NFAs and ε-NFAs

Winter 2020

1 NFAs and ε-NFAs

NFAs: A nondeterministic finite automaton (NFA) is a 5-tuple \((\Sigma, Q, q_0, A, \delta)\) where:

- \(\Sigma\) is a non-empty finite set whose elements are called symbols (or sometimes called letters). The set \(\Sigma\) is called the alphabet of the NFA.
- \(Q\) is a non-empty finite set whose elements are called states.
- \(q_0 \in Q\) is the initial state (sometimes called the starting state).
- \(A \subseteq Q\) is the set of accepting states (sometimes called final states).
- \(\delta: Q \times \Sigma \rightarrow 2^Q\) is the transition function, where \(2^Q\) denotes the power set of \(Q\) (the set of all subsets of \(Q\)).

The only difference between NFAs and DFAs is the definition of the transition function. For a DFA, the transition function is \(\delta: Q \times \Sigma \rightarrow Q\), so the result of \(\delta(q, a)\) for a state \(q\) and a symbol \(a\) is always a single state. For an NFA, the result of \(\delta(q, a)\) can be a set of states.

What happens when we transition from a single state to a set of multiple states? There are two common interpretations of what this means in an intuitive sense.

- The NFA is now “in multiple states at once”. The NFA accepts a word if at least one of the states it is in after reading the whole word is an accepting state.
- The NFA is “nondeterministically guessing” which state it should go to next, and the set of states represents all the possible options for its guesses. The NFA accepts a word if there is some sequence of guesses that leads to an accepting state.

Use whichever interpretation you find most intuitive. Either way, it is important to remember that in the NFA recognition algorithm, we keep track of a set of states instead of a single state, and we apply the transition function to every state in the set whenever we consume a letter.

ε-NFAs: An ε-NFA allows for the use of ε-transitions, which are transitions that can be taken without consuming any input. These are particularly useful for constructing automata that consist of multiple smaller automata connected together.

The only difference in the definition is that the transition function is changed to allow for transitions on ε:

\[\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q.\]

The ε-NFA recognition algorithm is also a bit different to account for the fact that it is possible to follow any number of ε-transitions in between consuming each symbol of the input. In between each symbol, we compute the ε-closure of the current state set, which is the set of all states we can reach from the current state set by following ε-transitions.
1.1 Exercises

1. Show that for every DFA, there is an NFA recognizing the same language.

**Solution:** Let $D = (\Sigma, Q, q_0, A, \delta)$ be a DFA. We want to construct an NFA recognizing the same language.

Since the only difference between DFAs and NFAs is the transition function, our NFA will be $(\Sigma, Q, q_0, A, \delta')$. All the components are carried over from the DFA except for the transition function.

Now, the issue with the transition function is that the DFA transition function has the form $\delta: Q \times \Sigma \rightarrow Q$, while the NFA transition function has the form $\delta': Q \times \Sigma \rightarrow 2^Q$. One outputs a state, and one outputs a set of states.

How do we make the NFA have the same behaviour as the DFA, while outputting a set instead of a single state? The answer is to just output a set with a single element:

$$\delta'(q, a) = \{\delta(q, a)\}.$$

This NFA transition function behaves exactly the same way as the DFA transition function, but it outputs a singleton set instead of a state. If you understand the NFA and DFA recognition algorithms, it should be easy to see that this NFA will recognize the exact same language as our original DFA.

End of solution.

2. Draw NFAs for the following languages over $\Sigma = \{0, 1\}$.

(a) Let $A$ be the language of strings ending in 1011.

**Solution:** We want to create an NFA for strings over $\{0, 1\}$ that end with 1011. We can start with an automaton that only recognizes the string “1011”:

Recall that when constructing a DFA for languages of words which end in a particular substring, we had to carefully make sure to backtrack to the right state whenever we read a “wrong” character. With an NFA, this problem is much simpler. We just need to add a loop on the initial state that can accept any number of 0’s and 1’s before moving on to the ending suffix 1011.

You can think of this NFA as “guessing” where the ending suffix will occur in the string. The NFA will accept a word as long as it is possible to “guess” correctly.

End of solution.

(b) Let $B$ be the language of strings ending in either 10 or 01.

**Solution:**

The NFA for this problem uses a similar idea to the previous one, but there are two possible paths it can take instead of one.
(c) Let $C$ be the language of strings beginning with $1000$.

**Solution:** We start by just accepting the string $1000$.

Now to accept any string that starts with $1000$, we simply need to accept any letter that follows the initial $1000$ prefix. We can do this with a loop on 0’s and 1’s in the accepting state.

Notice this diagram actually describes a DFA – there are no cases where we have two transitions out of a single state on the same symbol. This is fine, because NFAs are just a more powerful version of DFAs, so we could also interpret the diagram as an NFA. You should read the “nondeterministic” part of NFA as meaning “not necessarily deterministic” rather than “not ever deterministic”.

**End of solution.**

3. Draw an $\varepsilon$-NFA for the language $C(A \cup B)$, where $A$, $B$ and $C$ are the languages from the previous exercise.

**Solution:** To take the union of two languages using an $\varepsilon$-NFA, we create a new initial state, and add $\varepsilon$-transitions from this new state to the initial states of our original two NFAs.

To concatenate two languages, we add $\varepsilon$-transitions from each accepting state of the first language to the initial state of the second language. In this case the first language just has one accepting state. After adding the $\varepsilon$-transitions, we make the accepting states in the first language non-accepting (so we don’t accept words that are in the first language only and not in the concatenation).
There are various ways this \( \varepsilon \)-NFA could be simplified, but the above solution is valid.

End of solution.

2 Converting an NFA to a DFA

Recall that the only difference between an NFA and a DFA is the transition function. The transition function for an NFA maps state-symbol pairs to sets of states:

\[
\delta: Q \times \Sigma \rightarrow 2^Q.
\]

Whereas the transition function for an DFA maps state-symbol pairs to single states:

\[
\delta: Q \times \Sigma \rightarrow Q.
\]

This leads to a simple trick to turn NFAs into DFAs. The rough idea is to replace the state set \( Q \) of the NFA with \( 2^Q \). Now each “state” is a set, and the transition function maps “state-symbol pairs” (really set-symbol pairs) to “single states” (single sets). Now the transition function is deterministic, because for each state-symbol pair it outputs one state! This is known as the subset construction.

That’s the intuitive idea, but we have to be careful to make it formal. Suppose we have an NFA \( N = (\Sigma, Q, q_0, A, \delta) \) and we want to turn it into a DFA \( D = (\Sigma, Q', q'_0, A', \delta') \). Here is how each component of \( D \) is defined:

- \( \Sigma \) stays the same – we don’t change the alphabet when converting an NFA to a DFA.
- \( Q' \) is \( 2^Q \), the set of all subsets of \( Q \).
- \( q'_0 \) is \( \{q_0\} \), the set containing the initial state of \( Q \). Remember, the states of \( D \) are sets, so we have to add the curly braces!
- \( A' \) consists of all sets that contain a state from the NFA’s accepting state set \( A \). Formally, we define

\[
A' = \{S \in 2^Q : S \cap A \neq \emptyset \}.
\]

- Finally, the transition function \( \delta' : 2^Q \times \Sigma \rightarrow 2^Q \), is defined as follows. Given a set \( S \in 2^Q \) and a symbol \( a \in \Sigma \), the resulting set \( \delta'(S,a) \) is given by applying the NFA transition function \( \delta \) to each state in \( S \), and taking the union of the resulting sets. This is exactly what we do in the NFA recognition algorithm. Formally, we define:

\[
\delta'(S,a) = \bigcup_{q \in S} \delta(q,a).
\]

For example, if \( S = \{1, 2, 3\} \), \( \delta(1, a) = \{1, 4\} \), \( \delta(2, a) = \emptyset \), and \( \delta(3, a) = \{3, 5\} \), then we have

\[
\delta'([1, 2, 3], a) = \delta(1, a) \cup \delta(2, a) \cup \delta(3, a) = \{1, 4\} \cup \emptyset \cup \{3, 5\} = \{1, 3, 4, 5\}.
\]
In this worst case, the blow-up in states when converting from an NFA to a DFA is exponential. For example, if we have an NFA with 8 states, the equivalent DFA could have as many as $2^8 = 256$ states. One problem with the formal construction above is that it assumes we always need all the states in $2^Q$; sometimes we do, but this is just the worst-case scenario, and often we can make do with much fewer states.

For this reason, when converting NFAs to DFAs in practice, we use the following algorithm, which only finds the sets in $2^Q$ that are reachable from the initial set $\{q_0\}$. This algorithm is not guaranteed to produce a DFA with a minimal number of states, but it is better than assuming we need to include every possible set in $2^Q$.

This algorithm works by constructing a transition table. The rows of the table are states of the DFA, and the columns are letters of the alphabet.

1. Begin with one row, corresponding to the initial state $\{q_0\}$ of the DFA, and all columns empty.
2. For each empty row, let $S$ be the state corresponding to the row, and fill the row as follows. For each symbol $a \in \Sigma$, in column $a$ write the set $\delta'(S, a)$, where $\delta'(S, a)$ is as defined earlier.
   (What’s going on here? Recall that $\delta'(S, a)$ consists of all states which, in the original NFA, are reachable from a state in $S$ via the symbol $a$. We are looking at the states we have so far in our DFA, and figuring out which new sets are reachable from these states.)
3. After all the rows are filled, check if there any sets in the table which do not have a corresponding row. For each such set, add a new empty row.
   (Why do we add new empty rows? Because each set which has no row corresponds to a newly reached state. We need to see what states are reachable from these newly reached states, so we add rows for them and repeat the last step with these new rows.)
4. If there are empty rows, return to step 2. If there are no empty rows, the algorithm terminates.
   (Why do we terminate if there are no empty rows? No empty rows means we did not reach any new states in the previous step.)

After constructing the table, you can use it to draw a DFA diagram. You could also skip the table and just draw a DFA directly, adding new states and transitions when the algorithm says to “fill in columns”. However, the result might be more messy than constructing a table first and then drawing the diagram.

### 2.1 Exercise

Convert the following NFA to a DFA using the subset construction.

![NFA Diagram]

**Solution:** First, we make a transition table for the original NFA. This is not really necessary for the algorithm, but we need to be able to quickly compute values of the NFA transition function while doing the
algorithm, and it is easier to do so with a table than by looking at the diagram.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>1</td>
<td>{2}</td>
<td>{1, 3}</td>
</tr>
<tr>
<td>2</td>
<td>{1}</td>
<td>{3}</td>
</tr>
<tr>
<td>3</td>
<td>$\emptyset$</td>
<td>{1}</td>
</tr>
</tbody>
</table>

Note that state 3 on $a$ goes to the empty set $\emptyset$ because there are no transitions out of 3 on $a$.

Now we build the transition table for the DFA. We state with a row for state \{0\}:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
</tbody>
</table>

Now we figure out which subsets are reachable from \{0\} by using the NFA transition function. We can do this by just looking up 0 in our NFA transition table.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
</tbody>
</table>

Now we ask: are there any sets in the table that do not have corresponding rows? Yes, there are two: \{0, 1\} and \{2\}. We add empty rows for these sets.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>{2}</td>
<td>{2}</td>
<td>{2}</td>
</tr>
</tbody>
</table>

Since there are empty rows in the table, the algorithm continues. Now we fill out the empty rows. Filling out row \{2\} is straightforward, but \{0, 1\} contains two states. We have to look up both 0 and 1 in the NFA transition table, and take the union of the sets we get. For example, for 0 on $a$ we get \{0, 1\}, and for 1 on $a$ we get \{2\}, so we fill in \{0, 1, 2\} for column $a$ of row \{0, 1\}.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>{2}</td>
<td>{1}</td>
<td>{3}</td>
</tr>
</tbody>
</table>

Now there are four new sets that don’t have rows, so we add empty rows for them:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>{2}</td>
<td>{1}</td>
<td>{3}</td>
</tr>
</tbody>
</table>

The algorithm proceeds the same way as before. We will not show all the steps.
The final table looks like this:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0}</td>
<td>{0, 1}</td>
<td>{2}</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>{0, 1, 2}</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>{2}</td>
<td>{1}</td>
<td>{3}</td>
</tr>
<tr>
<td>{0, 1, 2}</td>
<td>{0, 1, 2}</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>{1, 2, 3}</td>
<td>{1, 2}</td>
<td>{1, 3}</td>
</tr>
<tr>
<td>{1}</td>
<td>{2}</td>
<td>{1, 3}</td>
</tr>
<tr>
<td>{3}</td>
<td>{0}</td>
<td>{1}</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>{1, 2}</td>
<td>{1, 3}</td>
</tr>
<tr>
<td>{1, 3}</td>
<td>{2}</td>
<td>{1, 3}</td>
</tr>
</tbody>
</table>

Note that the table does not include a row for \( \emptyset \). This is because \( \emptyset \) is an error state, and so it is not necessary to draw on the diagram. However, if you are writing a formal definition of a DFA you must include error states.

To draw the diagram, we need one more piece of information: what are the accepting states? A state in the DFA is accepting if it contains an accepting state of the original NFA. The only accepting state in the original NFA is 3, so all states containing 3 are accepting: \( \{3\} \), \( \{1, 3\} \) and \( \{1, 2, 3\} \).

The DFA diagram is below.

---

End of solution.

### 3 Converting an \( \varepsilon \)-NFA to a DFA

Converting a \( \varepsilon \)-NFA to a DFA is similar to converting an NFA without \( \varepsilon \)-transitions to a DFA. You just need to account for the \( \varepsilon \)-transitions when figuring out what states are reachable from a given set of states.
• Instead of always starting with initial state \( \{q_0\} \), the initial state should be:

\[
\{q_0\} \cup \{ q : q \text{ is reachable from } q_0 \text{ by taking one or more } \varepsilon\text{-transitions} \}.
\]

• When filling out row \( S \) of the table, compute \( \delta'(S, a) \) for each symbol \( a \) as usual. But before adding this set to the table, figure out which states are reachable from states in \( \delta'(S, a) \) by one or more \( \varepsilon \)-transitions, and throw in these states as well.

In other words, before adding a row or filling out a column in a row, we always want to take the \( \varepsilon \)-closure.

3.1 Exercise

Convert the following \( \varepsilon \)-NFA to a DFA.

Solution: As before, we start by creating a transition table for the NFA, only now we include a column for \( \varepsilon \). To fill out this column, for each state, find the set of states reachable from that state via one or more \( \varepsilon \)-transitions. Note in this example there are no “chains” of multiple \( \varepsilon \)-transitions in a row, but this could happen in general.

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( \varepsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( {0} )</td>
<td>( \emptyset )</td>
<td>( {1} )</td>
</tr>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( {2} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( {3} )</td>
<td>( {0} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>( {3} )</td>
<td>( {2} )</td>
</tr>
</tbody>
</table>

The algorithm is almost the same as the NFA to DFA case, but we must remember this essential rule:

Before you add a set to the table, add in all the states that are reachable from states in the set by one or more \( \varepsilon \)-transitions.

For example, right at the start of the algorithm, we need to add a row for \( \{0\} \), the initial state. But state 0 has a \( \varepsilon \)-transition leading to 1! So instead of writing \( \{0\} \) in the table, we must write \( \{0, 1\} \).

The set \( \{0, 1\} \), rather than \( \{0\} \), will be the initial state of our DFA.

Now we proceed as normal, remembering the essential rule. On \( a \), the set \( \{0, 1\} \) goes to \( \{0\} \). But before writing \( \{0\} \) in the table, we must add in states reachable by \( \varepsilon \)-transitions. So we add in 1, and this means we write \( \{0, 1\} \) in the table under column \( a \). For \( b \), the set \( \{0, 1\} \) goes to \( \{2\} \), which has no \( \varepsilon \)-transitions out of it, so we can just write in \( \{2\} \).
Now we add a new row for the newly reached set \{2\}.

\[
\begin{array}{c|cc}
\{0,1\} & a & b \\
\{2\} & \{0,1\} & \{2\}
\end{array}
\]

On \(a\), state 2 goes to \{3\}. But \{3\} has an \(\varepsilon\)-transition leading out of it to 2, so we add \{2, 3\} to the table. On \(b\), state 2 goes to \{0\}, which has an \(\varepsilon\)-transition to 1, so we add \{0, 1\} to the table.

\[
\begin{array}{c|cc}
\{0,1\} & a & b \\
\{2\} & \{0,1\} & \{2\} \\
\{2\} & \{2,3\} & \{0,1\}
\end{array}
\]

We add a new row for \{2, 3\}.

\[
\begin{array}{c|cc}
\{0,1\} & a & b \\
\{2\} & \{0,1\} & \{2\} \\
\{2,3\} & \{2,3\} & \{0,1\}
\end{array}
\]

On \(a\), set \{2, 3\} goes to \{3\}, which expands to \{2, 3\} after accounting for \(\varepsilon\)-transitions. On \(b\), set \{2, 3\} goes to \{0, 3\}, which expands to \{0, 1, 2, 3\} after accounting for \(\varepsilon\)-transitions.

\[
\begin{array}{c|cc}
\{0,1\} & a & b \\
\{2\} & \{0,1\} & \{2\} \\
\{2,3\} & \{2,3\} & \{0,1,2,3\}
\end{array}
\]

We add a new row for our new set \{0, 1, 2, 3\}. We leave it as an exercise to show that this set loops to itself on both \(a\) and \(b\).

\[
\begin{array}{c|cc}
\{0,1\} & a & b \\
\{2\} & \{0,1\} & \{2\} \\
\{2,3\} & \{2,3\} & \{0,1,2,3\} \\
\{0,1,2,3\} & \{0,1,2,3\} & \{0,1,2,3\}
\end{array}
\]

The accepting states are the sets which contain 2. Here is a diagram:

![Diagram]

End of solution.
Appendix: Converting an $\varepsilon$-NFA to an NFA

Sometimes it is useful to be able to convert an $\varepsilon$-NFA to an NFA without $\varepsilon$-transitions, rather than going all the way to a DFA. We can use the following algorithm to convert the $\varepsilon$-NFA $M = (Q, \Sigma, q_0, A, \delta)$ to the NFA $N = (Q', \Sigma, q'_0, A', \delta')$. Since this algorithm was not taught in lectures, you are not required to know it for the exams.

1. The start state $q'_0$ of the NFA is the same as the start state $q_0$ of the $\varepsilon$-NFA.
2. We take the state set $Q'$ of the NFA $N$ to be the start state $q_0$, as well as any other state $q \in Q$ which has at least one non-$\varepsilon$ transition going into it. In other words, states that can only be reached via $\varepsilon$ transitions get removed.
3. For each state in $q \in Q'$ and each letter $a \in \Sigma$, add a transition from $q$ to all states which can be reached from $q$ in the $\varepsilon$-NFA $M$ by consuming exactly one $a$ (possibly following multiple $\varepsilon$-transitions along the way).
4. Mark a state $q \in Q'$ as accepting if it was accepting in the original $\varepsilon$-NFA, or if it can reach an accepting state in the original $\varepsilon$-NFA by following only $\varepsilon$-transitions.

Exercise

Convert the $\varepsilon$-NFA from the previous section to an NFA using the above algorithm.

Solution: We begin with drawing the start state and states that have at least one non $\varepsilon$-transition going into them, namely states 0, 2 and 3.

Starting with state 0, notice that it can reach itself via $a$ and it can reach 2 via $b$ (by first following a $\varepsilon$-transition). We add the corresponding transitions.

Moving on to state 2, notice that it can reach both itself and 3 with an $a$, and 0 with a $b$. 


State 3 can reach everything with a $b$, and it can reach 2 and itself with an $a$.

![Diagram showing state transitions]

Finally, we mark both 2 and 3 as accepting, since 2 was accepting in the original $\varepsilon$-NFA, and 3 can reach 2 via $\varepsilon$-transitions in the original $\varepsilon$-NFA.

![Diagram showing final state transitions]

**End of solution.**