# From NFA to Boolean Circuits

Given a non-deterministic finite state automaton (an NFA) $\mathcal{A}$ over binary alphabet $\Sigma = \{0, 1\}$ and a fixed integer constant $n$, construct a Boolean circuit with input gates $x_1, \ldots, x_n$ that has value $\top$ for an input $x = x_1 x_2 \ldots x_n \in \Sigma^n$ iff $\mathcal{A}$ accepts $x$.

## Preliminary Notes

We may use $\top$ ($\bot$) to denote *true* (*false*). We will use Boolean circuits that are extensions to those described in Papadimitriou's book. First, an equivalence gate $a \Leftrightarrow b$ with in-degree 2 is equivalent to a sub-circuit $(a \wedge b) \vee (\neg a \wedge \neg b)$. This is shown in Figure 1. An OR-gate with in-degree $k$, $\bigvee_{x \in \{a_1, \ldots, a_k\}} x$, is defined to be (i) an input gate with sort $\bot$ if $k = 0$, (ii) the gate $a_1$ if $k = 1$ and (iii) for $k \geq 2$, the following sub-circuit of $k - 1$ OR-gates with in-degree 2: $(\ldots(a_1 \vee a_2) \vee a_3) \vee \cdots \vee a_k)$. Thus our extensions do not expand the expressive power of Boolean circuits but are only used for convenience.

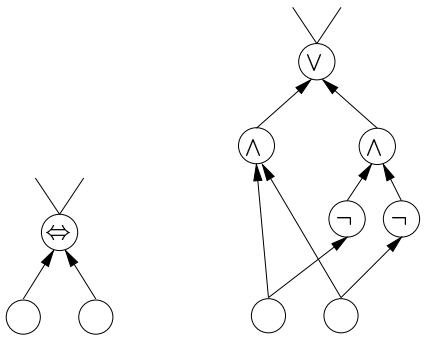


Figure 1: An equivalence gate and the corresponding AND/OR/NOT-sub-circuit.

## The Construction

Let the NFA be $\mathcal{A} = \langle Q, \Sigma, \Delta, \mathcal{F}, q_{init} \rangle$, where $Q = \{q_1, \ldots, q_k\}$ is the finite set of states, $\Sigma = \{0, 1\}$ is the binary alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is the (non-deterministic) transition relation, $\mathcal{F} \subseteq Q$ is the set of accepting states and $q_{init} \in Q$ is the initial state. A sequence $q'_1 \ldots q'_{j+1} \in Q^{j+1}$ of states is *legal* for input $x = x_1 x_2 \ldots x_n \in \Sigma^n$, $n \geq j$, iff $\langle q'_i, x_i, q'_{i+1} \rangle \in \Delta$ for all $1 \leq i \leq j$. $\mathcal{A}$ *accepts* $x = x_1 x_2 \ldots x_n \in \Sigma^n$ iff there is a legal sequence $q'_1 \ldots q'_{n+1} \in Q^{n+1}$ of states such that $q'_1 = q_{init}$ and $q'_{n+1} \in \mathcal{F}$.

The circuit $\mathcal{C} = \langle G, E \rangle$ is build as follows. The input gates of $\mathcal{C}$ consists of (i) the gates $q_{1,1}, \ldots, q_{k,1}$ with the sort $s(q_{i,1}) = \top$ if $q_i = q_{init}$ and $s(q_{i,1}) = \bot$ otherwise and (ii) the gates $x_1, \ldots, x_n$ with the sorts $s(x_i) = x_i$. The other gates are the gates $q_{i,j}$, $1 \leq i \leq k$ and $2 \leq j \leq n+1$, and the necessary gates for their sub-circuits:

$$q_{i,j} := \bigvee_{\langle q_{i'}, 1, q_i \rangle \in \Delta} (q_{i',j-1} \wedge x_{j-1}) \vee \bigvee_{\langle q_{i'}, 0, q_i \rangle \in \Delta} (q_{i',j-1} \wedge \neg x_{j-1}).$$

Finally, the output gate $o$ is defined $o := \bigvee_{q_i \in \mathcal{F}} q_{i,n+1}$. The intuition behind the construction is the following. Given a truth assignment on the input gates $x_1, \ldots, x_n$ corresponding to the input string, the gate $q_{i,j}$ has the value $\top$ iff there is a legal sequence $q_{init} \ldots q_i \in Q^j$ of $j$ states leading to the state $q_i$ on input $x_1 \ldots x_n$. The value of $o$ is clearly $\top$ iff the corresponding string $x_1 \ldots x_n$ is accepted by $\mathcal{A}$ (a simple inductive argument on the length of the input).

As an example, consider the automaton shown on the left in Figure 2. A corresponding circuit for $n = 3$ is shown on the right hand side.

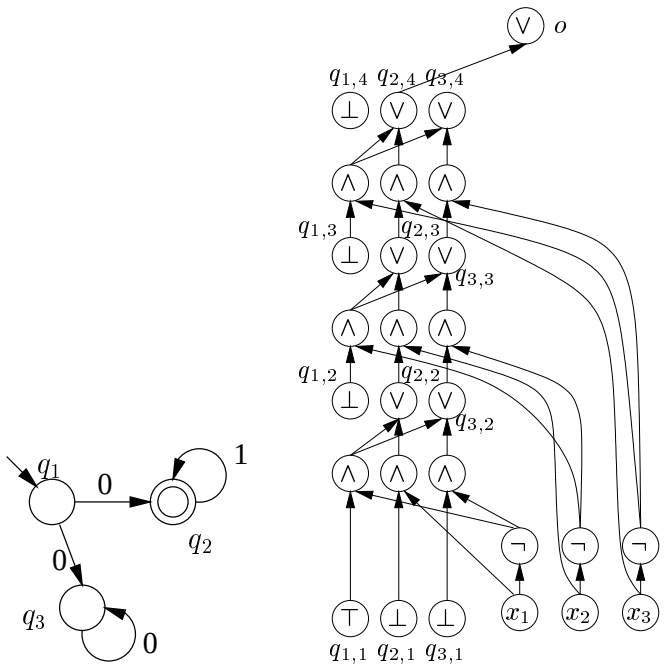


Figure 2: An NFA and a corresponding Boolean circuit for $n = 3$.

## 7.4.6

The Kleene star of a language $L$ is $L^\star = \{x_1 \dots x_k \mid k \geq 0 \text{ and } x_1, \dots, x_k \in L\}$. Our goal is to show that **P** and **NP** are closed under Kleene star, that is, if $L \in \mathbf{P}$, then $L^\star \in \mathbf{P}$ and similarly for **NP**.

Suppose that $L \in \mathbf{NP}$ i.e. $L$ is decided by a non-deterministic Turing machine $M$ in polynomial time. Then $L^\star$ is decided by a non-deterministic Turing machine $M'$ in polynomial time, as described in the following. Assume an input string $x$. If $x = \varepsilon$, then $M'$ accepts. Otherwise $M'$ splits $x$ non-deterministically into at most $|x|$ non-empty strings $x'_1, \dots, x'_k$ such that $x'_1 \dots x'_k = x$. Then $M'$ runs $M$ as a subroutine on each of the strings and accepts iff each simulation run answered "**yes**". It is obvious from the construction that $L(M') = L^\star$ and $M'$ works in polynomial time. Thus **NP** is closed under Kleene star.

Suppose that $L \in \mathbf{P}$ i.e. $L$ is decided by a deterministic Turing machine $M$ in polynomial time. Now we cannot use the previous splitting approach since there are $\sum_{k=0}^{n-1} \binom{n-1}{k}$, which is not polynomial, possible splittings for an input of length $n$. However, by using dynamic programming techniques, $L^\star$ is decided by the following algorithm, where $x^{i,j}$ means the substring of $x$ from the $i$th position to the $j$th position. The idea is to keep already computed information in a Boolean array (of polynomial size) and use it instead of recomputing.

Input: a string $x = x_1 \dots x_n$ and a machine $M$ deciding $L$
Output: *true* iff $x \in L^\star$
Variables: a 2D-array $inL^\star[n, n]$, initialized to *false*

**if** $x = \varepsilon$ **then return** *true*;
**for**$(i = n;\ i \geq 1;\ i--)$ {
  **if** $M(x^{i,n}) =$ **yes then** $inL^\star[i, n] := true$;
  **for**$(j = i;\ j < n;\ j++)$ {
    **if** $M(x^{i,j}) =$ **yes** $\wedge\ inL^\star[j + 1, n] = true$ **then** $inL^\star[i, n] := true$;
  }
}
**if** $inL^\star[1, n] = true$ **then return**(**yes**); **else return**(**no**);

The running time of the algorithm without the calls to the machine $M$ is $\mathcal{O}(n^k)$ for some fixed $k \geq 2$ depending of the implementation of the programming language operations, e.g. how substrings are extracted from a string, — certainly these can be done in polynomial time. The algorithm makes $\frac{n(n+1)}{2}$ queries to the machine $M$ deciding $L$, all of which are carried out in polynomial time. Thus $L^\star$ can be decided in deterministic polynomial time

and **P** is closed under Kleene star.

## 7.4.3 (a)

**Definition 1 (Definition 7.1 in Papadimitriou's book)** *Let* $f : \mathbb{N}_+ \to \mathbb{N}_+$. *$f$ is a* proper complexity function *if*

1. *$f$ is non-decreasing, i.e., $f(n+1) \geq f(n)$ for all $n \in \mathbb{N}_+$ and*
2. *there is a $k$-string Turing machine $M_f$ with input and output such that, given an input of length $n$, $M_f$ outputs $\sqcap^{f(n)}$ on its output string in time $\mathcal{O}(n+f(n))$ and uses $\mathcal{O}(f(n))$ space besides its input and output.*

Our assumption is that $f$ and $g$ are proper complexity functions, i.e., non-decreasing functions such that we $k_f$- and $k_g$-string Turing machines $M_f$ and $M_g$, respectively, calculating them and fulfilling the second condition in the above definition.

### Case: $f + g$

Clearly the function $(f+g)(n) = f(n) + g(n)$ is non-decreasing if $f$ and $g$ are. Now, since we have $M_f$ and $M_g$, we can construct a $k_f + k_g + 1$ -string Turing machine $M_{f+g}$ fulfilling the requirements in the definition for proper complexity fuctions as follows. Given an input, $M_{f+g}$ first runs $M_f$ on the input and thus calculates the value $f(n)$ on the $k_f$th string. $M_{f+g}$ then copies the input into the string $k_f + 1$ and then runs $M_g$ on it, computing the value $g(n)$ on the string $k_f + k_g$. Finally, $M_{f+g}$ concatenates the strings $k_f$ and $k_g$ on its output string $k_f + k_g + 1$. Clearly (i) the output string has the value $f(n)+g(n)$, (ii) the machine runs in time $\mathcal{O}(n+f(n)+n+g(n)+f(n)+g(n)) = \mathcal{O}(n+f(n)+g(n))$ and (iii) the maximum amount of space used by $M_{f+g}$ is $\mathcal{O}(\max\{f(n), g(n)\})$ (the input and output strings are not counted in). Therefore $f+g$ is also a proper complexity function.

### Case: $f \cdot g$

Clearly the function $(f \cdot g)(n) = f(n) \cdot g(n)$ is non-decreasing if $f$ and $g$ are. The construction for the machine $M_{f.g}$ is similar to the machine $M_{f+g}$ except that, in the last phase, the string $k_f$ is copied into the string $k_f + k_g + 1$ $k_g$ times. The time requirement now is $\mathcal{O}(n + f(n) + n + g(n) + f(n) \cdot g(n)) = \mathcal{O}(2n + 2f(n) \cdot 2g(n) - 2) = \mathcal{O}(n + f(n) \cdot g(n))$ and the space consumption

stays the same. Therefore $f \cdot g$ is also a proper compexity function if $f$ and $g$ are.

### Case: $2^g$

Clearly the function $2^g(n) = 2^{g(n)}$ is non-decreasing if $g$ is. We define the machine $M_{2^g}$ so that it first calculates the value $g(n)$ in unary on the string $k_g$ and then, using auxiliary strings $k_g + 1$ and $k_g + 2$, does the following. First, $M_{2^g}$ prints one $\sqcap$ symbol on the string $k_g + 2$. Then, using the string $k_g + 1$, it calculates in unary, how many symbols there are on the string $k_g + 2$ and then appends the same amount of new $\sqcap$ symbols on $k_g + 2$. $M_{2^g}$ repeats this as many times as there are symbols on the string $k_g$, that is, $g(n)$ times. Therefore, at the end, the string $k_g + 2$ will have $2^{g(n)}$ $\sqcap$s. Finally, $M_{2^g}$ copies the contest of the string $k_g + 2$ into its output string $k_g + 3$. The time consumption now is $\mathcal{O}(n + g(n) + 2^{g(n)}) = \mathcal{O}(n + 2^{g(n)})$ as required and the space consumption is clearly $\mathcal{O}(n + 2^{g(n)})$. Therefore $2^g$ is also a proper compexity function if $g$ is.

### Case: $f(g)$

Clearly the function $(f(g))(n) = f(g(n))$ is non-decreasing if $f$ and $g$ are. The (non-working) construction for the machine $M_{f(g)}$ is such that $M_{f(g)}$ first calculates the value $g(n)$ on the string $k_g$ and then uses it as input to the machine $M_f$ that calculates the value $f(g(n))$ into the output string $k_g + k_f$. This construction is *not* correct if $f$ grows stricly lower than the linear function $f(n) = n$. To see why this is the case, consider functions $g(n) = 2^n$ and $f(n) = \log n$. Now $f(g(n)) = n$ but the machine $M_{f(g)}$ uses time and space proportional to $2^n$. For functions $f$ that grow at least as fast as the linear function, the construction clearly fulfills the requirements for a machine computing proper complexity functions. Fortunately, (i) the time complexity functions are in practise always at least linear since a machine must be able to read its input and (ii) the only sub-linear space complexity functions used in this course are the log functions which, as we know, are proper. Therefore, the partial failure of this construction is not important for us.