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 textbook is given in CISC/CMPE-223 onQ pages.

Unreachable/useless states: 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.* This includes states that cannot be reached from the start state and states that have no outgoing path reaching an accepting state. By viewing a state diagram as a directed graph, 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.

Removing useless states is a first step in minimization, however, 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.

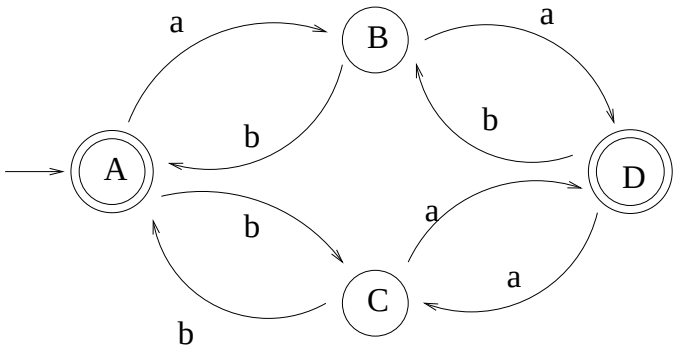

Figure 1: The states B and C can be merged into one state. Also states A and D can be merged.

Based on the above idea of merging indistinguishable states we present an algorithm to

minimize an arbitrary DFA.[1] More on on minimization of DFAs can be found in the textbook by P. Linz [3]. A link to the text is given on the course web site.

For the algorithm we need to define a notion of distinguishability for states of a DFA and for this purpose we use some formal notation. 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. Note that here M is a *complete DFA,* that is, $\delta : Q \times \Sigma \to Q$ is a total function. Earlier in the course we have considered also *incomplete DFAs* where some transitions could be undefined. An incomplete DFA can be easily completed by adding a so called "sink state" that is the target of all previously undefined transitions.[2]

Definition. 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.

[1] The minimization algorithm does not, in general, work for nondeterministic state diagrams.

[2] Adding the sink state may require adding a large number of transitions and this often makes drawing the state diagram messy. For this reason, often in examples the sink state is omitted. However, when dealing with the minimization algorithm we assume that all transitions are defined.

The idea behind 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. After finding all pairs of distinguishable states, we know that the remaining pairs are indistinguishable and, consequently, they can be merged into one state.

We call the algorithm "mark distinguishable pairs of states." As preprocessing we assume that states not reachable from the start state have been removed.

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 ith 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, that is, we are considering unordered pairs.

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 $(\delta(B, a), \delta(C, a)) = (D, D)$ and $(\delta(B, b), \delta(C, b)) = (A, A)$. Since a state is never distinguishable from itself, this verifies that the also the pair (B, C) will not be marked in stage 1. Since no new pairs were marked in stage 1, the algorithm terminates after stage 1.

The resulting minimized DFA is depicted in Figure 2.

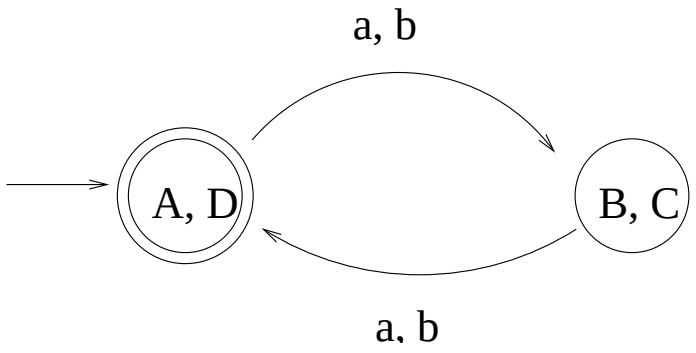

Figure 2: The minimized DFA equivalent to the DFA of Figure 1.

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.[3] The start state of the minimized

[3]More formally, indistinguishability is an *equivalence relation* on the set of states and partitions the set

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 the alphabet is $\Sigma = \{a, b\}$. The details will be done in class.

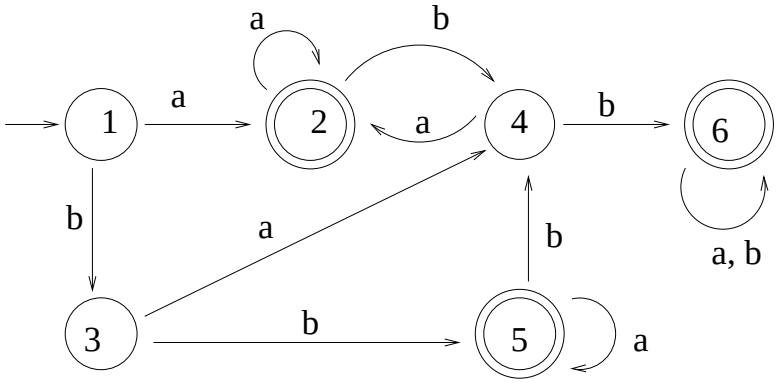

Figure 3: DFA M_2

It is easy to see that if a DFA M has n states the minimization algorithm always terminates after the $(n-1)$th stage.[4]

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' are any DFAs recognizing the same language, by applying the minimization algorithm to M and to M', respectively, yields the same minimal DFA.

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].

---

of states into equivalence classes – you may recall equivalence relations e.g. from the course CISC-203.

[4]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.

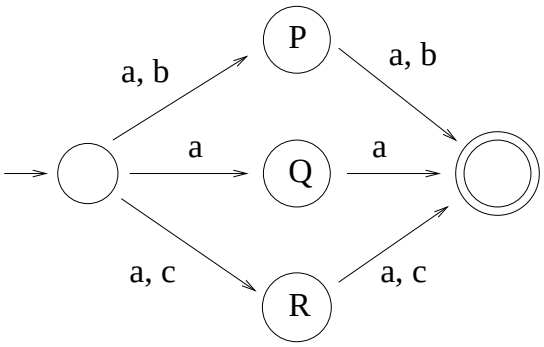

Figure 4: States P and Q can be merged or states Q and R can be merged. However, P, Q, R cannot all be merged.

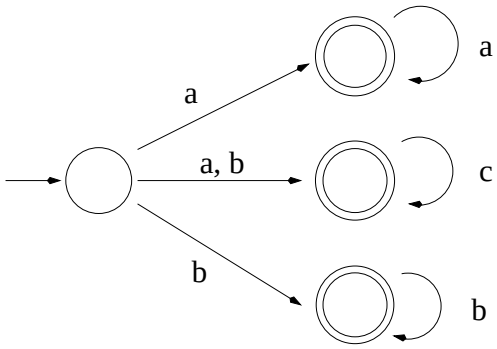

Figure 5: No two states can be merged, however, the NFA is not minimal.

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.

Analogously as was done for DFAs, we may attempt to reduce the number of states of an NFA 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.

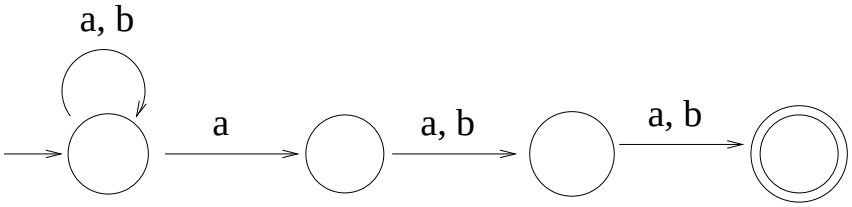

Figure 6: An NFA A_0 for which the equivalent minimal DFA has 8 states.

However, the phenomena illustrated by these examples imply that NFA minimization becomes a hard combinatorial problem and minimization, in general, is known to be intractable [1].[5]

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

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

[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.`