What we have proven is that any language described by a regular expression is also accepted by some finite automaton. In the process of proving this result, we have given a method how to design such an automaton. We will now see how this works on an example.

Video 9.1: Regular Expression to Finite Automaton

Using the construction from the proof of Lemma 9.2, design a finite automaton that accepts the language designed by the regular expression $\mathbf{01}^* + \mathbf{1}$.

9.2.2 From a Finite Automaton to a Regular Expression

In the previous section, we have shown how to design a finite automaton (in that case, an NFA with ϵ-transitions) that accepts the language described by some regular expression. This means that every language described by some regular expressions is a regular language, remembering that we defined regular languages as languages accepted by DFAs (and, equivalently, NFAs and NFAs with ϵ-transitions, as they all accept exactly the same class of languages). In this section, we go in the opposite direction. We show that if we are given a regular language L, we can find a regular expression that describes the language L. This means that the class of languages described by regular expressions is exactly the class of regular languages (which is how regular languages got their name in the first place). This is quite an astonishing results, to find two seemingly completely different models, finite automata and regular expressions, that accept/describe exactly the same class of languages. Adding to that the fact that different variations of the finite automata (deterministic, non-deterministic, non-deterministic with ϵ-transitions) accept that same class of languages tells us that in regular languages we have discovered quite an important class.

Lemma 9.3: From Finite Automata to Regular Expressions

Let L be a regular language. Then there exists a regular expression r such that r describes the language L.

We are not going to give a formal proof of this lemma. We are, instead, just going to show how to construct a regular expression that describes L. To actually prove that this regular expression describes exactly the language L is not overly hard, but it is quite tedious, so we will leave the formal proof out.

Constructing a regular expression that describes a regular language (and, hence, the language accepted by some finite automaton) is notably harder than it is to construct a finite automaton that accepts the language described by a regular expression. Let L be a regular language. This means that there exists a DFA M that accepts the language L. To come up with a regular expression that describes L, we will go through a series of steps. We will first introduce a new concept called a *generalised non-deterministic finite automaton* (or GNFA for short). GNFA is similar to a NFA, except that the transitions between the states are labelled with regular expressions, rather than with individual symbols. That is, a GNFA transits from some state p to some state p if it reads a sequence of input symbols that is described by a regular expression of the transition between p and q. We will then show how to convert the DFA M to a GNFA M' that accepts the same language, and then finally how to reduce M' to a new GNFA that has only two states. The label of a transition between these two states is then going to be a regular expression that describes exactly the language accepted by the DFA M. As our running example, we will use a regular language accepted by a DFA from Figure 9.1.

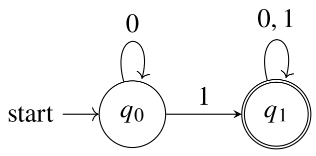


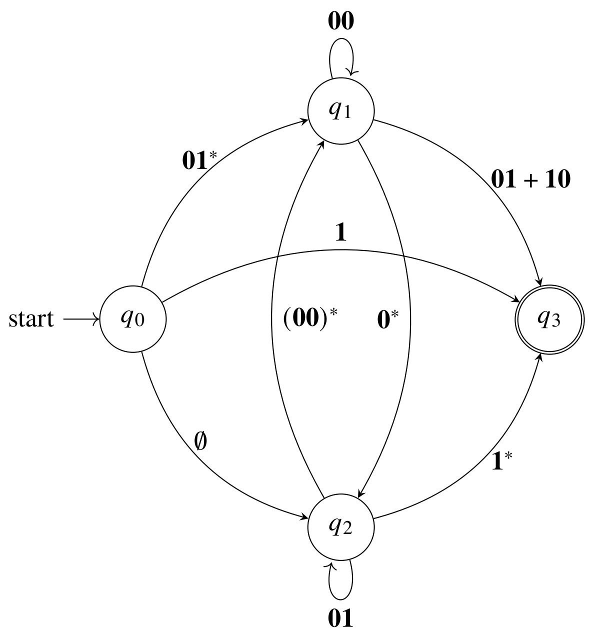
Figure 9.1: Example of a DFA M

Generalised Non-Deterministic Finite Automata

Definition 9.2: Generalised Non-Deterministic Finite Automaton (Informal)

Generalised Non-Deterministic Finite Automaton (GNFA) is similar to a NFA, except that the transitions between the states have regular expressions as labels.

Giving a formal definition of a GNFA, together with transition function on strings, is certainly possible, but too tedious for our purposes. It is enough to understand how a GNFA works, and that is best explained on an example. Consider the GNFA with the transition diagram below



A string w is accepted by this GNFA if it can be broken down into concatenation of strings $w_1w_2w_3\dots w_n$, such that there exists a path with n transitions from q_0 to q_3, with the transition i being labelled with a regular expression that describes string w_i. For example, $w = 01110$ is accepted, because it can be written as $(011)(10)$ and we can go from q_0 to q_1 on a transition $\mathbf{01}^*$ that describes the string 011 and then from q_1 to q_3 on transition $\mathbf{01+10}$ that describes the string 10. 01 is also accepted, because it can be written as $(01)\epsilon\epsilon$, and we can go from q_0 to q_1 on transition $\mathbf{01}^*$ that describes the string 01, from q_1 to q_2 on transition $\mathbf{0}^*$ that describes the string ϵ and finally from q_2 to q_3 on transition b^* that describes the string ϵ. Note that writing 01 as $(0)(1)\epsilon$ also works. On the other hand, string 11 is not accepted. There is no way to write this string as a concatenation of substrings such that there exists a path from q_0 to q_3 on labels that describe these substrings. This is easy to see - on reading the first input symbol 1, we can only go to q_3 from q_0 on transition **1**. But there is no transition out of q_2 that describes 1, nor are there transitions out of q_0 that describe 11 or ϵ. Therefore, the set of states in which the GNFA would be after it reads the string 11 is the empty set.

Remark 9.3: $\emptyset$ Transitions

Note that in our GNFA, there is a transition from q_0 to q_1 on $\emptyset$. When can this transition be made? The simple answer is - never. Recall that the language described by the regular expression $\emptyset$ is the empty set $\emptyset$. Therefore, there are no strings that are described by this regular expression. Note that this is different from a transition on regular expression ϵ, because this transition can be made on empty string ϵ, remembering that the language of regular expression ϵ is $\{\epsilon\}$. One might ask what the point of a transition on empty set is. The answer to that will become apparent below.

We are going to focus on a special class of GNFAs, which we can call *normalised GNFAs*. A normalised GNFA satisfies the following requirements:

1. The start state has transitions to every other state, but there are no transitions from other states to the start state.

2. There is only a single accepting state, and there are transitions to it from every other state. There are no transitions from the accepting state.

3. Except for the start and the accepting states, there is exactly one transition between any two states.

4. The start and the accepting state are not the same.

This does not result in loss of generality, as any GNFA can easily be converted into a normalised GNFA that accepts the same language. The property 1 is easily achieved for an arbitrary GNFA by adding a new start state with ϵ transition to the start state. Likewise, the property 2 can be achieved by adding a new accepting state, with ϵ-transitions from every accepting state of the original GNFA to it. This also satisfies the property 4. Finally, the property 3 can be achieved by adding transitions labelled $\emptyset$ between every two states between which a transition does not exist. If there are two or more transitions between some two states, we can merge them into a single transition. Let r_1 and r_2 be labels of two transition between states p and q. We can merge them into a single transition labelled $r_1 + r_2$. It is obvious that all of these changes do not change the language accepted by the GNFA. Therefore, from now on, we will assume that any GNFA we deal with is a normalised GNFA.

Converting a DFA into a GNFA

Converting some DFA M that accepts the regular language L into a GNFA that satisfies our four requirements is relatively simple. Firstly, we convert a DFA into a general GNFA by converting a transition between two states p and q on input symbol a into a transition on regular expression **a**. After that, we use the same procedure for converting this GNFA into a normalised GNFA. Using this procedure, the DFA from Figure 9.1 can be converted into a GNFA from Figure 9.2

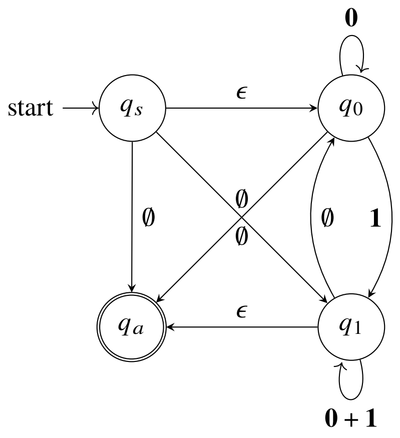


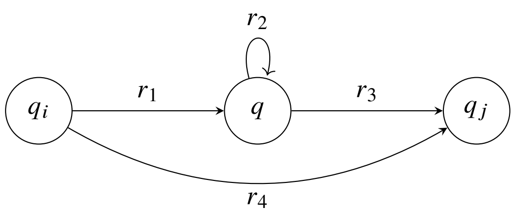
Figure 9.2: A Normalised GNFA from a DFA M from Figure 9.1

It is easy to see how this GNFA is obtained. We first add a new start state q_s, with transitions on regular expression ϵ from it to the start state of M (q_0), and transitions on regular expression $\emptyset$ to all other states. We also add a new accepting state q_a, with transitions on regular expression ϵ from the accepting state of M (q_1) on it, and transitions from every other state to it on a regular expression $\emptyset$. We finish by adding a transition on regular expression $\emptyset$ for any two states other than q_s and q_a for which there is no transition in M. In our case, that is just a transition from q_1 to q_0. The obtained GNFA indeed satisfies all four properties of a normalised GNFA.

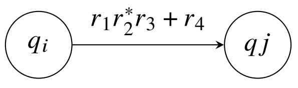
Reducing a GNFA to a Two-State GNFA

In general, to show that there exists a regular expression that describes the language accepted by some DFA M, we could just define a GNFA M' using the construction above of converting a DFA into a GNFA, and then show that it accepts the same language as M. The language accepted by a normalised GNFA can be described by a regular expression that describes a union (+ in the language of regular expression) of concatenations of all regular expressions that are obtained on some path from the start state to the accept state. There will also be infinite paths (because there are loops in our GNFA, e.g. from q_0 to q_0 on regular expression $\mathbf{0}$, and each loop can be followed an arbitrary number of times before going out of it), but these can be represented using the $*$ operator. For example, a path that loops from q_0 to q_0 an arbitrary number of times can be represented by a regular expression $\mathbf{0}^*$. However, we want to show more than just that a regular expression that describes the language accepted by M exists. We want to be able to precisely derive this regular expression, which would be very hard if we had to follow all the paths in a GNFA. For this purpose, the final step in our construction is reducing a normalised GNFA to a GNFA with just two states, the start state and the accepting state, and just one transition (obviously, from the start state to the accepting state). The label of this transition is going to be exactly the regular expression describing the language accepted by the GNFA.

Fortunately, there is a very easy method to eliminate a state (obviously, other than the start state and the accepting state) of a normalised GNFA and get the new normalised GNFA that accepts the same language, but has one state fewer. How do we eliminate a state from a GNFA? We simply need to update all the transitions in a GNFA to account for the missing state. Let us assume that we are eliminating the state q from a GNFA, and let us see how eliminating that state impacts a transition between some two states q_i and q_j. We need to compenstate for the regular expressions lost when we eliminated q (and, consequently, all the transitions to and out of q). In the case of q_i and q_j, removing q would mean that we lose the regular expression obtained by going from q_i to q, looping through the q an arbitrary number of times, and then going from q to q_j. This is depicted in the figure below.



Remember that a transition between q_i and q_j labelled with some regular expression r essentially means that our GNFA, if it starts in the state q_i, gets directly to the state q_j if it reads a string that is described by the regular expression r. In our new GNFA, we will not have state q, therefore we need to expand the label of a transition between q_i and q_j to cover also the regular expression that takes the GNFA from q_i to q_j via q. Strings described by this regular expression will take the new GNFA directly from q_i to q_j too. What regular expression describes the strings that take q_i to q and then to q_j? Firstly, a string that the regular expression r_1 describes takes the GNFA from q_i to q. From q, concatenation of any number of strings described by r_2 take the GNFA back to the state q. Finally, a string that the regular expression r_3 describes takes the GNFA from state q to state q_j. Therefore, the regular expression $r_1 r_2^* r_3$ describes the strings that take the GNFA from the state q_i to the state q_j via the state q. If to this we add r_4 that describes the strings that take the GNFA from q_i to q_j directly, we get that the regular expression $r_1 r_2^* r_3 + r_4$ describes all the strings that take the GNFA from q_i to q_j either directly or through q. These are all the strings that that the new GNFA from q_i to q_j directly, Therefore, if we take this as a label of the transition from q_i to q_j, we will have successfuly covered all the strings that take q_i to q_j directly in our new GNFA.



One might ask what happens with all the paths that go from q_i to q_j via q, but indirectly, through some other states too? They will be covered when we modify the transition between other pairs of states. Each path from q_i to q_j that passes through q will have a part that goes to some state p, then from p to q, loops through q possibly, and then goes from q to some state r. When we modify the transition from p to r, we will account for the lost part of the path through q.

With this, the method to reduce the number of states in a GNFA to 2 becomes quite clear. Let k be the number of states in the GNFA.

1. If $k = 2$, we are done. Otherwise, select any state q other than the start and the accepting state.

2. Take each pair (q_i, q_j) of the remaining states and modify the label of the transition from q_i to q_j to be $r_1 r_2^* r_3 + r_4$, where r_1 is the label of the transition from q_i to q, r_2 is the label of the transition from q to itself, r_3 is the label of the transition from q to q_j, and r_4 is the label of transition from q_i to q_j.

3. Remove the state q, as well as all the transitions to and from it, from the GNFA.

4. Go to the step 1.

The resulting GNFA will have exactly two states, the start state and the end state, and the transition between these two states is a regular expression that describes the same language as the initial GNFA. Therefore, if we start with an initial DFA M that accepts the language L, construct a GNFA M' that accepts the same language, and then use this method to reduce M' to a GNFA with just two states, we will get a regular expression that describes the language L.

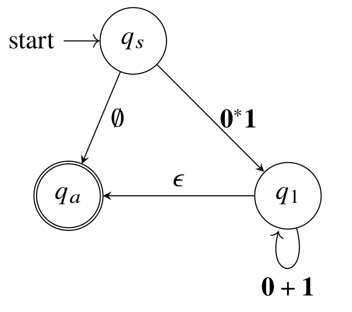
Let us see how our method works on our GNFA M' from Figure 9.2, keeping in mind that this GNFA accepts exactly the same language as the DFA from Figure 9.1. We have 4 states there, $\{q_s, q_0, q_1, q_a\}$. We want to end up with a GNFA with just the states $\{q_s, q_a\}$. Therefore, we can pick the state q_0 and eliminate it. Let us see how we modify the transitions of the GNFA to account for removal of q_0:

- Transition from q_s to q_1 becomes $\epsilon\mathbf{0}^*\mathbf{1} + \emptyset$. It is easy to see that this regular expression describes exactly the same set of strings as the regular expression $\mathbf{0}^*\mathbf{1}$.

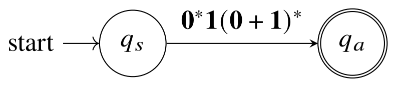
- Transition from q_1 to q_a becomes $\emptyset\mathbf{0}^*\emptyset + \epsilon$. It is easy to see that the regular expression ϵ describes exactly the same language.

- Transition from q_s to q_a becomes $\epsilon\mathbf{0}^*\emptyset + \emptyset$. The regular expression $\emptyset$ describes the same language.

Thus, our GNFA becomes



The next step is to eliminate the state q_1. We now only need to modify the transition from q_s to q_a, which becomes $\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*\epsilon + \emptyset$. The same language is described by the regular expression $\mathbf{0}^*\mathbf{1}(\mathbf{0}+\mathbf{1})^*$.



And with this, we are done. We have only two states in our GNFA, and the transition between them is labelled exactly with the regular expression that describes the language accepted by our original GNFA from Figure 9.2, which is the same language as the one accepted by our DFA from Figure 9.1.