Finite Automata and Formal Languages

TMV026/DIT321– LP4 2012

Lecture 10
Ana Bove

April 23rd 2012

Overview of today's lecture:

- Equivalence of Regular Languages
- Minimisation of Automata

Testing Equivalence of Regular Languages

There is no simple algorithm for testing this.

We have seen how one can prove that 2 RE are equal, hence the languages they represent are equivalent, but this is not an easy process.

We will see now how to test when 2 DFA describe the same language.

Testing Equivalence of States in DFA

We shall first answer the question: do states p and q behave in the same way?

Definition: We say that states p and q are *equivalent* if for all w, $\hat{\delta}(p, w)$ is an accepting state iff $\hat{\delta}(q, w)$ is an accepting state.

Note: We do not require that $\hat{\delta}(p, w) = \hat{\delta}(q, w)$.

Definition: If p and q are not equivalent, then they are *distinguishable*.

That is, there exists at least one w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is an accepting state and the other is not.

Table-Filling Algorithm

This algorithm finds pairs of states that are distinguishable.
Then, any pair of states that we do not find distinguishable are equivalent.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The algorithm is as follows:

Base case: If p is an accepting state and q is not, the (p, q) are distinguishable.

Inductive step: Let p and q be states such that for some symbol a, $\delta(p, a) = r$ and $\delta(q, a) = s$ with the pair (r, s) known to be distinguishable. Then (p, q) are also distinguishable.

(If w distinguishes r and s then aw must distinguish p and q since $\hat{\delta}(p, aw) = \hat{\delta}(r, w)$ and $\hat{\delta}(q, aw) = \hat{\delta}(s, w)$.)

Example: Table-Filling Algorithm

For the following DFA, we fill the table with an X at distinguishable pairs.

	a	b
$\rightarrow q_0$	q_1	q_2
$*q_1$	q_3	q_4
$*q_2$	q_4	q_3
q_3	q_5	q_5
q_4	q_5	q_5
$*q_5$	q_5	q_5

	q_0	q_1	q_2	q_3	q_4
q_5	X	X	X	X	X
q_4	X	X	X		
q_3	X	X	X		
q_2	X				
q_1	X				

Let us consider the base case of the algorithm.

Let us consider the pair (q_0, q_4).

Let us consider the pair (q_0, q_3).

Finally, let us consider the pairs (q_3, q_4) and (q_1, q_2).

Example: Table-Filling Algorithm

For the following DFA, we fill to table with an X at distinguishable pairs.

	a
$\rightarrow q_0$	q_1
$*q_1$	q_2
q_2	q_3
q_3	q_4
$*q_4$	q_5
q_5	q_0

	q_0	q_1	q_2	q_3	q_4
q_5	X	X		X	X
q_4	X		X	X	
q_3		X	X		
q_2	X	X			
q_1	X				

Let us consider the base case of the algorithm.

Let us consider the pair (q_0, q_5).

Let us consider the pair (q_0, q_2).

We go on with the remaining pairs.

Equivalent States

Theorem: *Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. If 2 states are not distinguishable by the table-filling algorithm then the states are equivalent.*

Proof: Let us assume there is a *bad pair* (p, q) such that p and q are distinguishable but the table-filling algorithm doesn't find them so.

If there are bad pairs, let (p', q') be a bad pair with the shortest string $w = a_1 a_2 \ldots a_n$ that distinguishes 2 states.

Note w is not ϵ otherwise (p', q') is found distinguishable in the base step.

Let $\delta(p', a_1) = r$ and $\delta(q', a_1) = s$. States r and s are distinguished by $a_2 \ldots a_n$ since this string takes r to $\hat{\delta}(p', w)$ and s to $\hat{\delta}(q', w)$.

Now string $a_2 \ldots a_n$ distinguishes 2 states and is shorter than w which is the shortest string that distinguishes a bad pair. Then (r, s) cannot be a bad pair and hence it must be found distinguishable by the algorithm.

Then the inductive part should have found (p', q') distinguishable.

This contradict the assumption that bad pairs exist.

Testing Equivalence of Regular Languages

We can use the table-filling algorithm to test equivalence of regular languages.

Let $\mathcal{L}$ and $\mathcal{M}$ be 2 regular languages.
Let $D_\mathcal{L} = (Q_\mathcal{L}, \Sigma, \delta_\mathcal{L}, q_\mathcal{L}, F_\mathcal{L})$ and $D_\mathcal{M} = (Q_\mathcal{M}, \Sigma, \delta_\mathcal{M}, q_\mathcal{M}, F_\mathcal{M})$ be their corresponding DFA.
Let us assume $Q_\mathcal{L} \cap Q_\mathcal{M} = \emptyset$ (easy to obtain by renaming).

We proceed now as follows.
Construct $D = (Q_\mathcal{L} \cup Q_\mathcal{M}, \Sigma, \delta, -, F_\mathcal{L} \cup F_\mathcal{M})$. (The initial state is irrelevant.)
δ is the union of $\delta_\mathcal{L}$ and $\delta_\mathcal{M}$ as a function.

One should now check if the pair $(q_\mathcal{L}, q_\mathcal{M})$ is equivalent.
If so, a string is accepted by $D_\mathcal{L}$ iff it is accepted by $D_\mathcal{M}$.
Hence $\mathcal{L}$ and $\mathcal{M}$ are equivalent languages.

Recap from Relations, Partitions and Equivalent Classes

Definition: A relation $R \subseteq S \times S$ is an equivalence relation iff R is

Reflexive: $\forall a \in S,\ aRa$

Symmetric: $\forall a, b \in S,\ aRb \Rightarrow bRa$

Transitive: $\forall a, b, c \in S,\ aRb \wedge bRc \Rightarrow aRc$

Definition: A set P is a *partition* over the set S if:

- Every element of P is a non-empty subset of S
- Elements of P are pairwise disjoint
- The union of the elements of P is equal to S

Definition: The *equivalent class* of a in S is $[a] = \{b \in S \mid aRb\}$.

Lemma: $\forall a, b \in S,\ [a] = [b]$ *iff* aRb.

Theorem: *The set of all equivalence classes in S w.r.t R form a partition.*

This partition is called the *quotient* and is denoted as S/R.

Equivalence of States: An Equivalence Relation

The relation "*state p is equivalent to state q*", which we shall denote $p \approx q$, is an equivalence relation. (Prove it as an exercise!)

Reflexive: every state p is equivalent to itself

$$\forall p,\ p \approx p$$

Symmetric: for any states p and q, if p is equivalent to q then q is equivalent to p

$$\forall p\ q,\ p \approx q \Rightarrow q \approx p$$

Transitive: for any states p, q and r, if p is equivalent to q and q is equivalent to r then p is equivalent to r.

$$\forall p\ q\ r,\ p \approx q \wedge q \approx r \Rightarrow p \approx r$$

(See Theorem 4.23 for a proof of the transitivity part.)

Partition of States

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA.
The table-filling algo. defines the "equivalence of states" relation over Q.
Since this is an equivalence relation we can define the quotient $Q/\approx$.
This quotient gives us a partition of the states into classes/blocks of mutually equivalent states.

Example: The partition for the example in slide 4 is the following (note the singleton classes!)

$$\{q_0\} \qquad \{q_1, q_2\} \qquad \{q_3, q_4\} \qquad \{q_5\}$$

Example: The partition for the example in slide 5 is the following

$$\{q_0, q_3\} \qquad \{q_1, q_4\} \qquad \{q_2, q_5\}$$

Note: Classes might also have more than 2 elements.

Minimisation of DFA

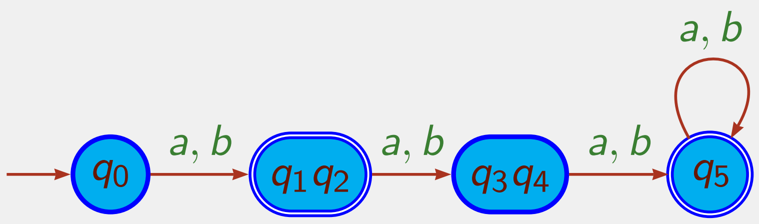
Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA.
$Q/\approx$ allows to build an equivalent DFA with the minimum nr. of states.
In addition, this minimum DFA is unique (modulo the name of the states).

The algorithm for building the minimum DFA $D' = (Q', \Sigma, \delta', q'_0, F')$ is:

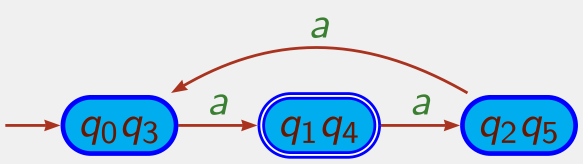
1. Eliminate any non accessible state
2. Partition the remaining states with the help of the table-filling algorithm
3. Use each block as a single state in the new DFA
4. The start state is the block containing q_0, the final states are all those blocks containing elements in F
5. $\delta'(S, a) = T$ if given any $q \in S,\ \delta(q, a) = p$ for some $p \in T$
(Actually, the partition guarantees that $\forall q \in S,\ \exists p \in T,\ \delta(q, a) = p$)

Examples

Example: The minimal DFA corresponding to the DFA in slide 4 is



Example: The minimal DFA corresponding to the DFA in slide 5 is



Does the Minimisation Algorithm Give a Minimal DFA?

Given a DFA D, the minimisation algorithm gives us a DFA D' with the minimal number of states with respect to those of D.
Can you see why?

But, could there exist a DFA A completely unrelated to D, also accepting the same language and with less states than those in D'?

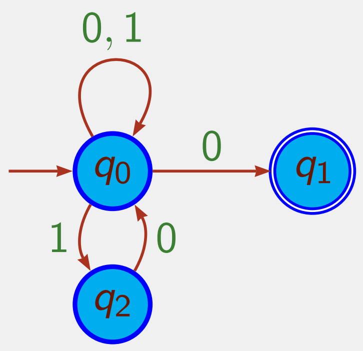
Section 4.4.4 in the book shows by contradiction that A cannot exist.

Theorem: *If D is a DFA and D' the DFA constructed from D with the minimisation algorithm described before, then D' has as few states as any DFA equivalent to D.*

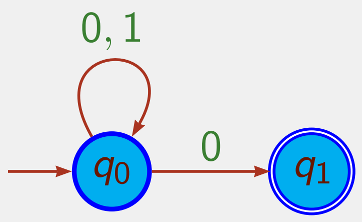
Can we Minimise a NFA?

One can of course find (in some cases) a smaller NFA, but the algorithm we presented before does not do the job.

Example: Consider the following NFA



The table-filling algorithm does not find equivalent states in this case. However, the following is a smaller and equivalent NFA for the language.



Functional Representation of the Minimisation Algorithm

OBS: we only construct the equivalent classes but not the new DFA!

```
data Q = ... deriving (Eq, Show)

data S = ...

delta :: Q -> S -> Q

final :: Q -> Bool


-- lists with all states and all the symbols
states :: [Q]
states = [...]

alphabet :: [S]
alphabet = [...]
```

Func. Representation of the Minimisation Alg. (Cont.)

```
-- swaps elements in a pair
swap :: (a,a) -> (a,a)
swap (p,q) = (q,p)

-- equality in pairs: order doesn't matter here
pair_eq ::  Eq a => (a,a) -> (a,a) -> Bool
pair_eq  p q = p == q || p == swap q

-- remove "repetition" from the list
clean :: Eq a => [(a,a)] -> [(a,a)]
clean = nubBy pair_eq

-- maps a function to the elements of the pair
map_pair :: (a -> b -> c) -> (a,a) -> b -> (c,c)
map_pair f (p,q) x = (f p x, f q x)
```

Func. Representation of the Minimisation Alg. (Cont.)

```
-- all possible pairs of states without repetition
all_pairs :: [(Q,Q)]
all_pairs = [ (p,q) | p <- states, q <- states, p /= q]


-- tests to distinguish if a state is final but not the other
dist :: (Q,Q) -> Bool
dist (p,q) = final p && not(final q) ||
             final q && not(final p)


--splits a list according to whether the elem. satisfies dist
splitBy :: [(Q,Q)] -> ([(Q,Q)],[(Q,Q)])
splitBy [] = ([],[])
splitBy (p:pps) = let (qs,ds) = splitBy pps
                  in if dist p then (qs,p:ds) else (p:qs,ds)
```

Func. Representation of the Minimisation Alg. (Cont.)

```
-- equiv ss pps dds gives the equivalent states
-- ss: all the symbols in the alphabet
-- pps: the pairs of states still to check if distinguishable
-- dds: all pairs of states already found distinguishable
equiv :: [S] -> [(Q,Q)] -> [(Q,Q)] -> [(Q,Q)]
equiv ss pps dds =
   let dqs = [ pq | pq <- pps,
                       or (map (\pp -> elem pp dds)
                            (map (map_pair delta pq) ss))]
       nds = union dds dqs
       nps = pps \\ dqs
   in if not (null dqs)
         then equiv ss nps nds
         else clean pps

-- if we return (clean nds) we give all distinguishable pairs
```

Func. Representation of the Minimisation Alg. (Cont.)

```
-- group the pairs into classes
group_classes :: Eq a => [(a,a)] -> [[a]]
group_classes [] = []
group_classes ((p,q):pps) =
   let pqs = (p,q):[ pr | pr <- pps,
                             (fst pr == p || fst pr == q |
                              snd pr == p || snd pr == q)]
       nps = pps \\ pqs
       qs = nub ([fst p | p <- pqs] ++ [snd p | p <- pqs])
  in qs : group_classes nps
```

Func. Representation of the Minimisation Alg. (Cont.)

```
-- add the classes with just one state
add_singel :: Eq a => [a] -> [[a]] -> [[a]]
add_singel [] pps = pps
add_singel (q:qs) pps | or (map (elem q) pps)
                      = add_singel qs pps
add_singel (q:qs) pps = [q] : add_singel qs pps


-- gives the base case of the test-filling algo. and the rest
(rest,base_dist) = splitBy all_pairs

-- returns all equivalent classes
equiv_classes :: [[Q]]
equiv_classes =
  add_singel states
             (group_classes (equiv alphabet rest base_dist))
```