Finite Automata and Formal Languages

TMV026/DIT321 – LP4 2010

Lecture 4

March 25rd 2010

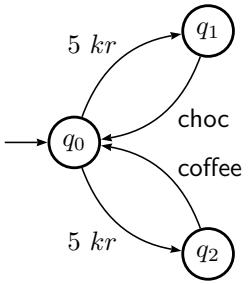
Overview of today's lecture:

- **Non-deterministic Finite Automata**
- **Equivalence between DFA and NFA**

Non-deterministic Finite Automata

A non-deterministic finite automata (NFA) can be in several states at once.

That is, given a state an the next symbol, the automata can "move" to many states.

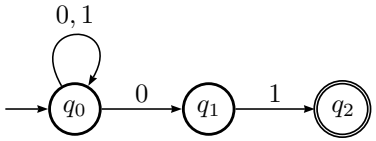


Intuitively, the vending machine can *choose* between different states.

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 successful computation if there is one.

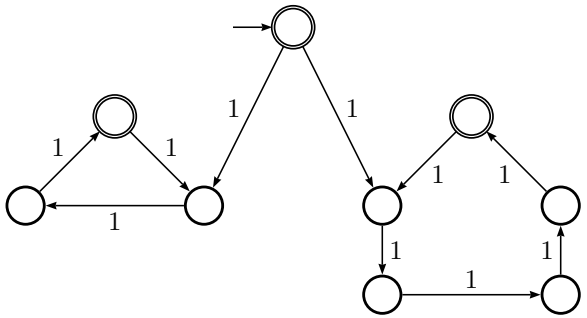


NFA accepting words that end in 01

What are all possible computations for the string 10101?

NFA Accepting Words of Length Divisible by 3 or by 5

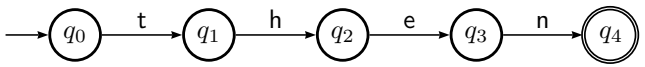
Let $\Sigma = \{1\}$.



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

What would be the equivalent DFA?

NFA Accepting the word "then"



Observe that we do not need a *dead* state here.

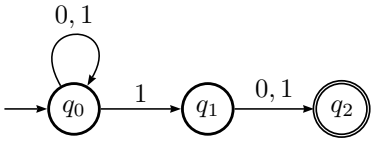
Non-deterministic Finite Automata

Definition: A ***non-deterministic finite automaton*** (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of:

1. A finite set Q of *states*
2. A finite set Σ of *symbols* (alphabet)
3. A *transition function* $\delta : Q \times \Sigma \to \mathcal{P}ow(Q)$
   (partial function that takes as argument a state and a symbol and returns a *set of states*)
4. A *start state* $q_0 \in Q$
5. A set $F \subseteq Q$ of *final* or *accepting* states

Example: NFA

Let us define an automaton accepting only the words such that the second last symbol from the right is 1.



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

The automaton *guesses* when the word finishes.

Extending the Transition Function to Strings

As before, we want to be able to determine $\hat{\delta}(q, x)$.

We define this by recursion on x.

Definition:

$$\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}ow(Q)$$
$$\hat{\delta}(q, \epsilon) = \{q\}$$
$$\hat{\delta}(q, ax) = \bigcup_{p \in \delta(q,a)} \hat{\delta}(p, x)$$

That is, if $\delta(q, a) = \{p_1, \ldots, p_n\}$ then

$$\hat{\delta}(q, ax) = \hat{\delta}(p_1, x) \cup \ldots \cup \hat{\delta}(p_n, x)$$

NFA as a Labelled Graphs

We could define the relation $p \xrightarrow{x} q$ iff $q \in \hat{\delta}(p, x)$ and see the NFA as a labelled graph.

The formal definition is done by induction on x.

- Basic case: $p \xrightarrow{\epsilon} p$
- Inductive step: $p \xrightarrow{ax} q$ iff $\exists r.\ r \in \delta(p, a)$ and $r \xrightarrow{x} q$

Note: We have $p \xrightarrow{a} q$ iff $q \in \delta(p, a)$.

Note: Now we can formally prove that $p \xrightarrow{x} q$ iff $q \in \hat{\delta}(p, x)$.

Another way of defining this relation is

$$\frac{}{p \xrightarrow{\epsilon} p} \qquad \frac{p \xrightarrow{a} r \qquad r \xrightarrow{x} q}{p \xrightarrow{ax} q}$$

Language Accepted by a NFA

Definition: The ***language*** accepted by the NFA $N = (Q, \Sigma, \delta, q_0, F)$ is the set $\mathcal{L}(N) = \{x \in \Sigma^* \mid \hat{\delta}(q_0, x) \cap F \neq \emptyset\}$.

That is, a word x is accepted if $\hat{\delta}(q_0, x)$ contains at least one accepting state.

An alternative definition is: $\mathcal{L}(N) = \{x \in \Sigma^* \mid \exists q \in F.\ q_0 \xrightarrow{x} q\}$.

Note: Again, we could write a program that simulates a NFA and let the program tell us whether a certain string is accepted or not.

Functional Representation of a NFA

Consider the following functions:

```
-- map f [x1, ... ,xn] = [f x1, ... ,f  xn]
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

-- [x1,...,xn] ++ [y1,...,ym] = [x1,...,xn,y1,...,ym]
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys

-- concat [xs1,...,xsn] = xs1 ++ ... ++ xsn
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```


Functional Representation of a NFA

```
data Q = ...
data S = ...

final :: Q -> Bool
...

delta :: S -> Q -> [Q]           -- Observe change in the type
...

run :: [S] -> Q -> [Q]           -- Idem
run [] q = [q]
run (a:xs) q = concat (map (run xs) (delta a q))

accepts :: [S] -> Bool
accepts xs = or (map final (run xs Q0))
```

Functional Representation of a NFA

A nicer way is to use "monadic" lists, which is a clever notation for programs using lists.

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

-- (>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)

run :: [S] -> Q -> [Q]
run [] q = return q
run (a:xs) q = delta a q >>= run xs
```

Note: The actual types of `return` and `(>>=)` are more general than those above...

Functional Representation of a NFA

Alternative notation for

```
run :: [S] -> Q -> [Q]
run [] q = return q
run (a:xs) q = delta a q >>= run xs
```

is

```
run :: [S] -> Q -> [Q]
run [] q = return q
run (a:xs) q = do p <- delta a q
                  run xs p
```


Transforming a NFA into a DFA

We have seen that for same examples it is much simpler to define a NFA than a DFA.

For example, the language with words of length divisible by 3 or by 5.

However, any language accepted by a NFA is also accepted by a DFA.

In general, the number of states of the DFA is about the number of states in the NFA although it often has many more transitions.

In the worst case, if the NFA has n states, a DFA accepting the same language might have 2^n states.

The *algorithm* transforming a NFA into an equivalent DFA is called the ***subset construction***.

The Subset Construction

Given a NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ we will construct a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $\mathcal{L}(D) = \mathcal{L}(N)$ as follows:

- $Q_D = \mathcal{P}ow(Q_N)$
- $\delta_D : Q_D \times \Sigma \to Q_D$ (that is, $\delta_D : \mathcal{P}ow(Q_N) \times \Sigma \to \mathcal{P}ow(Q_N)$)
  $\delta_D(X, a) = \bigcup_{q \in X} \delta_N(q, a)$
- $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$

Intuitive idea of the construction: there are only finitely many subsets of Q_N, hence only finitely many possible situations.

Remarks: Subset Construction

- If $|Q_N| = n$ then $|Q_D| = 2^n$.
  If some of the states in Q_D are not *accessible* from the start state of D we can safely remove them (we will see how to do this later on in the course).

- If $X = \{q_1, \ldots, q_n\}$ then $\delta_D(X, a) = \delta_N(q_1, a) \cup \ldots \cup \delta_N(q_n, a)$.
  In addition,

  $\delta_D(\emptyset, a) = \emptyset$ $\qquad$ $\delta_D(\{q\}, a) = \delta_N(q, a)$ $\qquad$ $\delta_D(X, a) = \bigcup_{q \in X} \delta_D(\{q\}, a)$

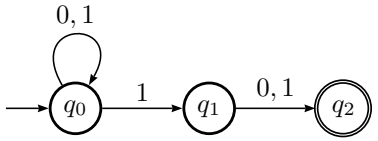
  and

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

- Each accepting state (set) S in F_D contains at least one accepting state of N.

Example: Subset Construction

Let us consider the NFA



The DFA we construct will start from $\{q_0\}$. Only accessible states matter ...

From $\{q_0\}$, if we get 0, we can only go to the state q_0 so $\delta_D(\{q_0\}, 0) = \{q_0\}$.

From $\{q_0\}$, if we get 1, we can go to q_0 or to q_1. We represent this by the state $\{q_0, q_1\}$ and the transaction $\delta_D(\{q_0\}, 1) = \{q_0, q_1\}$.

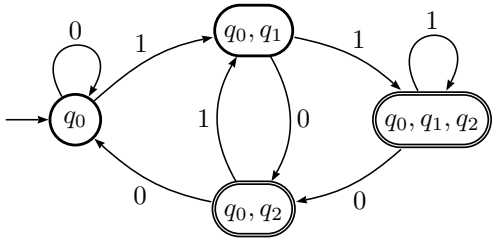
From $\{q_0, q_1\}$, if we get 0, we can go to q_0 or to q_2. Then we get a new state $\{q_0, q_2\}$ and also $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_2\}$.

From $\{q_0, q_1\}$, if we get 1, we can go to q_0 or q_1 or q_2. Then we get a new state $\{q_0, q_1, q_2\}$ and also $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_1, q_2\}$.

etc...

Example: Subset Construction (cont.)

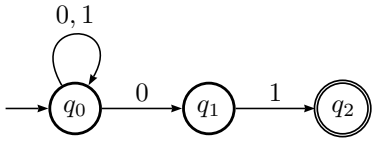
The complete (and simplified) DFA from the previous NFA is:



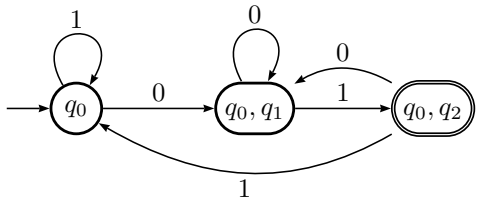
The DFA *remembers* the last two bits seen and accepts a word if the next-to-last bit is 1.

Example: Subset Construction

Given the NFA



we obtain the following DFA:



By only computing the *accessible* states (from the start state) we are able to keep the total number of states to 3 (and not 8).

Functional Representation of the Subset Construction

Given a (typed modified) δ_N function:

```
delta ::  S -> Q -> [Q]
```

we can define the (typed modified) δ_D function:

```
pDelta :: S -> [Q] -> [Q]
pDelta a qs = concat (map (delta a) qs)
```

or (with the monadic notation)

```
pDelta a qs = qs >>= delta a
```

or

```
pDelta a qs = do p <- qs
                 delta a p
```


Functional Representation of the Subset Construction

```
pFinal :: [Q] -> Bool
pFinal qs = or (map final qs)


pRun :: [S] -> [Q] -> [Q]
pRun [] qs = qs
pRun (a:xs) qs = pRun xs (pDelta a qs)


pAccepts :: [S] -> Bool
pAccepts xs = pFinal (pRun xs [Q0])
```


Testing the Correction of the Subset Construction

```
test :: [S] -> Bool
test xs = run xs Q0 == pRun xs [Q0]       -- see run in slides 11/12
```

Informally, let `xs` be `[x1,...,xn]`. Then:

```
run [x1,...,xn] q = delta x1 q >>= run [x2,...,xn]
= delta x1 q >>= (\p -> delta x2 p >>= run [...,xn])
= delta x1 q >>= (\p -> ... >>= (\r -> delta xn r >>= return)...)
= delta x1 q >>= delta x2 >>= .. >>= delta xn

pRun [x1,...,xn] [q] = pDelta xn (... (pDelta x1 [q])...)
= [q] >>= delta x1 >>= ... >>>= delta xn
= delta x1 q >>= delta x2 >>= .. >>= delta xn
```


Towards the Correction of the Subset Construction

Formally we have that

Proposition: $\forall q, x.\ \hat{\delta}_N(q, x) = \hat{\delta}_D(\{q\}, x)$.

Proof: By induction on x. Basic case is trivial.

The inductive step is:

$$
\begin{array}{rcll}
\hat{\delta}_N(q, ax) & = & \bigcup_{p \in \delta_N(q,a)} \hat{\delta}_N(p, x) & \text{by definition of } \hat{\delta}_N \\
& = & \bigcup_{p \in \delta_N(q,a)} \hat{\delta}_D(\{p\}, x) & \text{by IH} \\
& = & \hat{\delta}_D(\delta_N(q, a), x) & \text{see lemma below} \\
& = & \hat{\delta}_D(\delta_D(\{q\}, a), x) & \text{remark on slide 15} \\
& = & \hat{\delta}_D(\{q\}, ax) & \text{by definition of } \hat{\delta}_D
\end{array}
$$

Lemma: *For all words x and set of states S, $\hat{\delta}_D(S, x) = \bigcup_{p \in S} \hat{\delta}_D(\{p\}, x)$.*

Correction of the Subset Construction

Theorem: *Given a NFA N, if D is the DFA constructed from N by the subset construction then $\mathcal{L}(N) = \mathcal{L}(D)$.*

Proof: $x \in \mathcal{L}(N)$ iff $\hat{\delta}_N(q_0, x) \cap F_N \neq \emptyset$ iff $\hat{\delta}_N(q_0, x) \in F_D$.

By the previous proposition, this is equivalent to $\hat{\delta}_D(\{q_0\}, x) \in F_D$.

Since $\{q_0\}$ is the starting state in D the above is equivalent to $x \in \mathcal{L}(D)$.

Equivalence between DFA and NFA

Theorem: *A language $\mathcal{L}$ is accepted by some DFA iff $\mathcal{L}$ is accepted by some NFA.*

Proof: The "if" part is the result of the previous theorem (correctness of subset construction).

For the "only if" part we need to transform the DFA into a NFA.

Intuitively, each DFA can be seen as a NFA where there exists only one choice at each stage.

Formally, given $D = (Q, \Sigma, \delta_D, q_0, F)$ we define $N = (Q, \Sigma, \delta_N, q_0, F)$ such that, if $\delta_D(q, a) = p$ then $\delta_N(q, a) = \{p\}$.

It only remains to show (by induction on x) that if $\hat{\delta}_D(q_0, x) = p$ then $\hat{\delta}_N(q_0, x) = \{p\}$.

Application: Text Search

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

An useful way to proceed is to design a NFA that enters in an accepting state when it has recognised one of the keywords.

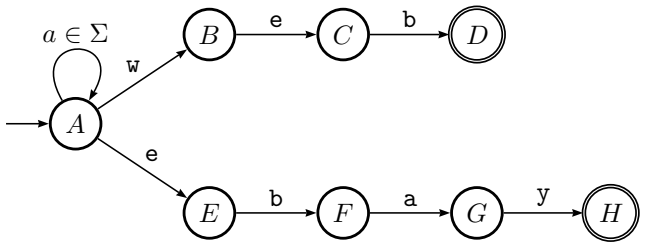
Then we could implement the NFA, or we could transform it to a DFA and get a deterministic (efficient) program.

Since we have proved the subset construction correct, we know the DFA will be correct (if the NFA is!).

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

Application: Text Search

The following (easy to write) NFA searches for the keyword `web` and `ebay`:



If one applies the subset construction one obtains the DFA of page 71 in the book.

Observe that the obtained DFA has the same number of states as the NFA.

Functional Representation: Text Search

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

delta :: Char -> Q -> [Q]
delta 'w' A = [A,B]
delta 'e' A = [A,E]
delta  _  A = [A]
delta 'e' B = [C]
delta 'b' C = [D]
delta 'b' E = [F]
delta 'a' F = [G]
delta 'y' G = [H]
delta  _  D = [D]
delta  _  H = [H]
delta  _  _ = []
```


Functional Representation: Text Search (cont.)

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

run :: String -> Q -> [Q]
run [] q = return q
run (a:xs) q = delta a q >>= run xs

accepts :: String -> Bool
accepts xs = or (map final (run xs A))
```