

Designing finite automata II

Problem: Design a DFA A such that $L(A)$ consists of all strings of \square and \diamond which are of length $3n$, for $n = 0, 1, 2, \dots$

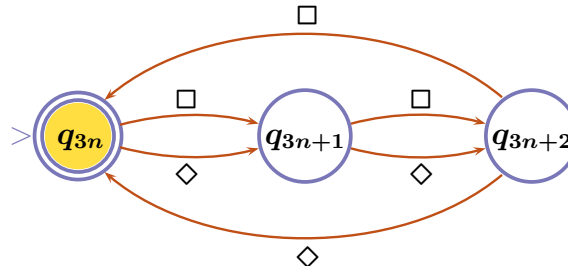
(1) Determine what to remember about the input string.

Assign a state to each of the possibilities: $q_{3n}, q_{3n+1}, q_{3n+2}$.

(2) Add the transitions telling how the possibilities rearrange,

select the initial state and the favourable states.

(3) Test the automaton.

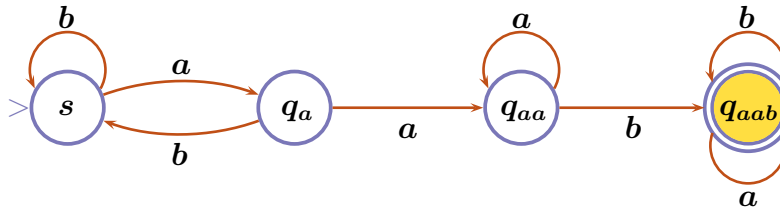


Pattern matching

Problem: Design a DFA A such that $L(A)$ consists of all strings of a 's and b 's which contain aab as a substring

There are four possibilities to remember:

- haven't seen any symbols of the pattern: (initial) state s
- have just seen an a : state q_a
- have just seen aa : state q_{aa} or
- have just seen the entire pattern aab : (favourable) state q_{aab} .



DFA as a theoretical model for programming

- unusual, different from conventional programming languages
- major concepts of imperative programming
(e.g., sequences, branching, loops) can be simulated
- elegant, simple to use, has a thoroughly developed theory
- all pattern matching (e.g., WWW search engines) can be described
- limited expressive power (the automaton is not capable of increasing any of its resources during computation)

What about 'subprogram calls' ?

Combining two automata

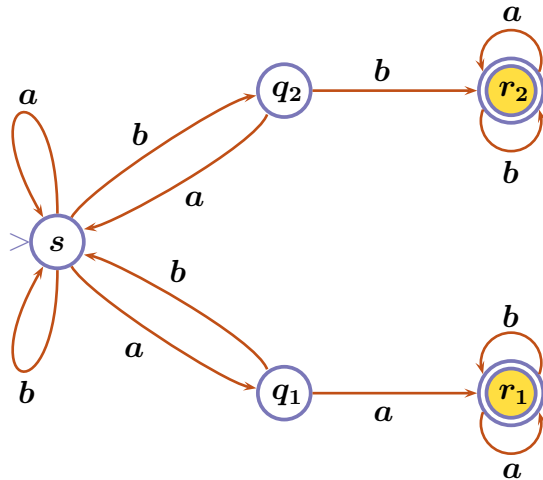
Problem: *Design an automaton that accepts all the strings that have either two consecutive a 's or two consecutive b 's*

- On page 24 (4), there is a DFA A_4 accepting all the strings that have two consecutive a 's. This automaton can easily be transformed into an automaton A'_4 that accepts all the strings that have two consecutive b 's.
- Since we already have 'programs' that solve the problem for aa 's and, respectively, bb 's, an obvious idea would be to design a 'program' that would just 'call' the given programs as 'subprograms.' A possible solution would be to 'combine' the initial states of A_4 and A'_4 .

Combining two automata (cont.)

Given: $L(A_4)$ = all words having two consecutive a 's
 $L(A'_4)$ = all words having two consecutive b 's

Needed: automaton A such that $L(A) = L(A_4) \cup L(A'_4)$



Nondeterminism

Observe that the state transition diagram on the previous slide **no longer** represents a DFA: there is more than one arrow coming out of the initial state s with identical labels (say, a).

We say that this automaton is **nondeterministic** because, being in state s and reading a from the input tape, the device now has a choice: it can choose to move either to state q_1 or to state s .

What can affect this choice?

We may imagine that the automaton just tries to 'guess' which way will lead it finally to a favourable state.

But real computers cannot 'guess'...

This argument can hardly be refuted.

However, one can justify this variant of finite automata by the following:

- Nondeterministic finite automata are much **easier to design** than deterministic ones

They allow us to model how 'programs' are built from 'subprograms'

- Whatever can be done by a nondeterministic finite automaton, it can also be done (though usually in a bit more complicated way) by a deterministic one.

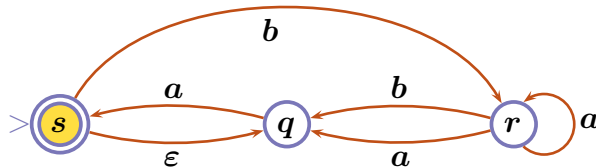
In other words, nondeterminism **does not increase the computational power** of finite automata

More nondeterminism

There are other features of nondeterministic finite automata (NFA):

- Just as from some states with some input symbols there may be several different possible next states, so with other combinations of states and input symbols there may be **no** possible moves at all (like in the example below, there is no a -arrow coming out of s).
- We also allow the machine to **'jump'** from a state to another state, without consuming any input symbol. These 'jumps' will be indicated on the state transition diagrams by arrows labelled with the empty word ϵ .

(Transition tables are no longer adequate to describe these automata.)



Acceptance of words by NFAs

A word w is **accepted** by an NFA A if **there is** a computation of A on input w ending up with a configuration (q, ε) for some favourable state q .

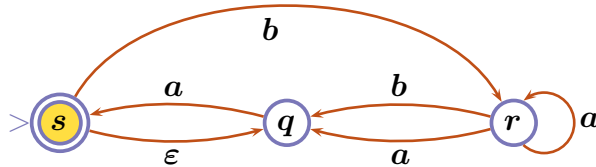
Otherwise, A **rejects** the input word.

So an NFA can reject an input word because **every** computation on this input is

- either 'stuck'
- or ends up with a configuration (q', ε) , but q' is not a favourable state.

There can be **more than one** computation on an input word !!

Example: how an NFA works



– Computations on input *baba*:

$(s, baba), (q, baba)$

stuck

$(s, baba), (r, aba), (r, ba), (q, a), (s, \varepsilon)$

accepted

– Computations on input *aa*:

$(s, aa), (q, aa), (s, a), (q, a), (s, \varepsilon)$

accepted

– Computations on input *babba*:

$(s, babba), (q, babba)$

stuck

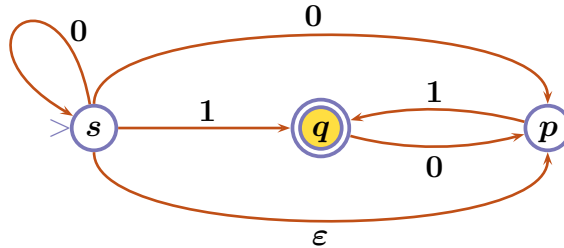
$(s, babba), (r, abba), (r, bba), (q, ba)$

stuck

$(s, babba), (r, abba), (q, bba)$

stuck, no more \leadsto rejected

NFA: another example



– Computations on input 11:

$(s, 11), (q, 1)$

stuck

$(s, 11), (p, 11), (q, 1)$

stuck, no more \leadsto **rejected**

– Computations on input 00:

$(s, 00), (s, 0), (s, \varepsilon); (s, 00), (s, 0), (p, \varepsilon)$

stuck

$(s, 00), (s, 0), (p, 0); (s, 00), (p, 0)$

stuck

$(s, 00), (p, 00)$

stuck, no more \leadsto **rejected**

– Computations on input 001:

$(s, 001), (s, 01), (s, 1), (q, \varepsilon)$

accepted

Nondeterministic Finite Automaton (NFA)

Formal definition:

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

$$A = (Q, \Sigma, \Delta, s, F) \text{ where}$$

- Q is a finite set of **states**
- Σ is a finite set, the **input alphabet**
- $s \in Q$ is the **initial state**
- $F \subseteq Q$ is the set of **favourable** (or **accepting**) **states**
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a ternary relation, the **transition relation**.

(If A , being in state $q \in Q$, has read the symbol $a \in \Sigma$, it enters one of the states q' for which (q, a, q') is in Δ . For any state q'' such that (q, ϵ, q'') is in Δ , A can 'jump' from state q to state q'' without reading any input symbol.)

Configurations of NFAs

Let $A = (Q, \Sigma, \Delta, s, F)$ be an NFA.

- A **configuration** of A is any pair of the form

(state in Q , word over Σ)

- Configuration (q, w) **yields** configuration (q', w') **in one step** if
 - either $w = \sigma w'$ for some σ in Σ and the triple (q, σ, q') is in Δ
 - or $w = w'$ and the triple (q, ε, q') is in Δ ('jump').

Observe that in general a configuration can yield **several** (or **no**) configurations in one step. In other words, a configuration does not determine uniquely the configuration it yields in one step.

That is why the automaton A is called **nondeterministic**

Computations of NFAs

A **computation of A on input word w** is **any** sequence of configurations

$$(q_1, w_1), (q_2, w_2), \dots, (q_k, w_k)$$

such that

- $(q_1, w_1) = (s, w)$ where s is the initial state of A
- every (q_i, w_i) yields (q_{i+1}, w_{i+1}) in one step, and
- either w_k is the empty word ϵ , or the configuration (q_k, w_k) yields no configuration in one step (**it gets stuck**)

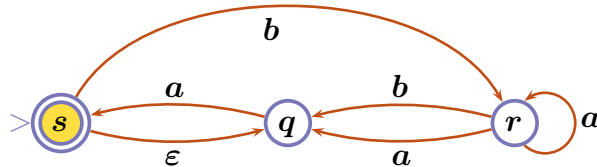
There can be **more than one** computation on an input word

NFAs and parallelism

Nondeterminism can be viewed as a kind of parallel computation wherein several 'processes' can be running concurrently.

- Suppose we are running an NFA on an input string and come to a state and read a symbol with multiple ways to proceed. Then the machine splits into multiple copies of itself and follows **all** the possibilities in parallel. If there are subsequent choices, the machine splits again.
- If the next input symbol does not appear on any arrows exiting the state occupied by a copy of the machine, that copy 'dies,' the branch of computation associated to it is 'stuck.'
- If a state with some exiting ϵ -arrows is encountered then, without reading any input, the machine splits into multiple copies: one following each exiting ϵ -arrow, and one staying at the current state.

Differences between NFAs and DFAs

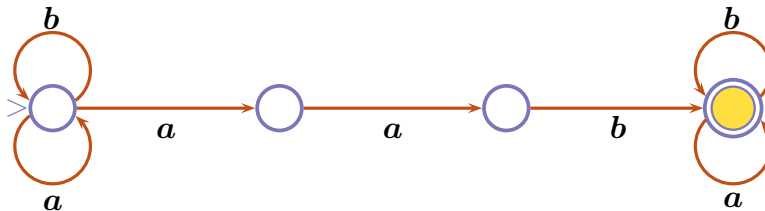


- There can be **more than one** arrow labelled by the same symbol coming out of a state (e.g., there are two *a*-arrows out of *r*).
- There can be **no** arrow labelled by some symbol coming out of a state (e.g., there is no *b*-arrow out of *q*).
- There can be arrows labelled by not symbols from the input alphabet but by the empty word ϵ . These arrows represent 'jumps' (e.g. there is a jump from *s* to *q*).

Every DFA is an NFA as well !

Pattern matching revisited

Problem: Design a finite automaton A such that $L(A)$ consists of all strings of a 's and b 's which contain aab as a substring



- It is much easier to design an NFA than a DFA.
- We will show now that every NFA can be converted to a DFA accepting precisely the same words

Equivalence of DFAs and NFAs

Two automata A and A' are said to be **equivalent** if they accept the same language

Claim: *For every NFA A , there is a DFA A' equivalent to A .*

Moreover, there is a 'mechanical method,' an **algorithm**, that carries out this conversion from a nondeterministic automaton to its deterministic equivalent

The 'subset construction'

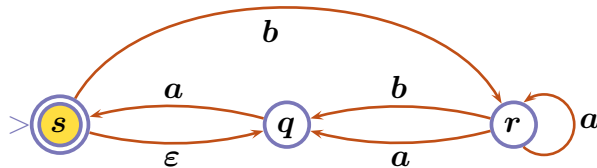
Let $A = (Q, \Sigma, \Delta, s, F)$ be a NFA. We will turn A to an equivalent DFA A' .

- The **states** of A' : all subsets of Q , that is, $\text{Pow}(Q)$
- The **initial state** of A' : the set containing the initial state s of A
plus all those states from Q that are reachable from s by ϵ -jumps.
- The **favourable states** of A' : all those subsets of Q that contain at least
one favourable state of A .

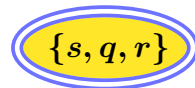
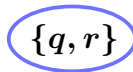
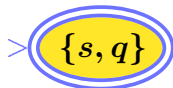
In other words, $P \subseteq Q$ represents a favourable state of A' if $P \cap F \neq \emptyset$

Example: the new states

NFA A :



the $2^3 = 8$ new **states** of DFA A' :



Subset construction: the new transition function

Remember, being in a state and reading an input symbol,
the transition function δ tells the machine's next state.

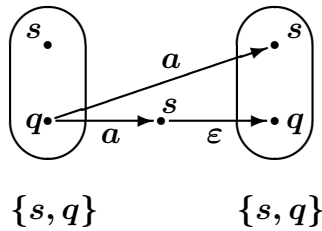
And the new states are **sets** of 'old' states now.

The **transition function** of A' : for every subset P of Q and every symbol a in Σ ,

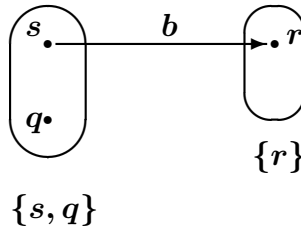
$\delta(P, a) =$ the set of all states that are reachable from
some state in P by an a -arrow,
followed by possibly one or more ε -jumps.

Computing the new transition function

$$\underline{\delta(\{s, q\}, a) = \{s, q\}:}$$



$$\underline{\delta(\{s, q\}, b) = \{r\}:}$$

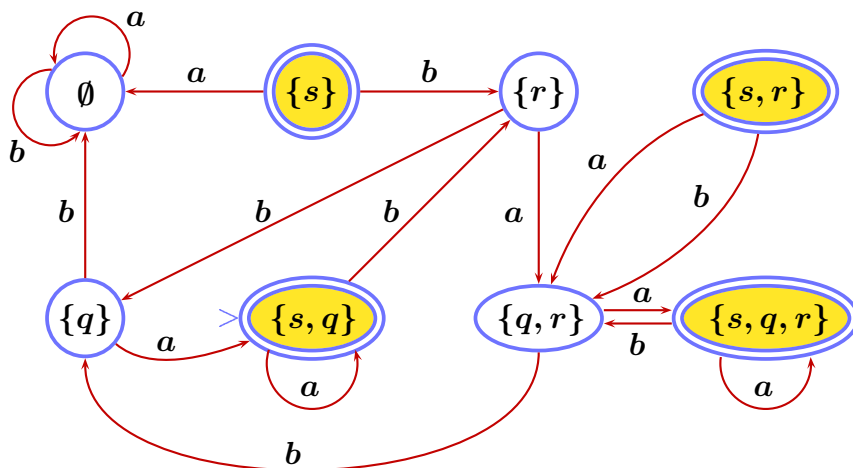


The new transition function

δ	a	b
$\{s\}$	\emptyset	$\{r\}$
$\{q\}$	$\{s, q\}$	\emptyset
$\{r\}$	$\{q, r\}$	$\{q\}$

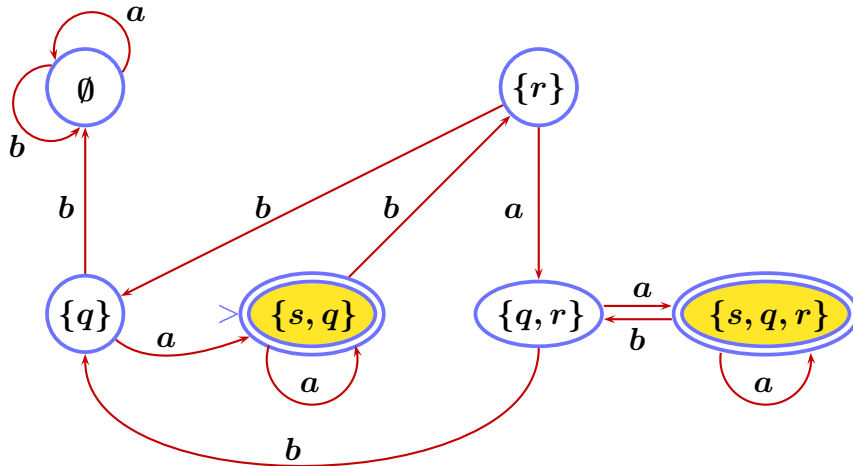
δ	a	b
$\{s, q\}$	$\{s, q\}$	$\{r\}$
$\{s, r\}$	$\{q, r\}$	$\{q, r\}$
\emptyset	\emptyset	\emptyset

δ	a	b
$\{q, r\}$	$\{s, q, r\}$	$\{q\}$
$\{s, q, r\}$	$\{s, q, r\}$	$\{q, r\}$



Removing the unreachable states

Observe that there is no computation going through the states $\{s\}$ and $\{s, r\}$, so we can safely remove them together with their outgoing arrows:



The initial state is never removable !

State minimisation problem

We saw that it is very easy to design nondeterministic finite automata for, say, pattern matching problems. However, an NFA cannot be directly implemented as a computer program. One must convert it into a DFA. But the number of states of the resulting DFA may increase significantly

2^{number of states of the original NFA} in the worst case

An important field where finite automata are being used is **hardware design**: some components of hardware are based on simulation of DFA. An important objective here is to use finite automata that have a **minimum** possible number of states. There are methods for doing this. (The 'subset construction' is just the initial phase of design.)