Finite Automata and Formal Languages

TMV026/DIT321– LP4 2011

Ana Bove

Lecture 4

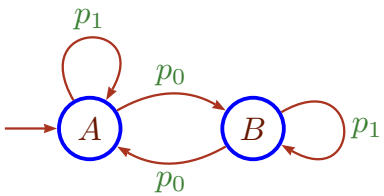
March 28th 2011

Overview of today's lecture:

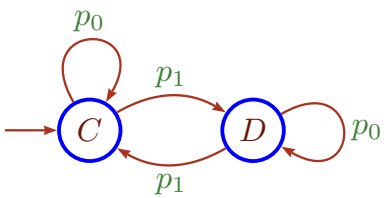
- **More on Deterministic Finite Automata**
- **Non-deterministic Finite Automata**
- **Equivalence between DFA and NFA**

Example: Product of Automata

Given an automaton that determines whether the number of p_0's is even or odd



and an automaton that determines whether the number of p_1's is even or odd



how to combine them so we keep track of the parity of *both* p_0 and p_1?

Product Construction

Definition: Given two DFA $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ with the *same alphabet* Σ, we can define the *product* $D = D_1 \times D_2$ as follows:

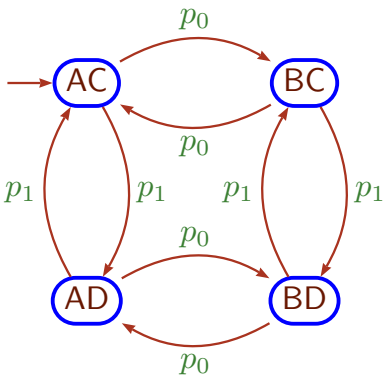
- $Q = Q_1 \times Q_2$
- $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$
- $q_0 = (q_1, q_2)$
- $F = F_1 \times F_2$

Proposition: $\hat{\delta}((r_1, r_2), x) = (\hat{\delta}_1(r_1, x), \hat{\delta}_2(r_2, x))$.

Proof: By induction on x.

Example: Product of Automata (cont.)

The product automaton that keeps track of the parity of *both* p_0 and p_1 is:



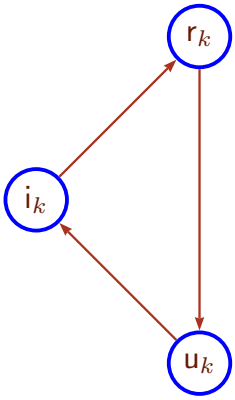
If after reading the word w we are in the state AD we know that w contains an even number of p_0's and an odd number of p_1's.

Example: Product of Automata

Let us model a system where users have three states: *idle*, *requesting* and *using*.

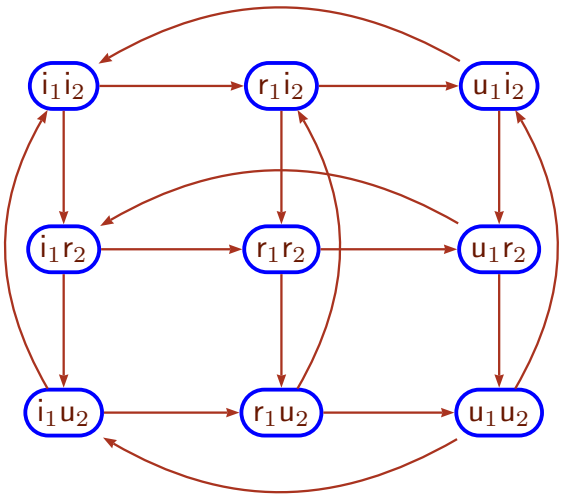
Let us assume we have 2 users.

Each user is represented by a simple automaton, for $k = 1, 2$:



Example: Product of Automata (cont.)

The complete system is represented by the product of these 2 automata and it has 3 * 3 = 9 states.



Language Accepted by a Product Automaton

Proposition: *Given two DFA D_1 and D_2, then* $\mathcal{L}(D_1 \times D_2) = \mathcal{L}(D_1) \cap \mathcal{L}(D_2)$.

Proof: $\hat{\delta}(q_0, x) = (\hat{\delta}_1(q_1, x), \hat{\delta}_2(q_2, x)) \in F$ iff $\hat{\delta}_1(q_1, x) \in F_1$ and $\hat{\delta}_2(q_2, x) \in F_2$, that is, $x \in \mathcal{L}(D_1)$ and $x \in \mathcal{L}(D_2)$.

Example: Let M_k be an automaton that accepts multiples of k such that $\mathcal{L}(M_k) = \{a^n \mid k \text{ divides } n\}$.

Then $M_6 \times M_9$ is M_{18} (6 divides k and 9 divides k iff 18 divides k.)

Note: It can be quite difficult to directly build an automaton accepting the intersection of two languages.

Example: Build a DFA for the language that contains the subword abb twice and an even number of a's.

Application: Automatic Theorem Proving

Assume $\Sigma = \{a, b\}$.

Let $\mathcal{L}$ be the set of $x \in \Sigma^*$ such that any a in x is followed by a b.

Let $\mathcal{L}'$ be the set of $x \in \Sigma^*$ such that any b in x is followed by a a.

How to prove that $\mathcal{L} \cap \mathcal{L}' = \{\epsilon\}$?

Intuitively:

- if $x \neq \epsilon$ in $\mathcal{L}$ we have that if $x = \ldots a \ldots$ then it should actually be $x = \ldots a \ldots b \ldots$
- if $x \neq \epsilon$ in $\mathcal{L}'$ we have that if $x = \ldots b \ldots$ then it should actually be $x = \ldots b \ldots a \ldots$

Hence a non-empty word in $\mathcal{L} \cap \mathcal{L}'$ should be infinite.

Application: Automatic Theorem Proving (cont.)

Formally we can automatically prove that $\mathcal{L} \cap \mathcal{L}' = \{\epsilon\}$ with an automaton.

Define a DFA D such that $\mathcal{L}(D) = \mathcal{L}$.

Define a DFA D' such that $\mathcal{L}(D') = \mathcal{L}'$.

Now we can compute $D \times D'$ and check that

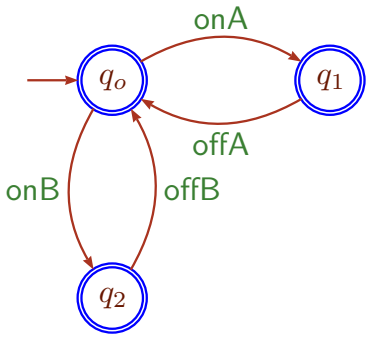
$$\mathcal{L} \cap \mathcal{L}' = \mathcal{L}(D \times D') = \{\epsilon\}$$

Application: Control System

Assume we have several machines working concurrently and that we need to forbid certain sequences of actions.

Example:

If we have two machines MA and MB we may want to make sure that MB cannot be on when MA is on.
The alphabet will contain: onA, offA, onB and offB.



Another condition may be that onA should always appear before onB.

We can take the product of the two automata to express the two conditions as one automaton representing a control system.

Variation of the Product

Definition: We define $D_1 \oplus D_2$ similarly to $D_1 \times D_2$ but with a different notion of accepting state:

$$\text{a state } (r_1, r_2) \text{ is accepting iff } r_1 \in F_1 \textit{ or } r_2 \in F_2$$

Proposition: *Given two DFA D_1 and D_2, then* $\mathcal{L}(D_1 \oplus D_2) = \mathcal{L}(D_1) \cup \mathcal{L}(D_2)$.

Example: We define the automaton accepting multiples of 3 or of 5 by taking $M_3 \oplus M_5$.

Complement

Definition: Given the automaton $D = (Q, \Sigma, \delta, q_0, F)$ we define the *complement* $\overline{D}$ of D as the automaton $\overline{D} = (Q, \Sigma, \delta, q_0, Q - F)$.

Proposition: *Given a DFA D we have that* $\mathcal{L}(\overline{D}) = \Sigma^* - \mathcal{L}(D)$.

Remark: We have that $D_1 \oplus D_2 = \overline{\overline{D_1} \times \overline{D_2}}$.

Regular Languages

Recall: Given an alphabet Σ, a *language* $\mathcal{L}$ is a subset of Σ^*, that is, $\mathcal{L} \subseteq \Sigma^*$.

Definition: A language $\mathcal{L} \subseteq \Sigma^*$ is *regular* iff there exists a DFA D on the alphabet Σ such that $\mathcal{L} = \mathcal{L}(D)$.

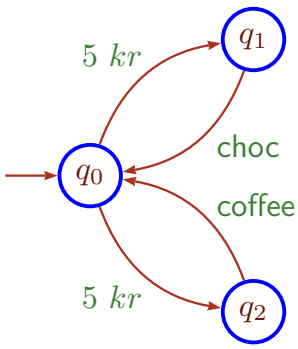
Proposition: *If $\mathcal{L}_1$ and $\mathcal{L}_2$ are regular languages then so are $\mathcal{L}_1 \cap \mathcal{L}_2$, $\mathcal{L}_1 \cup \mathcal{L}_2$ and $\Sigma^* - \mathcal{L}_1$.*

Proof: ...

Non-deterministic Finite Automata

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

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

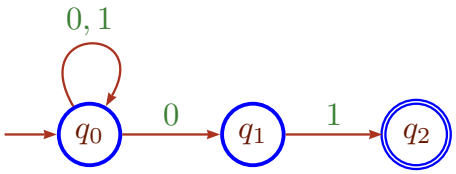


Intuitively, the vending machine can *choose* between different states.

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.

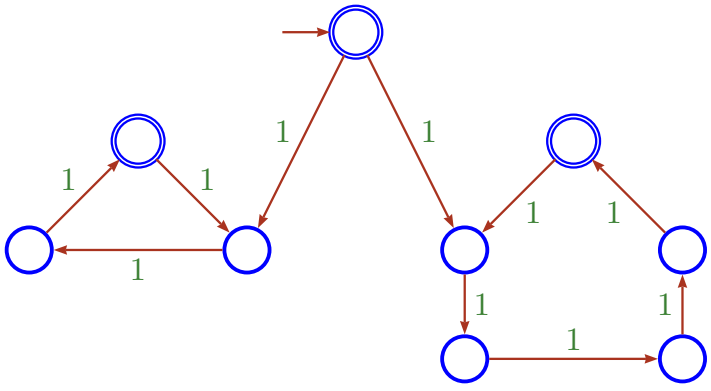


NFA accepting words that end in 01

What are all possible computations for the string 10101?

NFA Accepting Words of Length Divisible by 3 or by 5

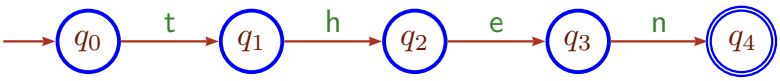
Let $\Sigma = \{1\}$.



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.

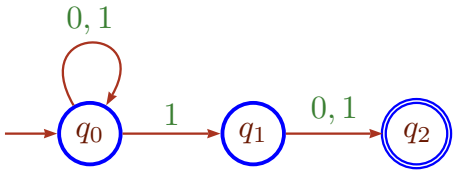
Non-deterministic Finite Automata

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
$\rightarrow q_0$	$\{q_0\}$	$\{q_0, q_1\}$
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:

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

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

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

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.

Note: Again, we could write a program that simulates a NFA and let it 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

An 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

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

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,

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

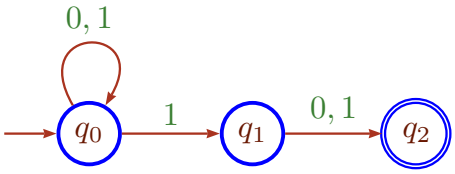
  and

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

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

Example: Subset Construction

Let us convert this NFA into a DFA



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 $\delta_D(\{q_0\}, 1) = \{q_0, q_1\}$.

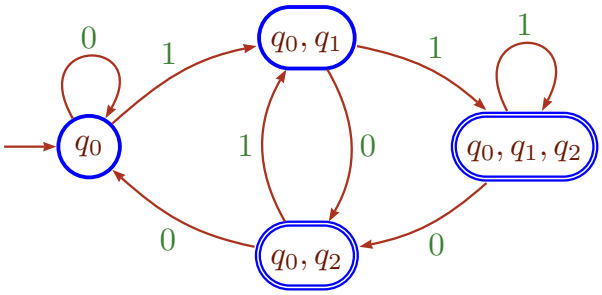
From $\{q_0, q_1\}$, if we get 0, we can go to q_0 or to q_2. Then we get a new state $\{q_0, q_2\}$ and also $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_2\}$.

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.