Lower Bounds **(CS294-152, UC Berkeley, Fall'18)**

The Hardest Functions and How to Find Them

Notes for 9/17/18

Ryan Williams

Announcements:

Turn in your report TODAY for last week's workshop! So far I have only four reports...

If you'd like office hours, please send me email. We can arrange a time and place in Soda.

1 Preliminaries

One might imagine, from our earlier lamentations about our inability to prove circuit lower bounds, that it is hopeless to find functions that require very (exponentially) large circuits. But this is not true. We can *find* such functions, but they appear to require lots of resources (time/space) to discover. The question of whether there are exponential-time computable functions that require large circuits is deeply connected to other major questions in complexity theory, in particular to the theory of derandomization.

1.1 Circuits over a Basis

First, let's look at the definition of a circuit. Let $\mathcal{B}$ be a basis set of functions that our circuits compute with.

Definition Boolean circuit C with n inputs and size s over a basis $\mathcal{B}$ (a.k.a. a *$\mathcal{B}$-circuit*):

– a DAG of $s+n$ nodes or *gates*, numbered $1,\ldots,s+n$, with n sources and one sink (the sth node).

– for $i=1,\ldots,n$, the ith gate is labeled by x_i, the ith input bit. (These nodes are often called *input gates*.)

– for $j=n+1,\ldots,n+s$, the jth gate is labeled with a function f_j from $\mathcal{B}$, where fan-in(f_j) = indegree(j).

The computation of a circuit C on an input x is denoted by $C(x)$. (I won't define this, it should be easy to define for yourself!)

Define $F_n := \{f : \{0,1\}^n \to \{0,1\}\}$, i.e. the set of all Boolean functions with n inputs.

C computes $f \in F_n$ if for all x, $f(x) = C(x)$.

> Circuit complexity of f over $\mathcal{B}$:
>
> $C_{\mathcal{B}}(f)$:= minimum size of a $\mathcal{B}$-circuit computing f.

1.1.1 What $\mathcal{B}$'s are interesting?

For circuits with bounded fan-in, there are two bases which are widely used. (For circuits with unbounded fan-in, there are many more.)

1. B_2 = the set of all $2^{2^2} = 16$ Boolean functions on two bits.

2. U_2 = informally speaking, uses only AND, OR, and NOT gates, where NOT gates are "free" (not counted towards size).

More formally, for a variable x, define $x^1 = x$ and $x^0 = \neg x$. Define

$$U_2 = \{f(x,y) = (x^{b_1} \wedge y^{b_2})^{b_3} \mid b_1, b_2, b_3 \in \{0,1\}\}.$$

Note U_2 has 8 different functions. If we hardwire constants into the circuit, we can get 14 functions with all of the possible projections onto 1 and 0 variables.

(Note, U_2 stands for "unate functions": functions which, for each variable, are either monotone increasing in that variable or non-monotone decreasing in that variable.)

Aside. If we have constants $0 - 1$ hardwired into the circuit, we can get more. By plugging $0 - 1$ constants into functions from U_2, we also obtain

$$f(x,y) = x, f(x,y) \quad = \neg x, f(x,y) = y, f(x,y) \quad = \neg y, f(x,y) = 0, f(x,y) \quad = 1.$$

That's six more functions, so we're now up to 14. The only two functions we're missing from B_2 is

$$XOR(x,y) = (x \wedge \neg y) \vee (\neg x \wedge y) = x \oplus y$$

and its complement, $EQUALS(x,y) = \neg XOR(x,y)$.
End Aside.

It is easy to see their circuit complexities are related to within constant factors:

Proposition 1. *For all f, $C_{B_2}(f) \leq C_{U_2}(f) \leq 3C_{B_2}(f)$.*

Proof. To get a U_2-circuit from a B_2-circuit, we only have to replace the XOR and NOT(XOR) gates with U_2 functions.

$$EQUALS(x, y) = (x \wedge y) \vee (\neg x \wedge \neg y),$$

Hence we can simulate EQUALS with 3 gates from U_2. Similarly we can get XOR:

$$XOR(x, y) = (\neg x \wedge y) \vee (x \wedge \neg y),$$

□

In fact, U_2 circuit complexity is essentially the same as B_2 circuit complexity minus XOR and negation of XOR:

Proposition 2. *For all f, $C_{U_2}(f) = C_{B_2-\{XOR,\neg XOR\}}(f) \pm 2$.*

(the ± 2 is for the constants 0-1, which may or may not be hard-wired)

Proof. $U_2 \subseteq B_2 - \{XOR, EQUALS\}$, so

$$C_{U_2}(f) \geq C_{B_2-\{XOR,EQUALS\}}(f).$$

Every function from $B_2 - \{XOR, EQUALS\}$ can be simulated with a function from U_2 (using built-in constants), so

$$C_{U_2}(f) \leq C_{B_2-\{XOR,EQUALS\}}(f) :$$

given a minimum size circuit over $B_2 - \{XOR, EQUALS\}$ with 0-1 constants, we can get a circuit of the same size over U_2. □

So one should only care about the distinction between U_2 or B_2 if you care about constant leading factors, since the two circuit complexities within (small) constant factors of each other. This is actually important for the state-of-the-art, since the best known U_2 circuit lower bound is about $5n$ for an explicit function, whereas for B_2 it is about $3.01n$.

1.1.2 More General Result

A basis $\mathcal{B}$ is *complete* if for all n and $f \in F_n$, there exists a $\mathcal{B}$-circuit C computing f.

Ex: $\mathcal{B} = \{AND\}$ is not a basis; $\mathcal{B} = \{NOR\}$ is a basis. (Exercise)

B_2 and U_2 are complete: any function f can be expressed in disjunctive normal form (DNF), huge OR of $O(2^n)$ ANDs, where each AND outputs 1 on exactly one input of f.

So writing f as a "tree" of up to 2^n ORs, where each leaf has n ANDs of variables and their negations, we can get a circuit for f over U_2 with size $O(n2^n)$.

Theorem 1.1. *Let $\mathcal{B}$ be a* complete *basis with constant fan-in for each function in the basis. Then for all n and $f \in F_n$, $C_{\mathcal{B}}(f) = \Theta(C_{U_2}(f)) = \Theta(C_{B_2}(f))$.*

1.2 Circuit Complexity

Let's start with a basic question: how large can circuit complexity get?

Question: **How large can $C_{B_2}(f)$ get, for $f \in F_n$?**

This should be some function of n, the number of inputs to f. We restrict attention to B_2 because we know it doesn't change much by a constant factor.

1.3 A Lower Bound

Theorem 1.2 (Shannon, 1949; Lupanov 1963)**.** *There is a universal $c \geq 1$ such that for all sufficiently large n, there is an $f_{hard} \in F_n$ such that $C_{B_2}(f_{hard}) \geq 2^n/(cn)$.*

While this lower bound is for B_2, it can easily be adapted to any other complete basis (to within constant factors).

Proof. Counting argument. What's the number of Boolean circuits of size S?

Claim: The total number of functions computed by $\mathcal{B}$-circuits of size S is at most

$$2^{O(S \log(n+S))}.$$

We can prove the claim by an encoding argument. A B_2-circuit can be specified by giving:

- a DAG on $n + S$ nodes where the first n nodes have indegree 0, and the other S nodes have indegree 2, and
- assigning some $g \in \mathcal{B}$ to each node of the DAG.

Each circuit of size S can be uniquely encoded with $T = cS\log(n+S)$ bits of information, for some constant $c \geq 1$:

- For each gate $i = n+1, \ldots, n+S$, we choose 2 gates that come before it as the inputs to that gate. It takes $O(\log(n+S))$ bits to encode each choice.
- Assigning a $g \in \mathcal{B}$ to each gate of the DAG takes $O(S)$ bits, because there are $16 = O(1)$ functions in $\mathcal{B}$.

It follows that the total number of circuits of size S is at most 2^T.

Finally, each such circuit computes one Boolean function, so the total number of functions with size-S circuits is at most 2^T as well. This proves the above claim.

Now the number of Boolean functions on n inputs is 2^{2^n}. When

$$2^{2^n} > 2^T = 2^{cS\log(n+S)},$$

the number of all Boolean functions exceeds the number of Boolean functions with size-S circuits; in that case, some function won't have size-S circuits. For all $S \ll 2^n/(cn)$, we have

$$cS\log(n+S) \ll (2^n/n)\cdot\log(n+2^n/(cn)) < 2^n$$

for large n. Thus there are functions without $2^n/(cn)$ size circuits. □

OK, some functions have exponential-in-n circuit complexity. Is this a rare occurrence? How many functions have high circuit complexity?

1.4 Random Functions Meet the Lower Bound

Theorem 1.3. *There is a constant $K > 1$ such that for large n,*

$$\Pr_{f\in F_n}\left[C_{B_2}(f) \geq \frac{2^n}{Kn}\right] \geq 1 - 1/2^n.$$

Proof. Set $S := 2^n/(2cn)$. Looking back at the previous proof, we have

$$\text{(number of functions with size-}S\text{ circuits)} \leq 2^{cS\log(n+S)} \leq 2^{2^n/2}$$

for large n. Therefore

$$\Pr_{f\in F_n}\left[C_{B_2}(f) < \frac{2^n}{Kn}\right] \leq 2^{2^n/2}/2^{2^n}.$$

□

1.5 Generic Upper Bounds

This lower bound is rather tight: for every n-bit Boolean function, no matter how hard it looks, we can always find a circuit of size roughly $O(2^n/n)$. Let's start with an $O(2^n)$ bound:

Theorem 1.4 (Folklore)**.** *Every $f \in F_n$ has a* **formula** *of size at most $O(2^n)$.*

(Recall a formula is a special case of a circuit, where the outdegree of each non-input gate is at most one.)

Proof. When $n = 1$, we can just use one gate. For $n > 1$, define the $(n-1)$-variable functions

$$f_0 := f(0, x_2, \ldots, x_n), f_1 := f(1, x_2, \ldots, x_n).$$

Then we can write f as:

$$f(x_1, ..., x_n) = (x_1 \wedge f_1(x_2, ..., x_n)) \vee ((\neg x_1) \wedge f_0(x_2, ..., x_n)).$$

Draw: Three gates, then two recursive calls to f_0 and f_1 on $n-1$ variables.

Recursively constructing a formula for f_0 and f_1, we can get a formula for f using 3 gates (OR and two ANDs). Then, the size of a formula for n variables satisfies the recurrence relations

$$S(n) \leq 2S(n-1) + 3, S(1) = 1.$$

Solving this recurrence, we obtain $S(n) \leq 4 \cdot 2^n$.

(Consider a complete binary tree of 2^n leaves. It has $2^n - 1$ interior nodes. Put a "cost" of 3 on each interior node, and cost of 1 on each leaf. Have $3(2^n - 1) + 2^n = 4 \cdot 2^n - 3$.) □

Having become sufficiently warmed up, we now show:

Theorem 1.5 (Lupanov / Shannon, 1949)**.** *For all $f \in F_n$, $C(f_n) \leq O(2^n/n)$.*

That is, every Boolean function on n bits has a circuit of size at most $O(2^n/n)$.

Intuition: Recall a circuit of size S where $S \geq n$ can be represented in $\Theta(S \log S)$ bits. So an $O(2^n/n)$-size circuit still takes $\Omega(2^n)$ bits to write down, so information-theoretically it is possible that we might represent a 2^n-bit string (i.e. $f : \{0,1\}^n \to \{0,1\}$) with it.

The idea is that a 2^n-bit string can be "compressed" down to a $O(2^n/n)$ size circuit, where the "wire connections" between gates are somehow encoding the 2^n-bit string.

The following "demultiplexer" lemma will be helpful:

Lemma 1.1. *Let $a_1, \ldots, a_{2^r}$ be an ordering of the r-bit strings. There is a size-$O(2^r)$ circuit D_r with r inputs and 2^r output gates $G_1, \ldots, G_{2^r}$, such that for all i, $G_i(x) = 1$ iff $x = a_i$.*

Proof. Divide variable $\{x_1, \ldots, x_r\}$ into two halves, with at most $r/2$ variables in each half. Compute all $O(2^{r/2})$ conjunctions on the first half of variables: i.e. expressions of the form

$$x_1^{b_1} \wedge \cdots \wedge x_{r/2}^{b_{r/2}},$$

where $b_i \in \{0, 1\}$, and $x^0 = \neg x$, $x^1 = x$. This takes $O(r2^{r/2})$ gates (r AND gates for each conjunction). Compute all $O(2^{r/2})$ conjunctions on the second half of variables, analogously. This takes $O(r2^{r/2})$ gates in total. We now have all possible conjunctions on the first half of variables, and on the second half.

For all 2^r conjunctions on r variables, each conjunction is the AND of a conjunction from the first half and one from the second half. Take these 2^r ANDs, and make each AND an output gate G_i. The overall size is $2^r + O(r2^{r/2}) \leq O(2^r)$. □

Proof of Theorem 1.5. Let $k \ll n$ be a parameter, and $f \in F_n$. Again the idea is to split the set of inputs into two parts. But now the first half has $n - k$ inputs and the second has k, where k will be very small (like $\Theta(\log n)$).

First, compute all k-bit Boolean functions simultaneously in $O(2^k 2^{2^k})$ gates: For *every* Boolean function $g \in F_k$, write down an $O(2^k)$-size formula. Since $|F_k| = 2^{2^k}$, this takes no more than $O(2^k 2^{2^k})$ gates.

Let C_1 be a size-$O(2^k 2^{2^k})$ circuit implementing all functions on k inputs, with k inputs and 2^{2^k} outputs.

Now we'll make another circuit C_2 which will help with the other $n-k$ inputs of f's inputs. For each of the $2^n/2^k$ assignments A to the first $n - k$ inputs, there is a function g_A on k bits that is induced. The idea is to use circuitry to efficiently "look up" this function g_A, and save some gates.

Set $C_2 := D_r$ from the previous lemma, with $r = n - k$. This C_2 has $n - k$ inputs and $t = 2^{n-k}$ outputs $G_1, \ldots, G_t$, where each output gate G_i is associated with a unique $n - k$ variable assignment A_i.

Now we connect the two circuits together. Fix an $f \in F_n$ to be simulated.

For every $a_i \in \{0, 1\}^{n-k}$, define the k-input Boolean function

$$f_{a_i}(y) := f(a_i, y).$$

Note that C_1 has already computed f_{a_i}, because it computed all $g \in F_k$.

So for every G_i in C_2 (corresponding to a_i), take the AND of G_i (from C_2) and f_{a_i} (from C_1). This introduces only 2^{n-k} more gates. Note that at most one of these gates can output 1. Take the OR of these gates. The overall circuit feeds its first $n-k$ inputs to C_2, and the last k of its inputs to C_1.

(Note: these ANDs with one input from C_1 and one from C_2 are the only part that depends on f; they are "encoding" f.)

Now we want to set k to minimize the overall circuit size.

- C_2 has size $O(2^{n-k})$,
- C_1 has size $O(2^k \cdot 2^{2^k})$,
- our AND-ing and OR-ing takes $O(2^{n-k})$ more gates.

Total size is $O(2^{n-k} + 2^k \cdot 2^{2^k})$. Set $k := \log(n/2)$. Then the size is

$$O(2^{n-\log(n/2)} + n/2 \cdot 2^{2^{\log(n/2)}}) \leq O(2^n/n).$$

□

Clearly the above can be optimized a bit more; it could even be applied recursively to k.

1.6 Improvements

The above circuit complexity upper and lower bounds can be sharpened very tightly.

Definition 1. *Let* $H(n) := \max_{f \in F_n} C_{B_2}(f)$.

Here, the H stands for "hard". $H(n)$ is the maximum circuit complexity achievable by a function $f \in F_n$. We have shown:

$$\Omega(2^n/n) \leq H(n) \leq O(2^n/n).$$

It turns out you can make the leading constants $(1 - o(1))$ and $(1 + o(1))$, in the lower and upper bounds, respectively.

Theorem 1.6 (Frandsen, Miltersen 2005)**.**

$$\frac{2^n}{n}(1 + (\log n)/n) - O(1/n)) \leq H(n) \leq \frac{2^n}{n}(1 + 3(\log n)/n + O(1/n)).$$

2 Constructing Hardest-Possible Functions

We have seen that a randomly chosen $f \in F_n$ needs at least $\Omega(2^n/n)$ size circuits whp, and every $f \in F_n$ has circuits of size at most $O(2^n/n)$. That is, "hard" functions are plentiful – most functions are hard! Two questions come to mind:

- Can we construct specific functions requiring large circuits?
- How efficiently can an algorithm *produce* a hard function?

Note: It is easy to produce a hard function using randomness... pick one at random... but then there is the question of how do we certify the function we picked is actually one of the hard ones? **How can we tell for sure that our output is indeed hard?** The algorithmic task of deciding if a truth table has high circuit complexity is a very interesting one (we will discuss it later).

We have to be careful how we formalize these questions about the complexity of hard functions. Remember that any $f \in F_n$ is a *finite* function. But in the usual complexity-theoretic setting, we study *languages* or decision problems, which (in the interesting case) are defined over *infinite* sets of strings.

(I will often use the words "problem" and "language" to describe the same thing: a function from *all* binary strings to $\{0,1\}$. If we get in a situation where this terminology makes stuff unclear, please let me know.)

When we talk about standard complexity classes, like $\mathsf{TIME}[2^n]$ (exponential time), we are talking about the class of languages / decision problems recognized by a single, uniform, $O(2^n)$ time algorithm.

To bring the two models together, we can talk about infinite families of Boolean functions: for every input length n, there is a Boolean function f_n on n-bits that we wish to simulate.

Every infinite language L on binary strings can be construed as an infinite family of finite problems $\{L_n\}$, where L_n is a Boolean function over inputs of length n.

2.1 Constructing Hard Functions With Algorithms

Let's define what it means for a decision problem to have small circuits:

Let $f : \{0,1\}^\star \to \{0,1\}$. Define $f_n : \{0,1\}^n \to \{0,1\}$ to be the function that agrees with f on all n-bit inputs. Let $s : \mathbb{N} \to \mathbb{N}$.

Definition 2. *We say that f* **has $s(n)$-size circuits** *if for all n, $C(f_n) \leq s(n)$.*

Let $\mathsf{SIZE}(s(n))$ be the class of f with $s(n)$-size circuits.

From our earlier results, we have:

Corollary 2.1. *For* **every** f, $f \in \mathsf{SIZE}(O(2^n/n))$.
There **exists** *a* $c \geq 1$ *and* f' *such that* $f \notin \mathsf{SIZE}(2^n/(cn))$.

How can we "construct" a function that needs $\Omega(2^n/n)$ size circuits, in the usual time/space complexity sense? Let us start with a "weak" algorithmic upper bound on constructing hard functions.

Theorem 2.1. *There is an* $f \in \mathsf{TIME}[2^{O(2^n)}]$ *such that for all* n, $C(f_n) = H(n)$.

Another way of saying this in complexity notation: $f \notin$ *i.o.*-$\mathsf{SIZE}(H(n))$.

Proof. Idea is simple: *Brute-force it!* Here is an algorithm for computing a hardest-possible function.

Given an input x of length n:
Let $C^\star(x_1, \ldots, x_n) := x_1 \wedge \neg x_1$; let $s^\star := size(C^\star)$.
For all $f \in F_n$, *[iterate over all* 2^n*-bit strings]*
  Try all circuits C of size 1, size 2, etc., until find first C_f such that $C_f \equiv f$.
  [Evaluate each C *on all* 2^n *inputs, check agreement with* f*]*
  If $size(C_f) > s^\star$, update $s^\star := size(C_f)$ and $C^\star := C_f$.
Output $C^\star(x)$.

After all loops have completed, we have a circuit $C^\star$ with $size(C^\star) = H(n)$, and the function implemented by $C^\star$ has no smaller circuit.

Note: Everything before the last line does not depend on x at all, only on $n = |x|$. So for every input of length n, we obtain the **same** $C^\star$.

Hence the above algorithm computes exactly $C^\star$ on inputs of length n. □

In fact, the above algorithm needs only $O(2^n)$ space: we need

- 2^n bits to store the current function f,
- $O(2^n)$ bits to store current $C^\star$ (because it's at most $O(2^n/n)$ size),
- $O(2^n)$ bits to store the current C,
- $O(n)$ to store s.

So we in fact obtain:

Theorem 2.2 (Meyer '73, Sholomov '75)**.** *There is an* $f \in \mathsf{SPACE}[2^n]$ *such that* $C(f_n) = H(n)$ *for every* n.

3 Time lower bounds from circuit upper bounds

From the above, we can obtain an intriguing "circuit upper bounds implies time lower bounds" connection. Recall $\mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}[n^k]$.

Theorem 3.1 (Folklore)**.** *If* $\mathsf{E} = \mathsf{TIME}[2^{O(n)}] \subset \mathsf{SIZE}(H(n) - 1)$*, then* $\mathsf{P} \neq \mathsf{PSPACE}$.

Proof. Contrapositive. If $\mathsf{P} = \mathsf{PSPACE}$ then $\mathsf{TIME}[2^{O(n)}] = \mathsf{SPACE}[2^{O(n)}]$. (Padding argument.) But the above says that there is an $f \in \mathsf{SPACE}[O(2^n)]$ without $(H(n)-1)$-size circuits. So $f \in \mathsf{TIME}[2^{O(n)}]$ as well. □

So if we could just "shave off one gate" from the circuit complexity of every exp-time problem, we would separate P from PSPACE! We will strengthen the above theorem soon.

3.1 Sidebar: Padding Arguments in Complexity

Proof of $\mathsf{P} = \mathsf{PSPACE} \Rightarrow \mathsf{SPACE}[2^{O(n)}] \subseteq \mathsf{TIME}[2^{O(n)}]$:

Assume $\mathsf{P} = \mathsf{PSPACE}$. Let $f \in \mathsf{SPACE}[2^{kn}]$, $f : \{0,1\}^\star \to \{0,1\}$. We'll give a $2^{O(n)}$-time algorithm for f. Define

$$f_{pad}(y) := \begin{cases} 1 & \text{if } y = x01^{2^{k|x|}} \text{ and } f(x) = 1 \\ 0 & \text{otherwise} \end{cases}$$

So f outputs 0, unless y has a huge number of ones on the end; in that case, f is computed on a tiny substring of y.

Claim: $f_{pad} \in \mathsf{SPACE}[n]$. Here's an $O(n)$-space algorithm:

> Given input y, if it does not have the form $y = x01^{2^{k|x|}}$, then *reject*.
> Run an $O(2^{k|x|})$ space algorithm for L on x and output the answer.

The key point is that we only run the $O(2^{k|x|})$ space algorithm when we already have $|y| \geq 2^{k|x|}$. So this algorithm runs in space LINEAR in $|y|$.

Now, by assumption, $f_{pad} \in \mathsf{P}$. Let A be a polytime algorithm for f_{pad}. The following algorithm computes f:

> Given x, compute $2^{k|x|}$, construct $y = x01^{2^{k|x|}}$, run $A(y)$ and output the answer.

The running time is polynomial in $|y|$, i.e. polynomial in $2^{k|x|}$. This is $2^{O(n)}$ time.

Note there are lots of other such lemmas: $\mathsf{P} = \mathsf{NP} \implies \mathsf{EXP} = \mathsf{NEXP}$, etc.

3.1.1 Why are these implications true? Why don't they point in the opposite direction?

My intuition: As the resource bounds get more relaxed, it becomes only more likely that the time class equals the space class. For example, it looks even more likely that $\mathsf{TIME}[2^{2^{O(n)}}] = \mathsf{SPACE}[2^{2^{O(n)}}]$.

Why is this? My intuition is that, in the "limit" where there is only a "finite" time limit as a function of the input and a "finite" space limit as a function of the input, we end up with the *same* class of problems: the deciable languages, the functions computable by algorithms in some finite time (or some finite space).

So as we relax the resource bounds and let them veer off into astronomical numbers, the difference between time and space matters less and less. We get closer to computability theory, and further away from complexity as we know it. (This is only intuition, though!)

Indeed, let $T(k, n)$ denote the function from $\mathbb{N}$ to $\mathbb{N}$ which is a tower of k n's. So $T(1, n) = n$, $T(2, n) = n^n$, and so on. Since $\mathsf{SPACE}[s] \subseteq \mathsf{TIME}[2^{O(s)}]$, we have $\mathsf{SPACE}[T(k, n)] \subseteq \mathsf{TIME}[T(k + 1, n)]$, and thus

$$\bigcup_k \mathsf{SPACE}[T(k, n)] = \bigcup_k \mathsf{TIME}[T(k, n)].$$

3.2 Reducing the Complexity of Finding Hard Function

We can reduce the complexity of the "hardest possible function" even further:

Theorem 3.2. *There is an* $L \in \mathsf{TIME}[2^{O(n)}]^{\Sigma_2 P} = \mathsf{E}^{\mathsf{NP}^{\mathsf{NP}}}$ *such that* $C(L_n) = H(n)$ *for every* n*.*

This complexity class $\mathsf{E}^{\Sigma_2 P}$ is "$2^{O(n)}$ Time with a $\Sigma_2\mathsf{P}$ oracle". Let's explain what that means. Recall $\Sigma_2\mathsf{P} = \mathsf{NP}^{\mathsf{NP}}$ (it came up in the lecture on SAT time-space tradeoffs):

> The class of all $f : \{0,1\}^\star \to \{0,1\}$ s.t. there's a polytime relation R and $k > 0$ such that
>
> $$f(x) = 1 \iff (\exists\, y : |y| \leq |x|^k)(\forall\, z : |z| \leq |x|^k)[R(x, y, z) \text{ is true}].$$

A canonical example problem in $\Sigma_2\mathsf{P}$:

Problem MIN-CIRCUIT:
Input: A Boolean circuit C (over B_2)
Decide: Is there a circuit smaller than C computing the same function as C?

Suppose our circuit C has n inputs and size s, encoded in $O(s \log(n+s))$ bits. How do we solve this problem in $\Sigma_2\mathsf{P}$?

Here is a $\Sigma_2\mathsf{P}$ "algorithm" for MIN-CIRCUIT:

Existentially guess a circuit C' s.t. $size(C') < s$.
Universally check for all n-bit inputs z that $C'(z) = C(z)$.

This takes $O(\text{poly}(s))$ time on a "$\Sigma_2\mathsf{P}$ machine."

To tie into the above definition of $\Sigma_2\mathsf{P}$, the polytime computable relation $R(C, C', z)$ would be:

(if ($size(C') \geq s$ or $C(z) \neq C'(z)$) then false, else true).

$\mathsf{E}^{\Sigma_2\mathsf{P}}$ = problems decidable by 2^{kn}-time algorithms which can make 2^{kn}-long queries to a $\Sigma_2\mathsf{P}$ problem, and get yes/no answers in 1 step.

This class $\mathsf{E}^{\Sigma_2\mathsf{P}}$ looks *way* more powerful than "just" exponential time: can find exponentially long solutions to problems.

So how do we prove the above theorem? Need to construct a function computable in $\mathsf{E}^{\Sigma_2\mathsf{P}}$ with maximum circuit complexity.

First, given the input length n, we need to compute what $H(n)$ is.

For two functions $f, g \in F_n$, we say that $g \leq f$ if, as strings, g is at most f in lexicographical order. (More formally, $tt(g) \leq_{lex} tt(f)$, where $\leq_{lex}$ is lex order on strings.)

Problem COMPLEX:
Input: (f, k) where f is a 2^n-bit string, $k \in \{1, \ldots, 2^n\}$
Decide: Is there a function $g \leq f$ that has circuit complexity at least k?

COMPLEX is in $\Sigma_2 P$, here's a Σ_2 algorithm:

Existentially guess a function $g \leq f$ (2^n bits),
Universally for all circuits C taking n inputs of size less than k, ($O(k \log k)$ bits)
check [$g \leq f$ and there is an x such that $C(x) \neq g(x)$].

For those of you who have heard of MCSP, this problem is in fact in NP^{MCSP}. (For those of you who haven't, don't worry about it!)

Now, here is an $\mathsf{E}^{\texttt{COMPLEX}}$ algorithm for computing the "lex first" function of maximum circuit complexity. We use the oracle in two ways.

Given an input x of length n,

• **Determine $H(n)$.**
Ask: *is* $(1^{2^n}, k) \in \texttt{COMPLEX}$*?*
on $k = 1, 2, \ldots$, until we get a "no" answer.

Then $H(n) = k - 1$. This takes $O(2^n)$ queries to COMPLEX.

• **Construct f with circuit complexity $H(n)$.**
Initialize $f = 1^{2^n}$, and $\ell = 1$.
While $\ell \leq 2^n$,
- Set ℓth bit of f to 0.
- If $(f, H(n)) \notin \texttt{COMPLEX}$ then set ℓth bit back to 1.
- Increment ℓ.

[Now: f is the lex first function with $C(f) = H(n)$.]
Output $f(x)$.

Note we just need $O(2^n)$ queries to COMPLEX oracle to determine f. So $f \in \mathsf{TIME}[O(2^n)]^{\texttt{COMPLEX}}$.

3.3 More time lower bounds from circuit upper bounds

We can use the above upper bound to get a stronger lower bound consequence from circuit upper bounds:

Theorem 3.3 (Folklore)**.** *If every $f \in \mathsf{E}$ has circuits of size $\leq H(n) - 1$, then $\mathsf{P} \neq \mathsf{NP}$.*

Proof. We prove the contrapositive. Suppose $\mathsf{P} = \mathsf{NP}$. Then $\Sigma_2\mathsf{P} = \mathsf{P}$. (This is in Arora-Barak if you haven't seen it.) This implies COMPLEX is in P. Therefore, in the previous theorem, all calls to COMPLEX can be simulated in time $2^{O(n)}$. So there is a function of circuit complexity $H(n)$ that's computable in E. □

In other words: $\texttt{COMPLEX} \in \mathsf{P} \Rightarrow \mathsf{EXP} \notin \mathsf{SIZE}(H(n) - 1)$.

3.4 Open Problems

How difficult is it to compute functions that are hard for circuit complexity, in terms of time/space complexity?

OPEN: Is there a function in $\Sigma_2\mathsf{EXP} = \mathsf{NEXP}^{\mathsf{NP}}$ such that $f \notin SIZE(H(n)-1)$?

There don't seem to be any crazy consequences of proving a yes-answer here, but it is a longstanding open problem that people have thought about. But a **no-answer** would imply $\mathsf{P} \neq \mathsf{NP}$:

Proposition 3. *If all functions in* $\mathsf{NEXP}^{\mathsf{NP}}$ *have* $(H(n)-1)$*-size circuits then* $\mathsf{P} \neq \mathsf{NP}$.

The proof is analogous to the previous theorem.

Note that if there is a function in $\mathsf{NTIME}[2^{O(n)}]$ that requires $H(n)$-size circuits, then we would already be able to prove some new derandomization results. In particular it would imply (approximately) that $MA = NP$.

MA = Merlin-Arthur Games, i.e. "NP with Randomness in the verifier"

The following are special cases of theorems in the literature:

NP/$f(n)$: NP machines with $f(n)$ non-uniform bits. Problems accepted by a list of nondeterministic polynomial time algorithms $\{A_n\}$ which have program size at most $O(f(n))$.

Theorem 3.4 (Impagliazzo-Kabanets-Wigderson '01)**.** *If there is a function in* $\mathsf{NTIME}[2^{O(n)}]$ *that requires* $H(n)$*-size circuits, then* $\mathsf{MA} \subset \mathsf{NP}/n^{\varepsilon}$ *for all* $\varepsilon > 0$.

Theorem 3.5 (Impagliazzo-Wigderson '97)**.** *If there is a function in* $\mathsf{TIME}[2^{O(n)}]$ *that requires* $H(n)$*-size circuits, then* $\mathsf{BPP} = \mathsf{P}$.

4 Intermediate Circuit Complexities of Functions

So we have shown that the "hardest" that a function can get is $H(n) = \Theta(2^n/n)$. Are there functions with intermediate circuit complexities?

Recall that in the usual time/space complexity, we have hierarchy theorems stating that as we are allotted more time to solve problems, there are strictly more problems that we can solve. For example:

Theorem 4.1 (Time Hierarchy)**.** *For all "nice"* $t(n) \geq n$, $\mathsf{TIME}[t] \subsetneq \mathsf{TIME}[t^2]$.

Here, "nice" means "time-constructible": there is an algorithm A, that given 1^n as input, outputs the number $t(n)$ and runs in less than $t(n)$ steps.

Can we compute strictly more functions, given bigger circuits to solve them?

Theorem 4.2 (Circuit Size Hierarchy)**.** *There is a universal constant c such that for all functions $s : \mathbb{N} \to \mathbb{N}$ satisfying $n \leq s(n) \leq o(2^n/n)$ for all n, $\mathsf{SIZE}[s] \subsetneq \mathsf{SIZE}[c \cdot s]$.*

Proof. NOT by universal simulation! It does not work like the time hierarchy theorem.

For a given n, we will design a function with $s < C(f) \leq c \cdot s$. Pick $n' < n$ such that $s(n) \leq 2^{n'}/n' \leq 3s(n)$. (Convince yourself that such an n' exists.)

By the tight circuit lower bound, there's a function $f_{n'}$ on n' bits such that $C(f_{n'}) \geq 3/42^{n'}/(n') > s(n)$. The upper bound gives us $C(f_{n'}) \leq d2^{n'}/n' \leq 3 * cs(n)$. To get an n-bit function, we can just add $n - n'$ "dummy variables" to the function that don't affect the output of the function. □