For two boolean functions f and g in n variables, we write $f \leqslant g$ if $f(x) \leqslant g(x)$ for all $x \in \{0,1\}^n$.

Definition 10.15. Let $f(x_1, \ldots, x_n)$ be a monotone boolean function. We say that f is *t-simple* if for every pair of integers $2 \leqslant s, r \leqslant n$ there exists a monotone s-CNF C consisting of at most $t \cdot (r-1)^s$ clauses, a monotone r-DNF D consisting of at most $t \cdot (s-1)^r$ monomials, and a subset $I \subseteq \{1, \ldots, n\}$ of $|I| \leqslant s-1$ indices, so that either

$$C \leqslant f \quad \text{or} \quad f \leqslant D \vee \Big(\bigvee_{i \in I} x_i\Big)$$

(or both) hold.

We have the following general lower bounds criterion for monotone boolean circuits. In fact, the same criterion holds for more general circuits, where we can use as gates not just boolean And and Or but also any *real-valued* functions $\varphi : \mathbb{R}^2 \to \mathbb{R}$ such that $\varphi(x_1, x_2) \leqslant \varphi(y_1, y_2)$ whenever $x_1 \leqslant y_1$ and $x_2 \leqslant y_2$. A similar criterion holds also with unbounded fan-in boolean gates (see Jukna 1999 for details).

Theorem 10.16. *If a monotone boolean function can be computed by a monotone circuit of size t, then it is t-simple.*

Proof. Given a monotone circuit, the idea is to approximate every intermediate gate (more exactly – the function computed at the gate) by an $(s-1)$-And-Or and an $(r-1)$-Or-And, and to show that doing so we do not introduce too many errors. If the function computed by the whole circuit is not t-simple, then it cannot be approximated well by such an And-Or/Or-And pair meaning that every such pair must make many errors. Since the number of errors introduced at each separate gate is small, the total number of gates must be large.

To make as few errors at each gate as possible we will use the following simple fact which allows us to convert an $(s-1)$-And-Or into a small $(r-1)$-Or-And and vice versa (cf. also Lemmas 7.4 and 10.7).

Claim 10.17. *For every $(s-1)$-And-Or f_0 there is an $(r-1)$-Or-And f_1 and an r-DNF D such that $|D| \leqslant (s-1)^r$ and*

$$f_1 \leqslant f_0 \leqslant f_1 \vee D.$$

Dually, for every $(r-1)$-Or-And f_1 there is an $(s-1)$-And-Or f_0 and an s-CNF C such that $|C| \leqslant (r-1)^s$ and

$$f_0 \wedge C \leqslant f_1 \leqslant f_0.$$

Proof. We prove the first claim (the second is dual). Let $f_0 = S_1 \wedge S_2 \wedge \cdots \wedge S_m$ be an $(s-1)$-And-Or. We associate with f_0 the following tree T of fan-out at most $s-1$. The first node of T corresponds to the first clause S_1, and the outgoing $|S_1|$ edges are labeled by the variables from S_1. Suppose we have reached a node v, and let M be the monomial consisting of the labels of edges from the root to v. If M intersects all the clauses of f_0, then v is a leaf. Otherwise, let S_i be the *first* clause such that $M \cap S_i = \emptyset$. Then the node v has $|S_i|$ outgoing edges labeled by the variables in S_i.

Each path from the root to a leaf of T gives us a monomial M which intersects all the clauses of f_0. The Or of paths in T from the root to leafs of height at most $r-1$ give us the desired $(r-1)$-Or-And f_1. It is clear that $f_1 \leqslant f_0$ because each monomial of f_1 intersects all the clauses of f_0. The OR of paths to the nodes of height r give us the desired r-DNF D. This DNF has at most $(s-1)^r$ monomials since every node of T has fan-out at most $s-1$. □

We now turn to the actual proof of Theorem 10.16. Let $F(x_1, \ldots, x_n)$ be a monotone boolean function, and suppose that F can be computed by a monotone circuit of size t. Our goal is to show that then the function F is t-simple. To do this, fix an arbitrary pair of integers $2 \leqslant s, r \leqslant n$.

Let $f = g * h$ be a gate in our circuit computing F. By an *approximator* of this gate we will mean a pair f_0, f_1, where f_0 is an $(s-1)$-And-Or (a *left* approximator of f) and f_1 is an $(r-1)$-Or-And (a *right* approximator of f) such that $f_1 \leqslant f_0$.

We say that such an approximator f_0, f_1 of f introduces a new error on input $x \in \{0,1\}^n$ if the approximators of g and of h did not make an error on x, but the approximator of f does. That is, $g_0(x) = g_1(x) = g(x)$ and $h_0(x) = h_1(x) = h(x)$, but either $f_0(x) \neq f(x)$ or $f_1(x) \neq f(x)$.

We define approximators inductively as follows.

Case 1: f is an input variable, say, $f = x_i$.

In this case we take $f_0 = f_1 \rightleftharpoons x_i$. It is clear that this approximator introduces no errors.

Case 2: f is an And gate, $f = g \wedge h$.

In this case we take $f_0 \rightleftharpoons g_0 \wedge h_0$ as the left approximator of f; hence, f_0 introduces no new errors. To define the right approximator of f we use Claim 10.17 to convert f_0 into an $(r-1)$-Or-And f_1; hence, $f_1 \leqslant f_0$. Let E_f be the set of inputs on which f_1 introduces a new error, i.e.,

$$E_f \rightleftharpoons \{x : f(x) = f_0(x) = 1 \text{ but } f_1(x) = 0\}.$$

By Claim 10.17, all these errors can be "corrected" by adding a relatively small r-DNF: there is an r-DNF D such that $|D| \leqslant (s-1)^r$ and $D(x) = 1$ for all $x \in E_f$.

Case 3: f is an Or gate, $f = g \vee h$.

In this case we take $f_1 \rightleftharpoons g_1 \vee h_1$ as the right approximator of f; hence, f_1 introduces no new errors. To define the left approximator of f we use Claim 10.17 to convert f_1 into an $(s-1)$-And-Or f_0; hence, $f_1 \leqslant f_0$. Let E_f be the set of inputs on which f_0 introduces a new error, i.e.,

$$E_f \rightleftharpoons \{x : f(x) = f_1(x) = 0 \text{ but } f_0(x) = 1\}.$$

By Claim 10.17, all these errors can be "corrected" by adding a relatively small s-CNF: there is an r-CNF C such that $|C| \leqslant (r-1)^s$ and $C(x) = 0$ for all $x \in E_f$.

Proceeding in this way we will reach the last gate of our circuit computing the given function F. Let F_0, F_1 be its approximator, and let E be the set of all inputs $x \in \{0,1\}^n$ on which F differs from at least of one of the functions F_0 or F_1. Since at input gates (= variables) no error was made, for every such input $x \in E$, the corresponding error should be introduced at some intermediate gate. That is, for every $x \in E$ there is a gate f such that $x \in E_f$ (approximator of f introduces an error on x for the first time). But we have shown that, for each gate, all these errors can be corrected by adding an s-CNF of size at most $(r-1)^s$ or an r-DNF of size at most $(s-1)^r$. Since we have only t gates, all such errors $x \in E$ can be corrected by adding an s-CNF C of size at most $t \cdot (r-1)^s$ and an r-DNF D of size at most $t \cdot (s-1)^r$, that is, for all inputs $x \in \{0,1\}^n$, we have

$$C(x) \wedge F_0(x) \leqslant F(x) \leqslant F_1(x) \vee D(x).$$

This already implies that the function F is t-simple. Indeed, if the CNF F_0 is empty (i.e., if $F_0 \equiv 1$) then $C \leqslant F$, and we are done. Otherwise, F_0 must contain some clause S of length at most $s-1$, say, $S = \bigvee_{i \in I} x_i$ for some I of size $|I| \leqslant s-1$. Since $F_0 \leqslant S$, the condition $F_1 \leqslant F_0$ implies $F \leqslant F_1 \vee D \leqslant F_0 \vee D \leqslant S \vee D$, as desired. This completes the proof of Theorem 10.16. □

10.6.2 Explicit lower bounds

In order to show that a given boolean function cannot be computed by a monotone circuit of size at most t, it is enough, by Theorem 10.16, to show that the function is not t-simple for at least one(!) choice of parameters s and r. In the following three sections we demonstrate how this can be used to derive exponential lower bounds for concrete boolean functions.

In applications, boolean functions f are usually defined as set-theoretic predicates. In this case we say that f accepts a set $S \subseteq \{1, \ldots, n\}$ if and only if f accepts its incidence vector. It is therefore convenient to also have a set-theoretic definition of t-simplicity, which we give now.

A set S is a *positive input* for f if $f(S) = 1$, and a *negative input* if $f(\overline{S}) = 0$, where $\overline{S}$ is the complement of S. Put otherwise, a positive (negative) input is a set of variables which, if assigned the value 1 (0), forces the function to take the value 1 (0) regardless of the values assigned to the remaining variables. Note that one set S can be both positive and negative input! For example, if $f(x_1, x_2, x_3)$ outputs 1 iff $x_1 + x_2 + x_3 \geqslant 2$, then $S = \{1, 2\}$ is both positive and negative input for f, because $f(1, 1, x_3) = 1$ and $f(0, 0, x_3) = 0$.

The following is a set-theoretic equivalent of Definition 10.15 (show this):