

Chapter 1

Our Adversary: The Circuit

Boolean (or switching) functions map each sequence of bits to a single bit 0 or 1. Bit 0 is usually interpreted as “false”, and bit 1 as “true”. The simplest of such functions are the product $x \cdot y$, sum $x \oplus y \bmod 2$, non-exclusive Or $x \vee y$, negation $\neg x = 1 - x$. The central problem of boolean function complexity—the lower bounds problem—is:

Given a boolean function, how many of these simplest operations do we need to compute the function on all input vectors?

The difficulty in proving that a given boolean function has high complexity lies in the nature of our adversary: the circuit. Small circuits may work in a counterintuitive fashion, using deep, devious, and fiendishly clever ideas. How can one prove that there is no clever way to quickly compute the function?

This is the main issue confronting complexity theorists. The problem lies on the border between mathematics and computer science: lower bounds are of great importance for computer science, but their proofs require techniques from combinatorics, algebra, analysis, and other branches of mathematics.

1.1 Boolean Functions

We first recall some basic concepts concerning boolean functions. The name “boolean function” comes from the boolean logic invented by George Boole (1815–1864), an English mathematician and philosopher. As this logic is now the basis of modern digital computers, Boole is regarded in hindsight as a forefather of the field of computer science.

Boolean values (or bits) are numbers 0 and 1. A *boolean function* $f(x) = f(x_1, \dots, x_n)$ of n variables is a mapping $f : \{0, 1\}^n \rightarrow \{0, 1\}$. One says that f *accepts* a vector $a \in \{0, 1\}^n$ if $f(a) = 1$, and *rejects* it if $f(a) = 0$.

A boolean function $f(x_1, \dots, x_n)$ need not to depend on all its variables. One says that f *depends* on its i -th variable x_i if there exist constants $a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$ in $\{0, 1\}$ such that

$$f(a_1, \dots, a_{i-1}, 0, a_{i+1}, \dots, a_n) \neq f(a_1, \dots, a_{i-1}, 1, a_{i+1}, \dots, a_n).$$

Since we have 2^n vectors in $\{0, 1\}^n$, the total number of boolean functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is doubly-exponential in n , is 2^{2^n} . A boolean function f is *symmetric* if it depends only on the number of ones in the input, and not on positions in which these ones actually reside. We thus have only 2^{n+1} such functions of n variables. Examples of symmetric boolean functions are:

- Threshold functions $\text{Th}_k^n(x) = 1$ iff $x_1 + \dots + x_n \geq k$.
- Majority function $\text{Maj}_n(x) = 1$ iff $x_1 + \dots + x_n \geq \lceil n/2 \rceil$.
- Parity function $\oplus_n(x) = 1$ iff $x_1 + \dots + x_n \equiv 1 \pmod 2$.
- Modular functions $\text{MOD}_k = 1$ iff $x_1 + \dots + x_n \equiv 0 \pmod k$.

Besides these, there are many other interesting boolean functions. Actually, *any* property (which may or may not hold) can be encoded as a boolean function. For example, the property “to be a prime number” corresponds to a boolean function PRIME such that $\text{PRIME}(x) = 1$ iff $\sum_{i=1}^n x_i 2^{i-1}$ is a prime number. It was a long-standing problem whether this function can be uniformly computed using a polynomial in n number of elementary boolean operations. This problem was finally solved affirmatively by Agrawal et al. (2004). The *existence* of small circuits for PRIME for every single n was known long ago.

To encode properties of graphs on the set of vertices $[n] = \{1, \dots, n\}$, we may associate a boolean variable x_{ij} with each potential edge. Then any 0–1 vector x of length $\binom{n}{2}$ gives us a graph G_x , where two vertices i and j are adjacent iff $x_{ij} = 1$. We can then define $f(x) = 1$ iff G_x has a particular property. A prominent example of a “hard-to-compute” graph property is the *clique function* $\text{CLIQUE}(n, k)$: it accepts an input vector x iff the graph G_x has a k -clique, that is, a complete subgraph on k vertices. The problem of whether this function can also be computed using a polynomial number of operations remains wide open. A negative answer would immediately imply that $\text{P} \neq \text{NP}$. Informally, the P vs. NP problem asks whether there exist mathematical theorems whose proofs are much harder to find than verify.

Roughly speaking, one of the goals of circuit complexity is, for example, to understand *why* the first of the following two problems is easy whereas the second is (apparently) very hard to solve:

1. Does a given graph contain at least $\binom{k}{2}$ edges?
2. Does a given graph contain a clique with $\binom{k}{2}$ edges?

The first problem is a threshold function, whereas the second is the clique function $\text{CLIQUE}(n, k)$. We stress that the goal of circuit complexity is not just to give an “evidence” (via some indirect argument) that clique *is* much harder than majority, but to understand *why* this is so.

A *boolean matrix* or a 0–1 matrix is a matrix whose entries are 0s and 1s. If $f(x, y)$ is a boolean function of $2n$ variables, then it can be viewed as a boolean $2^n \times 2^n$ matrix A whose rows and columns are labeled by vector in $\{0, 1\}^n$, and $A[x, y] = f(x, y)$.

x	y	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \rightarrow y$	x	$\neg x$
0	0	0	0	0	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	1	0		
1	1	1	1	0	1		

Fig. 1.1 Truth tables of basic boolean operations

We can obtain new boolean functions (or matrices) by applying boolean operations to the “simplest” ones. Basic boolean operations are:

- NOT (negation) $\neg x = 1 - x$; also denoted as \bar{x} .
- AND (conjunction) $x \wedge y = x \cdot y$.
- OR (disjunction) $x \vee y = 1 - (1 - x)(1 - y)$.
- XOR (parity) $x \oplus y = x(1 - y) + y(1 - x) = (x + y) \bmod 2$.
- Implication $x \rightarrow y = \neg x \vee y$ (Fig. 1.1).

If these operators are applied to boolean vectors or boolean matrices, then they are usually performed componentwise. Negation acts on ANDs and ORs via *DeMorgan rules*:

$$\neg(x \vee y) = \neg x \wedge \neg y \text{ and } \neg(x \wedge y) = \neg x \vee \neg y.$$

The operations AND and OR themselves enjoy the distributivity rules:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \text{ and } x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z).$$

Binary cube The set $\{0, 1\}^n$ of all boolean (or binary) vectors is usually called the *binary n -cube*. A *subcube of dimension d* is a set of the form $A = A_1 \times A_2 \times \cdots \times A_n$, where each A_i is one of three sets $\{0\}$, $\{1\}$ and $\{0, 1\}$, and where $A_i = \{0, 1\}$ for exactly d of the i s. Note that each subcube of dimension d can be uniquely specified by a vector $a \in \{0, 1, *\}^n$ with d stars, by letting $*$ to attain any of two values 0 and 1. For example, a subcube $A = \{0\} \times \{0, 1\} \times \{1\} \times \{0, 1\}$ of the binary 4-cube of dimension $d = 2$ is specified by $a = (0, *, 1, *)$.

Usually, the binary n -cube is considered as a *graph* Q_n whose vertices are vectors in $\{0, 1\}^n$, and two vectors are adjacent iff they differ in exactly one position (see Fig. 1.2). This graph is sometimes called the *n -dimensional binary hypercube*. This is a regular graph of degree n with 2^n vertices and $n2^{n-1}$ edges. Moreover, the graph is bipartite: we can put all vectors with an odd number of ones on one side, and the rest on the other; no edge of Q_n can join two vectors on the same side.

Every boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is just a coloring of vertices of Q_n in two colors. The bipartite subgraph G_f of Q_n , obtained by removing all edges joining the vertices in the same color class, accumulates useful information about the circuit complexity of f . If, for example, d_a denotes the average degree in G_f

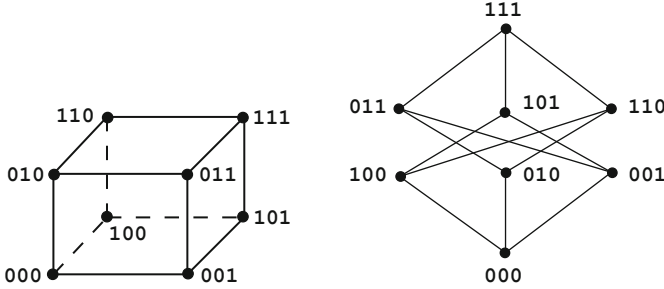


Fig. 1.2 The 3-cube and its Hasse-type representation (each level contains binary strings with the same number of 1s). There is an edge between two strings if and only if they differ in exactly one position

of vertices in the color-class $f^{-1}(a)$, $a = 0, 1$, then the product $d_0 \cdot d_1$ is a lower bound on the length of any formula expressing f using connectives \wedge , \vee and \neg (see Khrapchenko's theorem in Sect. 6.8).

CNFs and DNFs A trivial way to represent a boolean function $f(x_1, \dots, x_n)$ is to give the entire *truth table*, that is, to list all 2^n pairs $(a, f(a))$ for $a \in \{0, 1\}^n$. More compact representations are obtained by giving a covering of $f^{-1}(0)$ or of $f^{-1}(1)$ by not necessarily disjoint subsets, each of which has some “simple” structure. This leads to the notions of CNFs and DNFs.

A *literal* is a boolean variable or its negation. For literals the following notation is often used: x_i^1 stands for x_i , and x_i^0 stands for $\neg x_i = 1 - x_i$. Thus, for every binary string $a = (a_1, \dots, a_n)$ in $\{0, 1\}^n$,

$$x_i^1(a) = \begin{cases} 1 & \text{if } a_i = 1 \\ 0 & \text{if } a_i = 0 \end{cases} \quad \text{and} \quad x_i^0(a) = \begin{cases} 0 & \text{if } a_i = 1 \\ 1 & \text{if } a_i = 0. \end{cases}$$

A *monomial* is an AND of literals, and a *clause* is an OR of literals. A monomial (or clause) is *consistent* if it does not contain a contradicting pair of literals x_i and \bar{x}_i of the same variable. We will often view monomials and clauses as *sets* of literals.

It is not difficult to see that the set of all vectors *accepted* by a monomial consisting of k (out of n) literals forms a binary n -cube of dimension $n - k$ (so many bits are not specified). For example, a monomial $\bar{x}_1 \wedge x_3$ defines the cube of dimension $n - 2$ specified by $a = (0, *, 1, *, \dots, *)$. Similarly, the set of all vectors *rejected* by a clause consisting of k (out of n) literals also forms a binary n -cube of dimension $n - k$. For example, a clause $\bar{x}_1 \vee x_3$ rejects a vector a iff $a_1 = 1$ and $a_3 = 0$.

A *DNF* (disjunctive normal form) is an OR of monomials, and a *CNF* (conjunctive normal form) is an AND of clauses. Every boolean function $f(x)$ of n variables can be written both as a DNF $D(x)$ and as a CNF $C(x)$:

$$D(x) = \bigvee_{a: f(a)=1} \bigwedge_{i=1}^n x_i^{a_i} \quad C(x) = \bigwedge_{b: f(b)=0} \bigvee_{i=1}^n x_i^{1-b_i}.$$

Indeed, $D(x)$ *accepts* a vector x iff x coincides with at least one vector a accepted by f , and $C(x)$ *rejects* a vector x iff x coincides with at least one vector b rejected by f .

A DNF is a k -DNF if each of its monomials has at most k literals; similarly, a CNF is a k -CNF if each of its clauses has at most k literals.

DNFs (and CNFs) are the simplest models for computing boolean functions. The *size* of a DNF is the total number of monomials in it. It is clear that every boolean function of n variables can be represented by a DNF of size at most $|f^{-1}(1)| \leq 2^n$: just take one monomial for each accepted vector. This can also be seen via the following recurrence:

$$f(x_1, \dots, x_{n+1}) = x_{n+1} \wedge f(x_1, \dots, x_n, 1) \vee \neg x_{n+1} \wedge f(x_1, \dots, x_n, 0). \quad (1.1)$$

It is not difficult to see that some functions require DNFs of exponential size. Take, for example, the parity function $f(x_1, \dots, x_n) = x_1 \oplus x_2 \oplus \dots \oplus x_n$. This function accepts an input vector iff the number of 1s in it is odd. Every monomial in a DNF for f must contain n literals, for otherwise the DNF would be forced to accept a vector in $f^{-1}(0)$. Since any such monomial can accept only one vector, $|f^{-1}(1)| = 2^{n-1}$ monomials are necessary. Thus the lower bounds problem for this model is trivial.

Boolean functions as set systems By identifying subsets S of $[n] = \{1, \dots, n\}$ with their characteristic 0–1 vectors v_S , where $v_S(i) = 1$ iff $i \in S$, we can consider boolean functions as set-theoretic predicates $f : 2^{[n]} \rightarrow \{0, 1\}$. We will often go back and forth between these notations. One can identify a boolean function $f : 2^{[n]} \rightarrow \{0, 1\}$ with the family $\mathcal{F}_f = \{S : f(S) = 1\}$ of subsets of $[n]$. That is, there is a 1-to-1 correspondence between boolean functions and families of subsets of $[n]$:

boolean functions of n variables = families of subsets of $\{1, \dots, n\}$.

Minterms and maxterms A 1-term (resp., 0-term) of a boolean function is a smallest subset of its variables such that the function can be made the constant 1 (resp., constant 0) function by fixing these variables to constants 0 and 1 in some way. Thus after the setting, the obtained function does not depend on the remaining variables. Minterms (maxterms) are 1-terms (0-terms) which are minimal under the set-theoretic inclusion.

Note that one and the same set of variables may be a 1-term and a 0-term at the same time. If, for example, $f(x_1, x_2, x_3) = 1$ iff $x_1 + x_2 + x_3 \geq 2$, then $\{x_1, x_2\}$ is a 1-term of f because $f(1, 1, x_3) \equiv 1$, and is a 0-term of f because $f(0, 0, x_3) \equiv 0$.

If all minterms of a boolean function f have length at most k then f can be written as a k -DNF: just take the OR of all these minterms. But the converse does not hold! Namely, there are boolean functions f such that f *can* be written as a k -DNF even though some of its minterms are much longer than k (see Exercise 1.7).

Duality The *dual* of a boolean function $f(x_1, \dots, x_n)$ is the boolean function f^* defined by:

$$f^*(x_1, \dots, x_n) := \neg f(\neg x_1, \dots, \neg x_n).$$

For example, if $f = x \vee y$ then $f^* = \neg(\neg x \vee \neg y) = x \wedge y$. The dual of every threshold function $\text{Th}_k^n(x)$ is the threshold function $\text{Th}_{n-k+1}^n(x)$. A function f is *self-dual* if $f^*(x) = f(x)$ holds for all $x \in \{0, 1\}^n$. For example, the threshold- k function $f(x) = \text{Th}_k^{2k-1}(x)$ of $2k - 1$ variables is self-dual. Hence, if the number n of variables is odd, then the majority function Maj_n is also self-dual.

In set-theoretic terms, if $\bar{S} = [n] \setminus S$ denotes the complement of S , then the values of the dual of f are obtained by: $f^*(S) = 1 - f(\bar{S})$. Thus a boolean function f is self-dual if and only if $f(\bar{S}) + f(S) = 1$ for all $S \subseteq [n]$.

Monotone functions For two vectors $x, y \in \{0, 1\}^n$ we write $x \leq y$ if $x_i \leq y_i$ for all positions i . A boolean function $f(x)$ is *monotone*, if $x \leq y$ implies $f(x) \leq f(y)$. If we view f as a set-theoretic predicate $f : 2^{[n]} \rightarrow \{0, 1\}$, then f is monotone iff $f(S) = 1$ and $S \subseteq T$ implies $f(T) = 1$. Examples of monotone boolean functions are AND, OR, threshold functions $\text{Th}_k^n(x)$, clique functions $\text{CLIQUE}(n, k)$, etc. On the other hand, such functions as the parity function $\oplus_n(x)$ or counting functions $\text{Mod}_k^n(x)$ are not monotone.

Monotone functions have many nice properties not shared by other functions. First of all, their minterms as well as maxterms are just subsets of variables (no negated variable occurs in them). In set-theoretic terms, a subset $S \subseteq [n]$ is a minterm of a monotone function f if

$$f(S) = 1 \text{ but } f(S \setminus \{i\}) = 0 \text{ for all } i \in S,$$

and is a maxterm of f if

$$f(\bar{S}) = 0 \text{ but } f(\overline{S \setminus \{i\}}) = 1 \text{ for all } i \in S.$$

Let $\text{Min}(f)$ and $\text{Max}(f)$ denote the set of all minterms and the set of all maxterms of f . Then we have the following *cross-intersection property*:

$$S \cap T \neq \emptyset \text{ for all } S \in \text{Min}(f) \text{ and all } T \in \text{Max}(f).$$

Indeed, if S and T were disjoint, then for the vectors x with $x_i = 1$ for all $i \in S$, and $x_i = 0$ for all $i \notin S$, we would have $f(x) = 1$ (because S is a minterm) and at the same time $f(x) = 0$ (because $T \subseteq \bar{S}$ is a maxterm of f).

The next important property of monotone boolean functions is that every such function f has a *unique* representation as a DNF as well as a CNF:

$$f(x) = \bigvee_{S \in \text{Min}(f)} \bigwedge_{i \in S} x_i = \bigwedge_{T \in \text{Max}(f)} \bigvee_{i \in T} x_i.$$

Moreover, for every monotone boolean function f we have the following three equivalent conditions of their self-duality:

- $\text{Min}(f) = \text{Max}(f)$.
- Both families $\text{Min}(f)$ and $\text{Max}(f)$ are intersecting: $S \cap S' \neq \emptyset$ for all $S, S' \in \text{Min}(f)$, and $T \cap T' \neq \emptyset$ for all $T, T' \in \text{Max}(f)$.

- The family $\text{Min}(f)$ is intersecting and, for every partition of $[n]$ into two parts, at least one minterm lies in one of these parts.

Equivalence of the first condition $\text{Min}(f) = \text{Max}(f)$ with the definition of self-duality ($f(\bar{S}) = 1 - f(S)$ for all $S \subseteq [n]$) is not difficult to see. To show that also the second and the third conditions are equivalent, needs a bit more work.

In the rest of this section we recall some facts that turn out to be very useful when analyzing circuits. We include them right here both because they have elegant proofs and because we will use them later several times.

Functions with many subfunctions A *subfunction* of a boolean function $f(x_1, \dots, x_n)$ is obtained by fixing some of its variables to constants 0 and 1. Since each of the n variables has three possibilities (to be set to 0 or to 1 or remain unassigned), one function can have at most 3^n subfunctions.

If Y is some subset of variables, then a *subfunction of f on Y* is a boolean function of variables Y obtained from f by setting all the variables outside Y to constants 0 and 1, in some way. Some settings may lead to the same subfunction. So let $N_Y(f)$ denote the number *distinct* subfunctions of f on Y . It is not difficult to see that, if $|Y| = m$, then

$$N_Y(f) \leq \min\{2^{n-m}, 2^{2^m}\}.$$

Indeed, we have at most 2^{n-m} possibilities to assign constants to $n - |Y|$ variables, and there are at most 2^{2^m} distinct boolean functions on the same set Y of m variables. But some functions f may have fewer distinct subfunctions. For example, the parity function $\oplus_n(x) = x_1 \oplus x_2 \oplus \dots \oplus x_n$ has only $N_Y(\oplus_n) = 2$ different subfunctions. On the other hand, we will show later (in Sect. 6.5) that functions with many subfunctions cannot be “too easy”. So what functions have many subfunctions?

The simplest known example of a function with almost maximal possible number of distinct subfunctions is the *element distinctness function* $\text{ED}_n(x)$ suggested by Beame and Cook (unpublished). This is a boolean function of¹ $n = 2m \log m$ variables divided into m consecutive blocks Y_1, \dots, Y_m with $2 \log m$ variables in each of them; m is assumed to be a power of 2. Each of these blocks encode a number in $[m^2] = \{1, 2, \dots, m^2\}$. The function accepts an input $x \in \{0, 1\}^n$ if and only if all these numbers are distinct.

Lemma 1.1. *On each block, ED_n has at least $2^{n/2}/n$ subfunctions.*

Proof. It suffices to prove this for the first block Y_1 . So let $N = N_{Y_1}(\text{ED}_n)$, and consider the function f of m variables, each taking its value in $[m^2]$. The function accepts a string (a_1, \dots, a_m) of numbers in $[m^2]$ iff all these numbers are distinct. Thus $\text{ED}_n(x)$ is just a boolean version of f .

¹If not said otherwise, all logarithms in this book are to the basis of 2.

For a string $a = (a_2, \dots, a_m)$ of numbers $[m^2]$, let $f_a : [m^2] \rightarrow \{0, 1\}$ be the function $f_a(x) := f(x, a_2, \dots, a_m)$ obtained from f by fixing its last $m - 1$ variables. Note that N is exactly the number of distinct functions f_a .

The number of ways to choose a string $a = (a_2, \dots, a_m)$ with all the a_i distinct is $\binom{m^2}{m-1}(m-1)!$: each such string is obtained by taking an $(m-1)$ -element subset of $[m^2]$ and permuting its elements. If $b = (b_2, \dots, b_m)$ is another such string, and if b is not a permutation of a , then there must be an a_i such that $a_i \notin \{b_2, \dots, b_m\}$. But for such an a_i , we have that $f_a(a_i) = 0$ whereas $f_b(a_i) = 1$; hence, $f_a \neq f_b$. Since there are only $(m-1)!$ permutations of a , we obtain that $N \geq \binom{m^2}{m-1} \geq m^{m-1} \geq 2^{n/2}/n$. \square

Matrix decomposition A matrix B is *primitive* if it is boolean (has only entries 0 and 1) and has rank 1 over the reals. Each such matrix consists of one all-1 submatrix and zeros elsewhere. The *weight*, $w(B)$, of such a matrix is $r + c$, where r is the number of nonzero rows, and c the number of nonzero columns in B . Here is a primitive 4×5 matrix of weight $2 + 3 = 5$:

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Primitive matrices are important objects—we will use them quite often.

A *decomposition* of a boolean $m \times n$ matrix A is a set B_1, \dots, B_t of primitive $m \times n$ matrices such that A can be written as the sum $A = B_1 + B_2 + \dots + B_t$ of these matrices over the reals. That is, each 1-entry of A is a 1-entry in exactly one of the matrices B_i , and each 0-entry is a 0-entry in all matrices. The *weight* of such a decomposition is the sum $\sum_{i=1}^t w(B_i)$ of weights of the B_i . Let $\text{Dec}(A)$ denote the minimum weight of a decomposition of a boolean matrix A , and let $|A|$ denote the number of 1-entries in A .

Note that $\text{Dec}(A) \leq mn$: just decompose A into m primitive matrices corresponding to the rows of A . In fact, we have a better upper bound.

Lemma 1.2. (Lupanov 1956) *For every boolean $m \times n$ matrix,*

$$\text{Dec}(A) \leq (1 + o(1)) \frac{mn}{\log m}.$$

Proof. We first prove that for every boolean $m \times n$ matrix A and for every integer $1 \leq k \leq m$,

$$\text{Dec}(A) \leq \frac{mn}{k} + n2^{k-1}. \quad (1.2)$$

We first prove (1.2) for $k = n$, that is, we prove the upper bound

$$\text{Dec}(A) \leq m + n2^{n-1}. \quad (1.3)$$

Split the rows of A into groups, where the rows in one group all have the same values. This gives us a decomposition of A into $t \leq 2^n$ primitive matrices. For the i -th of these matrices, let r_i be the number of its nonzero rows, and c_i the number of its nonzero columns. Hence, $r_i + c_i$ is the weight of the i -th primitive matrix. Since each nonzero row of A lies in exactly one of the these matrices, the total weight of the decomposition is

$$\sum_{i=1}^t r_i + \sum_{i=1}^t c_i \leq m + \sum_{j=0}^n \sum_{i:c_i=j} j \leq m + \sum_{j=0}^n \binom{n}{j} \cdot j = m + n2^{n-1},$$

where the last equality is easy to prove: just count in two ways the number of pairs (x, S) with $x \in S \subseteq \{1, \dots, n\}$.

To prove (1.2) for arbitrary integer $1 \leq k \leq n$, split A into submatrices with k columns in each (one submatrix may have fewer columns). For each of these n/k submatrices, (1.3) gives a decomposition of weight at most $m + k2^{k-1}$. Thus, for every $1 \leq k \leq n$, every $m \times n$ matrix has a decomposition of weight at most $mn/k + n2^{k-1}$.

To finish the proof of the theorem, it is enough to apply (1.2) with k about $\log m - 2 \log \log m$. \square

Using a counting argument, Lupanov (1956) also showed that the upper bound given in Lemma 1.2 is almost optimal: $m \times n$ matrices A requiring weight

$$\text{Dec}(A) \geq (1 + o(1)) \frac{mn}{\log(mn)}$$

in any decomposition exist, even if the 1-entries in primitive matrices are allowed to overlap (cf. Theorem 13.18). Apparently, this paper of Lupanov remained unknown in the West, because this result was later proved by Tuza (1984) and Bublitz (1986).

Splitting a graph When trying to “balance” some computational models (decision trees, formulas, communication protocols, logical derivations) the following two structural facts are often useful.

Let G be a directed acyclic graph with one source node (the root) from which all leaves (nodes of outdegree 0) are reachable. Suppose that each non-leaf node has outdegree k . Suppose also that each vertex is assigned a non-negative weight which is *subadditive*: the weight of a node does not exceed the sum of the weights of its successors. Let r be the weight of the root, and suppose that each leaf has weight at most $l < r$.

Lemma 1.3. *For every real number ϵ between l/r and 1, there exists a node whose weight lies between $\epsilon r/k$ and ϵr . In particular, every binary tree with r leaves has a subtree whose number of leaves lies between $r/3$ and $2r/3$.*

Proof. Start at the root and traverse the graph until a node u of weight $> \epsilon r$ is found such that each of its successors has weight at most ϵr . Such a node u exists because each leaf has weight at most $l \leq \epsilon r$. Due to subadditivity of the weight function,

the (up to k) successors of u cannot *all* have weight $\leq \epsilon r/k$, since then the weight of u would be $\leq \epsilon r$ as well. Hence, the weight of at least one successor of u must lie between $\epsilon r/k$ and ϵr , as desired.

To prove the second claim, give each leaf of the tree weight 1, and define the weight of an inner node as the number of leaves in the corresponding subtree. Then apply the previous claim with $k = 2$ and $\epsilon = 2/3$. \square

The *length* of a path we will mean the number of nodes in it. The *depth* of a graph is the length of a longest path in it. The following lemma generalizes and simplifies an analogous result of Erdős et al. (1976). Let $d = 2^k$ and $1 \leq r \leq k$ be integers.

Lemma 1.4. (Valiant 1977) *In any directed graph with S edges and depth d it is possible to remove rS/k edges so that the depth of the resulting graph does not exceed $d/2^r$.*

Proof. A *labeling* of a graph is a mapping of the nodes into the integers. Such a labeling is *legal* if for each edge (u, v) the label of v is strictly greater than the label of u . A *canonical labeling* is to assign each node the length of a longest directed path that terminates at that node. If the graph has depth d then this gives us a labeling using only d labels $1, \dots, d$. It is easy to verify that this is a legal labeling: if (u, v) is an edge then any path terminating in u can be prolonged to a path terminating in v . On the other hand, since in any legal labeling, all labels along a directed path must be distinct, we have that the depth of a graph does not exceed the number of labels used by any legal labeling.

After these preparations, consider now any directed graph with S edges and depth d , and consider the canonical labeling using labels $1, \dots, d$. For $i = 1, \dots, k$ (where $k = \log d$), let E_i be the set of all edges, the binary representations of labels of whose endpoints differ in the i -th position (from the left) for the *first time*.

If E_i is removed from the graph, then we can relabel the nodes using integers $1, \dots, d/2$ by simply deleting the i -th bit in the binary representations of labels. It is not difficult to see that this is a legal labeling (of a new graph): if an edge (u, v) survived, then the first difference between the binary representations of the old labels of u and v were *not* in the i -th position; hence, the new label of u remains strictly smaller than that of v . Consequently, if any $r \leq k$ of the *smallest* sets E_i are removed, then at most rS/k edges are removed, and a graph of depth at most $d/2^r$ remains. \square

1.2 Circuits

In this section we recall the most fundamental models for computing boolean functions.

General circuits Let Φ be a set of some boolean functions. A *circuit* (or a *straight line program*) of n variables over the basis Φ is just a sequence g_1, \dots, g_t of $t \geq n$ boolean functions such that the first n functions are input variables $g_1 =$

$x_1, \dots, g_n = x_n$, and each subsequent g_i is an application $g_i = \varphi(g_{i_1}, \dots, g_{i_d})$ of some basis function $\varphi \in \Phi$ (called the *gate* of g_i) to some previous functions.

That is, the value $g_i(a)$ of the i -th gate g_i on a given input $a \in \{0, 1\}^n$ is the value of the boolean function φ applied to the values $g_{i_1}(a), \dots, g_{i_d}(a)$ computed at the previous gates. A circuit computes a boolean function (or a set of boolean functions) if it (or they) are among the g_i .

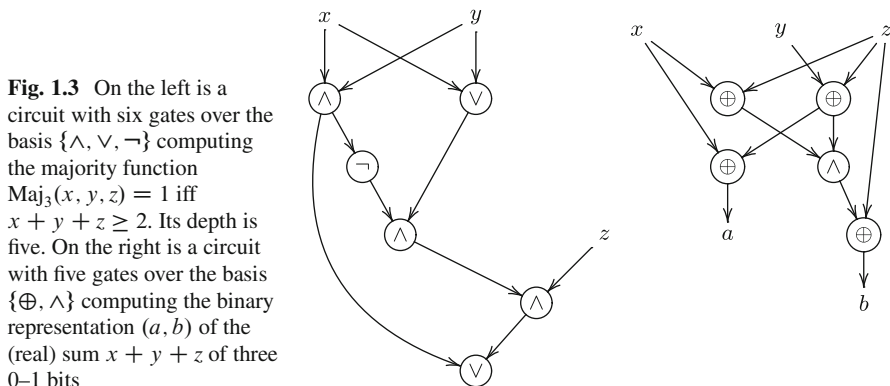
Each circuit can be viewed as a directed acyclic graph whose fanin-0 nodes (those of zero in-degree) correspond to variables, and each other node v corresponds to a function φ in Φ . One (or more) nodes are distinguished as outputs. The value at a node is computed by applying the corresponding function to the values of the preceding nodes (see Fig. 1.3).

In the literature circuits are usually drawn in a “bottom-up” manner: the first (lowest) level consists of inputs, and the last (highest) level consists of output gates. We will, however, mostly draw circuits in a more natural “top-down” manner: inputs at the top, and outputs at the bottom. Only where there already are established terms “top gate” and “bottom level” we will use bottom-up drawings.

The *size* of the circuit is the total number $t - n$ of its gates (that is, we do not count the input variables), and its *depth* is the length of a longest path from an input to an output gate. More precisely, input variables have depth 0, and if $g_i = \varphi(g_{i_1}, \dots, g_{i_d})$ then the depth of the gate g_i is 1 plus the maximum depth of the gates g_{i_1}, \dots, g_{i_d} . We will assume that every circuit can use constants 0 and 1 as inputs for free.

Formulas A *formula* is a circuit all whose gates have fanout at most 1. Hence, the underlying graph of a formula is a tree. The *size* of a formula is also the number of gates, and the *leafsize* of a is the number of input gates, that is, the number of leaves in its tree, and the *depth* of a formula is the depth of its tree. Note that the only (but crucial) difference of formulas from circuits is that in the circuit model a result computed at some gate can be used many times with no need to recompute it again and again, as in the case of formulas.

DeMorgan circuits A *DeMorgan circuit* is a circuit over the basis $\{\wedge, \vee\}$ but the inputs are variables and their negations. That is, these are the circuits over the basis



$\{\wedge, \vee, \neg\}$, where NOT gates are only applied to input variables; these gates do not contribute to the circuit size. Such circuits are also called *circuits with tight negations*. If there are no negated variables as inputs, then the circuit is *monotone*. By using DeMorgan rules $\neg(x \vee y) = \neg x \wedge \neg y$ and $\neg(x \wedge y) = \neg x \vee \neg y$, it can be easily shown that any circuit over $\{\wedge, \vee, \neg\}$ can be reduced to this form by at most doubling the total number of gates; the depth of the circuit remains the same. In the case of formulas, even the leafsize remains the same.

Probabilistic circuits Such circuits have, besides standard input variables x_1, \dots, x_n , some specially designed inputs r_1, \dots, r_m called random inputs. When these random inputs are chosen from a uniform distribution on $\{0, 1\}$, the output $C(x)$ of the circuit is a random 0–1 variable. A probabilistic circuit $C(x)$ computes a boolean function $f(x)$ if

$$\text{Prob}[C(x) = f(x)] \geq 3/4 \text{ for each } x \in \{0, 1\}^n.$$

There is nothing special about using the constant $3/4$ here—one can take any constant $> 1/2$ instead. The complexity would not change by more than a constant factor.

Can probabilistic circuits have much smaller size than usual (deterministic) circuits? We will answer this question *negatively* using the following simple (but often used) “majority trick”. It implies that if a random circuit errs on a fixed input with probability $< 1/2$, then the majority of not too many independent copies of such a circuit will err on this input with *exponentially* small probability. A Bernoulli random variable with success probability p is a 0–1 random variable taking the value 1 with probability p .

Lemma 1.5. (Majority trick) *If x_1, \dots, x_m are independent Bernoulli random variables with success probability $1/2 + \epsilon$, then*

$$\text{Prob}[\text{Maj}(x_1, \dots, x_m) = 0] \leq e^{-2\epsilon^2 m}.$$

Proof. Let \mathcal{F} be the family of all subsets of $[m] = \{1, \dots, m\}$ of size $> m/2$, and let $q := \text{Prob}[\text{Maj}(x_1, \dots, x_m) = 0]$. Then

$$\begin{aligned} q &= \sum_{S \in \mathcal{F}} \text{Prob}[x_i = 0 \text{ for all } i \in S] \cdot \text{Prob}[x_i = 1 \text{ for all } i \notin S] \\ &= \sum_{S \in \mathcal{F}} (1/2 - \epsilon)^{|S|} (1/2 + \epsilon)^{m-|S|} \\ &\leq \sum_{S \in \mathcal{F}} (1/2 - \epsilon)^{m/2} (1/2 + \epsilon)^{m/2} \\ &\leq 2^m (1/4 - \epsilon^2)^{m/2} = (1 - 4\epsilon^2)^{m/2} \leq e^{-2\epsilon^2 m}. \end{aligned}$$

The first inequality here follows by multiplying each term by

$$(1/2 - \epsilon)^{m/2 - |S|} (1/2 + \epsilon)^{|S| - m/2} \geq 1. \quad \square$$

Theorem 1.6. (Adleman 1978) *If a boolean function f of n variables can be computed by a probabilistic circuit of size M , then f can be computed by a deterministic circuit of size at most $8nM$.*

Proof. Let C be a probabilistic circuit that computes f . Take m independent copies C_1, \dots, C_m of this circuit (each with its own random inputs), and consider the probabilistic circuit C' that computes the majority of the outputs of these m circuits. Fix a vector $a \in \{0, 1\}^n$, and let x_i be an indicator random variable for the event “ $C_i(a) = f(a)$ ”. For each of these random variables we have that $\text{Prob}[x_i = 1] \geq 1/2 + \epsilon$ with $\epsilon = 1/4$. By the majority trick, the circuit C' will err on a with probability at most $e^{-2\epsilon^2 m} = e^{-m/8}$. By the union bound, the probability that the new circuit C' makes an error on at least one of all 2^n possible inputs a is at most $2^n \cdot e^{-m/8}$. If we take $m = 8n$, then this probability is smaller than 1. Therefore, there must be a setting of the random inputs which gives the correct answer for all inputs. The obtained circuit is no longer probabilistic, and its size is at most $8n$ times larger than the size of the probabilistic circuit. \square

Average time of computations Let $C = (g_1, \dots, g_s)$ be a circuit computing some boolean function $f(x)$ of n variables; hence, $g_s(x) = f(x)$. The number s of gates is the size of the circuit. One can also consider a notion of “computation time” on a given input $a \in \{0, 1\}^n$. For this, let us introduce one special boolean variable z , the *output* variable. Some of the gates may reset this variable, that is, set $z = g_i(a)$. In particular, gates of the form $z = 0$ and $z = 1$ are allowed. The last gate g_s always does this, that is, sets $z = g_s(a)$. Our goal however is to interrupt the computation sequence $g_1(a), \dots, g_s(a)$ as soon as the output variable already has the correct value $z = f(a)$.

To realize this goal, we declare some gates as “stop-gates”. Such a gate g stops the computation on an input a if $g(a) = 1$. Now, given an input $a \in \{0, 1\}^n$, a computation $g_1(a), g_2(a), \dots, g_i(a)$ continues until the first gate g_i is found such that g_i is a stop-gate and $g_i(a) = 1$. The computation on a then stops, and the output $C(a)$ of the circuit is the actual value of the output variable z at this moment (see Fig. 1.4). The computation time $t_C(a)$ of the circuit C on a is the number i of gates evaluated until the value was computed. The *average time* of the circuit C is

$$t(C) = 2^{-n} \sum_{a \in \{0, 1\}^n} t_C(a).$$

If we have no stop-gates at all, then $t_C(a) = s$ for all inputs a , and hence, the average time $t(C)$ of the circuit C is just the size s of C .

This model of *stop-circuits* was introduced by Chashkin (1997, 2000, 2004); he calls this model “non-branching programs with conditional stop”.

$z = 1$	$z = x_1 \vee x_2$ (stop)	$g_1 = x_1 \vee x_2$
$g_1 = x_1$ (stop)	$z = x_3 \vee x_4$	$g_2 = x_3 \vee x_4$
$g_2 = x_2$ (stop)		$z = g_1 \vee g_2$
$g_3 = x_3$ (stop)		
$g_4 = x_4$ (stop)		
$z = 0$		

Fig. 1.4 Three circuits computing the OR $x_1 \vee x_2 \vee x_3 \vee x_4$ of four variables. On input $a = (0, 1, 0, 0)$ the first circuit takes time $t_C(a) = 3$, the second takes time $t_C(a) = 1$, and the third (standard) circuit takes time $t_C(a) = 3$. The average time of the last circuit is $t(C) = 3$, whereas that of the middle circuit is $t(C) = \frac{1}{16}(12 \cdot 1 + 4 \cdot 2) = 5/4$

The average time, $t(f)$, of a boolean function f is the minimum average time of a circuit computing f . We always have that $t(f) \leq C(f)$. Chashkin (1997) showed that boolean functions f of n variables requiring $t(f) = \Omega(2^n/n)$ exist. But some functions have much smaller average time than $C(f)$.

Example 1.7. Consider the threshold-2 function $\text{Th}_2^n(x)$. Since every boolean function f , which depends on n variables, requires at least $n - 1$ gates, we have that $C(\text{Th}_2^n) \geq n - 1$. On the other hand, it is not difficult to show that $t(\text{Th}_2^n) = \mathcal{O}(1)$. To see this, let us first compute $z = \text{Th}_2^3(x_1, x_2, x_3)$. This can be done using 6 gates (see Fig. 1.3), and hence, can be computed in time 6. After that we compute $z = \text{Th}_2^3(x_4, x_5, x_6)$, and so on. Declare each gate re-setting the variable z as a stop-gate. This way the computations on $4^2 2^{n-6} = 2^{n-1}$ inputs will be stopped after 6 steps, the computations on $4^2 2^{n-6} = 2^{n-2}$ remaining inputs will be stopped after $6 \cdot 2 = 12$ steps and, in general, the computations on $4^t 2^{n-3t} = 2^{n-t}$ inputs will be stopped after $6t$ steps. Thus, the average computation times is at most $\sum_{t=1}^{n/3} 6t 2^{-t} = \mathcal{O}(1)$.

An interesting aspect of stop-circuits is that one can compute non-monotone boolean functions using monotone operations! For example, the following circuit over $\{0, 1\}$ computes the negation $\neg x$ of a variable x :

$$z = 0; g_1 = x \text{ (stop)}; z = 1$$

and the following circuit over $\{\wedge, \vee, 0, 1\}$ computes the parity function $x \oplus y$:

$$z = 0; g_1 = x \wedge y \text{ (stop)}; z = 1; g_2 = x \vee y \text{ (stop)}; z = 0.$$

Let $t_m(f)$ denote the minimum average time of a circuit over $\{\wedge, \vee, 0, 1\}$ computing f . Chashkin (2004) showed that there exist boolean functions f of n variables such that $t(f) = \mathcal{O}(1)$ but $t_m(f) = \Omega(\sqrt{2^n/n})$.

Arithmetic circuits Such circuits constitute the most natural and standard model for computing polynomials over a ring R . In this model the inputs are variables x_1, \dots, x_n , and the computation is performed using the arithmetic operations $+$, \times and may involve constants from R . The output of an arithmetic circuit is thus a polynomial (or a set of polynomials) in the input variables. Arithmetic circuits are a highly structured model of computation compared to boolean circuits. For example, when studying arithmetic circuits we are interested in syntactic computation of polynomials, whereas in the study of boolean circuits we are interested in the semantics of the computation. In other words, in the boolean case we are not interested in any specific polynomial representation of the function, but rather we just want to compute some representation of it, while in the arithmetic world we focus on a specific representation of the function. As such, one may hope that the P vs. NP question will be easier to solve in the arithmetical model. However, in spite of many efforts, we are still far from understanding this fundamental problem. In this book we will not discuss arithmetic circuits: a comprehensive treatment can be found in a recent survey by Shpilka and Yehudayoff (2010).

1.3 Branching Programs

Circuits and formulas are “parallel” models: given an input vector $x \in \{0, 1\}^n$, we process some pieces of x in parallel and join the results by AND or OR gates. The oldest “sequential” model for computing boolean functions, introduced already in pioneering work of Shannon (1949) and extensively studied in the Russian literature since about 1950, is that of switching networks; a modern name for these networks is “branching programs.”

Nondeterministic branching programs The most general of “sequential” models is that of *nondeterministic branching programs* (n.b.p.). Such a program is a directed acyclic graph with two specified nodes² s (source) and t (target). Each wire is either unlabeled or is labeled by a literal (a variable x_i or its negation $\neg x_i$). A labeled wire is called a *contact*, and an unlabeled wire is a *rectifier*.

The graph may be a multigraph, that is, several wires may have the same endpoints. The *size* of a program is defined as the number of contacts (labeled wires).

Each input $a = (a_1, \dots, a_n) \in \{0, 1\}^n$ switches the labeled wires On or Off by the following rule: the wire labeled by x_i is switched On if $a_i = 1$ and is switched Off if $a_i = 0$; the wire labeled by $\neg x_i$ is switched On if $a_i = 0$ and is switched Off if $a_i = 1$. The rectifiers are always considered On.

A nondeterministic branching program computes a boolean function in a natural way: it accepts the input a if and only if there exists a path from s to t which is

²We prefer to use the word “node” instead of “vertex” as well as “wire” instead of “edge” while talking about branching programs.

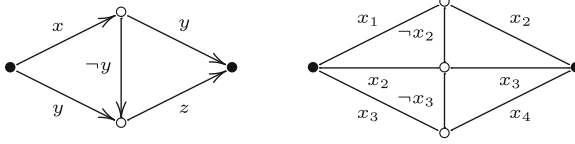


Fig. 1.5 A nondeterministic branching program computing the majority function $\text{Maj}_3(x, y, z) = 1$ iff $x + y + z \geq 2$, and a non-monotone switching network computing the threshold function $\text{Th}_2^4(x_1, x_2, x_3, x_4) = 1$ iff $x_1 + x_2 + x_3 + x_4 \geq 2$

consistent with a , that is, along which all wires are switched On by a . That is, each input switches the wires on or off, and we accept that input if and only if after that there is a nonzero conductivity between the nodes s and t (see Fig. 1.5). Note that we can have many paths consistent with one input vector a ; this is why a program is nondeterministic.

An n.b.p. is *monotone* if it does not have negated contacts, that is, wires labeled by negated variables. It is clear that every such program can only compute a monotone boolean function. For a monotone boolean function f , let $\text{NBP}_+(f)$ denote the minimum size of a monotone n.b.p. computing f , and let $\text{NBP}(f)$ be the non-monotone counterpart of this measure. Let also $l(f)$ denote the minimum length of its minterm, and $w(f)$ the minimum length of its maxterm.

Theorem 1.8. (Markov 1962) *For every monotone boolean function f ,*

$$\text{NBP}_+(f) \geq l(f) \cdot w(f).$$

Proof. Given a monotone n.b.p. program, for each node u define $d(u)$ as the minimum number of variables that need to be set to 1 to establish a directed path from the source node s to u . In particular, $d(t) = l(f)$ for the target node t .

For $0 \leq i \leq l(f)$, let S_i be the set of nodes u such that $d(u) = i$. If u is connected to v by an unlabeled wire (i.e., not a contact) then $d(u) \geq d(v)$, hence there are no unlabeled wires from S_i to S_j for $i < j$. Thus for each $0 \leq i < l(f)$, the set E_i of contacts out of S_i forms a cut of the branching program. That is, setting these contacts to 0 disconnects the graph, and hence, forces the program output value 0 regardless on the values of the remaining variables. This implies that the set $X(E_i)$ of labels of contacts in E_i must contain a maxterm of f , hence $|X(E_i)| \geq w(f)$ distinct variables. \square

For the threshold function Th_k^n we have $l(\text{Th}_k^n) = k$ and $w(\text{Th}_k^n) = n - k + 1$, so every monotone n.b.p. has at least $k(n - k + 1)$ contacts. Actually, this bound is tight, as shown in Fig. 1.6. Thus we have the following surprisingly tight result.

Corollary 1.9. (Markov 1962) $\text{NBP}_+(\text{Th}_k^n) = k(n - k + 1)$.

In particular, $\text{NBP}_+(\text{Maj}_n) = \Theta(n^2)$.

It is also worth noting that the famous result of Szelepcsényi (1987) and Immerman (1988) translates to the following very interesting simulation: there

Fig. 1.6 The naive monotone n.b.p. for Th_k^n has $k(n - k + 1)$ contacts; here $n = 9, k = 6$

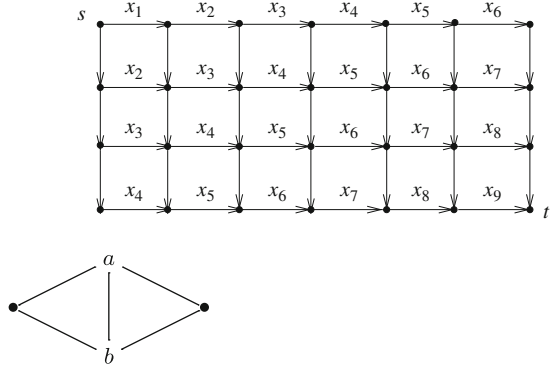


Fig. 1.7 A graph which is *not* parallel-serial: it has a “bridge” $\{a, b\}$ which is traversed in different directions

exists a constant c such that for every sequence (f_n) of boolean functions,

$$\text{NBP}(\neg f_n) \leq \text{NBP}(f_n)^c.$$

This is a “NP = co-NP” type result for branching programs.

A *parity branching program* is a nondeterministic branching program with the “counting” mode of acceptance: an input vector a is accepted iff the number s - t paths consistent with a is odd.

Switching networks A *switching network* (also called a *contact scheme*) is defined in the same way as an n.b.p. with the only difference that now the underlying graph is *undirected*. Note that in this case unlabeled wires (rectifiers) are redundant since we can always contract them.

A switching network is a *parallel-serial network* (or π -*scheme*) if its underlying graph consists of parallel-serial components (see Fig. 1.8). Such networks can be equivalently defined as switching networks satisfying the following condition: it is possible to direct the wires in such a way that every s - t path will turn to a directed path from s to t ; see Fig. 1.7 for an example of a switching network which is *not* parallel-serial.

It is important to note that switching networks include DeMorgan formulas as a special case!

Proposition 1.10. *Every DeMorgan formula can be simulated by a π -scheme of the same size, and vice versa.*

Proof. This can be shown by induction on the leafsize of a DeMorgan formula F . If F is a variable x_i or its negation $\neg x_i$, then F is equivalent to a π -scheme consisting of just one contact. If $F = F_1 \wedge F_2$ then, having π -schemes S_1 and S_2 for subformulas F_1 and F_2 , we can obtain a π -scheme for F by just identifying the target node of S_1 with the source node of S_2 (see Fig. 1.8). If $F = F_1 \vee F_2$ then,

Fig. 1.8 A π -scheme corresponding to the formula $x_1(x_2 \vee x_3)(x_3 \vee \bar{x}_4x_5(x_1 \vee \bar{x}_2))$

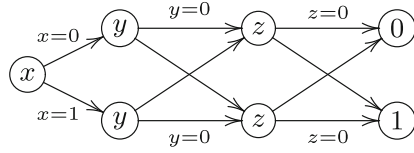
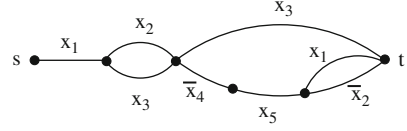
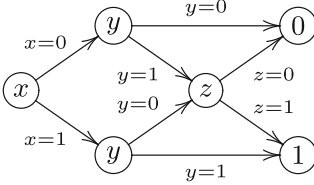


Fig. 1.9 A deterministic branching program computing the majority function $\text{Maj}_3(x, y, z) = 1$ iff $x + y + z \geq 2$, and such a program computing the parity function $\text{Parity}(x, y, z) = x + y + z \bmod 2$; wires left without a label in the latter program make tests $y = 1$ and $z = 1$, respectively

having π -schemes S_1 and S_2 for subformulas F_1 and F_2 , we can obtain a π -scheme for F by placing these two schemes in parallel and gluing their source nodes and their target nodes. \square

That the presence of unlabeled directed wires in a network makes a difference, can be seen on the example of the threshold function Th_2^n . Let $S(f)$ denote the minimum number of contacts in a switching network computing f , and let $S_+(f)$ denote the monotone counterpart of this measure. By Markov's theorem, $\text{NBP}_+(\text{Th}_2^n) = 2n - 3$, but it can be shown that $S_+(\text{Th}_2^n) = \Omega(n \log_2 n)$ (see Exercise 1.12). In fact, if n is a power of 2, then we also have $S_+(\text{Th}_2^n) \leq n \log_2 n$, even in the class of π -schemes (see Exercise 1.11). It can also be easily shown that in the class of *non-monotone* switching networks we have that $S(\text{Th}_2^n) \leq 3n - 4$ (see Fig. 1.5 for a hint).

Deterministic branching programs In a nondeterministic branching program as well as in a switching network one input vector $a \in \{0, 1\}^n$ can be consistent with *many* s - t paths. The deterministic version forbids this: every input vector must be consistent with exactly one path.

Formally, a *deterministic branching program* for a given boolean function f of n variables x_1, \dots, x_n is a directed acyclic graph with one source node and two sinks, that is, nodes of out-degree 0. The sinks are labeled by 1 (accept) and by 0 (reject). Each non-sink node has out-degree 2, and the two outgoing wires are labeled by the tests $x_i = 0$ and $x_i = 1$ for some $i \in \{1, \dots, n\}$; the node itself is labeled by the variable x_i (Fig. 1.9).

Such a program computes a boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ in a natural way: given an input vector $a \in \{0, 1\}^n$, we start in the source node and follow the unique path whose tests are consistent with the corresponding bits of a ; this path is the *computation* on a . In this way we reach a sink, and the input a is accepted iff this is the 1-sink.

Thus, a deterministic branching program is a nondeterministic branching program with the restriction that each non-sink node has fanout 2, and the two outgoing wires from each such node are labeled by the tests $x_i = 0$ and $x_i = 1$ on the *same* variable x_i . The presence of the 0-sink is just to ensure that each input vector can reach a sink.

A *decision tree* is a deterministic branching program whose underlying graph is a binary tree. The *depth* of such a tree is the maximum number of wires in a path from the source node to a leaf.

In the literature, branching programs are also called *binary decision diagrams* or shortly BDDs. This term is especially often used in circuit design theory as well as in other fields where branching programs are used to represent boolean functions. Be warned, however, that the term “BDD” in such papers is often used to denote a much weaker model, namely that of oblivious read-once branching programs (OBDD). These are deterministic branching programs of a very restricted structure: along every computation path all variables are tested in the same order, and no variable is tested more than once.

It is clear that $\text{NBP}(f) \leq \text{S}(f) \leq \text{BP}(f)$, where $\text{BP}(f)$ denotes the minimum size of a *deterministic* branching program computing f . An important result of Reingold (2008) translates to

$$\text{BP}(f_n) \leq \text{S}(f_n)^{\mathcal{O}(1)}.$$

This is a “P = NP” type result for branching programs.

1.4 Almost All Functions are Complex

We still cannot prove super-linear lower bounds for circuits with AND, OR and NOT gates. This is in sharp contrast with the fact, proved more than 60 years ago by Riordan and Shannon (1942) that most boolean functions require formulas of leafsize about $2^n / \log n$. Then Shannon (1949) showed a lower bound $2^n / n$ for circuits. Their arguments were the first applications of counting arguments in boolean function complexity: count how many *different* boolean functions of n variables can be computed using a given number of elementary operations, and compare this number with the total number 2^{2^n} of all boolean functions. After these works of Riordan and Shannon there were many results concerning the behavior of the so-called “Shannon function” in different circuit models.

Definition 1.11. (Shannon function) Given a circuit model with a particular their size-measure, the *Shannon function* for this model is $\mu(n) = \max \mu(f)$, where the maximum is taken over all boolean functions f of n variables, and $\mu(f)$ is the minimum size of a circuit computing f .

In other words, $\mu(n)$ is the smallest number t such that *every* boolean function of n variables can be computed by a circuit of size at most t .

Most bounds in circuit complexity ignore constant multiplicative factors. Moreover, boolean functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$ are parameterized by their number of variables n . Hence, under a boolean function f we actually understand an infinite sequence $\{f_n : n = 1, 2, \dots\}$ of boolean functions. So the claim “ f requires $\Omega(\varphi(n))$ gates” means that there exists a constant $\epsilon > 0$ such that, for infinitely many values of n , the function f_n cannot be computed using fewer than $\epsilon \cdot \varphi(n)$ gates. We will also say that f requires a “super-polynomial” number of gates, if $\varphi(n) \geq n^\alpha$ for some $\alpha \rightarrow \infty$ as $n \rightarrow \infty$, and that f requires an “exponential” number of gates, if $\varphi(n) \geq 2^{n^\epsilon}$ for a constant $\epsilon > 0$.

Through this section, by a circuit (formula) we will understand a circuit (formula) over the basis $\{\wedge, \vee, \neg\}$; similar results, however, also hold when all 16 boolean functions of two variables are allowed as gates. By B_n we will denote the set of all 2^{2^n} boolean functions of n variables x_1, \dots, x_n .

1.4.1 Circuits

Let $C(f)$ denote the minimum size of a fanin-two circuit over $\{\wedge, \vee, \neg\}$ computing f . Let also

$$\phi(n, t) := |\{f \in B_n : C(f) \leq t\}|$$

denote the number of distinct boolean functions $f \in B_n$ computable by circuits of size at most t . As before, we assume that the function computed by a circuit g_1, g_2, \dots, g_t is the function computed at its last gate g_t . So we now assume that every circuit computes only one boolean function. This implies that every class $F \subseteq B_n$ of $|F| > \phi(n, t)$ functions must contain a function requiring circuits of size $> t$. This was the main idea of Riordan–Shannon’s argument.

Lemma 1.12. $\phi(n, t) \leq t^t e^{2t+4n}$. In particular, $\phi(n, t) \leq 2^{t^2}$ for $t \geq n \geq 16$.

Proof. Clearly, we may suppose $n, t \geq 2$. Let g_1, \dots, g_t be names of the gates in a circuit. To describe a concrete circuit, it is sufficient to attach to each gate one of the connectives \wedge, \vee, \neg and an unordered pair of names of two other gates or literals. There are at most

$$\left(3^{\binom{t-1+2n}{2}}\right)^t \leq 2^t (t + 2n)^{2t}$$

such descriptions. Clearly, some of these descriptions do not represent a circuit satisfying all requirements, but every correct circuit may be described in this way. Note that the output does not have a special name. In a correct circuit, it is determined by the fact that it is the only gate not used in any other gate. It is easy to see that every function representable by a circuit of size at most t is also representable by a circuit of size exactly t satisfying the additional requirement that no two of its gates compute the same function. It is also easy to see that in a circuit satisfying the last mentioned property, each of the $t!$ permutations of the names of the gates leads to a different description of a circuit computing the same function. So using estimates $t! \geq (t/3)^t$ and $1 + x \leq e^x$, we can upper bound $\phi(n, t)$ by

$$\frac{2^t(t+2n)^{2t}}{t!} \leq \frac{2^t 3^t(t+2n)^{2t}}{t^t} = 6^t t^t \left(1 + \frac{2n}{t}\right)^{2t} \leq t^t 6^t e^{4n}. \quad \square$$

Lemma 1.13. (Kannan 1981) *For every integer $k \geq 1$, there is a boolean function of n variables such that f can be written as a DNF with n^{2k+1} monomials, but $C(f) > n^k$.*

Proof. We view a circuit computing a boolean function f as accepting the set of vectors $f^{-1}(1) \subseteq \{0, 1\}^n$, and rejecting the remaining vectors. Fix a subset $T \subseteq \{0, 1\}^n$ of size $|T| = nt^2 = n^{2k+1}$. By Lemma 1.12, we know that at most $2^{n^{2k}} < 2^{|T|}$ distinct subsets of T can be accepted by circuits of size at most n^k . Thus, some subset $S \subseteq T$ cannot be accepted by a circuit of size n^k . But this subset S can be accepted by a trivial DNF with $|S| \leq |T| = n^{2k+1}$ monomials: just take one monomial for each vector in S . \square

Since we have 2^{2^n} distinct boolean functions of n variables, setting $t := 2^n/n$ in Lemma 1.12 immediately implies the following lower bound on the Shannon function $C(n)$ in the class of circuits.

Theorem 1.14. *For every sufficiently large n , $C(n) > 2^n/n$.*

On the other hand, it is easy to see that $C(n) = \mathcal{O}(n2^n)$: just take the DNFs. Muller (1956) proved that $C(n) = \Theta(2^n/n)$ for any finite complete basis. Lupanov (1958a) used an ingenious construction to prove an asymptotically tight bound.

Theorem 1.15. (Lupanov 1958a) *For every boolean function f of n variables,*

$$C(f) \leq (1 + \alpha_n) \frac{2^n}{n} \text{ where } \alpha_n = \mathcal{O}\left(\frac{\log n}{n}\right). \quad (1.4)$$

Proof. We assume that the number n of variables is large enough. For a boolean vector $a = (a_1, \dots, a_n)$, let $\text{bin}(a) := \sum_{i=1}^n a_i \cdot 2^{n-i}$ be the unique natural number between 0 and $2^n - 1$ associated with a ; we call $\text{bin}(a)$ the *code* of a .

Let $H_{n,m}(i)$ denote the set of all boolean functions $h(x)$ of n variables such that $h(a) = 0$ if $\text{bin}(a) \leq m(i-1)$ or $\text{bin}(a) > mi$. That is, we arrange the vectors of $\{0, 1\}^n$ into a string of length 2^n according to their codes, split this string into consecutive intervals of length m , and let $H_{n,m}(i)$ to contain all boolean functions h that take value 0 outside the i -th interval:

$$\dots, 0, 0, \underbrace{*, \dots, *}_{\text{values on } m \text{ vectors}}, 0, 0, \dots$$

Thus,³ for each $i = 1, \dots, 2^n/m$, each function in $H_{n,m}(i)$ can only accept a subset of a fixed set of m vectors, implying that

³An apology to purists: for simplicity of presentation, we will often ignore ceilings and floors.

$$|H_{n,m}(i)| \leq 2^{m+1}$$

for all i . Since every input vector a has its unique weight, every boolean function $f(x)$ of n variables can be represented as a disjunction

$$f(x) = \bigvee_{i=1}^{2^n/m} f_i(x), \quad (1.5)$$

where $f_i \in H_{n,m}(i)$ is the functions such that $f_i(a) = f(a)$ for every a such that $m(i-1) < \text{bin}(a) \leq mi$. We can associate with every $a \in \{0, 1\}^n$ the *elementary conjunction*

$$K_a = x_1^{a_1} x_2^{a_2} \cdots x_n^{a_n}.$$

Recall that $x_i^\sigma = 1$ if $a_i = \sigma$, and $x_i^\sigma = 0$ otherwise. Hence, $K_a(b) = 1$ if and only if $b = a$, and we have 2^n such elementary conjunctions of n variables.

Claim 1.16. All elementary conjunctions of n variables can be simultaneously computed by a circuit with at most $2^n + 2n2^{n/2}$ gates.

Proof. Assume for simplicity that n is even. We first compute all $2^{n/2}$ elementary conjunctions of the first $n/2$ variables using a trivial circuit with at most $(n/2)2^{n/2}$ gates, and do the same for the conjunctions of the remaining $n/2$ variables. We now can compute every elementary conjunction of n variables by taking an AND of the corresponding outputs of these two circuits. This requires $2^{n/2} \cdot 2^{n/2} = 2^n$ additional gates, and the entire circuit has size at most $2^n + n2^{n/2}$. To include the case when n is odd, we just multiply the last term by 2. \square

We now turn to the actual construction of an efficient circuit for a given boolean function $f(x)$ of n variables. Let $1 \leq k, m \leq n$ be integer parameters (to be specified latter). By (1.5), we can write $f(x)$ as a disjunction

$$f(x) = \bigvee_a K_a(x_1, \dots, x_k) \wedge \bigvee_{i=1}^{2^n/m} f_{a,i}(x_{k+1}, \dots, x_n),$$

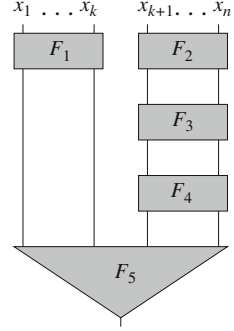
where a ranges over $\{0, 1\}^k$, and each $f_{a,i}$ belongs to $H_{n-k,m}(i)$. We will use this representation to design the desired circuit for f . The circuit consists of five subcircuits (see Fig. 1.10). The first subcircuit F_1 computes all elementary conjunctions of the first k variables. By Claim 1.16, this circuit has size

$$L(F_1) \leq 2^k + 2k2^{k/2}.$$

The second subcircuit F_2 also computes all elementary conjunctions of the remaining $n - k$ variables. By Claim 1.16, this circuit has size

$$L(F_2) \leq 2^{n-k} + 2(n-k)2^{(n-k)/2}.$$

Fig. 1.10 The structure of Lupanov's circuit



The third subcircuit F_3 computes all functions $f_{a,i}$ from the sets $H_{n-k,m}(i)$ using elementary conjunctions computed by F_2 . Since every function in $H_{n-k,m}(i)$ is an OR of at most m elementary conjunctions, each of length $n - k$, and since we have at most $2^{m+1} \cdot 2^{n-k}/m$ such functions, the subcircuit F_3 has size

$$L(F_3) \leq m2^{n-k+m+1}/m = 2^{n-k+m+1}.$$

The fourth subcircuit F_4 computes all functions

$$f_a(x_{k+1}, \dots, x_n) = \bigvee_{i=1}^{2^n/m} f_{a,i}(x_{k+1}, \dots, x_n)$$

using the functions $f_{a,i}$ computed by F_3 . Since we have at most 2^k such functions f_a , each of which is an OR of at most $2^{n-k}/m$ of the functions $f_{a,i}$, the subcircuit F_4 has size

$$L(F_4) \leq 2^k \cdot 2^{n-k}/m \leq \frac{2^n}{m} + 2^k.$$

The last subcircuit F_5 multiplies functions computed by F_3 by elementary conjunctions computed by F_1 , and computes the disjunction of these products. This subcircuit has size

$$L(F_5) \leq 2 \cdot 2^k.$$

Thus, the entire circuit F computes $f(x)$ and has size

$$L(F) \leq \frac{2^n}{m} + 4 \cdot 2^k + 2^{n-k} + 2n2^{k/2} + 2n2^{n-k} + 2^{n-k+m+1}.$$

Now set $k = n - 2 \log n$ and $m = n - 4 \log n$. Then all but the first terms are at most $\mathcal{O}(2^n/n^2)$, and we obtain that $L(F) \leq 2^n/m + \mathcal{O}(2^n/n^2)$. After simple computations, this implies $L(F) \leq (1 + \alpha)2^n/n$ where $\alpha \leq c(\log n)/n$ for a constant c . \square

Lozhkin (1996) improved (1.4) to

$$\alpha_n = \frac{\log n + \log \log n + \mathcal{O}(1)}{n}.$$

Lupanov (1963) also proved a lower bound

$$C(n) \geq (1 + \beta_n) \frac{2^n}{n} \quad \text{where} \quad \beta_n = (1 - o(1)) \frac{\log n}{n}. \quad (1.6)$$

The proof actually gives that the $o(1)$ factor is equal to $\mathcal{O}(1/\log n)$.

Redkin (2004) considered the behavior of the Shannon function when restricted to boolean functions accepting a small number of input vectors. Let $C(n, K)$ denote the smallest number t such that every boolean function f of n variables such that $|f^{-1}(1)| = K$ can be computed by a circuit over $\{\wedge, \vee, \neg\}$ of size at most t . Redkin (2004) proved that, if $2 \leq K \leq \log_2 n - c \log_2 \log_2 n$ holds for some constant $c > 1$, then

$$C(n, K) \sim 2n.$$

For the Shannon function $M(n)$ restricted to the class of all *monotone* boolean functions of n variables, Ugol'nikov (1976) and Pippenger (1976b) independently proved that

$$M(n) \sim \frac{1}{n} \binom{n}{\lfloor n/2 \rfloor}.$$

This holds for circuits with AND, OR and NOT gates. An important improvement by Andreev (1988b) shows that the upper bound is actually achieved by *monotone* circuits with only AND and OR gates!

1.4.2 Approximation Complexity

In a standard setting, a circuit $F(x)$ must compute a given boolean function $f(x)$ correctly on *all* input vectors $x \in \{0, 1\}^n$. We can relax this and only require that F computes f correctly on some given subset $D \subseteq \{0, 1\}^n$ of vectors; on other input vectors the circuit may output arbitrary values, 0 or 1. That is, we are asking for the smallest size $C(f)$ of a circuit computing a *partial* boolean function $f : \{0, 1\}^n \rightarrow \{0, 1, *\}$ defined on

$$D = f^{-1}(0) \cup f^{-1}(1).$$

Let $N = |D|$ be the size of the domain, and $N_1 = |f^{-1}(1)|$. It is clear that $C(f) = \mathcal{O}(nN)$. Actually, we have a much better upper bound:

$$C(f) \leq (1 + o(1)) \frac{N}{\log_2 N} + \mathcal{O}(n). \quad (1.7)$$

For functions with $\log_2 N \sim n$ this was (implicitly) proved already by Nechiporuk (1963, 1965, 1969a) in a series of papers devoted to rectifier networks; Pippenger (1976) gave an independent proof. Then Sholomov (1969) proved this for all $N \geq n \log^{1+\Omega(1)} n$, and finally Andreev (1988) proved this for arbitrary N . It is also known that

$$C(f) \leq (1 + o(1)) \frac{\log_2 \binom{N}{N_1}}{\log_2 \log_2 \binom{N}{N_1}} + \mathcal{O}(n).$$

For $\log_2 N_1 \sim n$ this was (implicitly) proved by Nechiporuk in the above mentioned papers, and by Pippenger (1976). Andreev et al. (1996) proved this in the case when $(1 + \epsilon) \log n < \log N_1 = \mathcal{O}(\log n)$ and $\log N = \Omega(n)$. Finally, Chashkin (2006) proved this for arbitrary N_1 .

Counting arguments (similar to those above) show that these upper bounds are asymptotically tight. The proofs of the upper bounds are, however, non-trivial: it took more than 40 years to find them!

Let us call a partial boolean function $f : D \rightarrow \{0, 1\}$ of n variables *dense* if the size $N = |D|$ of its domain satisfies $\log_2 N \sim n$. The proof of (1.7) for dense functions uses arguments similar to that we used in the proof of Theorem 1.15. Moreover, for dense functions, (1.7) holds without the additive factor $\mathcal{O}(n)$. The proof of (1.7) for functions that are not necessarily dense used interesting ideas which we will sketch right now. We will follow a simplified argument due to Chashkin (2006).

Let $f(x)$ be a partial boolean function which is not dense, that is, for which $\log_2 N \ll n$ holds. If f takes value 1 on fewer than N/n^2 input vectors, then we can compute f by a DNF using at most $n(N/n^2) = N/n$ gates. Thus, the difficult case is when f is not dense but is “dense enough”. The idea in this case is to express f as $f(x) = h(x) \oplus g(L(x))$, where h accepts only few vectors, $g : \{0, 1\}^m \rightarrow \{0, 1\}$ is a dense partial function, and $L : \{0, 1\}^n \rightarrow \{0, 1\}^m$ is an “almost” injective linear operator. Being *linear* means that $L(x) = Ax$ over $\text{GF}(2)$ for some boolean $m \times n$ matrix A . Both h and L have small circuits, and for g we can use the upper bound for dense functions.

Say that an operator $L : \{0, 1\}^n \rightarrow \{0, 1\}^m$ is *almost injective* on a subset $D \subseteq \{0, 1\}^n$ if $L(x) = L(y)$ for at most $2^{-m} \binom{|D|}{2}$ pairs $x \neq y$ of distinct vectors in D .

Lemma 1.17. *Let $D \subseteq \{0, 1\}^n$ be a set of vectors, and m a positive integer. Then there exists a linear operator $L : \{0, 1\}^n \rightarrow \{0, 1\}^m$ which is almost injective on D .*

Proof. We will use a simple (but useful) fact about random vectors in $\text{GF}(2)^n$. A random vector a in $\text{GF}(2)^n$ is obtained by flipping n times a fair 0–1 coin. Hence, $\text{Prob}[a = x] = 2^{-n}$ for each vector $x \in \text{GF}(2)^n$. It is easy to show (see Appendix A)

that $\text{Prob}[\langle a, x \rangle = \langle a, y \rangle] = 1/2$ holds for every two vectors $x \neq y$ in $\text{GF}(2)^n$, where $\langle a, x \rangle = \sum_{i=1}^n a_i x_i \bmod 2$ is the scalar product of a and x over $\text{GF}(2)$.

Now consider a random operator $L(x) = Ax$ where A is a random $m \times n$ matrix whose rows are random vectors in $\text{GF}(2)^n$. By the previous fact, every pair (x, y) of vectors $x \neq y$ in D is not separated by L with probability 2^{-m} . By the linearity of expectation, at most a fraction 2^{-m} of such pairs will not be separated by L . \square

Now let f be a partial boolean function of n variables defined on some domain $D \subseteq \{0, 1\}^n$ of size $N = |D|$.

Lemma 1.18. *If $\log N \geq n/3$ then $C(f) \leq (1 + o(1))N/\log N$.*

Proof. Let $D_0 = \{x \in D : f(x) = 0\}$ and $D_1 = \{x \in D : f(x) = 1\}$; hence, $D = D_0 \cup D_1$ is the set on which our function f is defined, and $N = |D|$. Set also $m = \lceil \log N + 3 \log n \rceil$.

Lemma 1.17 gives us a linear operator $L : \{0, 1\}^n \rightarrow \{0, 1\}^m$ which is almost injective on D . Consider a partial boolean function $g : \{0, 1\}^m \rightarrow \{0, 1\}$ defined on $L(D)$ by: $g(z) = 0$ if $z \in L(D_0)$, and $g(z) = 1$ otherwise. If necessary, specify arbitrary values of g on some vectors outside $L(D)$ until the domain of g has exactly N vectors.

And now comes the trick. We can write our function $f(x)$ as

$$f(x) = h(x) \oplus g(L(x)),$$

where

$$h(x) := f(x) \oplus g(L(x))$$

is a partial function defined on D . Thus, we only need to show that all three functions h , g and L can be computed by small circuits.

The operator $L(x)$ is just a set of $m \leq n$ parity functions, and hence, can be computed by a trivial circuit of size $\mathcal{O}(n^2)$, which is $o(N/n)$ because $\log N = \Omega(n)$, by our assumption.

The function h can be computed by a small circuit just because it accepts at most N/n^3 vectors $x \in D$. Indeed, $h(x) = 0$ for all $x \in D_0$ because then $L(x) \in L(D_0)$. Hence, h can accept a vector $x \in D$ only if $x \in D_1$ and $g(L(x)) = 0$, that is, if $x \in D_1$ and $L(x) = L(y)$ for some $y \in D_0$. Since the operator L is almost injective, and since $2^m \geq Nn^3$, there are at most $2^{-m} \binom{N}{2} \leq N/n^3$ pairs $(y, x) \in D_0 \times D_1$ such that $L(x) = L(y)$. Thus, the function h can accept at most N/n^3 vectors. By taking a DNF, this implies that h can be computed by a circuit of size $n(N/n^3) = o(N/n)$.

It remains therefore to compute the function g . Recall that g is a partial function of m variables defined on N vectors. Since $\log N \sim m$, the function g is dense, implying that $C(g) \leq (1 + o(1))N/\log N$. \square

We can now easily prove (1.7) for any partial function f . If $\log N \geq n/3$ then Lemma 1.18 gives the desired upper bound (without any additive term). Now suppose that $\log N \leq n/3$. In this case we take $m := \lceil 2 \log N \rceil$.

Lemma 1.17 gives us a linear operator $L : \{0, 1\}^n \rightarrow \{0, 1\}^m$ which is almost injective on D . But by our choice of m , the operator L is actually *injective* on D , because $2^{-m} \binom{N}{2} \leq 1/2 < 1$. Thus, in this case we do not need any “error correction” function h because now we have that $f(x) = g(L(x))$ for all $x \in D$, where g is defined as above using our new operator L . The function g has m variables and is defined on $|L(D)| = |D| = N$ vectors.

Since $m \leq \lceil 2 \log N \rceil \leq 3 \log N$, we can apply Lemma 1.18 to g and obtain $C(g) \leq (1 + o(1))N / \log N$. Since $C(L) = \mathcal{O}(n \log N)$, we obtain (1.7) with an additive factor $\mathcal{O}(n^2)$. One can reduce this factor to $\mathcal{O}(n)$ by using the existence of good linear codes computable by circuits of linear size; see Chashkin (2006) for details.

1.4.3 The Circuit Hierarchy Theorem

By using estimates of Shannon–Lupanov it is not difficult to show that one can *properly* increase the number of computed functions by “slightly” increasing the size of circuits. For a function $t : \mathbb{N} \rightarrow \mathbb{N}$, let $\text{Circuit}[t]$ denote the set of all sequences f_n , $n = 1, 2, \dots$ of boolean functions of n variables such that $C(f_n) \leq t(n)$.

Theorem 1.19. (Circuit Hierarchy Theorem) *If $n \leq t(n) \leq 2^{n-2}/n$ then*

$$\text{Circuit}[t] \subsetneq \text{Circuit}[4t].$$

Proof. Fix the maximal $m \in \{1, \dots, n\}$ such that $t(n) \leq 2^m/m \leq 2 \cdot t(n)$. This is possible: if m is the largest number with $2^m/m \leq 2 \cdot t(n)$, then $2^{m+1}/(m+1) > 2 \cdot t(n)$, which implies $t(n) \leq 2^m/m$. Consider the set $B_{n,m}$ of all boolean functions of n variables that depend only on m bits of their inputs. By the Shannon–Lupanov lower bound, there exists $f_n \in B_{n,m}$ such that $C(f_n) > 2^m/m \geq t(n)$. On the other hand, Lupanov’s upper bound yields $C(f_n) \leq 2 \cdot 2^m/m \leq 4 \cdot t(n)$. \square

Remark 1.20. Theorem 1.19 implies that $\phi(n, 4t) \geq \phi(n, t) + 1$; recall that $\phi(n, t)$ is the number of boolean functions of n variables computable by circuits of size at most t . Recently, Chow (2011) gave the following tighter lower bound: there exist constants c and $K > 1$ such that for all $t(n) \leq 2^{n-2}/n$ and all sufficiently large n ,

$$\phi(n, t + cn) \geq K \cdot \phi(n, t). \quad (1.8)$$

That is, when allowing an additional cn gates, the number of computable functions is multiplied by at least some constant factor $K > 1$. In particular, if $t(n) \gg n \log n$, then for any fixed d , $\phi(n, t) \geq n^d \cdot \phi(n, t/2)$ for all sufficiently large n . To prove (1.8), Chow sets $N = 2^n$ and lets $A \subseteq \{0, 1\}^N$ to be the set of all truth tables of boolean functions $f \in B_n$ computable circuits of size at most t .

A *truth table* is a 0–1 vector $a = (a_1, \dots, a_N)$, and it describes the unique function $f_a \in B_n$ defined by $f_a(x) = a_{\text{bin}(x)}$ where $\text{bin}(x) = \sum_{i=1}^n x_i 2^{i-1}$ is the number whose binary code is vector $x \in \{0, 1\}^n$. The *boundary* $\delta(A)$ of $A \subset \{0, 1\}^N$ is the set of all vectors $b \notin A$ that differ from at least one $a \in A$ in exactly one position. The discrete isoperimetric inequality (see, for example, Bezrukov (1994)) states that,

$$\sum_{i=0}^k \binom{N}{i} \leq |A| < \sum_{i=0}^{k+1} \binom{N}{i} \text{ implies } |\delta(A)| \geq \binom{N}{k+1}.$$

Using this and some simple properties of binomial coefficients, Chow shows that the boundary $\delta(A)$ of the set A of truth tables contains at least $\epsilon|A|$ vectors, for a constant $\epsilon > 0$. Now, if $b \in \delta(A)$, then there exists a vector $a \in A$ such that f_b differs from f_a on only one input vector x_0 . One can thus take a circuit for f_a , add additional cn gates to test the equality $x = x_0$, and obtain a circuit for f_b . Thus, using additional cn gates we can compute at least $K \cdot |A| = K \cdot \phi(n, t)$ boolean functions, where $K = (1 + \epsilon) > 1$.

Chow (2011) uses this result to show that the so-called “natural proofs barrier” in circuit lower bounds can be broken using properties of boolean functions of lower density; we shortly discuss the phenomenon of natural proofs in the Epilogue.

1.4.4 Switching Networks and Formulas

Let us now consider the Shannon function $S(n)$ in the class of switching networks. The worst-case complexity of switching networks is similar to that of circuits, and can be lower bounded using the following rough upper bound on the number of directed graphs with a given number of wires. Recall that multiple wires joining the same pair of nodes are here allowed.

Lemma 1.21. *There exist at most $(9t)^t$ graphs with t edges.*

Proof. Every set of t edges is incident with at most $2t$ nodes. Using these nodes, at most $r = (2t)^2$ their pairs (potential edges) can be built. Since $x_1 + \dots + x_r = t$ has $\binom{r+t-1}{t}$ integer solutions $x_i \geq 0$, and since $t! \geq (t/3)^t$ (by Stirling’s formula), the number of graphs with t edges is at most

$$\binom{r+t-1}{t} \leq \frac{(r+t-1)^t}{t!} \leq \frac{3^t(r+t-1)^t}{t^t} \leq \frac{3^{2t}t^{2t}}{t^t} = 3^{2t}t^t. \quad \square$$

Theorem 1.22. *For every constant $\epsilon > 0$ and sufficiently large n ,*

$$S(n) \geq (1 - \epsilon) \frac{2^n}{n}.$$

Proof. It is clear that if a boolean function can be computed by a network with at most t contacts then it can also be computed using exactly t contacts. By Lemma 1.21 we know that there are $(9t)^t$ graphs with t edges. Since we only have $2n$ literals, there are at most $(2n)^t$ ways to turn each such graph into a switching network by assigning literals to edges. Since every switching network computes only one boolean function, at most $(18nt)^t$ different boolean functions can be computed by switching networks with at most t contacts. Comparing this number when $t = (1 - \epsilon)2^n/n$ with the total number 2^{2^n} of all boolean functions, yields the result. \square

Shannon (1949) proved that $(1 - \epsilon)2^n/n < S(n) < 2^{n+3}/n$ holds for an arbitrarily small constant $\epsilon > 0$. Lupanov (1958b) obtained much tighter bounds:

$$\left(1 + \frac{2 \log n - \mathcal{O}(1)}{n}\right) \frac{2^n}{n} \leq S(n) \leq \left(1 + \frac{\mathcal{O}(1)}{\sqrt{n}}\right) \frac{2^n}{n}.$$

In the class of formulas over $\{\wedge, \vee, \neg\}$, that is, fanout-1 circuits constituting a subclass of switching networks (see Proposition 1.10), the behavior of Shannon's function is somewhat different: for some boolean functions, their formulas are at least $n/\log n$ times larger than circuits and switching networks.

When counting formulas, we have to count full binary tree, that is, binary trees where every vertex has either two children or no children. It is well known that the number of such trees with $n + 1$ leaves is exactly the n -th Catalan number:

$$C_n := \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}.$$

Let $L(f)$ denote the smallest number of gates in a formula over $\{\wedge, \vee, \neg\}$ computing f , and let $L(n)$ be the corresponding Shannon function.

Theorem 1.23. *For every constant $\epsilon > 0$ and sufficiently large n ,*

$$L(n) \geq (1 - \epsilon) \frac{2^n}{\log_2 n}.$$

Proof. We can assume that all negations are only applied to input gates (leaves). There are at most 4^t binary trees with t leaves, and for each such tree, there are at most $(2n + 2)^t$ possibilities to turn it into a DeMorgan formula: $2n$ input literals and two types of gates, AND and OR. Hence, the number of different formulas of leafsize at most t is at most $4^t(2n + 2)^t \leq (9n)^t$ for $n \geq 8$. Since, we have 2^{2^n} different boolean functions, the desired lower bound on t follows. \square

Using more precise computations, tighter estimates can be proved. Riordan and Shannon (1942) proved that

$$L(n) > (1 - \delta_n) \frac{2^n}{\log n} \text{ where } \delta_n = \mathcal{O}\left(\frac{1}{\log n}\right).$$

On the other hand, Lupanov (1960) showed that

$$L(n) \leq (1 + \gamma_n) \frac{2^n}{\log n} \text{ where } \gamma_n = \frac{2 \log \log n + \mathcal{O}(1)}{\log n}.$$

Lozhkin (1996) improved this to

$$\gamma_n = \mathcal{O}\left(\frac{1}{\log n}\right).$$

Interestingly, Lupanov (1962) showed (among other things) that $L(n)$ drops down from $2^n / \log n$ to

$$L(n) = \mathcal{O}(2^n / n),$$

if we allow just one of the basis functions AND, OR or NOT to have fanout 2. If we allow all three basis functions to have fanout 2, then even the asymptotic

$$L(n) \sim 2^n / n$$

holds. If only NOT gates are allowed to have fanout 2, then

$$L(n) \sim 2^{n+1} / n.$$

Savický and Woods (1998) gave tight estimates on the number of boolean functions computable by formulas of a given size. In particular, they proved that, for every constant k , almost all boolean functions of formula size n^k require circuits of size at least n^k / k .

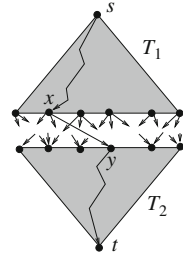
Nechiporuk (1962) considered the behavior of the Shannon function in cases when some of the gates are given for free. He proved that the smallest number of gates that is enough to compute any boolean function of n variables is asymptotically equal to:

- $2^n / n$ for formulas over $\{\vee, \neg\}$ when \vee -gates are for free;
- $\sqrt{2^{n+1}}$ for circuits over $\{\vee, \neg\}$ when \vee -gates are for free;
- $2^n / 2n$ for formulas over $\{\oplus, \wedge\}$ when \oplus -gates are for free;
- $\sqrt{2^n}$ for circuits over $\{\oplus, \wedge\}$ when \oplus -gates are for free,

Concerning the Shannon functions $\oplus\text{BP}(n)$ for parity branching programs and $\text{NBP}(n)$ for nondeterministic branching programs, Nechiporuk (1962) proved that

$$\oplus\text{BP}(n) \sim \sqrt{2^{n+1}}$$

Fig. 1.11 Construction of a nondeterministic branching program for an arbitrary boolean function on n variables. The program is *read-once* (along every s - t path, each variable is tested only once), and is *oblivious* (on each level, tests on the same variable are made)



and

$$\sqrt{2^{n+1}} \leq \text{NBP}(n) \leq 2\sqrt{2^n}. \quad (1.9)$$

The upper bound $\text{NBP}(n) \leq 4\sqrt{2^n}$ for an even n is easy to prove. Take a boolean function $f(x_1, \dots, x_n)$, and assume that $n = 2m$ is even. Let T_1 be a full decision tree on the first m variables, and T_2 a full decision tree on the remaining m variables. Turn T_2 “on its head”, and reverse the orientation of its wires. Draw a switch (unlabeled wire) from the leaf of T_1 reached by a vector $x \in \{0, 1\}^m$ to the leaf of T_2 reached by a vector $y \in \{0, 1\}^m$ if and only if $f(x, y) = 1$ (see Fig. 1.11). We have $|f^{-1}(1)|$ switches, but they are for free. The number of contacts in the trees T_1 and T_2 is smaller than $2 \cdot 2^{m+1} = 4\sqrt{2^n}$. Note that the constructed program is “read-once”: along each s - t path, each variable is tested only once. If the number of variables is odd, $n = 2m + 1$, then the above construction gives a program with at most $2(2^m + 2^{m+1}) = 3 \cdot 2^{m+1} = 3\sqrt{2^{n+1}}$ contacts. To obtain a better upper bound $2\sqrt{2^n}$, one can use more efficient contact schemes constructed by Lupanov (1958b).

The best known asymptotic bounds on the Shannon function restricted to *monotone* boolean functions can be found in a survey by Korshunov (2003).

1.4.5 Invariant Classes

Let B be the class of all boolean functions. A class $Q \subseteq B$ is *invariant* if together with every function $f(x_1, \dots, x_n)$ in Q it contains

- all subfunctions of f , and
- all function $f(x_{\pi(1)}, \dots, x_{\pi(n)})$ where $\pi : [n] \rightarrow [n]$ is a permutation.

For example, classes of all symmetric, all linear or all monotone functions are invariant. The class B itself is a trivial invariant class.

Let $Q(n)$ denote the set of all boolean functions $f \in Q$ of n variables; the functions need not depend on all their variables. Denote

$$\text{Lim}(Q) := \lim_{n \rightarrow \infty} |Q(n)|^{1/2^n}.$$

Theorem 1.24. *For every invariant class Q , $\text{Lim}(Q)$ exists and lies between 1 and 2.*

Proof. Let $f(x_1, \dots, x_{n+1})$ be an arbitrary boolean function in Q depending on $n+1$ variables. Recurrence (1.1) yields $|Q(n+1)| \leq |Q(n)|^2$. Hence, the sequence $|Q(n)|^{1/2^n}$ is non-increasing. If $Q \neq \emptyset$, then

$$1 = 1^{1/2^n} \leq |Q(n)|^{1/2^n} \leq (2^{2^n})^{1/2^n} = 2.$$

Thus $\text{Lim}(Q)$ exists and is a number in the interval $[1, 2]$. \square

By Theorem 1.24, every invariant class Q of boolean functions defines the unique real number $0 \leq \sigma \leq 1$ such that $\text{Lim}(Q) = 2^\sigma$. This number is an important parameter of the invariant class characterizing its cardinality. It also characterizes the maximum circuit complexity of functions in Q . We will therefore denote this parameter by writing Q_σ if σ is the parameter of Q .

For example, if P is the class of all linear boolean functions (parity functions), then $|P(n)| \leq 2^{n+1}$, implying that $\text{Lim}(P) = 1$, and hence, $\sigma = 0$. The same holds for the class S of all symmetric boolean functions. If M is the class of all monotone boolean functions, then

$$\binom{n}{n/2} \leq \log_2 |M(n)| \leq (1 + o(1)) \binom{n}{n/2}.$$

The lower bound here is trivial: consider monotone boolean functions whose minterms have length $n/2$. The upper bound was proved by Kleitman and Markowsky (1975) with the $o(1)$ factor being $\mathcal{O}(\log n/n)$. The number $|M(n)|$ is known as the *Dedekind number*, and was considered by many authors. Korshunov (1977, 1981) proved an asymptotically tight estimate

$$\log_2 |M(n)| \sim (1 + \alpha) \binom{n}{n/2} \quad \text{where } \alpha = \Theta(n^2/2^n).$$

Since $\binom{n}{n/2} = \Theta(2^n/\sqrt{n})$, we again have that $\text{Lim}(M) = 1$, and $\sigma = 0$. On the other hand, $\text{Lim}(B) = (2^{2^n})^{1/2^n} = 2$, and $\sigma = 1$.

Do there exist invariant classes Q with σ strictly between 0 and 1? Yablonskii (1959) showed that, for every real number $0 \leq \sigma \leq 1$ there exists an invariant class Q with $\text{Lim}(Q) = 2^\sigma$.

Example 1.25. As an example let us construct an invariant class with $\sigma = \frac{1}{2}$. For this, let $Q(n)$ consist of all boolean functions of the form $f(x_1, \dots, x_n) = l_S(x) \wedge g(x)$ where $l_S(x)$ is the parity function $\bigoplus_{i \in S} x_i$ or its negation, and g is an arbitrary boolean function depending only on variables x_i with $i \in S$. It is easy to see that Q is an invariant class. If we take $S = \{1, \dots, n\}$, then $l_S(x) = 1$ for 2^{n-1} vectors x . Hence, $|Q(n)| \geq 2^{2^{n-1}}$. On the other hand, for a

fixed $S \subseteq [n]$, there are at most $2^{2^{|S|-1}} \leq 2^{2^{n-1}}$ functions $f \in Q(n)$. Since we have only 2^{n+1} different linear functions on n variables, $|Q(n)| \leq 2^{n+1} 2^{2^{n-1}}$. Thus $\text{Lim}(Q) = \sqrt{2} \cdot \lim_{n \rightarrow \infty} 2^{n/2^n} = \sqrt{2}$.

Let $L_Q(n)$ denote the maximum, over all functions $f \in Q(n)$, of the minimum size of a DeMorgan circuit computing f . Yablonskii (1959) extended results of Shannon and Lupanov to all invariant classes.

Theorem 1.26. (Yablonskii 1959) *Let Q be an invariant class of boolean functions, and let $0 \leq \sigma \leq 1$ be its parameter. Then, for every constant $\epsilon > 0$,*

$$(1 - \epsilon)\sigma \frac{2^n}{n} \leq L_Q(n) \leq (1 + o(1))\sigma \frac{2^n}{n}.$$

The lower bound uses Shannon's counting argument and the fact that $Q(n)$ has about $2^{\sigma 2^n}$ boolean functions. The upper bound uses a construction similar to that used by Lupanov (1958a).

It is not difficult to verify that $\sigma < 1$ for every invariant class $Q \neq B$. Indeed, for some fixed m , there exists a boolean function $g(x_1, \dots, x_m) \notin Q$. Since the sequence $|Q(n)|^{1/2^n}$ is non-increasing, we have that

$$\lim_{n \rightarrow \infty} |Q(n)|^{1/2^n} \leq |Q(m)|^{1/2^m} \leq (2^{2^m} - 1)^{1/2^m} < 2.$$

Now suppose we have an algorithm constructing a sequence $F = (f_n : n = 1, 2, \dots)$ of boolean functions. Call such an algorithm *honest* if, together with the sequence F , it constructs some invariant class of boolean functions containing F . Specifying F as an element of an invariant class means that the sequence F is specified by its *properties*.

Theorem 1.27. (Yablonskii 1959) *Every honest algorithm constructing a sequence of most complex boolean functions must construct all boolean functions.*

Proof. Let us assume the opposite. That is, assume that some sequence $F = (f_n : n = 1, 2, \dots)$ of most complex boolean functions is a member of some invariant class $Q_\sigma \neq B$. Then $\sigma < 1$, and Theorem 1.26 implies that every boolean function $g_n(x_1, \dots, x_n) \in Q$ has a DeMorgan circuit of size at most $(1 - \lambda)2^n/n$ for some constant $\lambda > 0$. But the lower bound (1.6) implies that $C(f_n) > 2^n/n$. Comparing these bounds, we can conclude that the sequence F cannot be contained in any invariant class Q_σ with $\sigma < 1$. \square

This result serves as an indication that there (apparently) is no other way to construct a most-complex sequence of boolean function other than to do a “brute force search” (or “perebor” in Russian): just try all 2^{2^n} boolean functions.

1.5 So Where are the Complex Functions?

Unfortunately, the results above are not quite satisfactory: we know that almost all boolean functions are complex, but no *specific* (or *explicit*) complex function is known. This is a strange situation: we know that almost all boolean functions are complex, but we cannot exhibit any single example of a complex function! We also face a similar situation in other branches of mathematics. For example, in combinatorics it is known that a random graph on n vertices is a Ramsey-graph, that is, has no cliques or independent sets on more than $t = 2 \log n$ vertices. But where are these “mystical” graphs?

The best known explicit construction of non-bipartite t -Ramsey graphs due to Frankl and Wilson only achieves a much larger value t about $\exp(\sqrt{\log n \log \log n})$. In the bipartite case, t -Ramsey graphs with $t = n^{1/2}$ can be obtained from Hadamard matrices: Lindsey’s Lemma (see Appendix A) implies that such a matrix can have a monochromatic $a \times b$ submatrix only if $ab \leq n$. But even going below $t = n^{1/2}$ was only recently obtained by Pudlák and Rödl (2004), Barak et al. (2010), and Ben-Sasson and Zewi (2010). The paper of Barak et al. (2010) constructs bipartite t -Ramsey graphs with $t = n^\delta$ for an arbitrarily small constant $\delta > 0$.

The main goal of boolean complexity theory is to prove lower bounds on the complexity of computing explicitly given boolean functions in interesting computational models. By “explicitly given” researchers usually mean “belonging to the class NP”. This is a plausible interpretation since, on the one hand, this class contains the overwhelming majority of interesting boolean functions, and on the other hand, it is a sufficiently restricted class in which counting arguments seem not to apply. The second point is illustrated by a result of Kannan (1981) showing that already the class $\Sigma_2 \cap \Pi_2$, next after NP in the complexity hierarchy, contains boolean functions whose circuit size is $\Omega(n^k)$ for any fixed $k > 0$. The proof of this fact essentially uses counting arguments; we will present it in the Epilogue (see Theorem 20.13).

1.5.1 On Explicitness

We are not going to introduce the classes of the complexity hierarchy. Instead, we will use the following simple definitions of “explicitness”. Say that a sequence of boolean functions $g_{n,m}(x, y)$ of $n + m$ variables is “simple” if there exists a Turing machine (or any other algorithm) which, given n, m and a vector (x, y) , outputs the value $g_{n,m}(x, y)$ in time polynomial in $n + m$. Then we can treat a sequence of boolean functions $f_n(x)$ as “explicit” if there exists a sequence $g_{n,m}$ of simple functions with $m = n^{O(1)}$ such that

$$f_n(x) = 1 \text{ if and only if } g_{n,m}(x, y) = 1 \text{ for at least one } y \in \{0, 1\}^m.$$

In this case, simple functions correspond to the class P, and explicit functions form the class NP. For example, the parity function $x_1 \oplus \dots \oplus x_n$ is “very explicit”: to

determine its value, it is enough just to sum up all bits and divide the result by 2. A classical example of an explicit function (a function in NP) which is not known to be in P is the clique function. It has $n = \binom{v}{2}$ variables $x_{u,v}$, each for one possible edge $\{u, v\}$ on a given set V of n vertices. Each 0–1 vector x of length $\binom{v}{2}$ defines a graph $G_x = (V, E_x)$ in a natural way: $\{u, v\} \in E_x$ iff $x_{u,v} = 1$. The function itself is defined by:

$$\text{CLIQUE}(x) = 1 \text{ iff the graph } G_x \text{ contains a clique on } \sqrt{n} \text{ vertices.}$$

In this case, $m = n$ and the graphs G_y encoded by vectors y are k -cliques for $k = \sqrt{n}$. Since one can test whether a given k -clique is present in G_x in time about $\binom{k}{2} \leq n$, the function is explicit (belongs to NP). Thus a proof that CLIQUE requires circuits of super-polynomial size would immediately imply that $P \neq NP$.

Unfortunately, at the moment we are even not able to prove that CLIQUE requires, say, $10n$ AND, OR and NOT gates! The problem here is with NOT gates—we can already prove that the clique function requires $n^{\Omega(\sqrt{n})}$ gates, if no NOT gates are allowed; this is a celebrated result of Razborov (1985a) which we will present in Chap. 9.

1.5.2 Explicit Lower Bounds

The strongest known lower bounds for non-monotone circuits (with NOT gates) computing explicit boolean functions of n variables have the form:

- $4n - 4$ for circuits over $\{\wedge, \vee, \neg\}$, and $7n - 7$ for circuits over $\{\wedge, \neg\}$ and $\{\vee, \neg\}$ computing $\oplus_n(x) = x_1 \oplus x_2 \oplus \dots \oplus x_n$; Redkin (1973). These bounds are tight.
- $5n - o(n)$ for circuits over the basis with all fanin-2 gates, except the parity and its negation; Iwama and Morizumi (2002).
- $3n - o(n)$ for general circuits over the basis with all fanin-2 gates; Blum and Micali (1984).
- $n^{3-o(1)}$ for formulas over $\{\wedge, \vee, \neg\}$; Håstad (1998).
- $\Omega(n^2/\log n)$ for general fanin-2 formulas, $\Omega(n^2/\log^2 n)$ for deterministic branching programs, and $\Omega(n^{3/2}/\log n)$ for nondeterministic branching programs; Nechiporuk (1966).

We have only listed the strongest bounds for unrestricted circuit models we currently have (some other known bounds are summarized in Tables 1.1–1.4 at the end of this chapter). The bounds for circuits and formulas were obtained by gradually increasing previous lower bounds.

A lower bound $2n$ for general circuits was first proved by Kloss and Malyshev (1965), and by Schnorr (1974). Then Paul (1977) proved a $2.5n$ lower bound, Stockmeyer (1977) gave the same $2.5n$ lower bound for a larger family of boolean functions including *symmetric* functions, Blum and Micali (1984) proved the lower bound $3n - o(n)$. A simpler proof of this lower bound, but for much more complicated functions, was recently found by Demenkov and Kulikov (2011). They prove such a bound for any boolean function which is not constant on any affine

subspace of $\text{GF}(2)^n$ of dimension $o(n)$. A rather involved construction of such functions was given earlier by Ben-Sasson and Kopparty (2009).

For circuits over the basis with all fanin-2 gates, except the parity and its negation, a lower bound of $4n$ was obtained earlier by Zwick (1991b) (for a symmetric boolean function), then Lachish and Raz (2001) proved a $4.5n - o(n)$ lower bound, and finally Iwama and Morizumi (2002) extended this bound to $5n - o(n)$.

For formulas, the first nontrivial lower bound $n^{3/2}$ was proved by Subbotovskaya (1961), then a lower bound $\Omega(n^2)$ was proved by Khrapchenko (1971), and a lower bound of $\Omega(n^{2.5})$ by Andreev (1985). This was enlarged to $\Omega(n^{2.55})$ by Impagliazzo and Nisan (1993), and to $\Omega(n^{2.63})$ by Paterson and Zwick (1993), and finally to $n^{3-o(1)}$ by Håstad (1998).

The boolean functions for which these lower bounds are proved are quite “simple”. For general circuits, a lower bound $3n - o(n)$ is achieved by particular symmetric functions, that is, functions whose value only depends on the number of ones in the input vector.

The lower bound $5n - o(n)$ holds for any k -mixed boolean function with $k = n - o(n)$; a function is k -mixed if for any two different restrictions fixing the same set of k variables must induce different functions on the remaining $n - k$ variables. We will construct an explicit k -mixed boolean function for $k = n - \mathcal{O}(\sqrt{n})$ in Sect. 16.1. Amano and Tarui (2008) showed that some highly mixed boolean functions *can* be computed by circuits of size $5n + o(1)$; hence, the property of being mixed alone is not enough to improve this lower bound.

Almost-quadratic lower bounds for general formulas and branching programs are achieved by the element distinctness function (see Sects. 6.5 and 15.1 for the proofs).

The strongest known lower bounds, up to $n^{3-o(1)}$, for DeMorgan formulas are achieved by the following somewhat artificial function $A_n(x, y)$ (see Sect. 6.4). The function has $n = 2^b + bm$ variables with $b = \log(n/2)$ and $m = n/(2b)$. The last bm variables are divided into b blocks $y = (y_1, \dots, y_b)$ of length m , and the value of A_n is defined by $A_n(x, y) = f_x(\oplus_m(y_1), \dots, \oplus_m(y_b))$.

1.6 A $3n$ Lower Bound for Circuits

Existing lower bounds for general circuits were proved using the so-called “gate-elimination” argument. The proofs themselves consist of a rather involved case analysis, and we will not present them here. Instead of that we will demonstrate the main idea by proving weaker lower bounds.

The *gate-elimination* argument does the following. Given a circuit for the function in question, we first argue that some variable 1 (or set of variables) must fan out to several gates. Setting this variable to a constant will eliminate several gates. By repeatedly applying this process, we conclude that the original circuit must have had many gates.

To illustrate the basic idea, we apply the gate-elimination argument to threshold functions

$$\text{Th}_k^n(x_1, \dots, x_n) = 1 \text{ if and only if } x_1 + x_2 + \dots + x_n \geq k.$$

Theorem 1.28. *Even if all boolean functions in at most two variables are allowed as gates, the function Th_2^n requires at least $2n - 4$ gates.*

Proof. The proof is by induction on n . For $n = 2$ and $n = 3$ the bound is trivial. For the induction step, take an optimal circuit for Th_2^n , and suppose that the bottom-most gate g acts on variables x_i and x_j with $i \neq j$. This gate has the form $g = \varphi(x_i, x_j)$ for some $\varphi : \{0, 1\}^2 \rightarrow \{0, 1\}$. Notice that under the four possible settings of these two variables, the function Th_2^n has *three* different subfunctions Th_0^{n-2} , Th_1^{n-2} and Th_2^{n-2} . It follows that either x_i or x_j fans out to another gate h , for otherwise our circuit would have only *two* inequivalent sub-circuits under the settings of x_i and x_j . Why? Just because the gate $g = \varphi(x_i, x_j)$ can only take *two* values, 0 and 1.

Now suppose that it is x_j that fans out to h . Setting x_j to 0 eliminates the need of both gates g and h . The resulting circuit computes Th_2^{n-1} , and by induction, has at least $2(n-1) - 4$ gates. Adding the two eliminated gates to this bound shows that the original circuit has at least $2n - 4$ gates, as desired. \square

Theorem 1.28 holds for circuits whose gates are any boolean functions in at most two variables. For circuits over the basis $\{\wedge, \vee, \neg\}$ one can prove a slightly stronger lower bound. For this, we consider the parity function

$$\oplus_n(x) = x_1 \oplus x_2 \oplus \dots \oplus x_n.$$

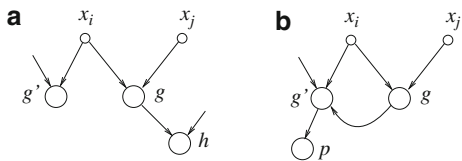
Theorem 1.29. (Schnorr 1974) *The minimal number of AND and OR gates in a circuit over $\{\wedge, \vee, \neg\}$ computing \oplus_n is $3(n-1)$.*

Proof. The upper bound follows since $x \oplus y$ is equal to $(x \wedge \neg y) \vee (\neg x \wedge y)$. For the lower bound we prove the existence of some x_i whose replacement by a suitable constant eliminates 3 gates. This implies the assertion for $n = 1$ directly and for $n \geq 3$ by induction.

Let g be the *first* gate of an optimal circuit for $\oplus_n(x)$. Its inputs are different variables x_i and x_j (see Fig. 1.12). If x_i had fanout 1, that is, if g were the only gate for which x_i is acting as input, then we could replace x_j by a constant so that gate g would be replaced by a constant. This would imply that the output became independent of the i -th variable x_i in contradiction to the definition of parity. Hence, x_i must have fanout at least 2. Let g' be the other gate to which x_i is an input.

We now replace x_i by such a constant that g becomes replaced by a constant. Since under this setting of x_i the parity is not replaced by a constant, the gate g cannot be an output gate. Let h be a successor of g . We only have two possibilities: either h coincides with g' (that is, g has no other successors besides g') or not.

Fig. 1.12 The two cases in the proof of Theorem 1.29



Case (a): $g' = h$. In this case g has fanout 1. We can set x_i to a constant so that g' will become set to a constant. This will eliminate the need for all three gates g , g' and p .

Case (b): $g' \neq h$. Then we can set x_i to a constant so that g will become set to a constant. This will eliminate the need for all three gates g , g' and h .

In either case we eliminate at least 3 gates. \square

Note that the same argument works if we allow as gates any boolean functions $\phi(x, y)$ with the following property: there exist constants $a, b \in \{0, 1\}$ such that both $\phi(a, y)$ and $\phi(x, b)$ are constants. The only two-variable functions that do not have this property is the parity function $x \oplus y$ and its negation $x \oplus y \oplus 1$.

1.7 Graph Complexity

As pointed out by Sipser (1992), one of the impediments in the lower bounds area is a shortage of problems of *intermediate* difficulty which lend insight into the harder problems. Most of known problems (boolean functions) are either “easy” (parity, majority, etc.) or are “very hard” (clique problem, satisfiability of CNFs, and all other NP-hard problems).

On the other hand, there are fields—like graph theory or matrix theory—with a much richer spectrum of known objects. It therefore makes sense to look more carefully at the graph structure of boolean functions: that is, to move from a “bit level” to a more global one and consider a given boolean function as a matrix or as a bipartite graph. The concept of graph complexity, as we will describe it below, was introduced by Pudlák et al. (1988), and was later considered by Razborov (1988, 1990), Chashkin (1994), Lokam (2003), Jukna (2006, 2010), Drucker (2011), and other authors.

A circuit for a given boolean function f generates this function starting from simplest “generators”—variables and their negations. It applies some boolean operations like AND and OR to these generators to produce new “more complicated” functions, then does the same with these functions until f is generated. Note however that there was nothing special to restrict ourselves to boolean functions—one can define, say, the complexity of graphs or matrices analogously.

A basic observation connecting graphs and boolean functions is that boolean functions can be treated as graphs. Namely, every boolean function $f(x_1, \dots, x_m, y_1, \dots, y_m)$ of $2m$ variables can be viewed as a bipartite $n \times n$

graph⁴ $G_f \subseteq V_1 \times V_2$ with $n = 2^m$, whose vertex-sets $V_1 = V_2 = \{0, 1\}^m$ are binary vectors, and $(u, v) \in G_f$ iff $f(u, v) = 1$. In particular, literals x_i^a and y_j^a for $a \in \{0, 1\}$ then turn to *bicliques* (bipartite complete graphs):

1. If $f = x_i^a$ then $G_f = \{u \in V_1: u_i = a\} \times V_2$.
2. If $f = y_j^a$ then $G_f = V_1 \times \{v \in V_2: v_j = a\}$.

Boolean operations AND and OR turn to set-theoretic operations:

$$G_{f \wedge g} = G_f \cap G_g \text{ and } G_{f \vee g} = G_f \cup G_g.$$

Thus, every (non-monotone!) DeMorgan formula (or circuit) for the function f turns to a formula (circuit) which can use any of $4m$ bicliques defined above, and apply the union and intersection operations to produce the entire graph G_f .

We thus can take a “vacation” from boolean functions, and consider the computational complexity of graphs: how many \cup and \cap operations do we need to produce a given bipartite graph G starting from bicliques?

Remark 1.30. In the context of arbitrary bipartite graphs, restriction to these special bicliques (1) and (2) as generators looks somewhat artificial. And indeed, if we use only these $4m = 4 \log n$ generators, then the complexity of *isomorphic* graphs may be exponentially different. In particular, there would exist a perfect matching of formula size $\mathcal{O}(m) = \mathcal{O}(\log n)$, namely that corresponding to the equality function defined by $f(x, y) = 1$ iff $(x = y)$, as well as a perfect matching requiring $\Omega(n)$ formula size; the existence can be shown by comparing the number $m^{\mathcal{O}(t)}$ of formulas of size t with the total number $n!$ of perfect matchings.

1.7.1 Clique Complexity of Graphs

In view of the previous remark, let us allow all 2^{2n} bicliques $P \times V_2$ and $V_1 \times Q$ with $P \subseteq V_1$ and $Q \subseteq V_2$ as generators. The *bipartite formula complexity*, $L_{\text{bip}}(G)$, of a bipartite $n \times n$ graph $G \subseteq V_1 \times V_2$, is then the minimum number of leaves in a formula over $\{\cap, \cup\}$ which produces the graph G starting from these generators.

By what was said above, we have that every boolean function f of $2m = 2 \log n$ variables requires non-monotone DeMorgan formulas with at least $L_{\text{bip}}(G_f)$ leaves. Thus any explicit bipartite $n \times n$ graph G with $L_{\text{bip}}(G) = \Omega(\log^K n)$ would immediately give us an explicit boolean function of $2m$ variables requiring non-monotone formulas of size $\Omega(m^K)$. Recall that the best known lower bound for formulas has the form $\Omega(m^3)$.

Note however that even if we have “only” to prove poly-logarithmic lower bounds for graphs, such bounds may be extremely hard to obtain. For example, we will prove later in Sect. 6.8 that, if f is the parity function of $2m$ variables, then

⁴Here and in what follows we will often consider graphs as *sets* of their edges.

any non-monotone DeMorgan formula computing f must have at least $\Omega(m^2) = \Omega(\log^2 n)$ leaves. But the graph G_f of f is just a union of two bicliques, implying that $L_{\text{bip}}(G) \leq 4$.

Another way to view the concept of bipartite complexity of graphs $G \subseteq V_1 \times V_2$ is to associate with subsets $P \subseteq V_1$ and $Q \subseteq V_2$ boolean variables (we call them *meta-variables*) $z_P, z_Q : V_1 \times V_2 \rightarrow \{0, 1\}$ interpreted as

$$z_P(u, v) = 1 \text{ iff } u \in P, \text{ and } z_Q(u, v) = 1 \text{ iff } v \in Q.$$

Then the set of edges accepted by z_P is exactly the biclique $P \times V_2$, and similarly for variables z_Q .

Remark 1.31. Note that in this case we do not need negated variables: for every $P \subseteq V_1$, the variable $z_{V_1 \setminus P}$ accepts exactly the same set of edges as the negated variable $\neg x_P$. Thus $L_{\text{bip}}(G)$ is exactly the minimum leafsize of a *monotone* DeMorgan formula of these meta-variables which accepts all edges and rejects all nonedges of G . Also, the depth of a decision tree for the graph G_f , querying the meta-variables, is *exactly* the communication complexity of the boolean function $f(x, y)$, a measure which we will introduce in Chap. 3.

1.7.2 Star Complexity of Graphs

Now we consider the complexity of graphs when only special bicliques—stars—are used as generators. A *star* is a bipartite graph formed by one vertex connected to all vertices on the other side of the bipartition. In this case the complexity of a given graph turns into a monotone complexity of monotone boolean functions “representing” this graph in the following sense.

Let $G = (V, E)$ be an n -vertex graph, and let $z = \{z_v : v \in V\}$ be a set of boolean variables, one for each vertex (not for each subset $P \subseteq V$, as before). Say that a boolean function (or a circuit) $g(z)$ *represents* the graph G if, for every input $a \in \{0, 1\}^n$ with exactly two 1s in, say, positions $u \neq v$, $g(a) = 1$ iff u and v are adjacent in G :

$$f(0, \dots, 0, \overset{u}{1}, 0, \dots, 0, \overset{v}{1}, 0, \dots, 0) = 1 \text{ if and only if } \{u, v\} \in E.$$

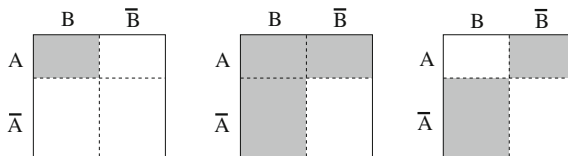


Fig. 1.13 The adjacency matrices of: (a) a complete bipartite graph $A \times B$ represented by $g = (\bigvee_{u \in A} z_u) \wedge (\bigvee_{v \in B} z_v)$, (b) a bipartite graph represented by an OR function $g = \bigvee_{v \in A \cup B} z_v$, and (c) a bipartite graph represented by a Parity function $g = \bigoplus_{v \in A \cup B} z_v$

If the graph is bipartite then we only require that this must hold for vertices u and v from different color classes. Note that in both cases (bipartite or not), on input vectors with fewer than two 1s as well as on vectors with more than two 1s the function g can take arbitrary values!

Another way to treat this concept is to view edges as 2-element sets of vertices, and boolean functions (or circuits) as accepting/rejecting subsets $S \subseteq V$ of vertices. Then a boolean function $f : 2^V \rightarrow \{0, 1\}$ represents a graph if it accepts all edges and rejects all non-edges. On subsets S with $|S| \neq 2$ the function can take arbitrary values.

Thus a single variable z_v represents a complete star around the vertex v , that is, the graph consisting of all edges connecting v with the remaining vertices. If we consider bipartite graphs with bipartition $V_1 \cup V_2$, then each single variable x_v with $v \in V_i$ represents the star consisting of all edges connecting v with vertices in V_{3-i} . If $A \subseteq V_1$ and $B \subseteq V_2$, then the boolean function

$$\left(\bigvee_{u \in A} z_u \right) \wedge \left(\bigvee_{v \in B} z_v \right)$$

represents the complete bipartite graph $A \times B$ (Fig. 1.13). Note also that every graph $G = (V, E)$ is represented by $\bigvee_{uv \in E} z_u \wedge z_v$. But this representation of n -vertex graphs is not quite compact: the number of gates in them may be as large as $\Theta(n^2)$. If we allow unbounded fanin OR gates then already $2n - 1$ gates are enough: we can use the representation

$$\bigvee_{u \in S} z_u \wedge \left(\bigvee_{v: uv \in E} z_v \right),$$

where $S \subseteq V$ is an arbitrary vertex-cover of G , that is, a set of vertices such that every edge of G has is endpoint in S .

We have already seen how non-monotone circuit complexity of boolean functions is related to biclique complexity of graphs. A similar relation is also in the case of star complexity.

As before, we consider a boolean function $f(x, y)$ of $2m$ variables as a bipartite $n \times n$ graph $G_f \subseteq U \times V$ with color classes $U = V = \{0, 1\}^m$ of size $n = 2^m$, in which two vertices (vectors) x and y are adjacent iff $f(x, y) = 1$. In the following lemma, by a “circuit” we mean an arbitrary boolean circuit with literals—variables and their negations—as inputs.

Lemma 1.32. (Magnification Lemma) *In every circuit computing $f(x, y)$ it is possible to replace its input literals by ORs of new variables so that the resulting monotone circuit represents the graph G_f .*

Proof. Any input literal x_i^a in a circuit for $f(x, y)$ corresponds to the biclique $U_i^a \times V$ with $U_i^a = \{u \in U : u_i = a\}$. Every such biclique is represented by an OR $\bigvee_{u \in U_i^a} z_u$ of $2^{m-1} = n/2$ new variables. \square

Instead of replacing input literals by ORs one can also replace them by any other boolean functions that compute 0 on the all-0 vector, and compute 1 on any input

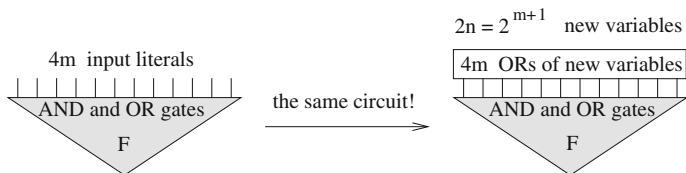


Fig. 1.14 Having a circuit F computing a boolean function f of $2m$ variables, we obtain a (monotone) circuit representing the graph G_f by replacing each input literal in F by an appropriate OR of new variables

vector with exactly one 1. In particular, parity functions also have this property, as well as any function $g(Z) = \varphi(\sum_{w \in S} z_w)$ with $\varphi : \mathbb{N} \rightarrow \{0, 1\}$, $\varphi(0) = 0$ and $\varphi(1) = 1$ does.

The Magnification Lemma is particularly appealing when dealing with circuits containing *unbounded* fanin OR (or unbounded fanin Parity gates) on the next to the input layer (Fig. 1.14). In this case the total number of gates in the circuit computing f is exactly the number of gates in the obtained circuit representing the graph G_f ! Thus if we could prove that some explicit bipartite $n \times n$ graph with $n = 2^m$ cannot be represented by such a circuit of size n^ϵ , then this would immediately imply that the corresponding boolean function $f(x, y)$ in $2m$ variables cannot be computed by a (non-monotone!) circuit of size $n^\epsilon = 2^{\epsilon m}$, which is already exponential in the number of variables of f . We will use Lemma 1.32 in Sect. 11.6 to prove truly exponential lower bounds for unbounded-fanin depth-3 circuits with parity gates on the bottom layer.

It is important to note that moderate lower bounds for graphs even in very weak circuit models (where strong lower bounds for boolean functions are easy to show) would yield impressive lower bounds for boolean circuits in rather nontrivial models. To demonstrate this right now, let $\text{cnf}(G)$ denote the smallest number of clauses in a monotone CNF (AND of ORs of variables) representing the graph G .

A bipartite graph is $K_{2,2}$ -free if it does not have a cycle of length 4, that is, if its adjacency matrix does not have a 2×2 all-1 submatrix.

■ **Research Problem 1.33.** Does there exist a constant $\epsilon > 0$ such that $\text{cnf}(G) \geq D^\epsilon$ for every bipartite $K_{2,2}$ -free graph G of average degree D ?

We will see later in Sect. 11.6 that a positive answer would give an explicit boolean function f of n variables such that any DeMorgan circuit of depth $\mathcal{O}(\log n)$ computing f requires $\omega(n)$ gates (cf. Research Problem 11.17). Thus graph complexity is a promising tool to prove lower bounds for boolean functions. Note, however, that even small lower bounds for graphs may be very difficult to prove. If, say, $n = 2^m$ and if $f(x, y)$ is the parity function of $2m$ variables, then any CNF for f must have at least $2^{2m-1} = n^2/2$ clauses. But the bipartite $n \times n$ graph G_f corresponding to this function consists of just two complete bipartite subgraphs; hence, G_f can be represented by a monotone CNF consisting of just four clauses.

1.8 A Constant Factor Away From $P \neq NP$?

Having warned about the difficulties when dealing with the graph complexity, in this section we sketch a potential (albeit very hard to realize) approach to proving strong lower bounds on circuit complexity of boolean functions using the graph complexity.

Recall that a DeMorgan circuit consists of fanin-2 AND and OR gates, and has all variables as well as their negations as inputs. A monotone circuit is a DeMorgan circuit without negated variables as inputs.

Proposition 1.34. *Almost all bipartite $n \times n$ graphs require monotone circuits of size $\Omega(n^2 / \log n)$ to represent them.*

Proof. Easy counting (as in the proof of Theorem 1.14) shows that there are at most $(nt)^{O(t)}$ monotone circuits with at most t gates. Since we have 2^{n^2} graphs, and different graphs require different circuits, the lower bound follows. \square

Thus the overwhelming majority of graphs require an almost-quadratic number of gates to represent. On the other hand, we are now going to show (Corollary 1.36 below) that any *explicit* bipartite $n \times n$ graph which cannot be represented by a monotone circuit with fewer than $7n$ gates would give us an explicit boolean function f in $2m$ variables which cannot be computed by a non-monotone(!) DeMorgan circuit with fewer than 2^m gates. That is, linear lower bounds on the monotone complexity of graphs imply exponential lower bounds on the non-monotone complexity of boolean functions.

When constructing the circuit for the graph G , as in the Magnification Lemma, we replace $4m$ input literals in a circuit for f_G by $4m = 4 \log n$ disjunctions of $2n = 2^{m+1}$ (new) variables. If we compute these disjunctions separately then we need about $mn = n \log n$ fanin-2 OR gates. The disjunctions can, however, be computed much more efficiently using only about n OR gates, if we compute all these disjunctions simultaneously. This can be shown using the so-called “transposition principle”.

Let $A = (a_{ij})$ be a boolean $p \times q$ matrix. Our goal is to compute the transformation $y = Ax$ over the boolean semiring. Such a transformation computes p boolean sums (disjunctions) of q variables x_1, \dots, x_q :

$$y_i = \bigvee_{j=1}^q a_{ij} x_j = \bigvee_{j: a_{ij}=1} x_j \quad \text{for } i = 1, \dots, p.$$

Thus, our question reduces to estimating the *disjunctive complexity*, $\text{OR}(A)$, of A defined as the minimum number of fanin-2 OR gates required to simultaneously compute all these p disjunctions.

By computing all p disjunctions separately, we see that $\text{OR}(A) < pq$. However, in some situations (as in the graph complexity) we have that the number p of disjunctions (rows) is much smaller than the number q of variables (columns). In the

context of graph complexity, we have $p = 4m$ and $q = 2^{m+1}$; hence, $p < 4 \log_2 q$. In such situations, it would be desirable to somehow “replace” the roles of rows and columns. That is, it would be desirable to relate the disjunctive complexity of a matrix A with the disjunctive complexity of the transposed matrix A^T ; recall that the transpose of a matrix $A = (a_{ij})$ is the matrix $A^T = (b_{ij})$ with $b_{ij} = a_{ji}$.

Transposition Principle. If A is a boolean matrix with p rows and q columns, then

$$\text{OR}(A^T) = \text{OR}(A) + p - q.$$

This principle was independently pointed out by Bordewijk (1956) and Lupanov (1956) in the context of rectifier networks. Mitiagin and Sadovskii (1965) proved the principle for boolean circuits, and Fiduccia (1973) proved it for bilinear circuits over any commutative semiring.

Proof. Let $A = (a_{ij})$ be a $p \times q$ boolean matrix, and take a circuit F with fanin-2 OR gates computing $y = Ax$. This circuit has q input nodes x_1, \dots, x_q and p output nodes y_1, \dots, y_p . At y_i the disjunction $\bigvee_{j: a_{ij}=1} x_j$ is computed.

Let $\alpha(F)$ be the number of gates in F . Since each non-input node in F has fanin 2, we have that $\alpha(F) = e - v + q$, where e is the total number of wires and v is the total number of nodes (including the q input nodes). Since the circuit F computes $y = Ax$ and has only OR gates, we have that $a_{ij} = 1$ if and only if there exists a directed path from the j -th input x_j to the i -th output y_i .

We now transform F to a circuit F' for $x = A^T y$ such that the difference $e' - v'$ between the numbers of wires and nodes in F' does not exceed $e - v$. First, we transform F so that no output gate is used as an input to another gate; this can be achieved by adding nodes of fanin 1. After that we just reverse the orientation of wires in F , contract all resulting fanin-1 edges, and replace each node of fanin larger than 2 by a binary tree of OR gates (see Fig. 1.15). Finally, assign OR gates to all n input gates of F (now the output gates of F').

It is easy to see that the new circuit F' computes $A^T y$: there is a path from y_i to x_j in F' iff there is a path from x_j to y_i in F . Moreover, since $e' - v' \leq e - v$, the new circuit F' has

$$\alpha(F') = e' - v' + p \leq e - v + p = \alpha(F) + p - q$$

gates. This shows that $\text{OR}(A^T) \leq \text{OR}(A) + p - q$, and by symmetry, that $\text{OR}(A) \leq \text{OR}(A^T) + q - p$. \square

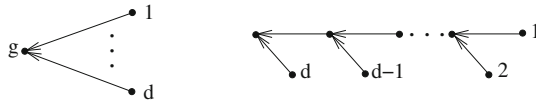


Fig. 1.15 We replace a node (an OR gate) g of fanin d by $d - 1$ nodes each of fanin 2. In the former circuit we have $e - v = d - 1$, and in the latter $e' - v' = 2(d - 1) - (d - 1) = d - 1 = e - v$

Corollary 1.35. *Let A be a boolean $p \times q$ matrix. Then, for every positive integer s dividing p ,*

$$\text{OR}(A) \leq sq + s2^{p/s} - 2p - s.$$

Proof. The proof is similar to that of Lemma 1.2. We want to compute a set Ax of p disjunctions on q variables. Split the transposed $q \times p$ matrix A^T into s submatrices, each of dimension $q \times (p/s)$. By taking a circuit computing all possible disjunction of p/s variables, we can compute disjunctions in each of these submatrices using at most $2^{p/s} - p/s - 1$ OR gates. By adding $q(s - 1)$ gates to combine the results of ORs computed on the rows of the submatrices, we obtain that $\text{OR}(A^T) \leq s2^{p/s} - p - s + q(s - 1)$ and, by the Transposition Principle,

$$\text{OR}(A) \leq \text{OR}(A^T) + q - p = sq + s2^{p/s} - 2p - s. \quad \square$$

In particular, taking $s = 1$, we obtain an upper bound $\text{OR}(A) \leq q + 2^p - 2p - 1$ which, as shown by Chashkin (1994) is optimal for $p \leq \log q$. Using a different argument (without applying the Transposition Principle), Pudlák et al. (1988) proved a slightly worse upper bound $\text{OR}(A) \leq q + 2^{p+1} - p - 2$.

Now we are able to give one consequence of the Transposition Principle for non-monotone circuits. Given a boolean function $f_{2m}(x, y)$ in $2m$ variables, its graph is a bipartite $n \times n$ graph G_f with $n = 2^m$ whose vertices are vectors in $\{0, 1\}^m$, and two vertices x and y from different parts are adjacent iff $f_{2m}(x, y) = 1$.

Corollary 1.36. *If a boolean function f_{2m} can be computed by a non-monotone DeMorgan circuit of size M , then its graph G_f can be represented by a monotone circuit of size $M + (6 + o(1))n$.*

Proof. Let $G_f = (V_1, V_2, E)$ be the graph of $f_{2m}(x, y)$. By Magnification Lemma, each of $2m = 2 \log n$ x -literals in a circuit computing f_{2m} is replaced by a disjunction on the set $\{z_u : u \in V_1\}$ of n variables. By Corollary 1.35 (with $p = 2 \log n$, $q = n$ and $s = 3$), all these disjunctions can be simultaneously computed using fewer than $3n + 3n^{2/3}$ fanin-2 OR gates. Since the same also holds for y -literals, we are done. \square

■ **Research Problem 1.37.** What is the smallest constant c for which the conclusion of Corollary 1.36 holds with $M + (6 + o(1))n$ replaced by $M + cn$?

By Corollary 1.36, any bipartite $n \times n$ graph requiring, say, at least $7n$ AND and OR gates to represent it gives a boolean function of $2m = 2 \log n$ variables requiring at least $\Omega(n) = \Omega(2^m)$ AND, OR and NOT gates to compute it. It is therefore not surprising that proving even linear lower bounds cn for explicit graphs may be a very difficult task. Exercise 1.10 shows that at least for $c = 2$ this task is still tractable.

■ **Research Problem 1.38.** Exhibit an explicit bipartite $n \times n$ graph requiring at least cn AND and OR gates to represent it, for $c > 2$.

Readers interested in this problem might want to consult the paper of Chashkin (1994) giving a somewhat tighter connection between lower bounds for graphs and the resulting lower bounds for boolean functions. In particular, he shows that the constant 6 in Corollary 1.36 can be replaced by 4, and even by 2 if the graph is unbalanced.

Exercises

1.1. Let, as before, $\text{Dec}(A)$ denote the minimum weight of a decomposition of a boolean matrix A . Suppose that A does not contain an $a \times b$ all-1 submatrix with $a + b > k$. Show that $\text{Dec}(A) \geq |A|/k$.

1.2. Let s_n be the smallest number s such that every boolean function of n variables can be computed by a DeMorgan formula of leafsize at most s . Show that $s_n \leq 4 \cdot 2^n - 2$. *Hint:* Use the recurrence (1.1) to show that $s_n \leq 4 \cdot 2^n - 2$, and apply induction on n .

1.3. Let $m = \lceil \log_2(n + 1) \rceil$, and consider the function $\text{Sum}_n : \{0, 1\}^n \rightarrow \{0, 1\}^m$ which, given a vector $x \in \{0, 1\}^n$ outputs the binary code of the sum $x_1 + x_2 + \dots + x_n$. Consider circuits where all boolean functions of two variables are allowed as gates, and let $C(f)$ denote the minimum number of gates in such a circuit computing f .

(a) Show that $C(\text{Sum}_n) \leq 5n$. *Hint:* Fig. 1.3.

(b) Show that $C(f_n) \leq 5n + o(n)$ for every symmetric function f_n of n variables.

Hint: Every boolean function g of m variables has $C(g) \leq 2^m/m$.

1.4. (Circuits as linear programs) Let $F(x)$ be a circuit over $\{\wedge, \vee, \neg\}$ with m gates. Show that there is a system $L(x, y)$ of $\mathcal{O}(m)$ linear constraints (linear inequalities with coefficients ± 1) with m y -variables such that, for every $x \in \{0, 1\}^n$, $F(x) = 1$ iff there is 0–1 vector y such that all constraints in $L(x, y)$ are satisfied.

Hint: Introduce a variable for each gate. For an \wedge -gate $g = u \wedge v$ use the constraints $0 \leq g \leq u \leq 1$, $0 \leq g \leq v \leq 1$, $g \geq u + v - 1$. What constraints to take for \neg -gates and for \vee -gates? For the output gate g add the constraint $g = 1$. Show that, if the x -variables have values 0 and 1, then all other variables are forced to have value 0 or 1 equal to the output value of the corresponding gate.

1.5. Write $g \leq h$ for boolean functions of n variables, if $g(x) \leq h(x)$ for all $x \in \{0, 1\}^n$. Call a boolean function h a *neighbor* of a boolean function g if either $g \oplus a \leq h \oplus a \oplus 1$ for some $a \in \{0, 1\}$, or $g \oplus x_i \leq g \oplus h$ for some $i \in \{1, \dots, n\}$. Show that:

(a) Constants 0 and 1 are neighbors of all non-constant functions.

(b) Neighbors of the OR gate \vee are all the two variable boolean functions, except the parity \oplus and the function \vee itself.

1.6. (Minimal circuits are very unstable) Let F be a circuit over some basis computing a boolean function f , and assume that F is *minimal*, that is, no circuit

with a smaller number of gates can compute f . In particular, minimal circuits are “unstable” with respect to deletion of its gates: the resulting circuit must make an error. The goal of this exercise is to prove that, in fact, minimal circuits are unstable in a much stronger sense: we cannot even *replace* a gate by another one. That is, the size of the resulting circuit remains the same but, nevertheless, the function computed by a new circuit differs from that computed by the original one.

Let F be a minimal circuit, v a gate in it of fanin m , and h be a boolean function of m variables. Let $F_{v \rightarrow h}$ be the circuit obtained from F as follows: replace the boolean function g attached to the gate v by h and remove all the gates that become redundant in the resulting circuit. Prove that, if h is a neighbor of g , then $F_{v \rightarrow h} \neq F$.

Hint: Since F is minimal, we cannot replace the gate v by a constant a , that is, there must be at least one vector $x \in \{0, 1\}^n$ such that $F_{v \rightarrow a}(x) \neq F(x)$.

1.7. Let $n = 2^r$ and consider two sequences of variables $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_r)$. Each assignment $a \in \{0, 1\}^r$ to the y -variables gives us a unique natural number $\text{bin}(a) = 2^{r-1}a_1 + \dots + 2a_{r-1} + a_r + 1$ between 1 and n ; we call $\text{bin}(a)$ the *code* of a . The *storage access function* $f(x, y)$ is a boolean function of $n + r$ variables defined by: $f(x, y) := x_{\text{bin}(y)}$.

Show that the monomial $K = x_1 x_2 \dots x_n$ is a minterm of f , but still f can be written as an $(r + 1)$ -DNF. *Hint:* For the second claim, observe that the value of $f(x, y)$ depends only on $r + 1$ bits y_1, \dots, y_r and $x_{\text{bin}(y)}$.

1.8. Let $G = ([n], E)$ be an n -vertex graph, and d_i be the degree of vertex i in G . Then G can be represented by a monotone formula $F = F_1 \vee \dots \vee F_n$, where

$$F_i = x_i \wedge \left(\bigvee_{j: \{i, j\} \in E} x_j \right).$$

A special property of this formula is that the i -th variable occurs at most $d_i + 1$ times. Prove that, if G has no complete stars, then *any* minimal monotone formula representing G must have this property.

Hint: Take a minimal formula F for G , and suppose that some variable x_i occurs $m_i > d_i + 1$ times in it. Consider the formula $F' = F_{x_i=0} \vee F_i$, where $F_{x_i=0}$ is the formula obtained from F by setting to 0 all m_i occurrences of the variable x_i . Show that F' represents G , and compute its leafsize to get a contradiction with the minimality of F .

1.9. Say that a graph is *saturated*, if its complement contains no triangles and no isolated vertices. Show that for every saturated graph $G = (V, E)$, its quadratic function $f_G(x) = \bigvee_{uv \in E} x_u x_v$ is the unique(!) monotone boolean function representing the graph G .

1.10. Let $G_n = K_{n-1} + E_1$ be a complete graph on $n - 1$ vertices $1, 2, \dots, n - 1$ plus one isolated vertex n . Let $F(x_1, \dots, x_n)$ be an arbitrary monotone circuit with fanin-2 AND and OR gates representing G_n . Show that G_n cannot be represented by a monotone circuit using fewer than $2n - 6$ gates.

Hint: Show that if $n \geq 3$ then every input gate x_i for $i = 1, \dots, n - 1$ has fanout at least 2.

1.11. Let $n = 2^m$ be a power of 2. Show that Th_2^n can be computed by a monotone DeMorgan formula with at most $n \log_2 n$ leaves.

Hint: Associate with each index $i \in [n]$ its binary code in $\{0, 1\}^m$. For $k \in [m]$ and $a \in \{0, 1\}$, let $F_{k,a}$ be the OR of all variables x_i such that the binary code of i has a in the k -th position. Show that the monotone formula $F = \bigvee_{k=1}^m F_{k,0} \wedge F_{k,1}$ computes Th_2^n .

1.12. (Hansel 1964) The goal of this exercise is to show that

$$S_+(\text{Th}_2^n) \geq \frac{1}{2} n \log_2 n.$$

Let $F(x)$ be a monotone switching network computing Th_2^n with the start node s and the target node t . Say that F is *canonical* if it has the following property: if a node v is joined to s or to t by a contact x_i , then no other edge incident with v has x_i as its label.

- (a) Suppose that $F(x) = 0$ for all input vectors x with at most one 1. Show that F can be made canonical without increasing the number of contacts.

Hint: Assume that some node u is joined to the source node s and to some other node v by edges with the same label x_i . Then $v \neq t$ (why?). Remove the edge $\{u, v\}$ and add the edge $\{s, v\}$ labeled by x_i . Show that the obtained network computes the same function.

- (b) Let F be a *minimal* canonical monotone network computing the threshold-2 function Th_2^n . Show that every node $u \notin \{s, t\}$ is adjacent with both nodes s and t .

Hint: If we remove a label of any contact in a minimal network, then the new network must make an error.

- (c) Let m be the number of contacts in a network F from (b). Show that $\text{Th}_2^n(x)$ can be written as an OR $F_1 \vee F_2 \vee \dots \vee F_t$ of ANDs

$$F_k(x) = \left(\bigvee_{i \in A_k} x_i \right) \wedge \left(\bigvee_{i \in B_k} x_i \right)$$

such that $A_k \cap B_k = \emptyset$ and $w \leq 2m$, where $w := \sum_{k=1}^t (|A_k| + |B_k|)$ is the total number of occurrences of variables in the formula.

- (d) Show that any expression of Th_2^n as in (c) must contain $w \geq n \log_2 n$ occurrences of variables.

Hint: For a variable x_i , let m_i be the number of ANDs F_k containing this variable. Show that $w = \sum_{i=1}^n m_i$. To lower bound this sum, throw a fair 0–1 coin for each of the ANDs F_k and remove all occurrences of variables x_i with $i \in A_k$ from the entire formula if the outcome is 0; if the outcome is 1, then remove all occurrences of variables x_i with $i \in B_k$. Let $X = X_1 + \dots + X_n$, where X_i is the indicator variable for the event “the variable x_i survives”. Since at most one variable can survive at the end (why?), we have that $E[X] \leq 1$. On the other hand, each variable x_i will survive with probability 2^{-m_i} (why?). Now use the linearity of expectation together with the arithmetic-geometric mean inequality $(\sum_{i=1}^n a_i)/n \geq (\prod_{i=1}^n a_i)^{1/n}$ with $a_i = 2^{-m_i}$ to obtain the desired lower bound on $\sum_{i=1}^n m_i$.

Table 1.1 Upper bounds for *any* symmetric boolean function f_n of n variables

$BP(f_n) \leq cn^2 / \log_2 n$	where $c = 2 + o(1)$; Lupanov (1965b)
$NBP(f_n) \leq n^{3/2}$	Lupanov (1965b)
$L(f_n) \leq n^{4.93}$	Khrapchenko (1972)
$C_*(f_n) \leq 4.5n + o(n)$	Demenev et al. (2010); this improves a simple upper bound $C_*(f_n) \leq 5n + o(n)$ which follows from a construction used by Lupanov (1965); see Exercise 1.3

Appendix: Known Bounds for Symmetric Functions

Here we summarize some (not all!) known results concerning bounds on the complexity of symmetric functions in various circuit models. Recall that a boolean function $f(x_1, \dots, x_n)$ is *symmetric* if its value only depends on the sum $x_1 + \dots + x_n$. Examples of symmetric functions are the parity function

$$\oplus_n(x) = 1 \text{ if and only if } x_1 + \dots + x_n \text{ is odd,}$$

all threshold functions

$$\text{Th}_k^n(x) = 1 \text{ if and only if } x_1 + \dots + x_n \geq k,$$

as well as the majority function

$$\text{Maj}_n(x) = 1 \text{ if and only if } x_1 + \dots + x_n \geq \lceil n/2 \rceil.$$

Let $C(f)$ and $L(f)$ denote, respectively, the minimum number of gates in a circuit and in a formula over $\{\wedge, \vee, \neg\}$ computing f . Let also $S(f)$, $BP(f)$ and $NBP(f)$ denote, respectively, the minimum number of contacts (labeled edges) in a switching network, in a deterministic and in a nondeterministic branching program computing f . Subscript “+” denotes the *monotone* versions of these measures, and subscript “*” means that all boolean functions of two variables can be used as gates.

Some relations between these basic measures are summarized in the following chain of inequalities (we will use $f \leq g$ to denote $f = \mathcal{O}(g)$):

$$C(f)^{1/3} \leq NBP(f) \leq S(f) \leq BP(f) \leq L(f) \leq NBP(f)^{\mathcal{O}(\log NBP(f))}.$$

Proofs are easy and can be found, for example, in Pudlák (1987).

Table 1.2 Bounds for the parity function

$S(\oplus_n) = 4n - 4$	Cardot (1952); apparently, this was the first nontrivial lower bound at all!
$C(\oplus_n) = 4n - 4$	Redkin (1973)
$L(\oplus_n) \leq \frac{9}{8}n^2$	Yablonskii (1954); see Theorem 6.29 below
$L(\oplus_n) \geq n^{3/2}$	Subbotovskaya (1961); see Sect. 6.3 below
$L(\oplus_n) \geq n^2$	Khrapchenko (1971); n is power of 2; see Sect. 6.8 below
$L(\oplus_n) \geq n^2 + c$	Rychkov (1994); $c = 3$ for odd $n \geq 5$, and $c = 2$ for even $n \geq 6$ which are not powers of 2

Table 1.3 Bounds for threshold functions in *non-monotone* models

$L(\text{Th}_2^n) \geq \frac{1}{4}n \log_2 n$	Krichevskii (1964) ^a
$L(\text{Th}_2^n) \geq n \lfloor \log_2 n \rfloor$	Lozhkin (2005)
$L_+(\text{Th}_2^n) \leq n \log_2 n$	if n is a power of 2; see Exercise 1.11
$L(\text{Th}_k^n) \geq k(n - k + 1)$	Khrapchenko (1971); see Sect. 6.8
$L_*(\text{Maj}_n) = \Omega(n \ln n)$	Fischer et al. (1982)
$L_*(\text{Th}_2^n) = \Omega(n \ln \ln n)$	Pudlák (1984)
$L_*(\text{Th}_k^n) \leq n^{3.13}$	Paterson et al. (1992)
$L(\text{Maj}_n) \leq n^{4.57}$	Paterson and Zwick (1993b)
$\text{BP}(\text{Th}_k^n) \leq n^{3/2}$	Lupanov (1965b)
$S(\text{Th}_k^n) \leq \frac{1}{p}n \ln^4 n$	where $p = (\ln \ln n)^2$; Krasulina (1987, 1988)
$\text{BP}(\text{Th}_k^n) \leq \frac{1}{p}n \ln^3 n$	where $p = (\ln \ln n)(\ln \ln \ln n)$; Sinha and Thathachar (1997)
$\text{BP}(\text{Maj}_n) = \Omega(np)$	where $p = \ln \ln n / \ln \ln \ln n$; Pudlák (1984)
$\text{BP}(\text{Maj}_n) = \Omega(np)$	where $p = \ln n / \ln \ln n$; Babai et al. (1990)
$S(\text{Maj}_n) = \omega(n)$	Grinchuk (1987, 1989)
$\text{NBP}(\text{Maj}_n) = \omega(n)$	Razborov (1990b)

^aKrichevskii (1964) actually proved an intriguing *structural* result: among minimal formulas computing Th_2^n there is a monotone formula of the form $F(x) = \bigvee_{k=1}^t (\bigvee_{i \in S_k} x_i) \wedge (\bigvee_{i \in T_k} x_i)$, where $S_k \cap T_k = \emptyset$ for all $k = 1, \dots, t$; see also Sect. 6.12

Table 1.4 Bounds for threshold functions in *monotone* models

$NBP_+(\text{Th}_k^n) = k(n - k + 1)$	Markov (1962); see Theorem 1.8 above
$NBP_+(\text{Th}_k^n) \leq pk(n - k)$	where $p = \ln(n - k)$, if no unlabeled edges (rectifiers) are allowed; Halldórsson et al. (1993)
$NBP_+(\text{Th}_k^n) = \Omega(pkn)$	where $p = \ln \frac{n}{k}$, if no unlabeled edges (rectifiers) are allowed; Radhakrishnan (1997)
$S_+(\text{Th}_2^n) = np + 2(n - 2^p)$	where $p := \lfloor \log_2 n \rfloor$; Krichevskii (1965), Hansel (1966)
$S(\text{Th}_2^n) \leq 3n - 4$	easy exercise, see Fig. 1.5
$S_+(\text{Maj}_n) \leq n^{4.99}$	Dubiner and Zwick (1992)
$L_+(\text{Maj}_n) \leq n^{5.3}$	Valiant (1984). As observed by Lozhkin and Semenov (1988), the proof actually gives $\mathcal{O}(k^{4.3}n \log^2 n)$ for every k .
$L_+(\text{Th}_k^n) \leq k^{6.3}n \log n$	Friedman (1986)
$L_+(\text{Th}_k^n) \leq k^{4.27}n \log n$	Boppana (1986)
$C(\text{Th}_k^n) \leq kn + p$	where $p = \mathcal{O}(n^{1-1/k})$; Dunne (1984)
$C(\text{Th}_k^n) \leq n \log k$	Kochol (1989); the proof is a simple application of a rather non-trivial result of Ajtai et al. (1983) stating that <i>all</i> threshold functions Th_k^n , $k = 1, \dots, n$, can be simultaneously computed by a monotone circuit of size $\mathcal{O}(n \log n)$ and depth $\mathcal{O}(\log n)$



<http://www.springer.com/978-3-642-24507-7>

Boolean Function Complexity

Advances and Frontiers

Jukna, S.

2012, XVI, 620 p., Hardcover

ISBN: 978-3-642-24507-7