DFA minimisation using the Myhill-Nerode theorem

Johanna Högberg Lars Larsson

Abstract

The Myhill-Nerode theorem is an important characterisation of regular languages, and it also has many practical implications. In this chapter, we introduce the theorem an present its proof. One consequence of the theorem is an algorithm for minimising DFAs that is outlined in the latter part of this paper. To clarify how the algorithm works, we conclude with an example of its application.

1 Introduction

This chapter gives an introduction to DFA minimisation using the Myhill-Nerode theorem. DFA minimisation is an important topic because it can be applied both theoretically and practically, in for instance compilers. Minimising a DFA increases its efficiency by reducing its amount of states and it also enables us to determine if two DFAs are equivalent.

The chapter is structured as follows. Following this introduction in Section 2, some relevant background information and definitions are stated, which will be used in the subsequent sections. In Section 3, the Myhill-Nerode theorem is stated and its proof presented. Section 4 contains the algorithm for DFA minimisation that uses the Myhill-Nerode theorem. An example showing how such a minimisation might be conducted in a specific case is found in Example 4.2.

2 Background information and definitions

The reader is advised to skim through this section to recall, at least at an intuitive level, the terms that will be used in this chapter. Following these intuitive descriptions, more formal definitions are provided.

A *deterministic finite automaton*, or DFA for short, may be considered a kind of very basic computational device. It processes strings of symbols of a certain alphabet and yields a result that is said to be either accepting or rejecting. Normally, an accepting result means that the string is part of a *recognised language*. The language is then defined as all strings that bring the DFA into an accepting state.

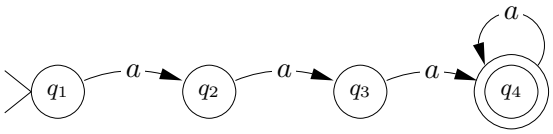

Figure 1: DFA accepting the language $a^n, n > 2$

Definition 2.1 (DFA) A deterministic finite automaton (DFA) A is a quintuple $A = (S, \Sigma, \delta, s, F)$ consisting of:

- a finite set of states S,
- a finite set of symbols Σ, called the alphabet,
- a transition function $\delta : S \times \Sigma \mapsto S$,
- a starting state $s \in S$, and
- a set of accepting states $F \subseteq S$.

We extend the definition of δ to the domain of Σ^* by letting

1. $\delta(q, \lambda) = q$, where λ denotes the empty string, and
2. $\delta(q, aw) = \delta(\delta(q, a), w)$, where $a \in \Sigma$ and $w \in \Sigma^*$.

Finally, A is said to *accept* the string $w \in \Sigma^*$ if $\delta(q_0, w) \in F$.

DFA are commonly represented by *state diagrams*, such as in Figure 1 and Figure 2. The states that are accepting have a double border, the initial state is the state with the angle on the left and the arrows or *transitions* signify what state should be entered after reading the symbol on the arrow.

There are some limitations as to what kind of language the DFA can recognise. The number of symbols in the alphabet as well as the number of states in the DFA has to be finite. This means that while it is entirely possible to construct a DFA that accepts strings that for instance look like $a^n, n > 2$ (let the starting state of the DFA be followed by two rejecting states and finally by a state that accepts any number of a's, see Figure 1) it is *not* possible to recognise a language such as $a^n b^n, n \in \mathbb{N}$ (the symbol a repeated n times, followed by the exact same number of b's) because that would require infinitely many states to keep track of the number of a's to ensure that the same number of b's are read. See Figure 2 for an example of how a DFA accepting the language $a^n b^n, n \leq 2$ might be constructed. Note the obvious difficulty that arises if n may be arbitrarily large as well as the difference between this language and a^n.

The class of languages that can be recognised by a DFA are called *regular languages* or Type-3 languages in the Chomsky hierarchy. Every regular language can be recognised by a read-only Turing machine, described by regular expression, generated by a regular grammar, and defined in a so-called monadic second-order logic.

As far as this chapter is concerned, the formal definition including syntax for regular languages has no useful meaning. Instead, it is sufficient to define regular languages as follows.

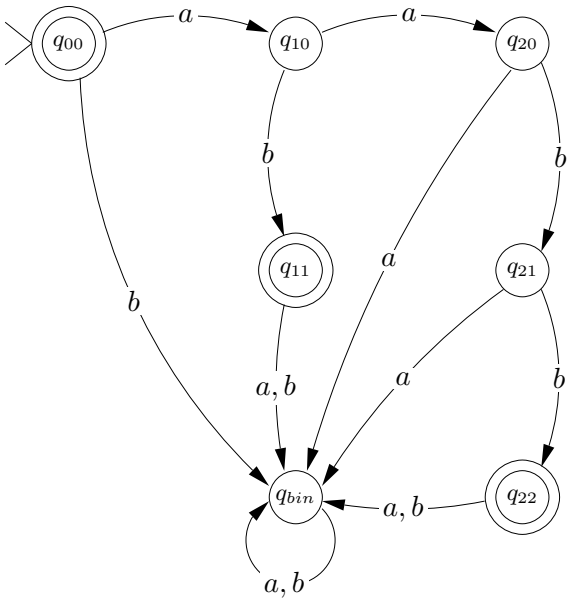

Figure 2: DFA accepting the language $a^n b^n, n \leq 2$

Definition 2.2 (regular language) The class of languages that can be recognised by a deterministic finite automaton.

The Myhill-Nerode theorem that will provide the theoretical background that allows us to minimise a DFA requires the following definitions that are related to equivalence relations.

Definition 2.3 (equivalence relation) An equivalence relation, $\sim$, is a binary relation on a set if it is

- Reflexive ($a \sim a$)
- Symmetric (if $a \sim b$ then $b \sim a$)
- Transitive (if $a \sim b$ and $b \sim c$ then $a \sim c$)

Definition 2.4 (equivalence classes, index) An equivalence relation $\sim$ on a set S imposes a partition $S/\sim$ on S, such that elements $e, e' \in S$ are in the same part of $S/\sim$ if $e \sim e'$. The parts of $S/\sim$ are called *equivalence classes*, and we write $[e]_\sim$ to denote the unique equivalence class of $S/\sim$ to which e belongs. The *index* of equivalence relation $\sim$ is $|S/\sim|$.

Definition 2.5 (right invariant) An equivalence relation $\sim$ on strings of symbols from some alphabet Σ is said to be *right invariant* if $\forall x, y \in \Sigma^*$ with $x \sim y$ and all $w \in \Sigma^*$ we have that $xw \sim yw$.

This last definition states that an equivalence relation has the right invariant property if two equivalent strings (x and y) that are in the language still are equivalent if a third string (w) is appended to the right of both of them.

In the rest of the chapter, the following two equivalence relations will be of particular importance.

Definition 2.6 (equivalence relation for DFA:s) Let A be a DFA. For strings $x, y \in \Sigma^*$, let $x \sim_A y$ iff the *same state* in the DFA is reached from the initial state by reading the strings x and y.

Definition 2.7 (equivalence relation for languages) Let L be any language over Σ. For strings $x, y \in \Sigma^*$, let $x \sim_L y$ iff $\forall w \in \Sigma^*$ we have that $xw \in L \iff yw \in L$.

Definition 2.8 (distinguishing string) Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The string $w \in \Sigma^*$ is a *distinguishing string* for states $p, q \in Q$ if exactly one of $\delta(p, w)$ and $\delta(q, w)$ is in F.

Intuitively, a distinguishing string s for states p and q in an automaton A is string which is mapped to an accepting state from exactly one of p and q. In other words, the existence of a distinguishing string for a pair of states $p, q \in Q$ proves that p and q are not equivalent.

3 Myhill-Nerode theorem

In this section, the Myhill-Nerode theorem [Ner58] and its proof are presented.

Theorem 3.1 Let $L \subseteq \Sigma^*$. Then the following statements are equivalent:

1. There is a DFA that accepts L (L is regular).
2. There is a right-invariant equivalence relation $\sim$ of finite index such that L is the union of some of the equivalence classes of $\sim$.
3. $\sim_L$ is of finite index.

Proof The proof that we give here is an adaption of that in [Vis05], which uses the implication order (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (1).

(1) $\Rightarrow$ (2). Since L is a regular language, there exits a DFA $A = (Q, \Sigma, \delta, q_0, F)$ that recognises L. Recall that $\sim_A$ is a right-invariant relation of finite index. It remains to observe that

$$L = \bigcup_{x:\delta(q_0,x)\in F} [x] \ ,$$

where $x \in \Sigma^*$.

(2) $\Rightarrow$ (3). Assume that L is the union of some equivalence classes of a right-invariant relation $\sim$ of finite index. Now, if (the partition imposed by) $\sim$ is a refinement of (the partition imposed by) $\sim_L$, then $\sim_L$ must be of finite index since it has fewer equivalence classes than $\sim$, which by assumption is of finite index. As we shall see, $\sim$ is indeed a refinement of $\sim_L$.

Since $\sim$ is right-invariant, we have that for every pair $x, y \in \Sigma$ such that $x \sim y$, it must hold that $\forall(z)\, xz \sim yz$. Moreover, we have that L is the union of some

equivalence classes of $\sim$, so if $x \sim y$, then $x \in L \leftrightarrow y \in L$. Combining these implications gives us

$$x \sim y \;\Rightarrow\; \forall(z)\, xz \sim yz \;\Rightarrow\; \forall(z)\, xz \in L \leftrightarrow yz \in L \;\Rightarrow\; x \sim_L y.$$

We now know that $x \sim y$ implies $x \sim_L y$, and draw the conclusion that $\sim$ is a refinement of $\sim_L$.

(3) $\Rightarrow$ (1). To prove that if $\sim_L$ is of finite index, then L is regular, it suffices to construct, for an arbitrary $\sim_L$, a DFA A which recognises L. The idea that underlies the construction is to use the equivalence classes of $\sim_L$ as states in A. First, we choose $x_1, \ldots, x_k$ as representatives for the k equivalence classes of $\sim_L$, and then assemble the DFA $A = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{[x_1], \ldots, [x_k]\}$,
- $\delta([x], a) = [xa]$,
- $q_0 = [\lambda]$, where λ denotes the empty string, and
- $F = \{[x] \mid x \in L\}$.

It should be obvious that δ is well defined and that $\delta(q_0, x) \in F$ iff $x \in L$. ∎

Please note that the Myhill-Nerode theorem lets us decide whether a language L is regular, by determining if either of Condition 2 or Condition 3 is met.

4 DFA minimisation

Not only does the Myhill-Nerode Theorem provide an informative characterisation of the recognisable languages, but it also suggests an algorithm for minimising DFAs. Before we attempt to describe this algorithm, we wish to make a few definitions that will shorten and simplify the later discussion.

Definition 4.1 (partition, block) Let S be a set. A (finite) *partition* of S is a set P of mutually disjoint non-empty sets $\{S_1, \ldots, S_k\}$, such that

- $S_i \subseteq S$, where $1 \leq i \leq k$, and
- the union of the sets in the partition equals the whole of S. Formally,

$$\bigcup_{i=1}^{k} S_i = S \;\;.$$

The components $S_1, \ldots, S_k$ are called the *blocks* or *parts* of P. If $s \in S$, then we denote by $[s]_P$ the unique block of P to which s belongs. When P is not important, or follows from the context, we omit the subscript of $[s]_P$.

In the proof of Theorem 3.1, it is established that if L is a regular language, and the index of $\sim_L$ is $i_L \in \mathbb{N}$, then it is both necessary and sufficient for a DFA to have i_L states in order to recognise L correctly. Now, let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognising the regular language $L(A)$. Furthermore, let $[q]$, where $q \in Q$, be the set of strings $\{x \mid x \in \Sigma^* \wedge \delta(q_0, x) = q\}$. We note that $\{[q] \mid q \in Q\}$

partitions Σ^*. We observe that for every $q \in Q$, and $x, y \in [q]$, the equivalence $x \sim_{L(A)} y$ holds and draw the conclusion that the partition imposed by Q is a refinement of the partition imposed by $\sim_{L(A)}$. This leads us to believe that if the the number of states of an automaton A exceeds the number of equivalence classes of $\sim_{L(A)}$, then the automaton can be reduced in size without altering the language that it recognises.

A first attempt to minimise A could be to find, for each equivalence class C in the partition of Σ^* that follows from $\sim_{L(A)}$, the subset Q' of Q such that $\cup_{q' \in Q'}[q'] = C$ and merge the states of Q' into a single state. The problem with this approach is that to be absolutely sure that two states p and q can be merged, we must examine $\delta(p, w)$ and $\delta(q, w)$ for *every* string in $\{w \mid |w| \leq |Q|\} \subseteq \Sigma^*$. A more efficient method is to start by assuming that all accepting states can be merged into one, and all rejecting into another, and then refine this initial partition until it is stable. By stable, we mean that if states p and q both belong to block B, then for all $\sigma \in \Sigma$, the state p goes to block B' on σ, if and only if q goes to B' on σ. If the refinement process is allowed to continue long enough, then the partition will eventually become a set of singletons, in which case it is stable and termination guaranteed.

The minimisation algorithm that stems from the latter method is listed in Algorithm 1 on page 10. Although it is obvious that a minimal automaton may not contain unreachable states, the partitioning procedure alone will not purge such states from the input automaton A. Instead, the removal of unreachable states is done in RemoveUnreachableStates, while Collapse merges the states of A in accordance with the partitioning produced by RefinePartition.

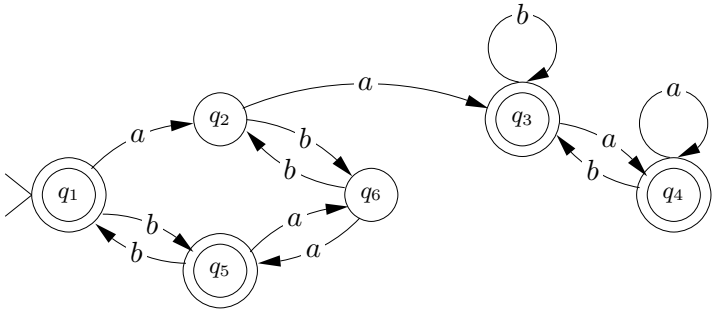

Figure 3: The DFA that is given as input to Algorithm 1 in Example 4.2.

Example 4.2 We shall now illustrate how RefinePartition of Algorithm 1 finds the coarsest stable partition with a running example. The input DFA with alphabet $\Sigma = \{a, b\}$ is given in Figure 3, and the reader should note that all of its states are reachable from the initial state q_1. The algorithm begins with the trivial partition,

$$P_0 = \{\mathrm{I}, \mathrm{II}\}, \text{ where } \mathrm{I} = \{q_1, q_3, q_4, q_5\}, \text{ and } \mathrm{II} = \{q_2, q_6\} \ ,$$

and determines, for each $q \in Q$ and $\sigma \in \Sigma$, the block $[\delta(q, \sigma)]_{P_0}$.

$$
\begin{array}{lll}
\text{I:} & [\delta(q_1,a)] = [q_2] = \text{II} & [\delta(q_1,b)] = [q_5] = \text{I} \\
& [\delta(q_3,a)] = [q_4] = \text{I} & [\delta(q_3,b)] = [q_3] = \text{I} \\
& [\delta(q_4,a)] = [q_4] = \text{I} & [\delta(q_4,b)] = [q_3] = \text{I} \\
& [\delta(q_5,a)] = [q_6] = \text{II} & [\delta(q_5,b)] = [q_1] = \text{I} \\
\\
\text{II:} & [\delta(q_2,a)] = [q_3] = \text{I} & [\delta(q_2,b)] = [q_6] = \text{II} \\
& [\delta(q_6,a)] = [q_5] = \text{I} & [\delta(q_6,b)] = [q_2] = \text{II}
\end{array}
$$

The partition P_0 is not stable, as q_1 and q_5 goes to a rejecting state on a, while q_3 and q_4 goes to an accepting state on the same input symbol. As the states in II behave identically on both input symbols, only I has to be refined, yielding the next partition in sequence

$$P_1 = \{\text{I, II, III}\}, \text{ where I} = \{q_1, q_5\}, \text{ II} = \{q_2, q_6\}, \text{ and III} = \{q_3, q_4\} \ .$$

P_1 is also unstable, which the algorithm determines by examining the behaviour of all $q \in Q$ for all input symbols.

$$
\begin{array}{lll}
\text{I:} & [\delta(q_1,a)] = [q_2] = \text{II} & [\delta(q_1,b)] = [q_5] = \text{I} \\
& [\delta(q_5,a)] = [q_6] = \text{II} & [\delta(q_5,b)] = [q_1] = \text{I} \\
\\
\text{II:} & [\delta(q_2,a)] = [q_3] = \text{III} & [\delta(q_2,b)] = [q_6] = \text{II} \\
& [\delta(q_6,a)] = [q_5] = \text{I} & [\delta(q_6,b)] = [q_2] = \text{II} \\
\\
\text{III:} & [\delta(q_3,a)] = [q_4] = \text{III} & [\delta(q_3,b)] = [q_3] = \text{III} \\
& [\delta(q_4,a)] = [q_4] = \text{III} & [\delta(q_4,b)] = [q_3] = \text{III}
\end{array}
$$

In this iteration, the only states that belong to the same block and disagrees are q_2 and q_6 of block II. After II is divided into $\{q_2\}$ and $\{q_6\}$, partition

$$
\begin{array}{l}
P_2 = \{\text{I}, \text{II}, \text{III}, \text{IV}\}, \\
\text{where I} = \{q_1, q_5\}, \text{ II} = \{q_2\}, \text{ III} = \{q_3, q_4\} \text{ and IV} = \{q_6\}
\end{array}
$$

remains. When deciding whether P_2 is stable, the algorithm could in theory take a shortcut and only examine block I, as it contains all states that were mapped to block III_{P_1} in the previous iteration, and is hence the only block that could have been affected by the refinement of III_{P_1}. Indeed, it turns out that I has indeed become inconsistent.

$$
\begin{array}{lll}
\text{I:} & [\delta(q_1,a)] = [q_2] = \text{II} & [\delta(q_1,b)] = [q_5] = \text{I} \\
& [\delta(q_5,a)] = [q_6] = \text{IV} & [\delta(q_5,b)] = [q_1] = \text{I}
\end{array}
$$

Block I is consequently split, yielding

$$
\begin{array}{l}
P_3 = \{\text{I}, \text{II}, \text{III}, \text{IV}, V\}, \\
\text{where I} = \{q_1\}, \text{ II} = \{q_2\}, \text{ III} = \{q_3, q_4\}, \text{ IV} = \{q_6\} \text{ and V} = \{q_5\} \ .
\end{array}
$$

The algorithm would need another iteration to notice that the partition does not change and that it is done, but we can establish this by a simple observation.

The only blocks that could have been affected by the last refinement are I, II and IV, but these are all singletons and cannot be refined further in any case.

It appears that the original DFA was close to minimal, as the minimised version contains only one state less. If we analyse the automaton, we see that it recognises the language L, given by

$$\begin{aligned} L &= L_1 \cup L_2, \text{ where} \\ L_1 &= \{w \in \Sigma^* \mid |w|_a \text{ is even}\}, \text{ and} \\ L_2 &= \{w \in \Sigma^* \mid w = uav, \text{ where } |u|_b \text{ is even, and } |u|_a \text{ is odd.}\} \ , \end{aligned}$$

where $|w|_a$ denotes the number of occurences of a in w. In the light of this, states q_1, q_2, q_5 and q_6 all remember a unique aspect of the the input string w read so far. State q_1 remembers that w has an even number of a's and b's, q_2 that it has and odd number of a's and b's, while q_5 and q_6 covers the remaining two combinations. As these four states are all both useful and distinct, no two of the can be merged. States q_3 and q_4 are both accepting and remember that w can be divided in the manner required by the definition of L_2. Since q_3 and q_4 are equivalent in all aspects that matter, they can be merged.

4.1 Distinguishing strings

One could easily modify Algorithm 1 to generate, for each (reachable) state q of the input automaton A, a string which distinguishes it from every state $q' \notin [q]_{L(A)}$: With every state q, we associate a string $s_q \in \Sigma^*$. When the algorithm first enters RefinePartition, all associated strings are empty. Every time a block B is found to be unstable with respect to some input symbol a and is consequently partitioned into smaller blocks, the symbol which forced the split is appended to s_q for all q that belonged to B. Eventually, the algorithm terminates by outputting the DFA $A' = (Q', \Sigma, \delta', q_0', F')$, the states of which are the equivalence classes of $\sim_{L(A)}$. It then holds for every pair of states $[q_1], [q_2] \in Q'$, that s_q, where $q \in [q_1] \cup [q_2]$, is a distinguishing string for $[q_1], [q_2]$. As a final observation, we note that the set $\{q_s | q \in Q\}$ is prefixed-closed and thus forms a tree over Σ.

5 Further reading

In 1971, Hopcroft presented a minimisation algorithm for DFAs that runs in time $O(n \log n)$, where n is the number of states of the input algorithm. Hopcrofts algorithm is similar to the one presented in this chapter, but uses the so-called "process the smaller half"-strategy to avoid unnecessary comparisons [Hop71].

During the fall semester of 2005, a course covering this topic and the background topics were given at the department of Computer Science at the University of Illinois by Prof. Viswanathan. The lecture notes from that course are quite extensive and may be suitable as further reading [Vis05].

References

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*, 1971.

[Ner58] A. Nerode. Linear automaton transformations. In *In Proc. of the American Mathematical Society 9*, pages 541–544, 1958.

[Vis05] Professor Mahesh Viswanathan. Lecture notes. CS475: Automata, Formal Languages, and Computational Complexity http://www.cs.uiuc.edu/class/fa05/cs475/Lectures/, 2005.

```
Input:  a deterministic finite automaton A
Output: the minimal DFA recognising L(A)

01: procedure Minimise (DFA A)
02:     A' := RemoveUnreachableStates (A) ;
03:     (Q, Σ, δ, q0, F) := A' ;
04:     P := RefinePartition (A', {Q \ F, F}) ;
05:     return Collapse (A', P) ;

01: procedure RemoveUnreachableStates (DFA (Q, Σ, δ, q0, F))
02:     Q' := q0 ;
03:     while ∃q ∈ Q' such that q is not marked
04:         choose q ∈ Q' such that q is not marked ;
05:         mark q ;
06:         for each a ∈ Σ
07:             if δ(q, a) ∉ Q'
08:                 add δ(q, a) to Q' ;
09:                 δ'(q, a) := δ(q, a) ;
10:
11:     return (Q', Σ, δ', q0, F ∩ Q') ;

01: procedure RefinePartition (DFA A = (Q, Σ, δ, q0, F), Partition P)
02:     P' := ∅ ;
03:     for each q in Q
04:         obs_q := ∅ ;
05:         for each symbol a in Σ
06:             add (a, [δ(q, a)]_P) to obs_q ;
07:         add q to [obs_q]_P' ;
08:
09:     if P' is equal to P
10:         return P' ;
11:     else
12:         return RefinePartition (A, P') ;

01: procedure Collapse (DFA A = (Q, Σ, δ, q0, F), Partition P)
02:     for each q ∈ Q
03:         add [q]_P to Q' ;
04:         for each a ∈ Σ
05:             δ'([q]_P, a) := [δ(q, a)]_P ;
06:
07:     for each q ∈ F
08:         add [q]_P to F' ;
09:
10:     return (Q', Σ, δ', [q0]_P, F') ;
```

Algorithm 1: A minimisation algorithm for DFAs.