To summarize, we have seen methods for translating between DFAs, NFAs, and regular expressions:

- Every DFA is a NFA.
- Every NFA can be converted to an equivalent DFA, by the subset construction.
- Every regular expression can be translated to an equivalent NFA, by the method in Section 5.4.2.
- Every DFA can be translated to a regular expression by the method in Section 7.1.3.

Notice that, in order to say that these translations work, i.e., are correct, we need to use the concept of formal language.

### 5.5 Minimization

Now we turn to examining how to reduce the size of a DFA such that it still recognizes the same language. This is useful because some transformations and tools will generate DFAs with a large amount of redundancy.

**Example 122.** Suppose we are given the following NFA:

```
0, 1
q0 ----> q1 ----> q2 ----> q3
0
1
```

The subset construction yields the following (equivalent) DFA:

```
1
p0 ----> p1 ----> p2 ----> p3 ----> p4 ----> p5
0
1
0
```

---

184
which has 6 reachable states, out of a possible $2^4 = 16$. But notice
that $p_3$, $p_4$, and $p_5$ are all accept states, and it’s impossible to ‘escape’ from
them. So you could collapse them to one big success state. Thus the DFA
is equivalent to the following DFA with 4 states:

There are methods for systematically reducing DFAs to equivalent ones
which are minimal in the number of states. Here’s a rough outline of a
minimization procedure:

1. Eliminate inaccessible, or unreachable, states. These are states for
which there is no string in $\Sigma^*$ that will take the machine to that state.
How is this done? We have already been doing it, somewhat infor-
mally, when performing subset constructions. The idea is to start in
$q_0$ and mark all states accessible in one step from it. Now repeat this
from all the newly marked states until no new marked state is pro-
duced. Any unmarked states at the end of this are inaccessible and
can be deleted.

2. Collapse equivalent states. We will gradually see what this means in
the following examples.

Remark. We will only be discussing minimization of DFAs. If asked to
minimize an NFA, first convert it to a DFA.

Example 123. The 4 state automaton
is clearly equivalent to the following 3 state machine:

Example 124. The DFA

recognizes the language

\[
\{0, 1\} \cup \{x \in \{0, 1\}^* \mid \text{len}(x) \geq 3\}
\]

Now we observe that \(q_3\) and \(q_4\) are equivalent, since both go to \(q_5\) on anything. Thus they can be collapsed to give the following equivalent DFA:

By the same reasoning, \(q_1\) and \(q_2\) both go to \(q_{34}\) on anything, so we can collapse them to state \(q_{12}\) to get the equivalent DFA

Example 125. The DFA
recognizes the language

\[ \{0^n \mid \exists k. n = 3k + 1\} \]

This DFA minimizes to

How is this done, you may ask.

The main idea is a process that takes a DFA and combines states of it in a step-by-step fashion, where each steps yields an equivalent automaton. There are a couple of criteria that must be observed:

- We never combine a final state and a non-final state. Otherwise the language recognized by the automaton would change.

- If we merge states \( p \) and \( q \), then we have to combine \( \delta(p, a) \) and \( \delta(q, a) \), for each \( a \in \Sigma \). Contrarily, if \( \delta(p, a) \) and \( \delta(q, a) \) are not equivalent states, then \( p \) and \( q \) can not be equivalent.

Thus if there is a string \( x = x_1 \cdots x_n \) such that running the automaton \( M \) from state \( p \) on \( x \) leaves \( M \) in an accept state and running \( M \) from state \( q \) on \( x \) leaves \( M \) in a non-accept state, then \( p \) and \( q \) cannot be equivalent. However, if, for all strings \( x \) in \( \Sigma^* \), running \( M \) on \( x \) from \( p \) yields the same acceptance verdict (accept/reject) as \( M \) on \( x \) from \( q \), then \( p \) and \( q \) are equivalent. Formally we define equivalence \( \approx \) as
Definition 38 (DFA state equivalence).

\[
p \approx q \text{ iff } \forall x \in \Sigma^*. \Delta(p, x) \in F \text{ iff } \Delta(q, x) \in F
\]

where \(F\) is the set of final states of the automaton.

**Question:** What is \(\Delta\)?

**Answer** \(\Delta\) is the extension of \(\delta\) from symbols (single step) to strings (multiple steps). Its formal definition is as follows:

\[
\Delta(q, \varepsilon) = q
\]
\[
\Delta(q, a \cdot x) = \Delta(\delta(q, a), x)
\]

Thus \(\Delta(q, x)\) gives the state after the machine has made a sequence of transitions while processing \(x\). In other words, it’s the state at the end of the computation path for \(x\), where we treat \(q\) as the start state.

**Remark.** \(\approx\) is an equivalence relation, i.e., it is reflexive, symmetric, and transitive:

- \(p \approx p\)
- \(p \approx q \Rightarrow q \approx p\)
- \(p \approx q \land q \approx r \Rightarrow p \approx r\)

An equivalence relation partitions the underlying set (for us, the set of states \(Q\) of an automaton) into disjoint equivalence classes. This is denoted by \(Q/\approx\). Each element of \(Q\) is in one and only one partition of \(Q/\approx\).

**Example 126.** Suppose we have a set of states \(Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}\) and we define \(q_i \approx q_j\) iff \(i \text{ mod } 2 = j \text{ mod } 2\), i.e., \(q_i\) and \(q_j\) are equivalent if \(i\) and \(j\) are both even or both odd. Then \(Q/\approx = \{\{q_0, q_2, q_4\}, \{q_1, q_3, q_5\}\}\).

The equivalence class of \(q \in Q\) is written \([q]\), and defined

\[
[q] = \{p \mid p \approx q\}.
\]

We have the equality

\[
\begin{align*}
\text{equivalence of states} & \iff (\text{equality of sets of states}) \\
p \approx q & \iff ([p] = [q])
\end{align*}
\]

The **quotient** construction builds equivalence classes of states and then treats each equivalence class as a single state in the new automaton.
Definition 39 (Quotient automaton). Let \( M = (Q, \Sigma, \delta, q_0, F) \) be a DFA. The quotient automaton is \( M/\approx = (Q', \Sigma, \delta', q_0', F') \) where

- \( Q' = \{ [p] \mid p \in Q \} \), i.e., \( Q/\approx \)
- \( \Sigma \) is unchanged
- \( \delta'([p], a) = [\delta(p, a)] \), i.e., transitioning from an equivalence class (where \( p \) is an element) on a symbol \( a \) is implemented by making a transition \( \delta(p, a) \) in the original automaton and then returning the equivalence class of the state reached.
- \( q_0' = [q_0] \), i.e., the start state in the new machine is the equivalence class of the start state in the original.
- \( F' = \{ [p] \mid p \in F \} \), i.e., the set of equivalence classes of the final states of the original machine.

Theorem 18. If \( M \) is a DFA that recognizes \( L \), then \( M/\approx \) is a DFA that recognizes \( L \). There is no DFA that both recognizes \( L \) and has fewer states than \( M/\approx \).

OK, OK, enough formalism! we still haven’t addressed the crucial question, namely how do we calculate the equivalence classes?

There are several ways; we will use a table-filling approach. The general idea is to assume initially that all states are equivalent. But then we use our criteria to determine when states are not equivalent. Once all the non-equivalent states are marked as such, the remaining states must be equivalent.

Consider all pairs of states \( p, q \) in \( Q \). A pair \( p, q \) is marked once we know \( p \) and \( q \) are not equivalent. This leads to the following algorithm:

1. Write down a table for the pairs of states
2. Mark \((p, q)\) in the table if \( p \in F \) and \( q \notin F \), or if \( p \notin F \) and \( q \in F \).
3. Repeat until no change can be made to the table:

   - if there exists an unmarked pair \((p, q)\) in the table such that one of the states in the pair \((\delta(p, a), \delta(q, a))\) is marked, for some \( a \in \Sigma \), then mark \((p, q)\).
4. Done. Read off the equivalence classes: if \((p, q)\) is not marked, then \(p \approx q\).

**Remark.** We may have to revisit the same \((p, q)\) pair several times, since combining two states can suddenly allow hitherto equivalent states to be markable.

**Example 127.** Minimize the following DFA

We start by setting up our table. We will be able to restrict our attention to the lower left triangle, since equivalence is symmetric. Also, each box on the diagonal will be marked with \(\approx\), since every state is equivalent to itself. We also notice that state \(D\) is not reachable, so we will ignore it.

<table>
<thead>
<tr>
<th></th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
<th>(E)</th>
<th>(F)</th>
<th>(G)</th>
<th>(H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A)</td>
<td>(\approx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(B)</td>
<td>(\approx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C)</td>
<td></td>
<td>(\approx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(D)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(E)</td>
<td></td>
<td></td>
<td></td>
<td>(\approx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(F)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(\approx)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(G)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(\approx)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(H)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(\approx)</td>
<td></td>
</tr>
</tbody>
</table>

Now we split the states into final and non-final. Thus, a box indexed by \(p, q\) will be labelled with an \(X\) if \(p\) is a final state and \(q\) is not, or vice versa.
Thus we obtain

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>X₀</td>
<td>X₀</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>E</td>
<td>X₀</td>
<td>–</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>F</td>
<td>X₀</td>
<td>–</td>
<td>≈</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>G</td>
<td>X₀</td>
<td>–</td>
<td>≈</td>
<td>–</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>X₀</td>
<td>–</td>
<td>≈</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

State C is inequivalent to all other states. Thus the row and column labelled by C get filled in with X₀. (We will subscript each X with the step at which it is inserted into the table.) However, note that C ≈ C is not filled in, since C ≈ C. Now we have the following pairs of states to consider:

\{AB, AE, AF, AG, AH, BE, BF, BG, BH, EF, EG, EH, FG, FH, GH\}

Now we introduce some notation which compactly captures how the machine transitions from a pair of states to another pair of states. The notation

\[ p₁p₂ \xleftarrow{0} q₁q₂ \xrightarrow{1} r₁r₂ \]

means \( q₁ \xrightarrow{0} p₁ \) and \( q₂ \xrightarrow{0} p₂ \) and \( q₁ \xrightarrow{1} r₁ \) and \( q₂ \xrightarrow{1} r₂ \). If one of \( p₁, p₂, r₁, \) or \( r₂ \) are already marked in the table, then there is a way to distinguish \( q₁ \) and \( q₂ \): they transition to inequivalent states. Therefore \( q₁ \not\approx q₂ \) and the box labelled by \( q₁q₂ \) will become marked. For example, if we take the state pair \( AB \), we have

\[ BG \xleftarrow{0} AB \xrightarrow{1} FC \]

and since \( FC \) is marked, \( AB \) becomes marked as well.
In a similar fashion, we examine the remaining unassigned pairs:

- \(BH \leftarrow ^0 AE \rightarrow ^1 FF\). Unable to mark.
- \(BC \leftarrow ^0 AF \rightarrow ^1 FG\). Mark, since \(BC\) is marked.
- \(BG \leftarrow ^0 AG \rightarrow ^1 FE\). Unable to mark.
- \(BG \leftarrow ^0 AH \rightarrow ^1 FC\). Mark, since \(FC\) is marked.
- \(GH \leftarrow ^0 BE \rightarrow ^1 CF\). Mark, since \(CF\) is marked.
- \(GC \leftarrow ^0 BF \rightarrow ^1 CG\). Mark, since \(CG\) is marked.
- \(GG \leftarrow ^0 BG \rightarrow ^1 CE\). Mark, since \(CE\) is marked.
- \(GG \leftarrow ^0 BH \rightarrow ^1 CC\). Unable to mark.
- \(HC \leftarrow ^0 EF \rightarrow ^1 FG\). Mark, since \(CH\) is marked.
- \(HG \leftarrow ^0 EG \rightarrow ^1 FE\). Unable to mark.
- \(HG \leftarrow ^0 EH \rightarrow ^1 FC\). Mark, since \(CF\) is marked.
- \(CG \leftarrow ^0 FG \rightarrow ^1 GE\). Mark, since \(CG\) is marked.
- \(CG \leftarrow ^0 FH \rightarrow ^1 GC\). Mark, since \(CG\) is marked.
- \(GG \leftarrow ^0 GH \rightarrow ^1 EC\). Mark, since \(EC\) is marked.

The resulting table is

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>≈</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>(X_1)</td>
<td>≈</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>(X_0)</td>
<td>(X_0)</td>
<td>≈</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>(X_1)</td>
<td>(X_0)</td>
<td>−</td>
<td>≈</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>(X_1)</td>
<td>(X_0)</td>
<td>−</td>
<td>(X_1)</td>
<td>≈</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>(X_1)</td>
<td>(X_0)</td>
<td>−</td>
<td>−</td>
<td>(X_1)</td>
<td>≈</td>
<td>−</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>(X_1)</td>
<td>(X_0)</td>
<td>−</td>
<td>(X_1)</td>
<td>(X_1)</td>
<td>≈</td>
<td>−</td>
</tr>
</tbody>
</table>
Next round. The following pairs need to be considered:

\{AE, AG, BH, EG\}

The previously calculated transitions can be re-used; all that will have changed is whether the ‘transitioned-to’ states have been subsequently marked with an $X_1$:

**AE**: unable to mark

**AG**: mark because $BG$ is now marked.

**BH**: unable to mark

**EG**: mark because $HG$ is now marked

The resulting table is

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$\approx$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>B</td>
<td>$X_1$</td>
<td>$\approx$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>C</td>
<td>$X_0$</td>
<td>$X_0$</td>
<td>$\approx$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>D</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>E</td>
<td>$X_1$</td>
<td>$X_0$</td>
<td>$\approx$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>F</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_0$</td>
<td>$-$</td>
<td>$X_1$</td>
<td>$\approx$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>G</td>
<td>$X_2$</td>
<td>$X_1$</td>
<td>$X_0$</td>
<td>$-$</td>
<td>$X_2$</td>
<td>$X_1$</td>
<td>$\approx$</td>
<td>$-$</td>
</tr>
<tr>
<td>H</td>
<td>$X_1$</td>
<td>$X_0$</td>
<td>$-$</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$X_1$</td>
<td>$\approx$</td>
</tr>
</tbody>
</table>

Next round. The following pairs remain: \{AE, BH\}. However, neither makes a transition to a marked pair, so the round adds no new markings to the table. We are therefore done. The quotiented state set is

\{\{A, E\}, \{B, H\}, \{F\}, \{C\}, \{G\}\}

In other words, we have been able to merge states $A$ and $E$, and $B$ and $H$. The final automaton is given by the following diagram.
5.6 Decision Problems for Regular Languages

Now we will discuss some questions that can be asked about automata and regular expressions. These will tend to be from a general point of view, i.e., involve arbitrary automata. A question that takes any automaton (or collection of automata) as input and asks for a terminating algorithm yielding a boolean (true or false) answer is called a decision problem, and a program that correctly solves such a problem is called a decision algorithm. Note well that a decision problem is typically a question about the (often infinite) set of strings that a machine must deal with; answers that involve running the machine on every string in the set are not useful, since they will take forever. That is not allowed: in every case, a decision algorithm must return a correct answer in finite time.

Here is a list of decision problems for automata and regular expressions:

1. Given a string \( x \) and a DFA \( M \), \( x \in L(M) \)?
2. Given a string \( x \) and an NFA \( N \), \( x \in L(N) \)?
3. Given a string \( x \) and a regular expression \( r \), \( x \in L(r) \)?
4. Given DFA \( M \), \( L(M) = \emptyset \)?
5. Given DFA \( M \), \( L(M) = \Sigma^* \)?
6. Given DFAs \( M_1 \) and \( M_2 \), \( L(M_1) \cap L(M_2) = \emptyset \)?