Finite Automata and Formal Languages

TMV027/DIT321– LP4 2013

Lecture 9

Ana Bove

April 25th 2013

Overview of today's lecture:

- Decision Properties for RL;
- Equivalence of RL;
- Minimisation of automata.

Decision Properties of Regular Languages

We want to be able to answer YES/NO to questions such as

- Is this language empty?
- Is string w in the language $\mathcal{L}$?
- Are these 2 languages equivalent?

In general languages are infinite so we cannot do a "manual" checking.

Instead we should work with the finite description of the languages (DFA, NFA. ϵ-NFA, RE).

Which description is the most convenient depends on the property and on the language.

Testing Emptiness of Regular Languages given FA

Given a FA for a language, testing whether the language is empty or not amounts to checking if there is a path from the start state to a final state.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

Recall the notion of *accessible states*: $\text{Acc} = \{\hat{\delta}(q_0, x) \mid x \in \Sigma^*\}$.

Proposition: *Given D as above, then $D' = (Q \cap \text{Acc}, \Sigma, \delta|_{Q \cap \text{Acc}}, q_0, F \cap \text{Acc})$ is a DFA such that $\mathcal{L}(D) = \mathcal{L}(D')$.*

In particular, $\mathcal{L}(D) = \emptyset$ if $F \cap \text{Acc} = \emptyset$.

(Actually, $\mathcal{L}(D) = \emptyset$ iff $F \cap \text{Acc} = \emptyset$ since if $\hat{\delta}(q_0, x) \in F$ then $\hat{\delta}(q_0, x) \in F \cap \text{Acc}$.)

Testing Emptiness of Regular Languages given FA

A recursive algorithm to test whether a state is accessible/reachable is as follows:

Base case: The start state q_0 is reachable from q_0.

Recursive step: If q is reachable from q_0 and there is an arc from q to p (with any label, including ϵ) then p is also reachable from q_0.

(This algorithm is an instance of *graph-reachability*.)

If the set of reachable states contains at least one final state then the RL is NOT empty.

Exercise: Program this!

Testing Emptiness of Regular Languages given RE

Given a RE for the language we can instead perform the following test:

Base cases: $\emptyset$ denotes the empty language while ϵ and a (any symbol from the alphabet) do not.

Inductive step: Let R be our RE.

- If $R = R_1 + R_2$ then $\mathcal{L}(R)$ is empty iff both $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ are empty;
- If $R = R_1 R_2$ then $\mathcal{L}(R)$ is empty iff either $\mathcal{L}(R_1)$ or $\mathcal{L}(R_2)$ is empty;
- If $R = R_1^*$ is never empty since it always contains the word ϵ.

Functional Representation of Testing Emptiness for RE

```
data RExp a = Empty | Epsilon | Atom a |
              Plus (RExp a) (RExp a) |
              Concat (RExp a) (RExp a) |
              Star (RExp a)


isEmpty :: RExp a -> Bool
isEmpty Empty = True
isEmpty (Plus e1 e2) = isEmpty e1 && isEmpty e2
isEmpty (Concat e1 e2) = isEmpty e1 || isEmpty e2
isEmpty _ = False
```

Testing Membership in Regular Languages

Given a RL $\mathcal{M}$ and a word w over the alphabet of $\mathcal{M}$, is $w \in \mathcal{M}$?

When $\mathcal{L}$ is given by a FA we can simply run the FA with the input w and see if the word is accepted by the FA.

We have seen an algorithm simulating the running of a DFA (and you have implemented algorithms simulating the running of NFA and ϵ-NFA :-).

Using *derivatives* (see exercises 4.2.3 and 4.2.5) there is a nice algorithm checking membership on RE.

Let $\mathcal{M} = \mathcal{L}(R)$ and $w = a_1 \dots a_n$.

Let $a\backslash R = D_a R = \{x \mid ax \in \mathcal{M}\}$ (in the book $\frac{d\mathcal{M}}{da}$).

$D_w R = D_{a_n}(\dots(D_{a_1} R)\dots)$.

It can then be shown that $w \in \mathcal{M}$ iff $\epsilon \in D_w R$.

Other Testing Algorithms on Regular Expressions

Tests if a RE contains ϵ.

```
hasEpsilon :: RExp a -> Bool
hasEpsilon Epsilon = True
hasEpsilon (Star _) = True
hasEpsilon (Plus e1 e2) = hasEpsilon e1 || hasEpsilon e2
hasEpsilon (Concat e1 e2) = hasEpsilon e1 && hasEpsilon e2
hasEpsilon _ = False
```

Other Testing Algorithms on Regular Expressions

Tests if $\mathcal{L}(R) \subseteq \{\epsilon\}$.

```
atMostEps :: RExp a -> Bool
atMostEps Empty = True
atMostEps Epsilon = True
atMostEps (Atom _) = False
atMostEps (Plus e1 e2) = atMostEps e1 && atMostEps e2
atMostEps (Concat e1 e2) = isEmpty e1 || isEmpty e2 ||
                           (atMostEps e1 && atMostEps e2)
atMostEps (Star e) = atMostEps e
```


Other Testing Algorithms on Regular Expressions

Tests if a regular expression denotes an infinite language.

```
infinite :: RExp a -> Bool
infinite (Star e) = not (atMostEps e)
infinite (Plus e1 e2) = infinite e1 || infinite e2
infinite (Concat e1 e2) = (infinite e1 && notIsEmpty e2) ||
                          (notIsEmpty e1 && infinite e2)
  where notIsEmpty e = not (isEmpty e)
infinite _ = False
```

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).

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 $w = a_1 a_2 \dots a_n$ 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 $a_2 \dots a_n$ since this string takes r to $\hat{\delta}(p', w)$ and s to $\hat{\delta}(q', w)$.

Now string $a_2 \dots a_n$ 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 step should have found (p', q') distinguishable.

This contradicts 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 $\mathcal{M}$ and $\mathcal{N}$ be 2 regular languages.
Let $D_{\mathcal{M}} = (Q_{\mathcal{M}}, \Sigma, \delta_{\mathcal{M}}, q_{\mathcal{M}}, F_{\mathcal{M}})$ and $D_{\mathcal{N}} = (Q_{\mathcal{N}}, \Sigma, \delta_{\mathcal{N}}, q_{\mathcal{N}}, F_{\mathcal{N}})$ be their corresponding DFA.

Let us assume $Q_{\mathcal{M}} \cap Q_{\mathcal{N}} = \emptyset$ (easy to obtain by renaming).

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

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

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 13 is the following (note the singleton classes!)

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

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.

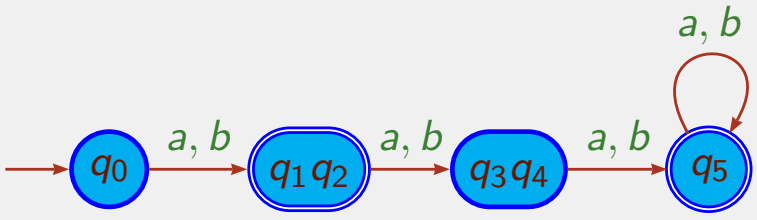
In addition, this minimum DFA is unique (modulo the name of the states).

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 $\forall q \in S.\ \exists p \in T.\ \delta(q, a) = p$.)

Example

Example: The minimal DFA corresponding to the DFA in slide 13 is



Exercise: Program the minimisation algorithm!

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'?

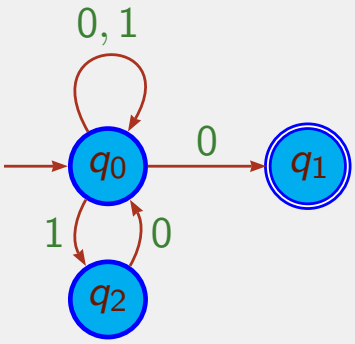
Section 4.4.4 in the book shows by contradiction that A cannot exist.

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?

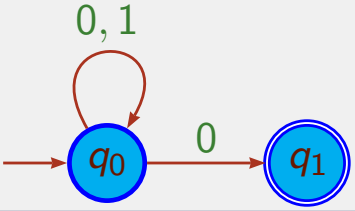
One can of course find (in some cases) a smaller NFA, but the algorithm we presented before does not do the job.

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.



Overview of Next Lecture

Guest lecture by *Martin Fabian*

Supervisory Control Theory – A Practical Application of Automata Theory

and sections 5–5.2:

- Context free grammars;
- Derivations and parse trees.