Nonregular languages and minimization of DFAs

While the regular languages have nice properties (as we have seen in the early part of the course), unfortunately many languages that we encounter in practice are nonregular. We discuss a method to prove that a given language is nonregular. This material is from Section 9.4 in the textbook.

As the last topic on regular languages we consider an algorithm to minimize deterministic state diagrams. More on this topic can be found in the textbook by P. Linz [3]. A link to the text is given on the course web site.

Pumping lemma for regular languages

All regular languages L have the following property:

- corresponding to L there is a constant value called "pumping length" such that all strings in the language of length at least the "pumping length" can be "pumped", that is, some substring can be repeated arbitrarily many times and the resulting string remains in the language L.

The result is stated in the below pumping lemma. The pumping lemma gives a general technique for showing that certain languages are not regular.

Pumping lemma. For every regular language L there exists a constant n such that any string $x \in L$ of length at least n can be written as

$$x = p \cdot q \cdot r$$

where

(P1) $q \neq \varepsilon$

(P2) $|p \cdot q| \leq n$

(P3) $pq^k r \in L$ for all $k \geq 0$.

<u>Note</u>: In the proof of the pumping lemma we can choose n to be the number of states of a state diagram accepting L.

Examples. The following languages are not regular:

$\{0^i 1^i \mid i \geq 0\}$

$\{0^{2^i} \mid i \geq 0\}$ (this language consists of all strings of 0's having a length that is a power of 2)

How would you use the pumping lemma to show that the above languages are not regular? (Will be done in class.)

<u>Note.</u> If L is a finite language, any string $x \in L$ cannot be written in three parts $p \cdot q \cdot r$ such that conditions (P1) and (P3) hold. Do finite languages satisfy the conditions of the pumping lemma? Why or why not? Are all finite languages regular?

The general method of using the pumping lemma to show that a given language L is not regular can be described as follows. We use *proof by contradiction.*

1. For the sake of contradiction we <u>assume</u> that L is regular. Then the pumping lemma gives us the pumping length n. We don't know what n is (it can be arbitrarily large), we only know that it is a constant (positive) integer.

2. Choose a string $x \in L$ of length at least n.

3. Consider <u>all</u> the possible decompositions of x into three parts (as in the pumping lemma). If none of them satisfy the conditions (P1), (P2), (P3) from the pumping

lemma simultaneously, we obtain a *contradiction* and can conclude that L cannot be regular.

The crucial part in using the pumping lemma is usually to select a suitable string $x \in L$ in 2. above. The string x should be selected so that it cannot be pumped (without going "out" of the language L). Note that $x \in L$ has to be specified using the "unknown" constant n.

Above in stage 2., we can choose the string $x \in L$ in a way that we expect to cause problems with the "pumping property". Note, however, that in the following stage 3. we need to show that *any* decomposition of x into three parts does not satisfy the conditions (P1), (P2), (P3). That is, it is *not* sufficient to consider one particular decomposition.

Sometimes in order to show that given languages are not regular, together with the pumping lemma, we can rely on *closure properties* of the family of regular languages.

We observe the following: if S and T are regular languages then also the following languages are regular:

$S \cap T$

$\overline{S} = \{w \in \Sigma^* \mid w \notin S\}$

How can you establish the above closure properties?

Example. Define S to consist of all strings over alphabet $\{0, 1\}$ that have an equal number of occurrences of 0's and 1's. We show that S is not regular.

Proof by contradiction: If S is regular, then also the language

$$S \cap 0^*1^* = \{0^i1^i \mid i \geq 0\}$$

is regular. (Why?) We have in the earlier example shown that the language $\{0^i1^i \mid i \geq 0\}$ is not regular. We conclude that S cannot be regular.

Minimizing deterministic state diagrams (DFAs)

A state diagram, whether deterministic or nondeterministic, may have states that cannot be reached in computations on any input word, such states are called *useless.* Useless states can be found using a straigthforward graph reachability algorithm. How? Once the useless states are identified, they can be simply deleted from the state diagram.

A DFA with no useless states need not be minimal. Consider the example given in Figure 1. Here states B and C are indistinguishable (as defined more precisely below) in the sense that any string w takes the state B to an accepting state if and only if w takes C to an accepting state. Indistinguishable states can be merged into one state. In the DFA of Figure 1 also states A and D are indistinguishable.

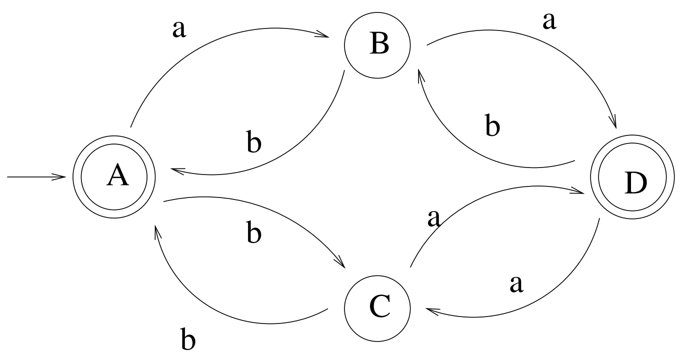

Figure 1: The states B and C can be merged into one state. Also states A and D can be merged.

Based on the above idea of merging indistinguishable states we present an algorithm to minimize an arbitrary DFA. More on this topic can be found in the textbook by P. Linz [3]. A link to the text is given on the course web site.

Recall that a DFA was defined as a tuple $M = (Q, \Sigma, \delta, s, F)$ where Q is the set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $s \in Q$ is the start state and $F \subseteq Q$ is the set of accepting states. We extend δ as a function $\hat{\delta} : Q \times \Sigma^* \to Q$ by defining inductively

1. $\hat{\delta}(q, \varepsilon) = q$ for all $q \in Q$, and,

2. $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ for all $q \in Q$, $a \in \Sigma$ and $w \in \Sigma^*$.

In the following also the extended transition function $\hat{\delta}$ is denoted simply by δ. For a state $q \in Q$ and string $w \in \Sigma^*$, $\delta(q, w)$ is the unique state that the DFA M is in after reading the string w assuming the computation starts in state q.

With the above notation we can now define that states q_1 and q_2 are *indistinguishable* if

$$(\forall w \in \Sigma^*) \;\; \delta(q_1, w) \in F \text{ iff } \delta(q_2, w) \in F.$$

The states q_1 and q_2 are *distinguishable (via string v)* if v violates the above condition, that is, if $\delta(q_1, v) \in F$ and $\delta(q_2, v) \notin F$ or vice versa.

The idea of the minimization algorithm is to find all pairs of distinguishable states. As the starting point of the algorithm we note that any accepting state (an element of F) is always distinguishable from a nonaccepting state (an element of $Q - F$). Why? Initially the algorithm marks all such pairs as distinguishable.

Algorithm: *Mark distinguishable pairs of states.* The input for the algorithm is a DFA $M = (Q, \Sigma, \delta, s, F)$ where all states are reachable from the start state. The algorithm marks all pairs of states (q_1, q_2) such that q_1 and q_2 are distinguishable.

- Stage 0: Mark all pairs (q_1, q_2) where $q_1 \in Q - F$ and $q_2 \in F$, or vice versa.
- Repeat the following until at some stage no new pairs are marked:
  Stage i $(i \geq 1)$: For each unmarked pair (q_1, q_2) and each $b \in \Sigma$ do the following. If the pair $(\delta(q_1, b), \delta(q_2, b))$ has been marked distinguishable at stage $i - 1$, then mark (q_1, q_2) as distinguishable.

Note that a pair (q_1, q_2) is distinguishable if and only if (q_2, q_1) is distinguishable. This means that when implementing the algorithm (or when tracing the algorithm "by hand") it is sufficient to consider unordered pairs of states.

Example 1. Consider the DFA M_1 of Figure 1. We have already observed that M_1 is not minimal.

When applying the marking algorithm to M_1, at stage 0 we mark pairs (A, B), (A, C), (B, D) and (C, D). As observed above, distinguishability is symmetric and hence marking (A, B) implies that also (B, A) is marked.

At stage one of the algorithm we now need to consider the unmarked pairs (A, D) and (B, C). We note that $(\delta(A, a), \delta(D, a)) = (B, C)$ and $(\delta(A, b), \delta(D, b)) = (C, B)$ where B and C are indistinguishable at previous stage 0. Hence the pair (A, D) is not marked at stage 1. Similarly it is observed that neither the pair (B, C) will be marked and the algorithm terminates after stage 1.

The minimized DFA is depicted in Figure 2.

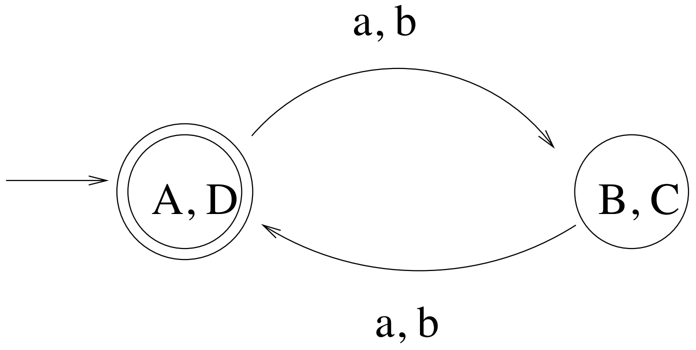

Figure 2: The minimized DFA equivalent to the DFA of Figure 1.

As indicated in the above example, the minimized DFA is obtained from the original DFA M by merging together into one class all states q_1, q_2 such that the pair (q_1, q_2) remains unmarked after the execution of the algorithm "*mark distinguishable pairs of states*". When merging q_1 and q_2, also the corresponding outgoing transitions are "merged" together. This is always possible because the pair (q_1, q_2) remaining unmarked means that the states q_1 and q_2 are indistinguishable, and hence, for any $b \in \Sigma$, the outgoing transitions from q_1 and q_2 on symbol b end up in a pair of indistinguishable states (that are also merged into one state).

More formally, assume that the original DFA M has a transition from state q to state p

on input symbol b. Then the minimized DFA has a transition on input b from the "merged together class" containing q to the class containing p. The start state of the minimized DFA is the class that contains the start state of the original DFA. All classes consisting of accepting states are accepting. (Note that an accepting state can never belong to the same class as a non-accepting state.)

Example 2. As a slightly bigger example, we use the algorithm to minimize the DFA M_2 of Figure 3. Here $\Sigma = \{a, b\}$. The details will be done in class.

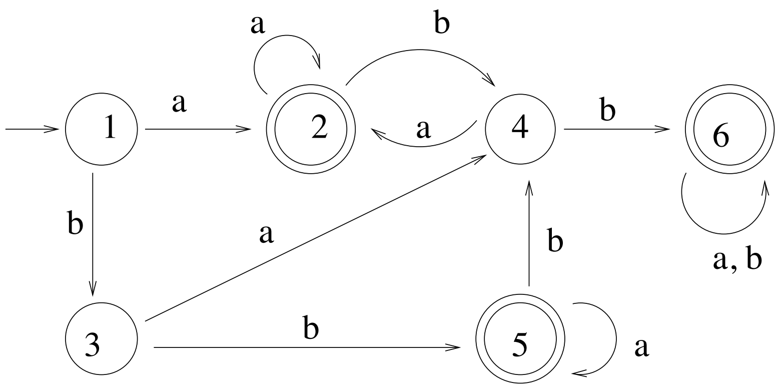

Figure 3: DFA M_2

It is easy to see that if a DFA M has n states the minimization algorithm always terminates after the $(n-1)$th stage.[1]

It can be shown that the algorithm produces a minimal DFA equivalent to the original DFA M and, furthermore, the minimal DFA is unique for any regular language [3]. That is, if M and M' recognize the same language, by applying the minimization algorithm to M and M', respectively, yields the same minimal DFA.

[1]At stage i the unmarked pairs define a partition of the state set into subsets where any two states in the same subset cannot be distinguished by any string of length at most i. At stage $i+1$ the partition is a refinement of the stage i partition and, if the total number of states is n, the refinement cannot be done more than $n-1$ times.

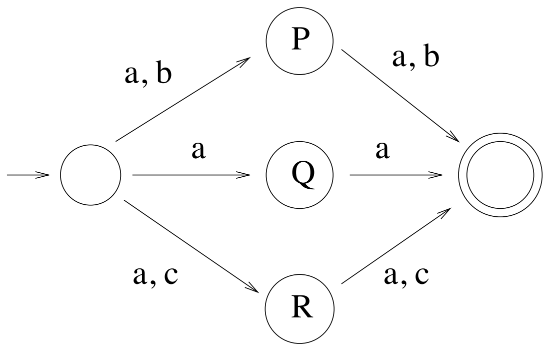

Figure 4: States P and Q can be merged or states Q and R can be merged. However, P, Q, R cannot all be merged.

A naive implementation of the minimization algorithm runs in cubic time. A considerably more sophisticated variant can be made to run in $O(n \cdot \log n)$ time [2].

Simplification of NFAs and regular expressions

Exactly as for deterministic state diagrams, in a given NFA we can identify and eliminate the useless states using a simple graph reachability algorithm. Here by useless states we mean states that cannot be reached from the start state on any string, or states from which an accept state cannot be reached on any string.

Naturally we may attempt to reduce the number of states of an NFA also by merging together "equivalent" states. However, the end result may depend on the order in which we choose to merge the states and, furthermore, a nonminimal NFA need not have any mergible states. This is illustrated by the two examples given in Figures 4 and 5.

The simple NFAs of Figures 4 and 5 can be easily minimized using exhaustive search. However, the phenomena illustrated by these examples imply that NFA minimization becomes a combinatorial problem and minimization, in general, is known to be intractable [1].[2]

Also the simplification of regular expressions is a hard problem. The regular expressions produced by the "state elimination algorithm" discussed in the previous section are often

[2]Using technical language, the algorithmic problem of NFA minimization is PSPACE-complete.

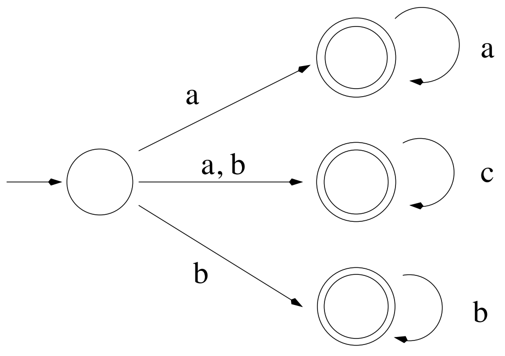

Figure 5: No two states can be merged, however, the NFA is not minimal.

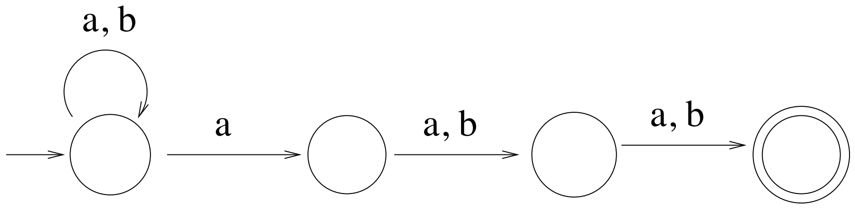

Figure 6: An NFA A_0 for which the equivalent minimal DFA has 8 states.

very large. The state elimination algorithm is implemented in the software package *Grail.* Using *Grail* we can determinize the NFA A_0 depicted in Figure 6 and the resulting DFA has 8 states. When, again using *Grail,* we apply the state elimination algorithm to the 8 state DFA, the regular expression has size over 32000 bytes. For this language the "obvious" regular expression is $(a + b)^* a(a + b)(a + b)$, however, when given the corresponding DFA as input, the state elimination algorithm cannot find any regular expression of reasonable size.

The size of the regular expressions produced by the state elimination algorithm can be reduced by various heuristic simplification techniques. There is no known general simplification algorithm and regular expression simplification is a current research topic.

References

[1] M. Holzer and M. Kutrib, Descriptional and computational complexity of finite automata — A survey. *Information and Computation* 209 (2011) 456–470.

[2] J.E. Hopcroft, An $n \cdot \log n$ algorithm for minimizing the states in a finite automaton. In: Z. Zohavi, A. Paz (Eds.), *International Symposium on the Theory of Machines and Computations,* Academic Press, 1971, pp. 189–196.

[3] P. Linz, *An Introduction to Formal Languages and Automata,* (section 2.4). Jones and Bartlett Publishers. `A link to the on-line edition can be found on the course web page.`