framework* within which *we* can formulate and verify an *existential MATHEMATICAL statement* of the type

for a given input x, an accepting path exists (1)

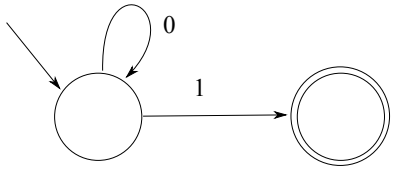
Given an acceptable input, *the NFA does NOT actually guess* "correct" moves (from among a set of choices), either in a hidden manner (*consulting the Oracle in Delphi, for example!*), or in an explicit computational manner (e.g., parallelism, backtracking) toward *finding* an accepting path for said x.

Simply, the NFA formalism allows *us* to *state*, and provides tools so that we can *verify*, the statement (1) above by verifying an accepting path exists! (0.4.1)

This is analogous with the fact that *the language of logic allows us to state statements such as* $(\exists y)\mathscr{F}(y,x)$, and offers tools to us to prove them.

In the case of NFA, an independent agent, which *could be ourselves* or a FA —YES, we will see that *every NFA can be simulated by some FA!*— can effect the verification that indeed an accepting path labeled x exists.

0.4.3 Example. The following is a NFA but not a FA (why? Compare with 0.1.1). It decides the language $\{0^n1 : n \geq 0\}$.

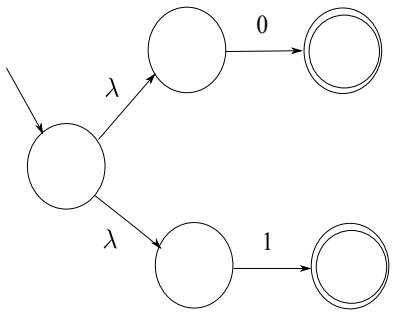


□

0.4.4 Example. NFA are much easier to construct than FA, partly because of the convenience of the λ moves, and the ability to "guess" (cf. earlier discussion about "guessing").

Also, partly due to *lack of concern for totalness*: we *do not have to worry about "installing" a trap state*.

For example, the following NFA over $A = \{0, 1\}$ decides/recognises just its alphabet A and nothing else as we can trivially see that there are just two accepting paths: one named "0" and one named "1".



□

0.5. From FA to NFA and Back

We noted earlier that any FA is a NFA (Def. 0.4.1), thus *the NFA are at least as powerful as the FA*.

They can do all that the deterministic model can do.

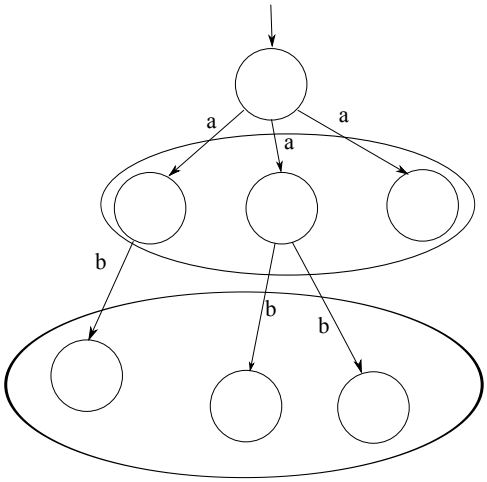
It is *a bit of a surprise* that the opposite is also true: *For every NFA M we can* construct *a FA N, such that $L(M) = L(N)$.*

- Thus, in the case of these very simple machines, non-determinism ("guessing") buys *ONLY convenience*, but not real power.

How does one simulate a NFA on an input x?

The most straightforward idea is to trace *all possible paths* labeled x (due to nondeterminism they may be more than one —or none at all) *in parallel* and accept iff one (*or more*) of those is accepting.

The principle of this idea is illustrated below.



Say, the input to the NFA M is $x = ab\ldots$ Suppose that a leads the start state —which is at "level 0"— to three states; we draw all three. These are at level 1.

We repeat for *each state at level 1* on input b:

Say, for the sake of discussion, that, of the three states at level 1, *the first leads to one state* on input b, *the second leads to two* and *the third leads to none*.

We draw these three states obtained on input b; they are at level 2. Etc.

An FA can keep track of all the states at the various levels since **they can be no more than the totality of states of the NFA M!**

The amount of information at each level is *independent of the input size* —i.e., it is a constant— and moreover can be coded as a single FA-state (*depicted in the figure by an ellipse*) that uses a *"compound" name*, *consisting of all the NFA state names at that level*.

This has led to the idea that *the simulating FA must have as states nodes whose names are* sets of *state names of the NFA*.

Clearly, for this construction, *state names are important*, through which we can keep track of and describe what we are doing.

Here are the details:

0.5.1 Definition. (a-successors) Let M be a NFA over an input alphabet Σ, q be a state, and $a \in \Sigma$.

A state p is *an a-successor* of q iff there is an edge from q to p, *labeled a*. □

In a NFA a-successors need not be unique, nor need to exist —for all pairs (q, a).

On the other hand, *in a FA they exist and are unique*.

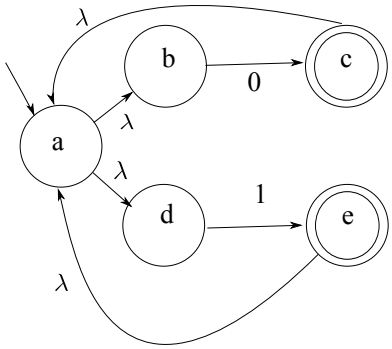
0.5.2 Definition. (λ-closure) Let M be a NFA with state-set Q and let $S \subseteq Q$. The λ-closure of S, denoted by $\lambda(S)$, is defined to be *the smallest set that includes* S but *also includes all* $q \in Q$, *such that* there is a path, named λ, *from some* $p \in S$ *to* q.

When we speak of the λ-closure of a state q, we mean that of the set $\{q\}$ and write $\lambda(q)$ rather than $\lambda(\{q\})$.

□

Note that a path named λ will have **all** its edges named λ since the concatenation of a sequence of strings is λ iff each string in the sequence is.

0.5.3 Example. Consider the NFA below.



We compute some λ-closures: $\lambda(a) = \{a, b, d\}$; $\lambda(c) = \{c, a, b, d\}$. □

0.5.4 Theorem. *Let M be a NFA with state set Q and input alphabet Σ. Then there is a FA N that has as state set a subset of $\mathcal{P}(Q)$ —the power set of Q— and* the same input alphabet as that of M.

N satisfies $L(M) = L(N)$.

We say that two automata M and N (whether both are FA or both are NFA, or we have one of each kind) are *equivalent* iff $L(M) = L(N)$.

Thus, the above says that *for any NFA there is an equivalent FA*.

In fact, this can be strengthened as the proof shows: We can *construct* the equivalent FA.

We show how *in the definition below, BEFORE we start the proof proper*.

0.5.5 Definition. (NFA to FA Construction)

- The start state of N is $\lambda(q_0)$, where q_0 is the start state of NFA M.
- A state of N is accepting **iff** its *name* contains at least one accepting state *name* of NFA M.
- Let S be a state of N and let $a \in \Sigma$. The unique a-successor of S in N is constructed as follows:

  (1) Construct the set of *all* a-successors in M of *all component-names* of S. Call T this set of a-successors.

  (2) Construct $\lambda(T)$; this is *the* a-successor of S in N.

□

As an illustration, we compute some *0-successors* in the FA constructed as above if the given NFA is that of Example 0.5.3.

(I) For state $\{a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the λ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor.

(II) For state $\{c, a, b, d\}$ step (1) yields $\{c\}$. Step (2) yields the λ-closure of $\{c\}$: The state $\{c, a, b, d\}$ is the 0-successor; that is, *the 0-edge loops back to where it started*: at state $\{c, a, b, d\}$.

Proof. Of 0.5.4.

With the FA N constructed as in 0.5.5 from the NFA M, we need to prove two things:

Direction1. $L(M) \subseteq L(N)$.

Direction2. $L(N) \subseteq L(M)$.

- *$L(M) \subseteq L(N)$ direction*:

  Let

  $$x = a_1 a_2 \cdots a_n \in L(M) \tag{1}$$

  $$\text{Prove that } x \in L(N) \tag{2}$$

  Without loss of generality, we have an accepting path in M that is labeled as follows:

  $$x = \lambda^{j_1} a_1 \lambda^{j_2} a_2 \lambda^{j_3} a_3 \cdots \lambda^{j_n} a_n \lambda^{j_{n+1}} \tag{3}$$

  where each λ^{j_i} depicts $j_i \geq 0$ consecutive path edges, each labeled λ, *where $j_i = 0$ in this context means that the j_i group has no λ-moves*.

An accepting path for the exact string —λ and all— in (3) is the zig-zag path depicted in Fig. 2 below.

To prove (2) we need a path in FA N *the edges of which are labeled by the a_i in $x = a_1 a_2 \ldots a_n$ in the indicated order*, while the nodes are states of N with the first node being the initial one, and that last one is an accepting state.

Here is how to do this:

▶ It suffices to show that for each level $i = 0, 1, 2, \ldots$, the N-path *of N-states and labelled edges*, consists of the indicated ellipses in Fig. 2 —and partially shown in Fig. 1— which contain in their name the enclosed "*horizontal*" M-nodes shown. *Why "suffices"? Read on!*

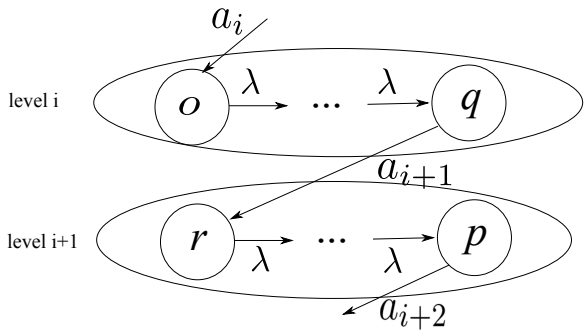


Figure 1: Idea for $L(M) \subseteq L(N)$ proof

▶ Regarding the above Figure, if we assume that at level i the elliptical N-node indeed includes all the indicated M-nodes in its name (*these are M-nodes from the M-computation!*), then so does the N-node at level $i + 1$ —that is, the *a_{i+1}-successor of the N-node at level i.*

This is so by Def. 0.5.5 since at level $i + 1$ we have the λ-closure of all a_{i+1}-successors —in M. But r is ONE such successor and thus all horizontal nodes will be in the name too! ◀

Now, clearly, $\lambda(q_0)$, THE start state of N —depicted by the level-0 ellipse in Fig. 2— will contain all horizontal nodes shown.

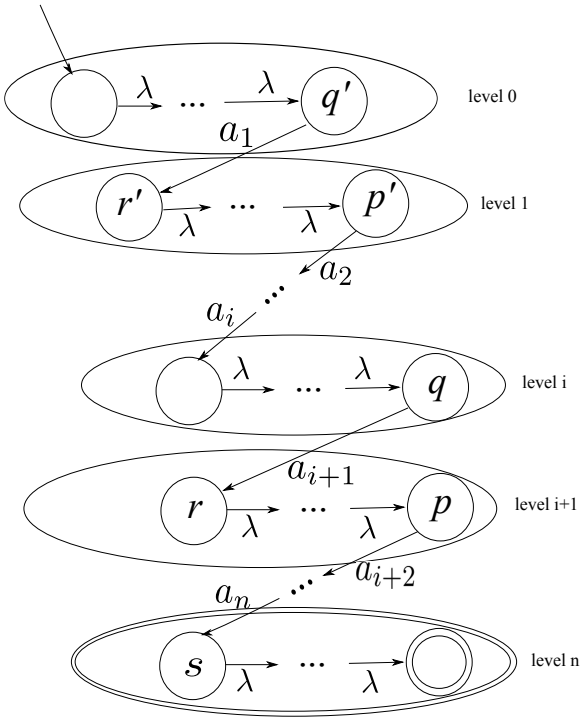


Figure 2: Equivalence of NFA and FA

Then —by (1) and (3) on p.36— the next ellipse (N-state) at *level 1* must contain r' (by Def. 0.5.5) and hence also *all horizontal nodes (as sub-names) shown at level 1*.

By the "Induction step" in the ▶ ◀-passage on p.37 all depicted elliptical nodes of N in Fig. 2 contain the nodes from M shown.

Clearly Fig. 2 depicts and FA N-computation that consumes $x = a_1a_2 \dots a_n$ and ends with an accepting N-state. It is *ACCEPTING* because it *contains in its name an accepting M-state*.

All in all: $x \in L(N)$. We proved (2) (p.36).

- *$L(N) \subseteq L(M)$ direction*:

  So let $x = a_1a_2 \dots a_n \in L(N)$ this time. (†)

  We will argue that also

  $$x \in L(M) \qquad (\ddagger)$$

  We will reuse Fig. 2.

  Observe that by (†) we have a path in N from the elliptical start-state to some *accepting elliptical N-state*.

We will construct an *accepting path for x in the NFA M*.

In our construction we start from the end (accepting N-state) and *proceed BACKWARDS towards the N start state*.

All the work is shown in Fig. 2 where now we retrace the M-path backwards.

OK. The accepting N state must have an *accepting NFA state* in its name.

▶ How did this get there?

- Either as an a_n-successor in NFA M of some name *found in the elliptical state immediately above*, or, more generally,
- It is at the end of a λ-path starting at an *a_n-successor in NFA M* —here named "*s*"— found in the last ellipse. *This general case is depicted in Fig. 2*.

To understand *how the construction propagates UPWARDS* (BACKWARDS) imagine that $a_n = a_{i+2}$.

Then the question is "whose a_{i+2}-successor (in M) is s? *Well, we named it p in Fig. 2*.

The next question is: "How did p get in the name of the ellipse at level $n - 1 = i + 1$?"

Well, as above,

– Either as an a_{i+1}-successor in NFA M of some name *found in the elliptical state immediately above* —at level $n-2=i$,

or, more generally,

– p is at the end of a λ-path starting at an a_{i+1}-*successor* r *in NFA* M found in the last ellipse. *This general case is depicted in Fig. 2.*

Continuing the construction like this we find that the presence of q' in the start state of N is either that it is the same as the state "q_0" of the NFA M, OR q' is connected to q_0 by a backwards λ-path, in general, as depicted in Fig. 2.

We have just constructed a path labelled

$$\lambda^{j_1}a_1\lambda^{j_2}a_2\lambda^{j_3}a_3\cdots\lambda^{j_n}a_n\lambda^{j_{n+1}} = a_1a_2a_3\cdots a_n$$

in the NFA M from its q_0 to some accepting state!

Thus $x \in L(M)$. □

In theory, to construct a FA for a given NFA we draw all the states of the latter —*named by all subsets of the state-set Q of the NFA*— and then determine the interconnections via edges, for each state-pair of the FA and each member of the input alphabet Σ.

In practice we may achieve significant economy of effort if we start building the FA "*from the start state down*": That is, starting with the start state (level 0) we determine *all* its (elliptical) a-successors, *for each* $a \in \Sigma$.

At the end of this step we will have drawn all states at "level 1".

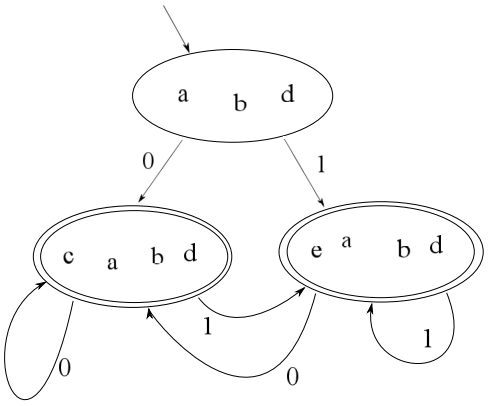
In the next step for each state at level 1, draw its a-successors, for each $a \in \Sigma$. And so on.

This sequence of steps *terminates* since there are only a finite number of states in the FA and *we cannot keep writing* new *ones*

Sooner or later we will stop introducing *new* states: edges will point *"back" to existing states*.

See the following example.

0.5.6 Example. We convert the NFA of 0.5.3 to a FA. See below, and review the above comment and the proof of 0.5.4, in particular the three bullets on p.34, to verify that the given is correct, and follows procedure.



You will notice the aforementioned economy of effort achieved by our process. We have only three states in the FA as opposed to the predicted 32 ($= 2^5$) of the proof of Theorem 0.5.4. *But what happened to the other states?* Why are they not listed by our procedure?

Because *OUR procedure* only constructs FA states that *are accessible FROM the start state via a computation path*.

These are the only ones that can possibly participate in an accepting path. The others are irrelevant to accepting computations —indeed to any computations that start with the start state— and can be omitted without affecting the set decided by the FA. □

0.5.7 Example. Suppose that we have converted a NFA M into a FA N.

Let *a be in the input alphabet*.

What is the a-successor of the state named $\emptyset$ *in* N?

Well, there are no states in $\emptyset$ *to start the deterministic a-successor process* of Def. 0.5.5!

So the set of successor states we get is empty; we are back to $\emptyset$.

Thus, the set of a-successors (*in* M) of states from $\emptyset$ is itself the empty set. In other words, the a-successor of $\emptyset$ *in* N is $\emptyset$. The edge labeled a loops back to it.

Therefore, in the context of the NFA-to-FA conversion, $\emptyset$ is a *trap state* in N. □