1 Circuit Complexity

1.1 Definitions

A Boolean circuit C on n inputs $x_1, \ldots, x_n$ is a directed acyclic graph (DAG) with n nodes of in-degree 0 (the inputs $x_1, \ldots, x_n$), one node of out-degree 0 (the output), and every node of the graph except the input nodes is labeled by AND, OR, or NOT; it has in-degree 2 (for AND and OR), or 1 (for NOT). The Boolean circuit C computes a Boolean function $f(x_1, \ldots, x_n)$ in the obvious way: the value of the function is equal to the value of the output gate of the circuit when the input gates are assigned the values $x_1, \ldots, x_n$.

The *size* of a Boolean circuit C, denoted $|C|$, is defined to be the total number of nodes (gates) in the graph representation of C. The *depth* of a Boolean circuit C is defined as the length of a longest path (from an input gate to the output gate) in the graph representation of the circuit C.

A Boolean *formula* is a Boolean circuit whose graph representation is a tree.

Given a family of Boolean functions $f = \{f_n\}_{n \geq 0}$, where f_n depends on n variables, we are interested in the sizes of smallest Boolean circuits C_n computing f_n. Let $s(n)$ be a function such that $|C_n| \leq s(n)$, for all n. Then we say that the Boolean function family f is computable by Boolean circuits of size $s(n)$. If $s(n)$ is a polynomial, then we say that f is computable by polysize circuits.

It is not difficult to see that every language in P is computable by polysize circuits. Note that given any language L over the binary alphabet, we can define the Boolean function family $\{f_n\}_{n \geq 0}$ by setting $f_n(x_1, \ldots, x_n) = 1$ iff $x_1 \ldots x_n \in L$.

Is the converse true? No! Consider the following family of Boolean functions f_n, where $f_n(x_1, \ldots, x_n) = 1$ iff TM M_n halts on the empty tape; here, M_n denotes the nth TM in some standard enumeration of all TMs. Note that each f_n is a constant function, equal to 0 or 1. Thus, the family of these f_n's is computable by linear-size Boolean circuits. However, this family of f_n's is *not* computable by any algorithm (let alone any polytime algorithm), since the Halting Problem is undecidable. Thus, in general, the Boolean circuit model of computation is strictly more powerful that the Turing machine model of computation.

Still, it is generally believed that NP-complete languages cannot be computed by polysize circuits. Proving a superpolynomial circuit lower bound for any NP-complete language would imply that $\mathsf{P} \neq \mathsf{NP}$. (Check this!) In fact, this is one of the main approaches that was used in trying to show that $\mathsf{P} \neq \mathsf{NP}$. So far, however, nobody was able to disprove that every language in NP can be computed by linear-size Boolean circuits of logarithmic depth!

[1] This lecture is a modification of notes by Valentine Kabanets

1.2 TMs that take advice

The Boolean circuit model of computation is *nonuniform*, i.e., different algorithms (circuits) are used for inputs of different sizes, and there may be no uniform algorithm (TM) that, given n, will generate the nth Boolean circuit C_n.

This uniformity can be regarded as another kind of computational resources. We can imagine a TM equipped with a special read-only tape, called *advice tape*, where some advice string $a(n)$ appears when the TM is given an input x of length n. Note that the same advice string $a(n)$ is used for all inputs of length n. If such a TM decides a language L, then we say that L is accepted by a TM with advice.

More formally, we say that $L \in \mathsf{Time}(t(n))/f(n)$ if there is a family of advice strings $\{a_n\}_{n\geq 0}$ such that $|a_n| \leq f(n)$ for all n, and a $t(n)$-time TM M such that, for any string x of length n,

$$x \in L \Leftrightarrow (x, a_n) \in L(M).$$

The most important class of languages accepted by TM with advice is the class $\mathsf{P/poly} = \cup_k \mathsf{Time}(n^k)/n^k$.

Theorem 1. *A language L is computable by polysize Boolean circuits iff $L \in \mathsf{P/poly}$.*

Proof Sketch. $\Rightarrow$. Encode a Boolean circuit C_n as an advice string a_n. Then a polytime TM with advice a_n, can decide if $x \in L$ by decoding the Boolean circuit for L from a_n and evaluating this circuit on x.

$\Leftarrow$. Let $\{a_n\}_{n\geq 0}$ be the family of polysize advice strings for L, and let M be a polytime TM such that $x \in L$ iff $(x, a_{|x|}) \in L(M)$. We know that every polytime decidable language is computable by polysize Boolean circuits. Let $\{C_n\}_{n\geq 0}$ be a polysize circuit family computing the language $L(M)$. We have that $x \in L$ iff $C_m(x, a_{|x|}) = 1$, where $m = |x| + |a_{|x|}| \in \mathsf{poly}(|x|)$. By hardwiring the advice string a_n into a corresponding circuit C_m, we obtain a circuit family deciding the language L. The sizes of all these new circuits are bounded by some polynomial. □

1.3 $\mathsf{NP} \overset{?}{\subset} \mathsf{P/poly}$

There is an interesting connection between NP having polysize circuits and the collapse of a polytime hierarchy.

Theorem 2 (Karp-Lipton). *If $NP \subset \mathsf{P/poly}$, then $\mathsf{PH} = \Sigma_2^p$.*

Proof Sketch. As we argued earlier, to show that $\mathsf{PH} = \Sigma_2^p$, it suffices to prove that $\Sigma_2^p = \Pi_2^p$. To prove the latter, it actually suffices to argue that $\Pi_2^p \subseteq \Sigma_2^p$. (Show that this is indeed sufficient!)

Since $\mathsf{NP} \subset \mathsf{P/poly}$, there is a polysize family of circuits computing SAT. Moreover, since there is a polytime algorithm for finding a satisfying assignment, given access to an algorithm for SAT, we conclude that there is a polysize family of circuits C_n with the following property:

Given a propositional formula ϕ of size n, $C_n(\phi)$ outputs a satisfying assignment for ϕ if one exists, or outputs the string of 0s if ϕ is unsatisfiable.

Now, consider any language $L \in \Pi_2^p$. By definition, there is a polytime polybalanced relation R such that $x \in L$ iff $\forall y \exists z R(x, y, z)$. Consider the language $L' = \{(x, y) \mid \exists z R(x, y, z)\}$. Obviously, $L' \in \mathsf{NP}$. By our assumption, there is a polysize circuit family such that $C_m(x, y)$ outputs a satisfying assignment z for $R(x, y, z)$ if one exists; here, $m = |x| + |y|$.

We can test if $x \in L$ by the following polytime polybalanced formula: $\exists C_m \forall y R(x, y, C_m(x, y))$. Indeed, if $x \in L$, then there will be a polysize circuit C_m that would produce a satisfying z-assignment for $R(x, y, z)$ for every y. Conversely, if there is some small circuit C_m that produces a satisfying z-assignment for $R(x, y, z)$ for every y, then the formula $\forall y \exists z R(x, y, z)$ must be true, and hence, $x \in L$. Thus, we have shown that $L \in \Sigma_2^p$, as required. □

1.4 Hard Boolean functions

Every Boolean function f on n variables is computable by a Boolean circuit of size $O(n2^n)$: consider a DNF formula, which is an OR of at most 2^n ANDs, where each AND is a conjunction of n literals for each x such that $f(x) = 1$. A more careful argument shows that every Boolean function on n variables is computable by a Boolean circuit of size $\frac{2^n}{n}(1 + o(1))$.

A simple counting argument shows that almost all Boolean functions are hard in the sense that they require Boolean circuits of size $\Omega(2^n/n)$. The total number of n-variable Boolean functions is $B(n) = 2^{2^n}$. On the other hand, the total number of Boolean circuits of size s on n variables is at most (very roughly) $C(n, s) = ((n + 3)s^2)^s$; there are $n + 3$ gate types; each gate has at most two inputs; there are s gates. When $s < 2^n/cn$, we have $C(n, s) \ll B(n)$. So, most functions require $\Omega(2^n/n)$ circuit size. Similar argument for formulas shows that most n-variable Boolean functions require $\Omega(2^n/\log n)$ formula size. Remark: more careful arguments give $2^n/n$ and $2^n/\log n$ lower bounds for circuits and formulas, respectively.

Thus, we know that hard Boolean functions abound, but we cannot get our hands on any particular hard function. That is, we do not know whether any language in NEXP requires superpolynomial circuit size.

2 Parallel Computation

Imagine a Boolean formula on n variables. Suppose that we apply the appropriate electric currents to the inputs. How long will take for these currents to "propagate" through the formula, yielding the value of the formula on the given inputs? A moment's thought suggests that this time is proportionate to the *depth* of the formula. Thus, the smaller the depth, the faster we can compute the formula value on any given input.

The considerations above show the importance of the following complexity classes (actively studied by Nick Pippenger, and named in his honor NC, for "Nick's Class", by Steve Cook):

$$\mathsf{NC}_i = \{L \mid L \text{ is decided by a family of polysize circuits of depth } O(\log^i n)\}$$

The class $\mathsf{NC} = \cup_i \mathsf{NC}_i$.

Thus, NC_1 is the class of languages decided by polysize circuits of logdepth. In general, almost all Boolean functions need circuits of linear depth. So, those Boolean functions that can be computed by *shallow* circuits are the functions computable *in parallel*, as the depth of a circuit corresponds to the parallel time.

Some comments on the definition of NC. For NC_1, it does not matter whether we consider polysize circuits of logdepth or *formulas* of logdepth! This is because any polysize circuit of depth $O(\log n)$ can be easily converted into a formula of the same depth $O(\log n)$, by "unwinding" the underlying graph into a tree (i.e., each gate gives rise to as many copies of itself as there are paths from that gate to the output gate of the circuit). (Check this!)

Also, the class of languages computable by logdepth formulas is the same as that computable by polysize formulas (without any depth restrictions)! The reason is that any given polysize formula can be "re-balanced" to become of logdepth. The details follow.

Let $F(x_1, \ldots, x_n)$ be a formula of size s, where $s \in \mathsf{poly}(n)$. Then it is possible to show that F will contain a subformula F' of size t, where $s/3 \leq t \leq 2s/3$. (Think of a tree with two subtrees: left and right. If either left or right subtree is of size between 1/3 and 2/3 of the size of the whole tree, then we are done. Otherwise, pick the subtree that is bigger than 2/3 of the size of the original tree, and continue with that subtree. Sooner or later, we will come across a subtree whose size is in the required range, since after each step our subtree looses at least one leaf.) Let $\hat{F}(x_1, \ldots, x_n, z)$ be the formula F with the subformula F' replaced by a new variable z. Then

$$F(x_1, \ldots, x_n) = (\hat{F}(x_1, \ldots, x_n, 1) \wedge F'(x_1, \ldots, x_n)) \vee (\hat{F}(x_1, \ldots, x_n, 0) \wedge \neg F'(x_1, \ldots, x_n)).$$

Now, we recursively re-balance the formulas $\hat{F}(x_1, \ldots, x_n, 1)$ and $F'(x_1, \ldots, x_n)$. Then we plug the resulting balanced formulas into the right-hand side of the expression for F given above.

Each recursive call adds at most 3 to the depth of the formula. On the other hand, since after each recursive call the size of the formula gets shrunk by a factor 2/3, there can be at most $\log_{3/2} |F|$ nested recursive calls (i.e., the depth of the recursion is at most $O(\log n)$). Thus, in total, the depth of the formula obtained at the end of this recursive re-balancing will be $O(\log n)$.

3 Examples of problems in NC_1

Boolean matrix multiplication

Given two $n \times n$ Boolean-valued matrices A, B, the goal is to compute their product $C = AB$. Note that $C[i, j] = \vee_{k=1}^{n} A[i, k] \wedge B[k, j]$. For each triple i, k, j, we can compute the AND of $A[i, k]$ and $B[k, j]$ in depth 1. Then, for each pair i, j, we can construct a binary tree of depth $\log n$ that computes the OR of the n terms $A[i, k] \wedge B[k, j]$. Thus, each entry of the matrix C can be computed in $O(\log n)$ depth.

4 Constant-depth circuits: AC^0

We also consider Boolean circuits of constant depth. If the fan-in remains at most 2, such circuits compute functions that do not depend on all of its inputs. So in NC^0 we can only compute constant functions.

To make things more interesting, we allow the fan-in to be unbounded. The resulting class of polysize circuits of constant depth (and unbounded fan-in) is called AC^0. This class is especially important since it corresponds to first-order logic in the model checking setting.

Theorem 3. *$FO(+,*)$ captures AC^0, where $FO(+,*)$ is the first-order logic with built-in predicates for addition and multiplication.*

Proof sketch. The intuitive idea behind the proof is that for a first-order formula of constant length, we can represent $\forall$ and $\exists$ as unbounded gates with as many inputs as there are elements in the universe of our structure. So for every structure size there will be a different circuit corresponding to the given formula on that structure. For the other direction, we do need $+$ and $*$ to talk about the order of inputs and gates; without $+$ and $*$ first-order logic is strictly weaker than AC^0. □

AC^0 is a relatively weak class. For example, the Parity of n-bit strings cannot be computed in AC^0. (However, the proof of this result is rather involved, and is one of the few successes of complexity theory in proving some kind of circuit lower bounds.) On the other hand, adding two n-bit numbers $a = a_1 \dots a_n$ and $b = b_1 \dots b_n$ can be done in AC^0.

The idea is to compute for each bit position i, whether there is carry into that position. This computation can be done independently (in parallel) for each bit position i. It is easy to see that the carry into position i is 1 iff there exists an index $j > i$ that generated a carry (i.e., $a_j = b_j = 1$) and that carry was propagated all the way to i (i.e., for each $i < k < j$, we have $a_k = 1$ or $b_k = 1$). It is now easy to construct a constant-depth (unbounded fan-in) circuit computing the carry c_i for each position i. Then using this carry computing circuit, we can easily compute each bit in the sum $a + b$ in AC^0.