Finite Automata Theory and Formal Languages

TMV027/DIT321– LP4 2017

Lecture 10
Ana Bove

April 24th 2017

Overview of today's lecture:

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

Recap: Regular Languages

- We can convert between FA and RE;
- Hence both FA and RE accept/generate regular languages;
- We use the Pumping lemma to show that a language is NOT regular;
- RL are closed under:
  - Union, complement, intersection, difference, concatenation, closure;
  - Prefix, reversal;
- Closure properties can be used both to prove that a language IS regular or that a language is NOT regular.

Decision Properties of Regular Languages

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

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

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

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

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

Testing Membership in Regular Languages

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

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, right? :-).

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 $w = a_1 \ldots a_n$.

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

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

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

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 define the function:

$$
\begin{aligned}
&\mathit{isEmpty} : \mathit{RE} \to \text{Bool} \\
&\mathit{isEmpty}(\emptyset) = \text{True} \\
&\mathit{isEmpty}(\epsilon) = \text{False} \\
&\mathit{isEmpty}(a) = \text{False} \\
&\mathit{isEmpty}(R_1 + R_2) = \mathit{isEmpty}(R_1) \wedge \mathit{isEmpty}(R_2) \\
&\mathit{isEmpty}(R_1 R_2) = \mathit{isEmpty}(R_1) \vee \mathit{isEmpty}(R_2) \\
&\mathit{isEmpty}(R^*) = \text{False}
\end{aligned}
$$

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
```

Other Testing Algorithms on Regular Expressions

Tests if a RE generates ϵ.

$$
\begin{aligned}
&hasEpsilon : RE \to \text{Bool}\\
&hasEpsilon(\emptyset) = \text{False}\\
&hasEpsilon(\epsilon) = \text{True}\\
&hasEpsilon(a) = \text{False}\\
&hasEpsilon(R_1 + R_2) = hasEpsilon(R_1) \vee hasEpsilon(R_2)\\
&hasEpsilon(R_1 R_2) = hasEpsilon(R_1) \wedge hasEpsilon(R_2)\\
&hasEpsilon(R^*) = \text{True}
\end{aligned}
$$

Other Testing Algorithms on Regular Expressions

Tests if R generates at most ϵ: $\mathcal{L}(R) \subseteq \{\epsilon\}$.

$$
\begin{aligned}
&atMostEps : RE \to \text{Bool}\\
&atMostEps(\emptyset) = \text{True}\\
&atMostEps(\epsilon) = \text{True}\\
&atMostEps(a) = \text{False}\\
&atMostEps(R_1 + R_2) = atMostEps(R_1) \wedge atMostEps(R_2)\\
&atMostEps(R_1 R_2) = isEmpty(R_1) \vee isEmpty(R_2) \vee\\
&\qquad\qquad (atMostEps(R_1) \wedge atMostEps(R_2))\\
&atMostEps(R^*) = atMostEps(R)
\end{aligned}
$$

Other Testing Algorithms on Regular Expressions

Tests if a regular expression generates an infinite language.

$$
\begin{aligned}
&\mathit{infinite} : \mathit{RE} \to \mathrm{Bool}\\
&\mathit{infinite}(\emptyset) = \mathrm{False}\\
&\mathit{infinite}(\epsilon) = \mathrm{False}\\
&\mathit{infimote}(a) = \mathrm{False}\\
&\mathit{infinite}(R_1 + R_2) = \mathit{infinite}(R_1) \vee \mathit{infinite}(R_2)\\
&\mathit{infinite}(R_1 R_2) = (\mathit{infinite}(R_1) \wedge \neg(\mathit{isEmpty}(R_2)) \vee\\
&\qquad\qquad\qquad\quad (\neg(\mathit{isEmpty}(R_1)) \wedge \mathit{infinite}(R_2))\\
&\mathit{infinite}(R^*) = \neg(\mathit{atMostEps}(R))
\end{aligned}
$$

Testing Equivalence of Regular Languages

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

How to answer the question *"do states p and q behave in the same way"*?

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

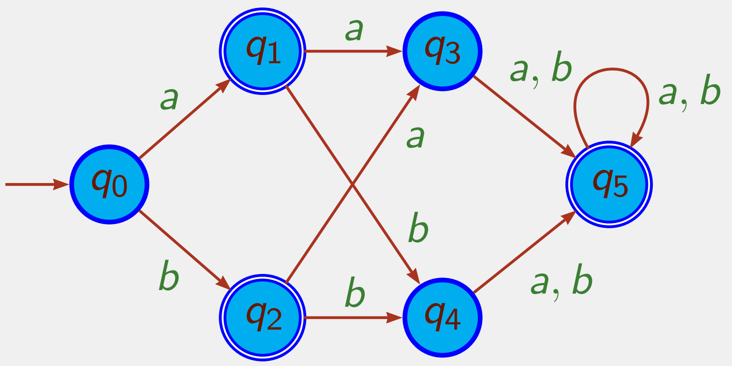
Note: We do not require that $\hat{\delta}(p, w) = \hat{\delta}(q, w)$!

Definition: If p and q are <u>not</u> 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.

Example: Identifying Distinguishable Pairs

Let us find the distinguishable pairs in the following DFA.



	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				

If p is accepting and q is not, then the word ϵ distinguish them.

$\delta(q_1, a) = q_3$ and $\delta(q_5, a) = q_5$. Since (q_3, q_5) is distinguishable so must be (q_1, q_5).

What about $\delta(q_2, a)$ and $\delta(q_5, a)$?

What about the pairs (q_0, q_3) and (q_0, q_4) with the input a?

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

Table-Filling Algorithm

This algorithm finds pairs of states that are distinguishable.

Any 2 states that we do not find distinguishable are equivalent (see slide 16).

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

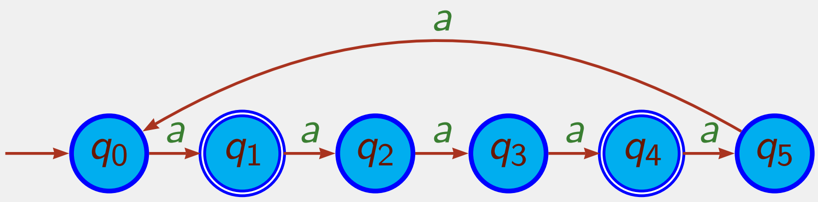
Base case: If p is an accepting state and q is not, then (p, q) are distinguishable.

Recursive step: Let p and q be such that for some a, $\delta(p, a) = r$ and $\delta(q, a) = s$ with (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

Let us fill the table of distinguishable pairs in the following DFA.



	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).

Let us consider (q_2, q_3) and (q_3, q_5).

Finally, let us consider 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 step should have found (p', q') distinguishable.

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*", denoted $p \approx q$, is an *equivalence relation*.

Reflexive: $\forall p.\ p \approx p$;

Symmetric: $\forall p\ q.\ p \approx q \Rightarrow q \approx p$;

Transitive: $\forall p\ q\ r.\ p \approx q \wedge q \approx r \Rightarrow p \approx r$.

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

Exercise: Prove these properties!

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.

This is an equivalence relation so we can define the quotient $Q/\approx$.

This quotient gives us a *partition* into classes of mutually equivalent states.

Example: The partition for the example in slide 13 is the following (note the singleton classes!)

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

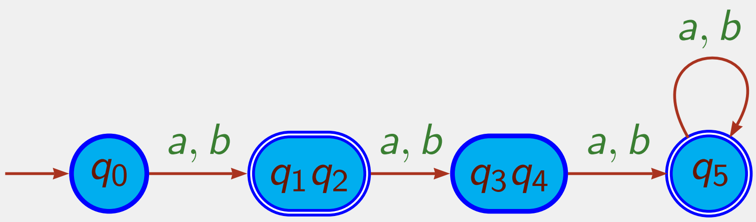
Example: The partition for the example in slide 15 is the following

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

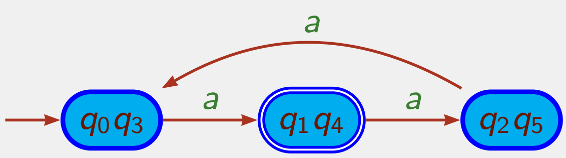
Note: Classes might also have more than 2 elements.

Example: Minimisation of DFA

How to use the partition into equivalent states to minimise the DFA in slide 13?



Example: The minimal DFA corresponding to the DFA in slide 15 is



Exercise: Program the minimisation algorithm!

Minimisation of DFA: The Algorithm

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.

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 using the table-filling algorithm;
3. Use each class as a single state in the new DFA;
4. The start state is the class containing q_0;
5. The final states are all those classes containing elements in F;
6. $\delta'(S, a) = T$ if given any $q \in S$, $\delta(q, a) = p$ for some $p \in T$.
   (Actually, the partition guarantees that $\forall q \in S.\ \exists p \in T.\ \delta(q, a) = p$.)

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.

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

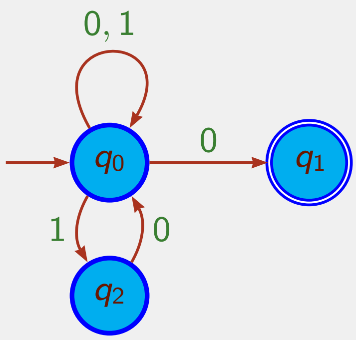
Section 4.4.4 in the book shows by contradiction that A cannot exist.

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?

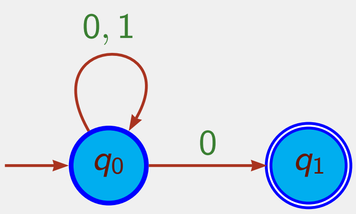
One could find a smaller NFA, but not with this algorithm.

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.



Learning Outcome of the Course (revisited)

After completion of this course, the student should be able to:

- Explain and manipulate the different concepts in automata theory and formal languages;
- Have a clear understanding about the equivalence between (non-)deterministic finite automata and *regular expressions*;
- Acquire a *good* understanding of the power and the limitations of regular languages and context-free languages;
- Prove properties *of languages*, grammars and *automata* with rigorously formal mathematical methods;
- Design automata, *regular expressions* and context-free grammars accepting or generating a certain language;
- Describe the language accepted by an automata, or *generated by a regular expression* or a context-free grammar;
- *Simplify automata* and context-free grammars;
- Determine if a certain word belongs to a language;
- Define Turing machines performing simple tasks;
- Differentiate and manipulate formal descriptions of *languages*, automata and grammars.

Overview of Next Lecture

Sections 5–5.2.2:

- Context-free grammars;
- Derivations;
- Parse trees;
- Proofs in grammars.