1.4 Boolean functions and Boolean circuits

A *Boolean function* is a mapping $f : \{0,1\}^n \to \{0,1\}$. The values 0,1 are sometimes identified with the values False, True and the variables in $f(x_1, \ldots, x_n)$ are sometimes called *Boolean variables*, *Boolean variables* or *bits*. In many algorithmic problems, there are n input Boolean variables and one output bit. For example: given a graph G with N nodes, suppose we want to decide whether it has a Hamiltonian cycle. In this case, the graph can be described with $\binom{N}{2}$ Boolean variables: the nodes are numbered from 1 to N and x_{ij} $(1 \le i < j \le N)$ is 1 if i and j are connected and 0 if they are not. The value of the function $f(x_{12}, x_{13}, \ldots, x_{n-1,n})$ is 1 if there is a Hamilton cycle in G and 0 if there is not. Our problem is the computation of the value of this (implicitly given) Boolean function.

There are only four one-variable Boolean functions: the identically 0, identically 1, the identity and the *negation*: $x \to \overline{x} = 1 - x$. We also use the notation $\neg x$. There are 16 Boolean functions with 2 variables (because there are 2^4 mappings of $\{0,1\}^2$ into $\{0,1\}$). We describe only some of these two-variable Boolean functions: the operation of *conjunction* (logical AND):

$$x \wedge y = \begin{cases} 1 & \text{if } x = y = 1, \\ 0 & \text{otherwise,} \end{cases}$$

this can also be considered the common, or mod 2 multiplication, the operation of *disjunction* (logical OR)

$$x \vee y = \begin{cases} 0 & \text{if } x = y = 0, \\ 1 & \text{otherwise,} \end{cases}$$

the *binary addition* (logical exclusive OR of XOR)

$$x \oplus y = x + y \bmod 2.$$

Among Boolean functions with several variables, one has the logical AND, OR and XOR defined in the natural way. A more interesting function is MAJORITY, whici is defined as follows:

$$\text{MAJORITY}(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if at least } n/2 \text{ of the variables is 1;} \\ 0, & \text{otherwise.} \end{cases}$$

These bit-operations are connected by a number of useful identities. All three operations AND, OR and XOR are associative and commutative. There are several distributivity properties:

$$\begin{aligned} x \wedge (y \vee z) &= (x \wedge y) \vee (x \wedge z) \\ x \vee (y \wedge z) &= (x \vee y) \wedge (x \vee z) \end{aligned}$$

and

$$x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z).$$

The DeMorgan identities connect negation with conjunction and disjunction:

$$\begin{aligned} \overline{x \wedge y} &= \overline{x} \vee \overline{y}, \\ \overline{x \vee y} &= \overline{x} \wedge \overline{y}. \end{aligned}$$

Expressions composed using the operations of negation, conjunction and disjunction are called *Boolean polynomials.*

(1.1) **Lemma** *Every Boolean function is expressible as a Boolean polynomial.*

Proof Let $a_1, \ldots, a_n \in \{0, 1\}$. Let

$$z_i = \begin{cases} x_i & \text{if } a_i = 1, \\ \overline{x}_i & \text{if } a_i = 0, \end{cases}$$

and $E_{a_1,\ldots,a_n}(x_1, \ldots, x_n) = z_1 \wedge \cdots \wedge z_n$. Notice that $E_{a_1,\ldots,a_n}(x_1, \ldots, x_n) = 1$ holds if and only if $(x_1, \ldots, x_n) = (a_1, \ldots, a_n)$. Hence

$$f(x_1, \ldots, x_n) = \bigvee_{f(a_1,\ldots,a_n)=1} E_{a_1,\ldots,a_n}(x_1, \ldots, x_n).$$

∎

The Boolean polynomial constructed in the above proof has a special form. A Boolean polynomial consisting of a single (negated or unnegated) variable is called a *literal.* We call an *elementary conjunction* a Boolean polynomial in which variables and negated variables are joined by the operation "$\wedge$". (As a degenerate case, the constant 1 is also an elementary conjunction, namely the empty one.) A Boolean polynomial is a *disjunctive normal form* if it consists of elementary conjunctions, joined by the operation "$\vee$". We allow also the empty disjunction, when the disjunctive normal form has no components. The Boolean function defined by such a normal form is identically 0. In general, let us call a Boolean polynomial *satisfiable* if it is not identically 0. As we see, a nontrivial disjunctive normal form is always satisfiable.

By a *disjunctive k-normal form*, we understand a disjunctive normal form in which every conjunction contains at most k literals.

(1.2) **Example** Here is an important example of a Boolean function expressed by disjunctive normal form: the *selection function*. Borrowing the notation from the programming language C, we define it as

$$x?y : z = \begin{cases} y & \text{if } x = 1, \\ z & \text{if } x = 0. \end{cases}$$

It can be expressed as $x?y : z = (x \wedge y) \vee (\neg x \wedge z)$. It is possible to construct the disjunctive normal form of an arbitrary Boolean function by the repeated application of this example. $\diamondsuit$

Interchanging the role of the operations "$\wedge$" and "$\vee$", we can define the *elementary disjunction* and *conjunctive normal form*. The empty conjunction is also allowed, it is the constant 1. In general, let us call a Boolean polynomial a *tautology* if it is identically 1.

We found that all Boolean functions can be expressed by a disjunctive normal form. From the disjunctive normal form, we can obtain a conjunctive normal form, applying the distributivity property repeatedly. We have seen that this is a way to decide whether the polynomial is a tautology. Similarly, an algorithm to decide whether a polynomial is satisfiable is to bring it to a disjunctive normal form. Both algorithms can take very long time.

In general, one and the same Boolean function can be expressed in many ways as a Boolean polynomial. Given such an expression, it is easy to compute the value of the function. However, most Boolean functions can be expressed only by very large Boolean polynomials; this may even be so for Boolean functions that can be computed fast.

(1.3) **Example** [Majority Function] Let $f(x_1, \ldots, x_n) = 1$ if and only if at least half of the variables are 1. $\diamondsuit$

One reason why a computation might be much faster than the size of the Boolean polynomial is that the size of a Boolean polynomial does not reflect the possibility of reusing partial results. This deficiency is corrected by the following more general formalism.

Let G be a directed graph with numbered nodes that does not contain any directed cycle (i.e. is *acyclic*). The sources, i.e. the nodes without incoming edges, are called *input nodes*. We assign a literal (a variable or its negation) to each input node.

The sinks of the graph, i.e. those of its nodes without outgoing edges, will be called *output nodes*. (In what follows, we will deal most frequently with circuits that have a single output node.)

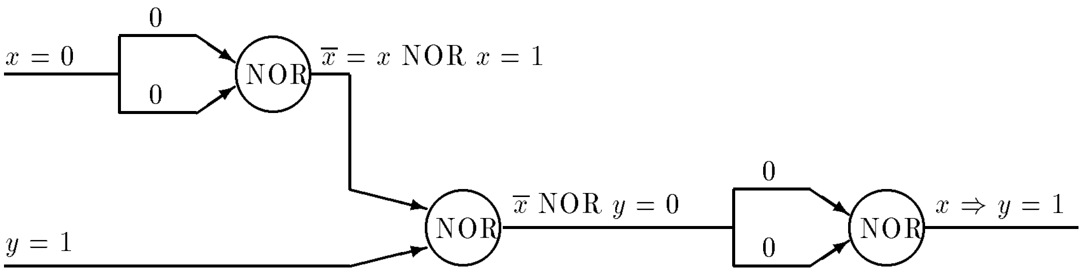

Figure 1.4: A NOR circuit computing $x \Rightarrow y$, with assignment on edges

To each node v of the graph that is not a source, i.e. which has some degree $d = d^{+}(v) > 0$, a "gate" is given, i.e. a Boolean function $F_v : \{0,1\}^d \to \{0,1\}$. The incoming edges of the node are numbered in some increasing order and the variables of the function F_v are made to correspond to them in this order. Such a graph is called a *circuit*.

The *size* of the circuit is the number of gates; its *depth* is the maximal length of paths leading from input nodes to output nodes.

Every circuit H determines a Boolean function. We assign to each input node the value of the assigned literal. This is the *input assignment*, or *input* of the computation. From this, we can compute to each node v a value $x(v) \in \{0,1\}$: if the start nodes $u_1, \ldots, u_d$ of the incoming edges have already received a value then v receives the value $F_v(x(u_1), \ldots, x(u_d))$. The values at the sinks give the *output* of the computation. We will say about the function defined this way that it is *computed* by the circuit H.

1.1 **Exercise** Prove that in the above definition, the circuit computes a unique output for every possible input assignment. $\diamondsuit$

(1.4) **Example** A NOR circuit computing $x \Rightarrow y$. We use the formulas

$$x \Rightarrow y = \neg(\neg x \text{ NOR } y), \ \neg x = x \text{ NOR } x.$$

If the states of the input lines of the circuit are x and y then the state of the output line is $x \Rightarrow y$. The assignment can be computed in 3 stages, since the longest path has 3 edges. See Figure 1.4. $\diamondsuit$

(1.5) **Example** For a natural number n we can construct a circuit that will simultaneously compute all the functions $E_{a_1,\dots,a_n}(x_1,\dots,x_n)$ (as defined above in the proof of Lemma 1.1) for all values of the vector $(a_1,\dots,a_n)$. This circuit is called the *decoder circuit* since it has the following behavior: for each input $x_1,\dots,x_n$ only one output node, namely $E_{x_1,\dots,x_n}$ will be true. If the output nodes are consecutively numbered then we can say that the circuit decodes the binary representation of a number k into the k-th position in the output. This is similar to addressing into a memory and is indeed the way a "random access" memory is addressed. Suppose that a decoder circuit is given for n. To obtain one for $n+1$, we split each output $y = E_{a_1,\dots,a_n}(x_1,\dots,x_n)$ in two, and form the new nodes

$$\begin{aligned} E_{a_1,\dots,a_n,1}(x_1,\dots,x_{n+1}) &= y \wedge x_{n+1}, \\ E_{a_1,\dots,a_n,0}(x_1,\dots,x_{n+1}) &= y \wedge \neg x_{n+1}, \end{aligned}$$

using a new copy of the input x_{n+1} and its negation. $\diamondsuit$

Of course, every Boolean function is computable by a trivial (depth 1) circuit in which a single (possibly very complicated) gate computes the output immediately from the input. The notion of circuits is interesting if we restrict the gates to some simple operations (AND, OR, exclusive OR, implication, negation, etc.). If each gate is a conjunction, disjunction or negation then using the DeMorgan rules, we can push the negations back to the inputs which, as literals, can be negated variables anyway. If all gates are disjunctions or conjunctions then the circuit is called *Boolean*. The in-degree of the nodes is often restricted to 2 or to some fixed maximum. (Sometimes, bounds are also imposed on the out-degree. This means that a partial result cannot be "freely" distributed to an arbitrary number of places.)

Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function and let

$$f(x_1,\dots,x_n) = E_1 \vee \dots \vee E_N$$

be its representation by a disjunctive normal form. This representation corresponds to a depth 2 circuit in the following manner: let its input points correspond to the variables $x_1,\dots,x_n$ and the negated variables $\overline{x}_1,\dots,\overline{x}_n$. To every elementary conjunction E_i, let there correspond a vertex into wich edges run from the input points belonging to the literals occurring in E_i, and which computes the conjunction of these. Finally, edges lead from these vertices into the output point t which computes their disjunction.

We can consider each Boolean circuit as an algorithm serving to compute some Boolean function. It can be seen immediately, however, that circuits "can do" less than e.g. Turing machines: a circuit can deal only with inputs and outputs of a given size. It is also clear that (since the graph is acyclic) the number of computation steps is bounded. If, however, we fix the length of the input and the number of steps then by an appropriate circuit, we can already simulate the work of every Turing machine computing a single bit. We can express this also by saying that every Boolean function computable by a Turing machine in a certain number of steps is also computable by a suitable, not too big, Boolean circuit.

(1.6) **Theorem** *For every Turing machine T and every pair $n, N \geq 1$ of numbers there is a Boolean circuit with n inputs, depth $O(N)$, indegree at most 2, that on an input $(x_1, \ldots, x_n) \in \{0,1\}^n$ computes 1 if and only if after N steps of the Turing machine T, on the 0'th cell of the first tape, there is a 1.*

(Without the restrictions on the size and depth of the Boolean circuit, the statement would be trivial since every Boolean function can be expressed by a Boolean circuit.)

Proof Let us be given a Turing machine $T = \langle k, \Sigma, \alpha, \beta, \gamma \rangle$ and $n, N \geq 1$. For simplicity, let us assume $k = 1$. Let us construct a directed graph with vertices $v[t, g, p]$ and $w[t, p, h]$ where $0 \leq t \leq N$, $g \in \Gamma$, $h \in \Sigma$ and $-N \leq p \leq N$. An edge runs into every point $v[t+1, g, p]$ and $w[t+1, p, h]$ from the points $v[r, g', p+\varepsilon]$ and $w[r, p+\varepsilon, h']$ ($g' \in \Gamma$, $h' \in \Sigma$, $\varepsilon \in \{-1, 0, 1\}$). Let us take n input points $s_0, \ldots, s_{n-1}$ and draw an edge from s_i into the points $w[0, i, h]$ ($h \in \Sigma$). Let the output point be $w[N, 0, 1]$.

In the vertices of the graph, the logical values computed during the evaluation of the Boolean circuit (which we will denote, for simplicity, just like the corresponding vertex) describe a computation of the machine T as follows: the value of vertex $v[t, g, p]$ is true if after step t, the control unit is in state g and the head scans the p-th cell of the tape. The value of vertex $w[t, p, h]$ is true if after step t, the p-th cell of the tape holds symbol h.

Certain ones among these logical values are given. The machine is initially in the state START, and the head starts from cell 0:

$$v[0, g, p] = 1 \quad \Leftrightarrow \quad g = START \wedge p = 0,$$

further we write the input onto cells $0, \ldots, n-1$ of the tape:

$$w[0, p, h] = 1 \Leftrightarrow ((p < 0 \vee p \geq n) \wedge h = *) \vee (0 \leq p \leq n-1 \wedge h = x_p).$$

The rules of the Turing machine tell how to compute the logical values corresponding to the rest of the vertices:

$$\begin{aligned} v[t+1,g,p]=1 &\quad\Leftrightarrow\quad \exists g'\in\Gamma,\ \exists h'\in\Sigma: \alpha(g',h')=g \wedge v[t,g',p-\gamma(g',h')]=1 \\ &\qquad\quad \wedge w[t,p-\gamma(g',h'),h']=1. \\ w[t+1,p,h]=1 &\quad\Leftrightarrow\quad (\exists g'\in\Gamma,\ \exists h'\in\Sigma: v[t,g',p]=1 \wedge w[t,p,h']=1 \wedge \beta(g',h')=h) \\ &\qquad\quad \vee(w[t,p,h]=1 \wedge \forall g'\in\Gamma: w[t,g',p]=0). \end{aligned}$$

It can be seen that these recursions can be taken as logical functions which turn the graph into a Boolean circuit computing the desired functions. The size of the circuit will be $O(N^2)$, its depth $O(N)$. Since the in-degree of each point is at most $3|\Sigma|\cdot|\Gamma|=O(1)$, we can transform the circuit into a Boolean circuit of similar size and depth. ∎

(1.7) **Remark** Our construction of a universal Turing machine in Theorem 1.1 is inefficient and unrealistic. For most commonly used transition functions α,β,γ, a table is a very inefficient way to store the description. A Boolean circuit (with a Boolean vector output) is often a vastly more economical representation. It is possible to construct a universal one-tape Turing machine V_1 taking advantage of such a representation. The beginning of the tape of this machine would not list the table of the transition function of the simulated machine, but would rather describe the Boolean circuit computing it, along with a specific state of this circuit. Each stage of the simulation would first simulate the Boolean circuit to find the values of the functions α,β,γ and then proceed as before. ♢