Topics in Circuit Complexity **(CS354, Spring'16)**

Week 2: Really Hard Functions

Notes for 4/7/2016

Ryan Williams

Office hours: By email.

1 Really Hard Functions

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

Let's start with a basic question. Let $\mathcal{B}_n$ be the set of functions $f : \{0,1\}^n \to \{0,1\}$. Let B be our basis set of functions. Typically we have $\mathcal{B} = B_2$.

Recall: $C_B(f)$ = min size of any circuit computing f.

How large can $C_B(f)$ get, for $f \in B_n$?

This should be some function of n, the number of inputs to f.

1.1 A Lower Bound

Theorem 1.1 (Shannon, 1949; Lupanov 1963)**.** *Suppose $|\mathcal{B} = c$, and each $g \in \mathcal{B}$ has fan-in at most f. There is a universal constant $d > 0$ such that for every n, there is an $F : \{0,1\}^n -> \{0,1\}$ such that $C_{\mathcal{B}}(F) \geq 2^n/(dnf \log c)$.*

Proof. Counting argument. For simplicity, we'll think of the fan-in as 2. What's the number of Boolean circuits of size S?

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

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

Why? Because we can specify a circuit by giving:

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

Each such DAG can be represented with $\leq O(S \cdot flog(n + S))$ bits of information.

For each gate $i = n+1, \ldots, n+S$, we pick f gates that come before it as the inputs to that gate.

It takes about $O(f \log(n + S))$ bits (omitting ceilings) to encode this choice.

Assigning some $g \in \mathcal{B}$ to each node takes $O(S \log c)$ bits, because there are c functions in $\mathcal{B}$.

Therefore every circuit of size S can be encoded within

$$T = (S \log c) + fS \log(n + S)$$

bits. The total number of circuits of size S is at most $K = 2^T$, and the claim above is established.

Of course, each circuit only computes one Boolean function, so the total number of functions with size-S circuits is at most 2^T as well.

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

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

the number of all Boolean functions exceeds the number of Boolean functions with size S circuits. For all $S \ll O(2^n/(n \cdot (f + logc)))$, the inequality holds. So there are functions which do not have $O(2^n/(n \cdot (f + logc)))$ size circuits. □

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

1.2 Random Functions Meet the Lower Bound

Theorem 1.2. *There is a constant $K > 1$ such that*

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

Proof. Suppose we set $S = 2^n/(Kn)$. For K sufficiently large, we can make

$$\text{(number of functions with size-}S\text{ circuits)} \leq L = 2^{2^n/2},$$

So if you pick a random 2^n-bit string, that string is among these L special functions with probability only $2^{2^n/2}/2^{2^n} \leq 2^{-2^n/2}$. So with probability at least $1 - 1/2^{-2^n/2}$, a randomly chosen n-bit function requires at least $2^n/(Kn)$ size circuits. □

1.3 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 a simple observation along these lines:

Observation 1. *There is an $O(n2^n)$-size depth-two circuit for any function.*

Proof. Take the truth table of the function, and use it to express the function in *disjunctive normal form*. That is, for every entry e of the truth table which gives a 1-output, we create a conjunction on all n variables which evaluates to 1 if and only if the variable assignment corresponds to entry e. This is an OR of at most 2^n ANDs. The overall size is no more than $O(n2^n)$. □

We can drop the leading n factor with a little more cleverness:

Theorem 1.3 (Folklore)**.** *Every $f : \{0,1\}^n \to \{0,1\}$ has a formula of size at most $O(2^n)$.*

(Recall a formula is 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 $n > 1$, we can write every boolean function f as a formula

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

where f_0 and f_1 depend only on $n-1$ variables $x_2, \ldots, x_n$.

If we think of f as a 2^n-bit vector

$$[f(0^n), f(0^{n-1}1), f(0^{n-2}10), \ldots\ldots, f(10^{n-1}), \ldots, f(1^n),$$

then f_0 is given by the first half of this table, and f_1 is the second half. (These halves are 2^{n-1} bits each, corresponding to $n-1$ variable functions.)

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

Size of the 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 tree of 2^n leaves. Has $2^n - 1$ interior nodes. Put "cost" of 3 on each interior node, cost 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 1.4 (Shannon, 1949)**.** *For all $f \in B_n$, $C(f_n) \leq O(2^n/n)$. That is, every Boolean function on n bits has a circuit of size at most $O(2^n/n)$.*

Intuition: 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; the interesting part is that we can save this factor of n.

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 the gates are 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 circuit D with r inputs and 2^r output gates $G_1, \ldots, G_{2^r}$, such that for all i, $W_i(x) = 1$ iff $x = a_i$. The size of D is only $O(2^r)$.*

Proof. Divide the set of r variables into two halves of at most $r/2$ variables in each half. (For simplicity assume r is divisible by 2.) With $O(r2^{r/2})$ size, compute all $2^{r/2}$ 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 $x^0 = \neg x$, $x^1 = x$. Compute all $2^{r/2}$ conjunctions on the second half of variables, analogously. This takes $O(r2^{r/2})$ gates in total, and these two circuits have $2^{r/2}$ outputs each.

Then for every pair of outputs, one from one half and one from the other half, take their AND, and make that AND one of the output gates G_i. The size of the overall circuit is $2^r + O(r2^{r/2}) \leq O(2^r)$. □

Proof of Theorem 1.4. Let $k \ll n$ be a parameter, and $f \in B_n$. We can compute all k-bit Boolean functions simultaneously in $O(2^k 2^{2^k})$ gates:

For *every* Boolean function $g \in B_k$, write down a $O(2^k)$-size formula. Since $|B_k| = 2^{2^k}$, this takes no more than $O(2^k 2^{2^k})$ gates.

Let C_1 be this circuit, with k inputs and 2^{2^k} outputs.

Now we'll make another circuit C_2 which handles the other $n-k$ inputs of f's inputs. Build a truth table for the first $n-k$ inputs of f. For each of the $2^n/2^k$ assignments, there is a function on k bits that is induced.

Set $C_2 := D$ 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.

For every $f \in B_n$, suppose the first $n - k$ inputs are assigned to a_i. This induces a k-bit Boolean function f_{a_i} on the remaining k inputs. But C_1 has already computed f_{a_i}, because it computed all $g \in B_k$.

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 2^{n-k} more gates; note that at most one of these gates can output 1. Take the OR of these gates.

The final circuit feeds $n - k$ of its input bits to C_2, and k of its input bits to C_1. We want to set k to minimize the overall circuit size.

C_2 has size $O(2^{n-k})$, C_1 has size $O(2^{2^k})$, our AND-ing and OR-ing takes $O(2^{n-k})$ more gates. Total size is $O(2^{n-k} + 2^{2^k})$. Set $k := \log(n/2)$. Then the total size is is

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

□

1.4 Sidebar: Computing Multiple Copies

What if we want to compute many copies of the same function? How hard is this problem?

Here is a very intriguing theorem of Uhlig from the 70's:

Theorem 1.5. *Theorem [Uhlig '74]: Let $f : \{0,1\}^n \to \{0,1\}$. For all $k \leq 2^{o(n/\log n)}$, There is a circuit C_f such that $C_f(x_1, ..., x_k) = (f(x_1), ..., f(x_k))$ for all $x_i \in \{0,1\}^n$, and $size(C_f) \leq O(2^n/n)$.*

That means that, for very hard functions $f \in B_n$ where $C(f) \sim 2^n/n$, evaluating it k different times can be done without much extra overhead. (Naively, one might think that $size(C_f) \geq k \cdot C(f)$! But this isn't true!)

This is a generic problem that comes up in many situations in complexity theory:

> THE DIRECT SUM PROBLEM:
>
> Given: problem Π, and it takes r resources to solve one instance of Π. For k instances of Π, does it take $r \cdot k$ resources to solve these k instances?

The great thing about this problem is that it takes on so many guises! You can ask it for time complexity, space complexity, circuit complexity, ...

Example: Inner product function (mod 2). Requires $\Omega(n)$ size circuits. But can compute n of these with fast matrix multiplication.

For problems with very high circuit complexity, the answer is apparently no!

1.5 Improvements

The above circuit complexity upper and lower bounds can be sharpened very tightly. The leading constant is 1, at least over B_2. In particular, the upper and lower bound is $2^n/n + \Theta(2^n(\log n)/n^2)$.

Definition 1. *Let $H(n)$ be the maximum value of $C_{B_2}(f)$, over all $f \in B_n$.*

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

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

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

Rather technical, so we'll skip it...

2 Constructing Hard Functions

So we have seen that a randomly chosen $f \in B_n$ needs at least $2^n/n$ size circuits whp, and every $f \in B_n$ has circuits of size at most $2^n/n$. That is, "hard" functions are plentiful – most functions are hard!

- Can we construct specific functions that require large circuits?
- What is the *time/space complexity* of constructing such a hard function?
- How efficiently can an algorithm *produce* a hard function?

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?

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

When we talk about standard complexity classes, like $\mathsf{TIME}[2^n]$ (exponential time), we are talking about the class of languages that are 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.

(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.)

2.1 Reminder: Circuit Families

Let $L \subseteq \{0,1\}^\star$. Define $L_n = L \cap \{0,1\}^n$.

Definition 2. *We say that a* language $L \subseteq \{0,1\}^\star$ has $s(n)$-size circuits *if for all* n, $C(L_n) \leq s(n)$.

From Theorem 1.4, we have:

Proposition 1. *Every* $L \subseteq \{0,1\}^\star$ *has* $O(2^n/n)$ *size circuits.*

How can we "construct" a function that needs $2^n/n$ 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 a language* $L \in \mathsf{TIME}[2^{O(2^n)}]$ *such that* $C(L_n) = H(n)$ *for all* n.

Proof. Idea: *Brute-force it!*

Here is an algorithm for computing a hardest-possible function. In the following, let $tt(C)$ be the truth table of a circuit C.

Given an input x of length n:
Let $C^\star(x_1, \ldots, x_n) := x_1 \wedge \neg x_1$; let $s := size(C^\star)$.
For every $f \in B_n$, (iterate over all 2^n-bit strings)
  Iterate through all circuits C of size 1, size 2, etc., until you find the first C_f such that $C_f \equiv f$.
  (Evaluate each C on all 2^n inputs obtaining $tt(C)$, check if $tt(C) = f$.)
  If $size(C_f) > s$, update $s := size(C_f)$ 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 $C^\star$,
- $O(2^n)$ 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 a language $L \in \mathsf{SPACE}[2^n]$ such that $C(L_n) = H(n)$ for every n.*

3 Time lower bounds from circuit upper bounds: A connection

From the above, we can already obtain a "circuit upper bounds implies time lower bounds" connection. Recall $\mathsf{EXP} = \bigcup_k \mathsf{TIME}[2^{n^k}]$, $\mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}[n^k]$, $\mathsf{EXPSPACE} = \bigcup_k \mathsf{SPACE}[2^{n^k}]$.

Theorem 3.1. *If every language in EXP has circuits of size at most $H(n) - 1$, then $\mathsf{P} \neq \mathsf{PSPACE}$.*

Proof. Contrapositive. If $\mathsf{P} = \mathsf{PSPACE}$ then $\mathsf{EXP} = \mathsf{EXPSPACE}$. (The proof is a padding argument.) But the above says that there is an $L \in \mathsf{EXPSPACE}$ that does not have $(H(n) - 1)$-size circuits. So $L \in EXP$ as well. □

3.1 Sidebar: Padding Arguments in Complexity

Proof of $\mathsf{P} = \mathsf{PSPACE} \Rightarrow \mathsf{EXPSPACE} \subseteq \mathsf{EXP}$:

Assume $\mathsf{P} = \mathsf{PSPACE}$. Let $L \in \mathsf{SPACE}[2^{n^k}]$.

Define the language $L_{padding} = \{x01^{2^{|x|^k}} \mid x \in L\}$.

Claim: $L_{padding} \in \mathsf{SPACE}[n]$. Here's an algorithm running in linear space:

> Given a string y, determine if it has the form $y = x01^{2^{|x|^k}}$. If not, reject. If yes, run an $O(2^{|x|^k})$ space algorithm for L on x and output the answer.

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

By assumption, $L_{padding} \in \mathsf{P}$. Let A be a polytime algorithm for $L_{padding}$.

Now the following algorithm decides if $L \in \mathsf{TIME}[2^{O(n^k)}]$:

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

The running time is polynomial in $|y|$, i.e. polynomial in $2^{|x|^k}$. 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 languages: 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 Refining the connection

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

Theorem 3.2 (Folklore?). *There is an* $L \in \mathsf{EXP}^{\Sigma_2 P} = \mathsf{EXP}^{\mathsf{NP}^{\mathsf{NP}}}$ *such that* $C(L_n) = H(n)$ *for every* n.

This complexity class $\mathsf{EXP}^{\Sigma_2 P}$ is "Exponential Time with a $\Sigma_2\mathsf{P}$ oracle". Let's explain what that means.

Recall first what $\Sigma_2\mathsf{P} = \mathsf{NP}^{\mathsf{NP}}$ is:

the class of all languages L s.t. there's a polytime relation R and $k > 0$ such that

$$x \in L \iff (\exists\, y : |y| \leq |x|^k)(\forall\, z : |z| \leq |x|^k)[R(x, y, z) \text{ is true}]$$

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 with the same functionality?

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 definition of $\Sigma_2\mathsf{P}$, note our relevant polytime computable relation $R(C, C', z)$ is: (if ($size(C') \geq s$ or $C(z) \neq C'(z)$) then false, else true)

$\mathsf{EXP}^{\Sigma_2 P}$ = languages decidable by exp-time algorithms which can make exp-long queries to a $\Sigma 2P$ language, and get yes/no answers in 1 step.

This class $\mathsf{EXP}^{\Sigma_2 P}$ 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 $\mathsf{EXP}^{\Sigma_2 P}$ with maximum circuit complexity.

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

Define a problem `COMPLEX`:

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

We say that $g \leq f$ if $tt(g) \leq_{lex} tt(f)$, where $\leq_{lex}$ is lex order on strings.

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 $C(x) \neq g(x)$].

Now, here is an $\mathsf{EXP}^{\text{COMPLEX}}$ algorithm for computing the "lex first" function of maximum circuit complexity:

Given an input x of length n,

• In $\mathsf{EXP}^{\text{COMPLEX}}$,
Determine $H(n)$:
Ask: *is* $(1^{2^n}, k) \in$ 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.

• In $\mathsf{EXP}^{\text{COMPLEX}}$,
Determine the lex. first function f with circuit complexity $H(n)$,
Ask: *is* $(f, H(n)) \in$ COMPLEX*?*
on decreasing values of f, until the answer is no. More formally:

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

[The resulting f is the lex smallest function with $C(f) = H(n)$.]
Output $f(x)$.

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

3.3 More time lower bounds from circuit upper bounds

Now we can get a stronger consequence from circuit upper bounds:

Theorem 3.3. *If every language in* EXP *has circuit complexity less than* $H(n) - 1$*, then* $\mathsf{P} \neq \mathsf{NP}$.

Proof. We prove the contrapositive. Suppose $\mathsf{P} = \mathsf{NP}$. Then the polynomial hierarchy collapses, and $\Sigma_2\mathsf{P} = \mathsf{P}$. (Do you see why?) This implies that $\mathsf{EXP}^{\Sigma_2\mathsf{P}} = \mathsf{EXP}^{\mathsf{P}}$: whatever $\Sigma_2\mathsf{P}$ language we are using as an oracle, we can substitute that with a P-language.

But $\mathsf{EXP}^{\mathsf{P}} = \mathsf{EXP}$:

consider each oracle call to "polynomial time". The oracle call is $2^{O(n^k)}$ length, and it can be simulated directly by just running a $2^{O(n^k)}$-time machine.

So $\mathsf{EXP}^{\Sigma_2\mathsf{P}} = \mathsf{EXP}$. That means (by the previous theorem) that there is a function of circuit complexity $H(n)$ that's computable in EXP. □

In other words: if $\texttt{COMPLEX} \in \mathsf{P}$ then EXP has functions of circuit complexity $H(n)$.

3.4 Open Problems

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

> **OPEN:** Can we construct a function in $\Sigma_2\mathsf{EXP} = \mathsf{NEXP}^{\mathsf{NP}}$ that requires $H(n)$-size circuits?

There don't seem to be any crazy consequences of proving a yes-answer here. Of course, a no-answer would be incredible:

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

What can we expect to prove here? 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"

NP/f(n): NP machines with f(n) non-uniform bits. Languages 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?

Define $\mathsf{SIZE}[s] = \{L \subseteq \{0,1\}^\star \mid L \text{ has size-}s \text{ circuits}\}$.

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

Think about how you might prove this...

Corollary 4.1 (of Circuit Size Hierarchy)**.** *For all functions s(n) such that* $n \leq s(n) \leq o(2^n/n)$*, there is an* $f : \{0,1\}^n \to \{0,1\}$ *such that* $s < C(f) \leq 4s$.