Finite Automata

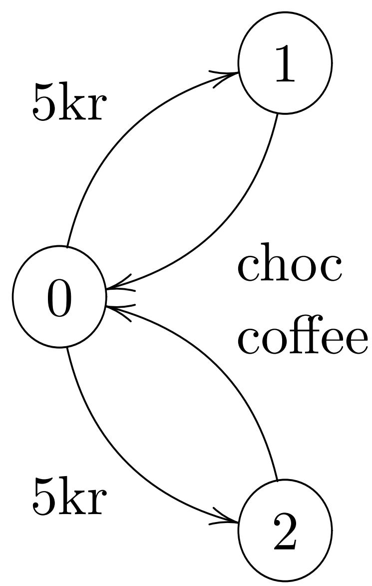
We present one application of finite automata: non trivial text search algorithm

Given a finite set of words find if there are occurences of one of these words in a given text

Nondeterministic Finite Automata

A nondeterministic finite automaton (NFA) is one for which the next state is not uniquely determined by the current state and the coming symbol

Informally, the automaton can *choose* between different states



A nondeterministic vending machine

Nondeterministic Finite Automata

When does *nondeterminism* appear??

Tossing a coin (probabilistic automata)

When there is incomplete information about the state

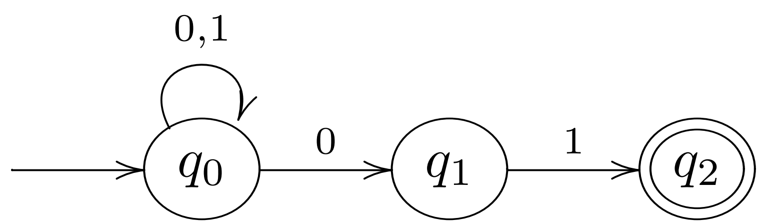
For example, the behaviour of a distributed system might depend on messages from other processes that arrive at unpredictable times

Nondeterministic Finite Automata

When does a NFA accepts a word??

Intuitively, the automaton accepts w iff there is *at least one* computation path starting from the start state to an accepting state

It is helpful to think that the automaton can *guess* the succesful computation (if there is one)

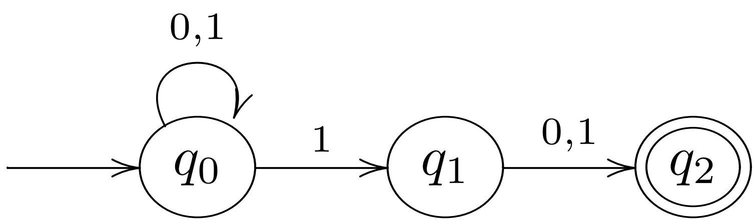


NFA accepting all words that end in 01

What are all possible computations for the word 1010??

Nondeterministic Finite Automata

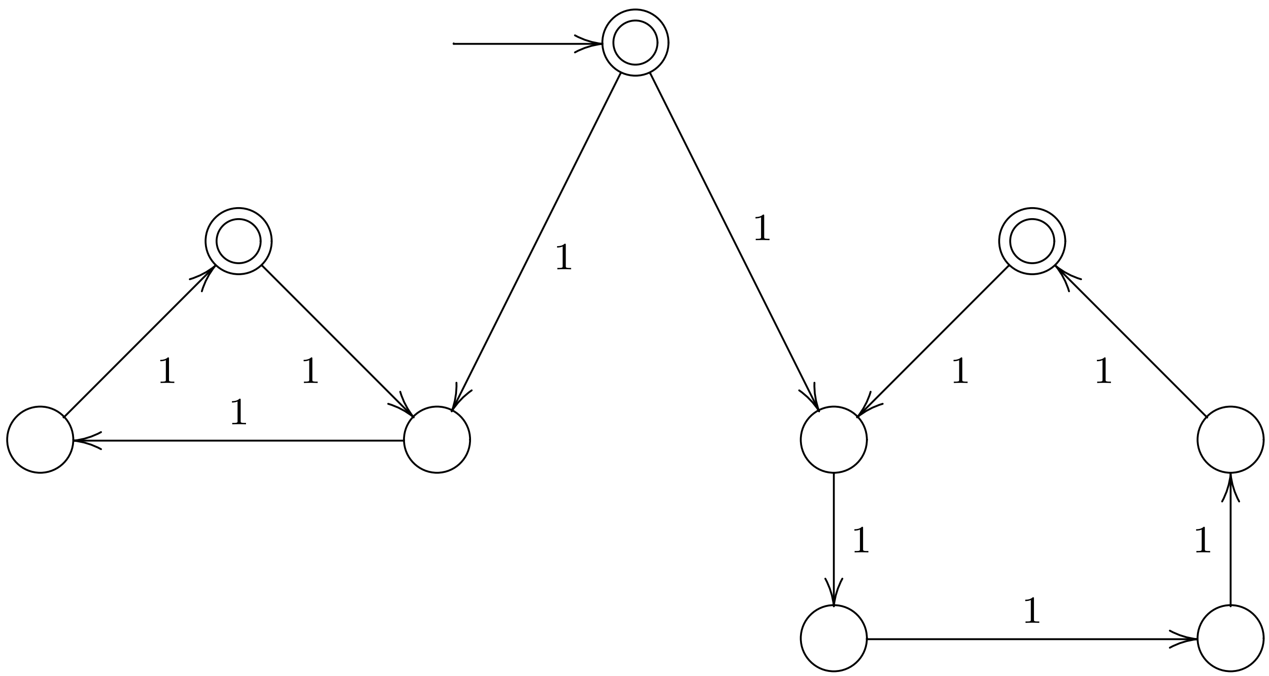
Another example: automaton accepting only the words such that the second last symbol from the right is 1



The automaton "guesses" when the word finishes

Nondeterministic Finite Automata

$\Sigma = \{1\}$



NFA accepting all words of length multiple of 3 *or* 5

The automaton *guesses* the right direction, and then *verifies* that $|w|$ is correct!

How to define mathematically a non deterministic machine??

NFA and DFA

We saw on examples that it is much easier to build a NFA accepting a given language than to build a DFA accepting this language

We are going to give an *algorithm* that produces a DFA from a given NFA accepting the same language

This is surprising because a DFA cannot “guess”

First we have to define *mathematically* what is a NFA

Both this definition and the algorithm uses in a crucial way the *powerset* operation

if A is a set, we denote by $Pow(A)$ the set of all *subsets* of A

(in particular the empty set $\emptyset$ is in $Pow(A)$)

Nondeterministic Finite Automata

Definition A nondeterministic finite automaton (NFA) consists of

1. a finite set of *states* (often denoted Q)
2. a finite set Σ of *symbols* (alphabet)
3. a *transition function* that takes as argument a state and a symbol and returns a **set of states** (often denoted δ); this set can be empty
4. a *start state*
5. a set of *final* or *accepting* states (often denoted F)

We have, as before, $q_0 \in Q$ $F \subseteq Q$

Nondeterministic Finite Automata

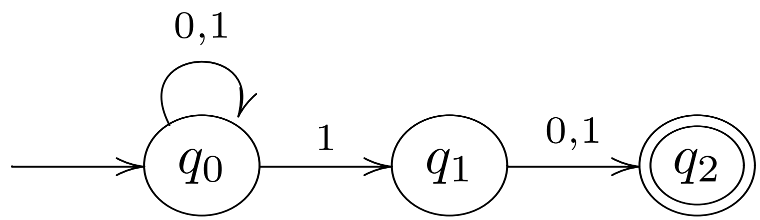
The transition function of a NFA is a function

$\delta : Q \times \Sigma \to Pow(Q)$

Each symbol $a \in \Sigma$ defines a *binary relation* on the set Q

$q_1 \xrightarrow{a} q_2$ iff $q_2 \in \delta(q_1, a)$

Nondeterministic Finite Automata



has for transition table

	0	1
$\rightarrow q_0$	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

$\delta(q_0, 1) = \{q_0, q_1\}$; we have $\delta(q_0, 1) \in Pow(Q)$

Extending the Transition Function to Strings

We define $\hat{\delta}(q, x)$ by induction

BASIS $\hat{\delta}(q, \epsilon) = \{q\}$

INDUCTION suppose $x = ay$

$\hat{\delta}(q, ay) = \hat{\delta}(p_1, y) \cup \ldots \cup \hat{\delta}(p_k, y)$ where $\delta(q, a) = \{p_1, \ldots, p_k\}$

$\hat{\delta}(q, ay) = \bigcup_{p \in \delta(q,a)} \hat{\delta}(p, y)$

We write $q.x \in Pow(Q)$ instead of $\hat{\delta}(q, x)$

Extending the Transition Function to Strings

A word x is accepted iff $q_0.x \cap F \neq \emptyset$ i.e. there is at least one accepting state in $q_0.x$

$\hat{\delta} : Q \times \Sigma^* \to Pow(Q)$ and each *word* x defines

a binary relation on Q: $q_1 \xrightarrow{x} q_2$ iff $q_2 \in q_1.x$

$L(A) = \{x \in \Sigma^* \mid q_0.x \cap F \neq \emptyset\}$

Extending the Transition Function to Strings

Intuitively: $q_1 \xrightarrow{x} q_2$ means that there is one path from q_1 to q_2 having x for sequence of events

We can define $q_1 \xrightarrow{x} q_2$ inductively

BASIS: $q_1 \xrightarrow{\epsilon} q_2$ iff $q_1 = q_2$

STEP: $q_1 \xrightarrow{ay} q_2$ iff there exists $q \in \delta(q_1, a)$ such that $q \xrightarrow{y} q_2$

Then we have $q_1 \xrightarrow{x} q_2$ iff $q_2 \in q_1.x$

Representation in functional programming

```
next :: Q -> E -> [Q]

run :: Q -> [E] -> [Q]

run q [] = [q]
run q (a:x) = concat (map (\ p -> run p x) (next q a))
```

Representation in functional programming

We use

```
-- map f [a1,...,an] = [f a1,...,f an]

 map f [] = []
 map f (a:x) = (f a):(map f x)

-- concat [x1,...,xn] = x1 ++ ... ++ xn

 concat [] = []
 concat (x:xs) = x ++ concat xs
```

Representation in functional programming

It is nicer to take

```
next :: E -> Q -> [Q]
```

we define

```
run :: [E] -> Q -> [Q]

run [] q = [q]
run (a:x) q = concat (map (run x) (next a q))
```

Representation in functional programming

In the monadic notation (with the list monad)

```
run :: [E] -> Q -> [Q]

run [] q = return q
run (a:x) q = next a q >>= run x

accept :: [E] -> Bool
accept x = or (map final (run x q0))
```

Representation in functional programming

List monad: clever notations for programs with list

```
-- return :: a -> [a]

return x = [x]

--  (>>=) :: [a] -> (a->[b]) -> [b]

xs >>= f = concat (map f xs)
```

This is exactly what is needed to define run (a:x) q

Representation in functional programming

Other notation: do notation

```
run :: [E] -> Q -> [Q]

run [] q = return q
run (a:x) q = next a q >>= run x
```

is written

```
run :: [E] -> Q -> [Q]

run [] q = return q
run (a:x) q =
  do p <- next a q
     run x p
```

The Subset Construction

This corresponds closely to Ken Thompson's implementation

We can now indicate how, given a NFA, to build a DFA that accepts the same language.This DFA may require more states.

Intuitive idea of the construction for a NFA N: there are only finitely many subsets of Q, hence only finitely many possible situations

Extending the Transition Function to Strings

We start from a NFA $N = (Q, \Sigma, \delta, q_0, F)$ where

$\delta : Q \times \Sigma \to Pow(Q)$

We define

$\delta_D : Pow(Q) \times \Sigma \to Pow(Q)$

$\delta_D(X, a) = \bigcup_{q \in X} \delta(q, a)$

If $X = \{p_1, \ldots, p_k\}$ then

$\delta_D(X, a) = \delta(p_1, a) \cup \ldots \cup \delta(p_k, a)$

$\delta_D(\emptyset, a) = \emptyset, \qquad \delta_D(\{q\}, a) = \delta(q, a)$

The Subset Construction

This function satisfies also

$\delta_D(X_1 \cup X_2, a) = \delta_D(X_1, a) \cup \delta_D(X_2, a)$

$\delta_D(X, a) = \bigcup_{p \in X} \delta_D(\{p\}, a)$

The Subset Construction

We build the following DFA

$Q_D = Pow(Q)$

$\delta_D : Pow(Q) \times \Sigma \to Pow(Q)$

$q_D = \{q_0\} \in Q_D$

$F_D = \{X \subseteq Q \mid X \cap F \neq \emptyset\}$

Representation in functional programming

Given

```
next :: E -> Q -> [Q]
```

we define its parallel version

```
pNext :: E -> [Q] -> [Q]

pNext a qs = concat (map (next a) qs)
```

Representation in functional programming

With the monadic notation

```
pNext :: E -> [Q] -> [Q]
pNext a qs = qs >>= next a
```

```
pNext :: E -> [Q] -> [Q]
pNext a qs =
 do
  q <- qs
  next a q
```

Representation in functional programming

We can now define

```
run' :: [E] -> [Q] -> [Q]

run' [] qs = qs
run' (a:x) qs = run' x (pNext a qs)
```

Representation in functional programming

We state that we have for all x

```
run' x [q] = run x q

run' [a1,a2] [q]
 = pNext a2 (pNext a1 [q])
 = [q] >>= next a1 >>= next a2
 = next a1 q >>= next a2

run [a1,a2] q
 = next a1 q >>= run [a2]
 = next a1 q >>= (\ p -> next a2 p >>= return)
 = next a1 q >>= next a2
```

The Subset Construction

Lemma 1: *For all word z and all set of states X we have* $\hat{\delta_D}(X, z) = \bigcup_{p \in X} \hat{\delta_D}(\{p\}, z)$

Lemma 2: *For all words x we have* $q.x = \hat{\delta_D}(\{q\}, x)$

Proof: By induction. The inductive case is when $x = ay$ and then

$$
\begin{array}{rcll}
q.(ay) & = & \bigcup_{p \in \delta(q,a)} p.y & \text{by definition} \\
 & = & \bigcup_{p \in \delta(q,a)} \hat{\delta_D}(\{p\}, y) & \text{by induction} \\
 & = & \hat{\delta_D}(\delta(q,a), y) & \text{by lemma 1} \\
 & = & \hat{\delta_D}(q, ay) & \text{by definition}
\end{array}
$$

The Subset Construction

Lemma: For all words x we have $q.x = \hat{\delta_D}(\{q\}, x)$

Theorem: The language accepted by the NFA N is the same as the language accepted by the DFA $(Q_D, \Sigma, \delta_D, q_D, F_D)$

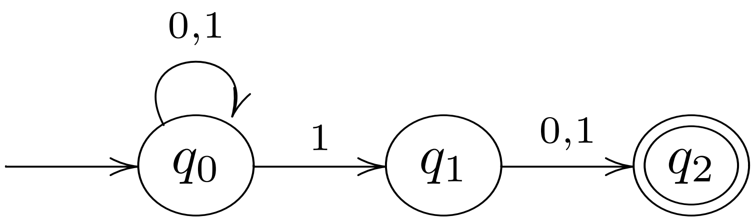
Proof: We have $x \in L(N)$ iff $\hat{\delta}(q_0, x) \cap F \neq \emptyset$ iff $\hat{\delta}(q_0, x) \in F_D$ iff $\hat{\delta_D}(q_D, x) \in F_D$. We use the Lemma to replace $\hat{\delta}(q_0, x)$ by $\hat{\delta_D}(\{q_0\}, x)$ which is the same as $\hat{\delta_D}(q_D, x)$ Q.E.D.

The Subset Construction

It seems that if we start with a NFA that has n states we shall need 2^n states for building the corresponding DFA

In practice, often a lot of states are not accessible from the start state and we don't need them

The Subset Construction



We start from A= $\{q_0\}$ (only one start state)

If we get 0, we can only go to the state q_0

If we get 1, we can go to q_0 *or* to q_1. We represent this by going to the state B= $\{q_0, q_1\} = \delta_D(A, 1)$

From B, if we get 0, we can go to q_0 or to q_2; we go to the state C= $\{q_0, q_2\} = \delta_D(B, 0)$

From B, if we get 1, we can go to q_0 or q_1 or q_2; we go to the state D= $\{q_0, q_1, q_2\} = \delta_D(B, 1)$

etc...

The Subset Construction

We get the following automaton

A= $\{q_0\}$

B= $\{q_0, q_1\}$

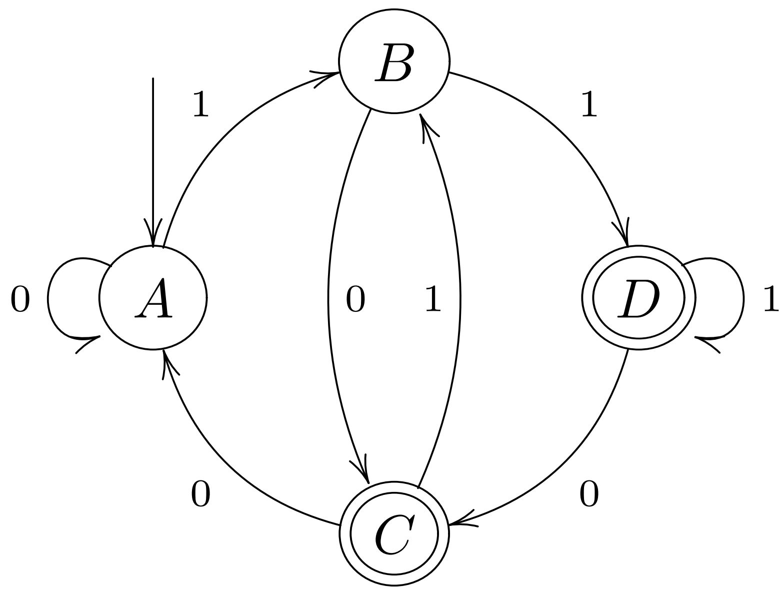
C= $\{q_0, q_2\}$

D= $\{q_0, q_1, q_2\}$

	0	1
$\rightarrow$A	A	B
B	D	C
$*$C	A	B
$*$D	C	D

The Subset Construction

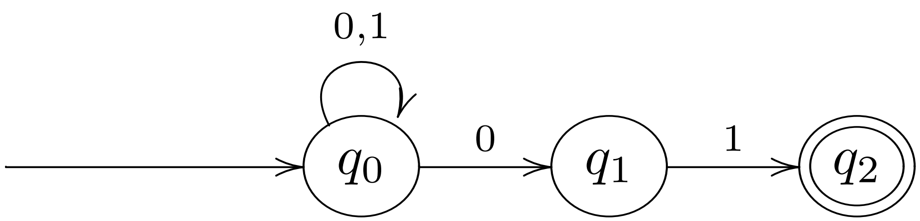
Same automaton, as a transition system



The DFA "remembers" the last two bits seen and accepts if the next-to-last bit is 1

The Subset Construction

Another example: words ending by 01



The new states are

A = $\{q_0\}$

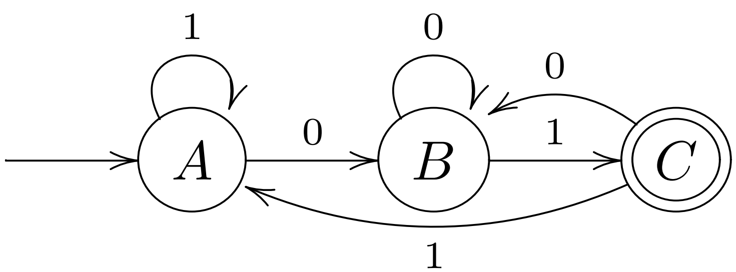
B = $\{q_0, q_1\}$

C = $\{q_0, q_2\}$

	0	1
→A	B	A
B	B	C
∗C	B	A

The Subset Construction

The DFA is



A = $\{q_0\}$

B = $\{q_0, q_1\}$

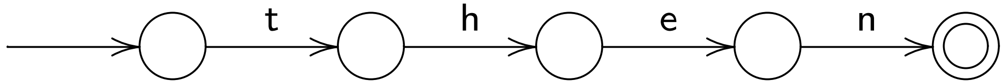
C = $\{q_0, q_2\}$

This DFA has only 3 states (and not 8). It is correct i.e. accepts only the word ending by 01 *by construction*

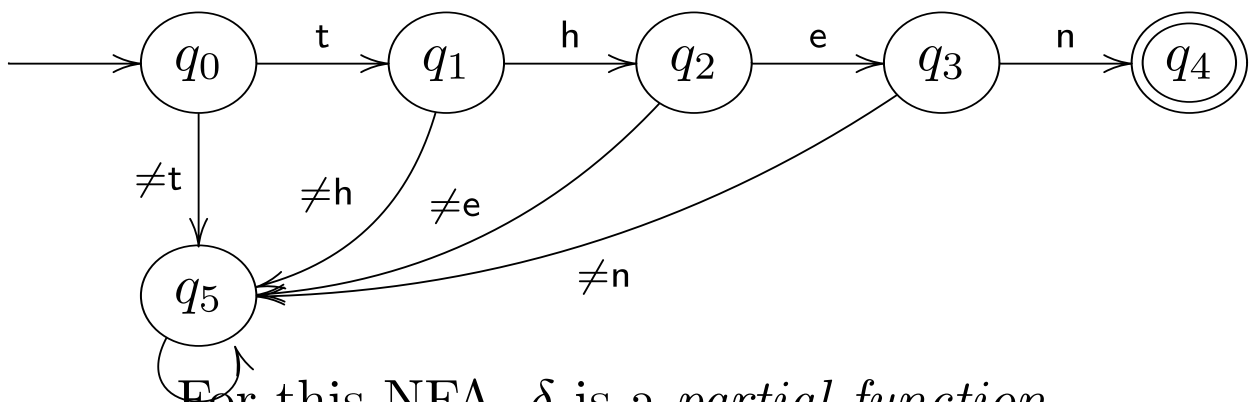
We had only to prove the general correctness of the subset construction

Example: password

If we apply the subset construction to the NFA



we get exactly the following DFA



For this NFA, δ is a *partial function*

with a "stop" or "dead" state $q_5 = \emptyset$

An Application: Text Search

Suppose we are given a set of words, called *keywords*, and we want to find occurences of any of these words.

For such a problem, a useful way to proceed is to design a NFA which recognizes, by entering in an accepting state, that it has seen one of the keywords.

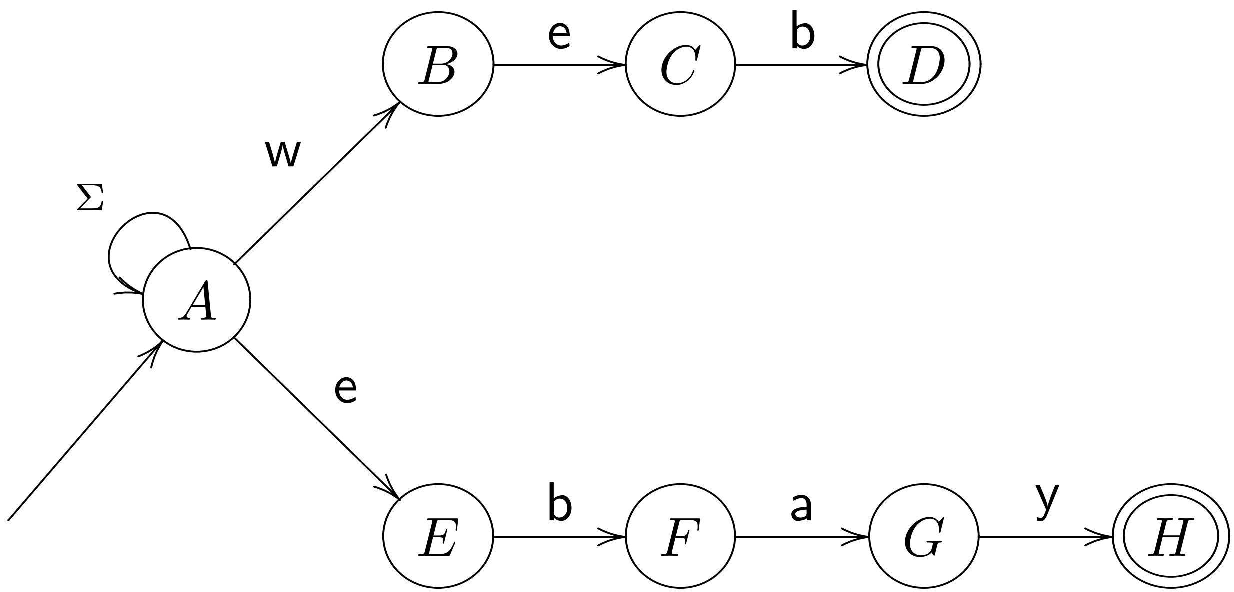
The NFA is only a nondeterministic program, but we can run it using lists or transform it to a DFA and get a deterministic (efficient) program

Once again, we know that this DFA will be correct by construction

This is a good example of a derivation of a *program* (DFA) from a *specification* (NFA)

An Application: Text Search

The following NFA searches for the keyword web and ebay



Almost no thinking needed to write this NFA

What is a corresponding DFA?? Notice that this has the *same* number of states as the NFA

Representation in functional programming

```
\slideheading{Representation in functional programming}

data Q = A | B | C | D | E | F| G | H

next 'w' A = [A,B]
next 'e' A = [A,E]
next _ A = [A]
next 'e' B = [C]
next 'b' C = [D]
next 'b' E = [F]
next 'a' F = [G]
next 'y' G = [H]
next _ D = [D]
next _ H = [H]
next _ _ = []
```

Representation in functional programming

```
run :: String -> Q -> [Q]
run [] q = return q
run (a:x) q = next a q >>= run x

final :: Q -> Bool
final D = True
final H = True
final _ = False

accept :: String -> Bool
accept x = or (map final (run x A))
```

An Application: Text Search

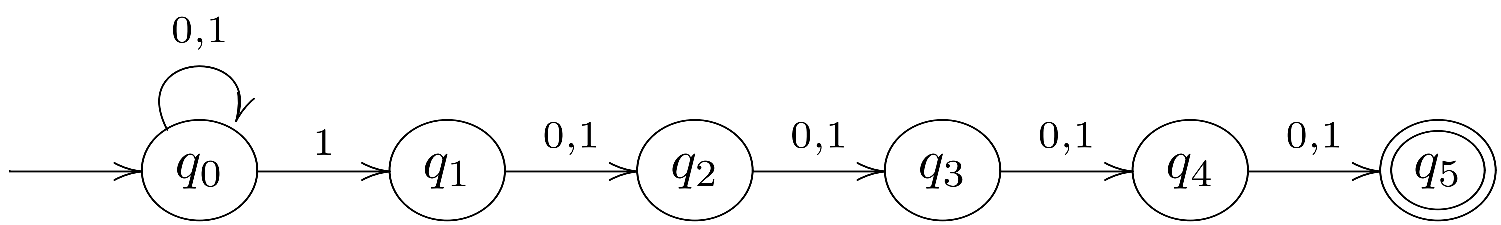
Even for searching an occurence of *one* keyword this gives an interesting program

This is connected to the Knuth-Morris-Pratt string searching algorithm

Better than the naive string searching algorithm

A Bad Case for the Subset Construction

Theorem: Any DFA recognising the same language as the NFA



has at least $2^5 = 32$ states!

A Bad Case for the Subset Construction

Lemma 1: If A is a DFA then

$$q.(xy) = (q.x).y$$

for any $q \in Q$ and $x, y \in \Sigma^*$

We have proved this last time

A Bad Case for the Subset Construction

We define $L_n = \{x1y \mid x \in \Sigma^*,\ y \in \Sigma^{n-1}\}$

$A = (Q, \Sigma, \delta, q_0, F)$

Theorem: If $|Q| < 2^n$ then $L(A) \neq L_n$

Lemma 2: If $|Q| < 2^n$ there exists $x, y \in \Sigma^*$ and $u, v \in \Sigma^{n-1}$ with $q_0.(x0u) = q_0.(y1v)$

Proof of the Theorem, given Lemma 2: If $L(A) = L_n$ we have

$y1v \in L(A)$ and $x0u \notin L(A)$, so

$q_0.(y1v) \in F$ and $q_0.(x0u) \notin F$

This contradicts $q_0.(x0u) = q_0.(y1v)$ Q.E.D.

A Bad Case for the Subset Construction

Proof of the lemma: The map $z \longmapsto q_0.z, \quad \Sigma^n \to Q$ is *not* injective because $|Q| < 2^n = |\Sigma^n|$

So we have $a_1 \dots a_n \neq b_1 \dots b_n$ with

$$q_0.(a_1 \dots a_n) = q_0.(b_1 \dots b_n) \qquad (*)$$

We can assume $a_i = 0,\ b_i = 1$. We take
$x = a_1 \dots a_{i-1},\ y = b_1 \dots b_{i-1}$ and
$u = a_{i+1} \dots a_n 0^{i-1},\ v = b_{i+1} \dots b_n 0^{i-1}$

Notice then that $(*)$ implies, by Lemma 1

$$q_0.(a_1 \dots a_n 0^{i-1}) = q_0.(b_1 \dots b_n 0^{i-1})$$

Q.E.D.