UCT

Circuits

Klaus Sutner

Carnegie Mellon University

Spring 2024

1 **Parallelism and Non-Uniformity**

2 **Small Circuits**

3 NC **and** AC

4 **Branching Programs**

Any decision problem with finitely many instances is automatically decidable, albeit for entirely the wrong reasons.

To wit, we can hardwire the answers. Sort the instances in length-lex order

x_1	x_2	x_3	$\ldots$	x_{n-1}	x_n
b_1	b_2	b_3	$\ldots$	b_{n-1}	b_n

Here b_i is a bit that encodes the answer.

The problem is that the correct bit-vector $b_1, b_2, \ldots, b_n$ exists, basta.

Alas, we may not know what it is. We know a decision algorithm exists, but we may not be able to produce it.

Think about SAT and all formulae up to a fixed size N, say, $N = 10^{12}$. The corresponding table **exists** in set theory la-la land, but that is almost meaningless:

- The table would be gigantic and utterly impossible to implement.
- Even if we could somehow store all this information, we don't know how to determine the table entries in the first place.

Question: Is there are way to rule out such lookup tables?

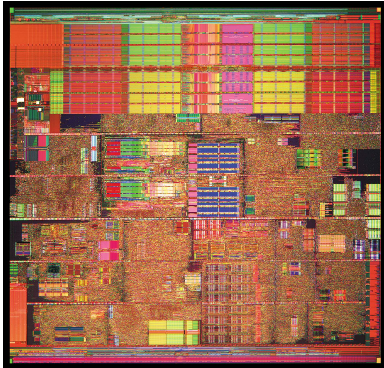
One interesting answer is to define the complexity of finite strings in terms of minimal programs that generate the string.

So we would insist that $\boldsymbol{b} = b_1, \ldots, b_n$ has small Kolmogorov-Chaitin complexity in order be acceptable as a lookup table.

Kolmogorov-Chaitin complexity has lots of interesting properties and has attracted an enormous amount of attention in the last half-century. But it has one annoying feature: everything is highly uncomputable. There seems to be no conceivable link to practical computation.

Here is a different approach that is better aligned with actual computation.

The idea that one should consider an "algorithm" that applies only to fixed size inputs is antithetical to our standard Turing machine approach, or equivalent attempts based on programs. In fact, it is always a central requirement that these devices work on all inputs.



But the hardware of all of our digital computers is based on working on a fixed number of bits.

This naturally leads to the idea of a (digital) circuit: a device with a fixed number of inputs that generates some output by performing a simple sequence of elementary (algebraic or logical) operations on the given data.

Thus, there is an evaluation map eval so that eval$(C)(x)$ is the result of performing these operations on circuit C and input x. Different possible domains come to mind, but for us the most important case is Boolean: we are manipulating single bits.

As we will see from the formal definition, eval for Boolean circuits is clearly linear time and, more importantly, can be handled in parallel.

Write $\mathcal{C}_n$ for the collection of n-input circuits over D defined formally as follows: we have an acyclic digraph G such that

- G has n nodes of in-degree 0, called sources (also inputs).
- G has one node of out-degree 0, called terminal (also output, sink).
- The non-source nodes ν of G are called gates, and are labeled by functions $D^{\text{indeg}(\nu)} \to D$.

Alternatively, we may allow for multiple outputs.

Note that given values for the inputs, we can propagate them to the output layer by layer (so the depth of the circuit will be important).

One often speaks about fan-in and fan-out instead of in-degree and out-degree, and one may refer to the edges as wires.

Unless the depth of the circuit is critical, fan-in 2 is essentially the same as bounded fan-in: we can build a little $\log k$ depth tree of in-degree 2 gates to simulate a in-degree k.

The same holds for fan-out.

Note that high values of fin-in/out make no sense for realizations in terms of digital circuitry: there are only so many wires one can attach to some presumably small gizmo.

Again, evaluation is straightforward: we traverse the digraph from the sources to the terminal, propagating the values upward. Given a reasonable representation of the circuit, and constant time functions at the gates, the whole evaluation is easily linear in the size of the circuit.

Hence, every n-input circuit $C \in \mathcal{C}_n$ over D defines a function

$$\mathcal{F}_c : D^n \to D \qquad\qquad \mathcal{F}_C(\boldsymbol{x}) = \mathsf{eval}(C)(\boldsymbol{x})$$

We are mostly interested in Boolean circuits where $D = \mathbf{2}$ and the fan-in is at most 2, so we have linear time evaluation.

Measuring Circuits

There are two essential parameters describing a circuit:

Size The total number of nodes in the circuit.

Depth The depth of the associated digraph (longest path).

So we are interested in circuits in some subclass $\mathcal{C}_n(s(n), d(n))$.

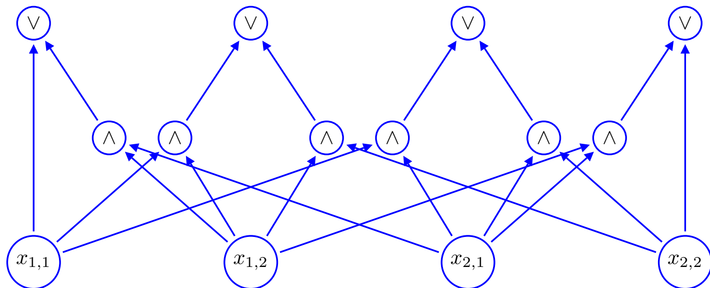
In particular, we would like to understand small, shallow circuits: say, polynomial size and logarithmic depth: $\mathcal{C}_n(\text{poly}, \text{log})$.

One central reason for the importance of circuits is that they can be evaluated in parallel: we traverse the circuit layer by layer, starting at the sources going up to the root. Each gate is associated with it's own processor and can evaluate independently of all the others.

For example, a circuit in $\mathcal{C}_n(\text{poly}, \log)$ would admit a logarithmic time evaluation if we can assign enough processors to the gates.

Of course, we need sufficiently man processors to do this. More precisely, we need the number of gates at each level to be reasonably small (really a red herring, there are only logarithmically many levels).

Things also get tricky with large fan-outs (inter-processor communication is the bane of parallel computation).



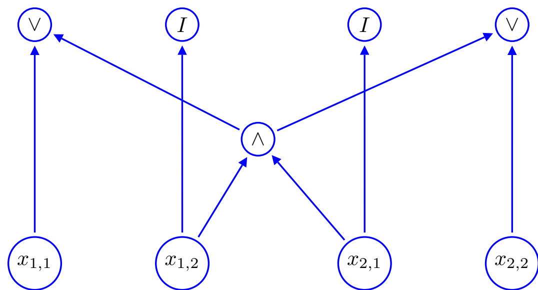
This 4-terminal circuit in $\mathcal{C}_4$ computes the square of a Boolean matrix $\left(\begin{smallmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{smallmatrix}\right) \in \mathbf{2} \times \mathbf{2}$.

Boolean matrix $A = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$

produces

$$A \cdot A = \begin{pmatrix} x_{11} + x_{12}x_{21} & x_{11}x_{12} + x_{12}x_{22} \\ x_{11}x_{21} + x_{21}x_{22} & x_{12}x_{21} + x_{22} \end{pmatrix}$$

In the circuit, the leftmost terminal corresponds to $x_{11} + x_{12}x_{21}$.



Similarly, this one computes the transitive closure.

Exercise

Figure out what a transitive closure circuit would look like for an $n \times n$ matrix. For simplicity assume $n = 2^k$.

And Boolean Formulae?

Is there any difference between a Boolean formula and a Boolean circuit?

It depends on the degrees:

- Standard Boolean operations like conjunction and disjunction have two arguments (though they naturally generalize), so that corresponds to fan-in at most 2.
- For fan-out higher than 1 we need to duplicate subexpressions in order to translate a circuit into a formula.

So circuits are a strict generalization.

Correspondingly, we should expect circuit SAT (CSAT), the decision problem analogous to SAT, to be $\mathbb{NP}$-complete:

Problem:	**Circuit** SAT **(**CSAT**)**
Instance:	A Boolean circuit C.
Question:	Is C satisfiable?

Theorem

CSAT *is* $\mathbb{NP}$*-complete.*

For hardness, we transform a Boolean formula from SAT into a corresponding circuit. Note that the circuit could actually be a bit smaller thanks to subexpression sharing.

Since there are 2^{2^n} Boolean functions of n arguments, one should not expect corresponding circuits to be small, at least not most of the time.

Theorem (Shannon 1949)

Most Boolean functions of n arguments require circuits of size $(1 - \varepsilon)\, 2^n / n$ for all positive ε.

Similarly, for Boolean formulae, we get a size of $(1 - \varepsilon)\, 2^n / \log n$.

Of course, interesting functions like parity, majority, counting and so on may well have much smaller circuits.

In order to use circuits to recognize a language L over $\mathbf{2}$ we need one circuit C_n with n inputs for each $n \in \mathbb{N}$:

$$\forall\, n\, \exists\, C_n \in \mathcal{C}_n \left(C_n \text{ recognizes } L \cap \mathbf{2}^n \right)$$

We can then define the language of this family of circuits as the set of words $x \in \mathbf{2}^\star$ such that $C_{|x|}$ on input x evaluates to true.

Theorem (Quadratic Circuits)

Let t be reasonable, $t(n) \geq n$.
Then $L \in \mathrm{TIME}(t)$ has a circuit family of size $O(t^2)$ and depth $O(t)$.

The key idea here is not new: we can represent a computation of a Turing machine running in time $N = t(n)$ by a tableau, a grid of size $N \times N$ (this approach was used e.g. in the old tiling problem). As usual, row i represents the configuration at time $i \leq N$, in the standard $\Sigma^\star \, Q \, \Sigma^\star$ format.

Acceptance can be expressed in terms of having reached the appropriate state at time N and we may safely assume that the head has returned to its original position (configuration $q_y \boldsymbol{w}$).

Moving towards Boolean circuits, let $k = \max(\log|Q|, \log|\Sigma|)$. We represent each symbol by a block of $k+1$ bits, say, $0\boldsymbol{a}$ for tape symbols and $1\boldsymbol{s}$ for states. So there will be exactly one block in each row where the indicator bit is 1, the state block.

Unless a block is adjacent to or the state block, the bits do not change from row i to $i+1$. The bits in the state block and the two adjacent blocks are updated according to the transition function of the Turing machine.

For example, if the transition involves moving the head to the right this may look locally like

$$\begin{array}{cccccc} \ldots & 0\boldsymbol{a} & 1\boldsymbol{s} & 0\boldsymbol{b} & 0\boldsymbol{c} & \ldots \\ \ldots & 0\boldsymbol{a} & 0\boldsymbol{b}' & 1\boldsymbol{s}' & 0\boldsymbol{c} & \ldots \end{array}$$

Clearly this can be handled by a Boolean circuit.

Exercise

Figure out the details.

Consider an arbitrary language $L \subseteq \mathbf{2}^\star$. Clearly, for each n, there is a circuit C_n^L that recognizes the finite language $L \cap \mathbf{2}^n$. For example, think of the latter as a Boolean function, and express it as the standard DNF formula (depth 2 plus negation with unbounded fan-in).

But that means that L is recognized by the potentially exponential size circuit family $(C_n^L \mid n \geq 0)$.

- This works even if L is highly undecidable (say, arithmetic truth).
- Also, there are uncountably many circuit families.

This may seem like a bridge too far. As we will see, it is actually a useful concept.

How do we pare things back to a more practical notion?

Size: Insist on small circuit size.

Computability: Insist that C_n is computable from n.

Because of the computability constraint there are only countably many such circuits. And, any such computable family represents an actual decision algorithm.

This is another example of turning an existential quantifier "there is a circuit C_n such that" into something more constructive: we want the circuit C_n to be computable from n. For example, for the quadratic size circuit theorem, we can construct the circuit directly from the Turing machine.

This leads to the critical distinction between uniform versus non-uniform families of circuits.

Uniform: There is a Turing machine that, on input n, constructs the corresponding circuit C_n.

Non-Uniform: Each circuit exists (whatever that may mean; e.g., we could prove existence in ZFC), but we may not know how to construct them.

One might suspect that uniform circuits generated by simple Turing machines cannot do much.

- A circuit family (C_n) is $\mathbb{P}$-uniform if there is a polynomial time Turing machine $\mathcal{M}$ such that $\mathcal{M}(0^n) = C_n$.
- A circuit family (C_n) is logspace-uniform if there is an implicitly logspace computable function f such that $f(0^n) = C_n$.

Implicitly log space computable means that $x, i \mapsto \mathsf{bit}(f(x), i)$ is log space computable (just a single bit, not all of $f(x)$).

For the sake of clarity, here is one possible way to pin down a reasonable description of a circuit family. We think of C_n as a labeled digraph (labels are Boolean operations) with vertex set $[m]$, $m = s(n) = O(n^c)$. The first n nodes are input, the last node is output.

For each n, we need to be able to compute

- the type of each gate (vertex in the digraph)
- the edges of the digraph (some bit in the adjacency matrix)

The number of bits needed to represent a vertex is $\log m = O(\log n)$.

Theorem

$\mathbb{P}$-uniform circuit families recognize exactly $\mathbb{P}$.

Sketch of proof.

For each instance x, $n = |x|$, we run the circuit machine $\mathcal{M}$ on 0^n to get C_n. Feed x to the circuit and return the result.

For $L \in \mathbb{P}$, let $\mathcal{M}$ be a polynomial time TM recognizing L. For given n, we can construct a circuit C_n that simulates $\mathcal{M}$ on all inputs of length n.

□

1 Parallelism and Non-Uniformity

2 **Small Circuits**

3 NC and AC

4 Branching Programs

Definition

The class P/poly consists of languages decidable by a circuit family of polynomial size.

This a non-uniform class, we are only interested in the existence of a small circuit, not how to construct it. So we are getting help from some mysterious entity that knows everything about circuits.

But note that the smallness requirement rules out some brute-force lookup table approach for, say, SAT. This is vaguely similar to reining in the power of an all powerful prover by a computationally limited verifier.

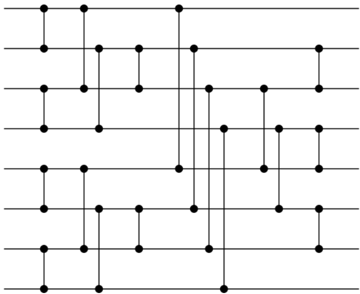
If you prefer, you can think about P/poly as being given a polynomial time Turing machine $\mathcal{M}$ (wrto the first argument) together with a sequence (a_n) of advice strings such that for all n

$$x \in L \cap \mathbf{2}^n \iff \mathcal{M}(x, a_n) \text{ accepts}$$

Our friendly advice strings come from la-la land, they don't have to be computed in any particular way, they just exist in the blissful realm of celestial spheres, also known as Zermelo-Fraenkel set theory with Choice.

A Turing machine is oblivious if its head position at time t depends only on the length $|x|$ of the input, not the actual word $x \in \mathbf{2}^\star$.

This may sound bizarre, but it is just the idea of a non-adaptive algorithm transferred into Turing world.



As a vague analogy, consider sorting: a standard sorting algorithm like quicksort adapts its execution pattern to the actual input. But a sorting network (like Batcher sort) only depends on input length.

Proposition

Every Turing machine can be simulated by an oblivious one, with quadratic increase in running time (with usual assumptions).

This is quite similar to the simulation of a multi-tape machine on a single tape machine: keep sweeping the tape head from one end to the other. The following is much harder to prove.

Theorem

The oblivious simulation can be handled in time $O(t \log t)$.

Theorem

An oblivious time t Turing machine can be simulated by a circuit of size $O(t)$.

Proof. Again we translate a tableau of a computation into a circuit.

By obliviousness, an instantaneous description requires only the state and the tape inscription, the head position is not needed. This amount of information can be handled in $O(t)$ bits.

But then a constant size circuit is sufficient to handle the parts of an ID that change during one single time step in the computation.

By piling up these one-step circuits (and pass-through wires for the parts that do not change) we get a simulation of the whole computation using a circuit of size $O(t)$.

□

Theorem (Adleman)

BPP $\subseteq$ P/poly.

Proof.

We essentially showed this in the lecture on BPP under the label "feeble derandomization."

For all $n \in \mathbb{N}$ there is a special witness $u_n \in \mathbf{2}^{p(n)}$ such that

$$\forall\, x \in \mathbf{2}^n \left(x \in L \iff \mathcal{M}(x, u_n) \text{ accepts } \right)$$

$\square$

But recall that the argument was probabilistic and utterly non-constructive, we know the advice string exists, but we have no cheap way to construct it.

By the oblivious TM theorem, $\mathbb{P} \subseteq$ P/poly in a strong sense.

But note, the other direction is blatantly false: every tally language $L \subseteq 0^\star$, no matter how undecidable, is in P/poly: just use the language itself for advice.

The following connection between $\mathbb{NP}$ and P/poly is known.

Theorem

If $\mathbb{NP} \subseteq$ P/poly, then the polynomial hierarchy collapses at level 2.

1 Parallelism and Non-Uniformity

2 Small Circuits

3 NC and AC

4 Branching Programs

As we have seen, it is really the size of a circuit that makes it interesting. So, it is natural to define various "small size" circuit classes and study their computational power.

Definition (NC)

A language L is in NC^d if there is a bounded fan-in circuit family (C_n) that decides L, the size of C_n is polynomial in n and the depth of C_n is $O(\log^d n)$.

NC is the union of all NC^d.

This is interesting, since we can evaluate a circuit in parallel in depth of the circuit many steps (given enough processors).

For example, it is easy to see that parity testing is in NC^1.

Claim:
A problem admits an efficient parallel algorithm iff it lies in NC.

The key idea is that, given enough processors, we can evaluate C_n in $O(\log^{d+1} n)$ steps, assuming that our parallel algorithm can send an output bit produced at gate to all the recipients in $O(\log n)$ steps.

For the opposite direction, we build a circuit that simulates the parallel algorithm. Its width will be the number of processors and its depth the number of rounds in the parallel algorithm.

Definition (AC)

A language L is in AC^d if there is an unbounded fan-in circuit family (C_n) that decides L, the size of C_n is polynomial in n and the depth of C_n is $O(\log^d n)$.

AC is the union of all AC^d.

So AC disregards physical realizability considerations: unbounded fan-in is an illusion in the world of electronic circuits.

Note that NC^0 is not interesting, but AC^0 might be: we could check for the existence of an input-bit 1.

Lemma

$\mathsf{AC}^d \subseteq \mathsf{NC}^{d+1}$. *Hence* $\mathsf{AC} = \mathsf{NC}$.

As defined, these classes are non-uniform: the circuits exist without necessarily being computable. Similarly we can define obvious uniform versions.

Proposition

Addition is in logspace-uniform AC^0.

Proof.

Let $\boldsymbol{a}, \boldsymbol{b} \in 2^n$ be the inputs, written in reverse binary (LSD first). Write c_i for the carry in position i, $c_1 = 0$. Note that the standard kindergarten algorithm does not work: it is bounded fan-in, but linear depth since we need to scan the bits from left to right, one at a time, to get the carries right.

Wild Idea: Maybe there is a clever, non-standard method to compute the carries?

The idea is that $c_{i+1} = 1$ if

- $a_i = b_i = 1$, or
- $a_i = 1$ or $b_i = 1$ and $a_{i-1} = b_{i-1} = 1$, or
- $\ldots$

In other words, the carries march down the line.

In terms of a Boolean formula we can express this as follows:

$$c_{i+1} = \bigvee_{k \leq i} (a_k \wedge b_k) \wedge (a_{k+1} \vee b_{k+1}) \wedge \ldots \wedge (a_i \vee b_i)$$

This condition can be handled by a constant-depth unbounded fan-in circuit.

□

A much more difficult result shows that Parity is difficult. By Parity we mean the problem of computing $\sum x_i \bmod 2$. Clearly we can handle this function in logarithmic depth, NC^1, just build a tree of binary xor-gates. Could we flatten out this circuit if we allow for unbounded fan-in?

Theorem (Ajtai 1983; Furst-Saxe-Sipser 1984)

Parity is not in AC^0.

In other words, any constant-depth circuit that computes Parity must be exponentially large.

We can define Boolean matrix multiplication analogously to numerical matrix multiplication by interpreting plus as disjunction, and times as conjunction.

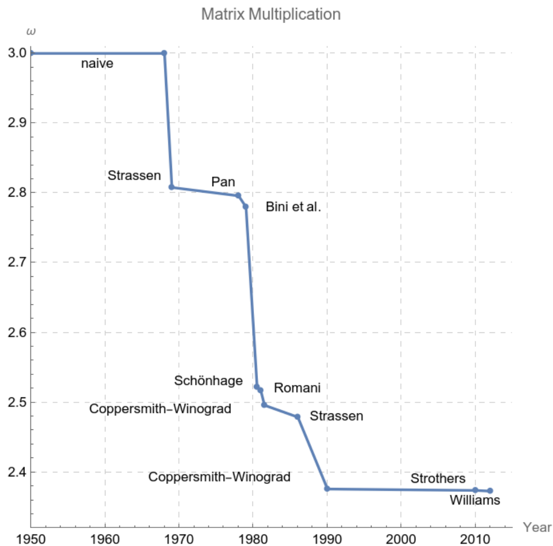
$$A \cdot B = C \qquad C(i,j) = \sum_k A(i,k)B(k,j)$$

In particular, if A is the adjacency matrix of a graph, then A^k describes paths of length exactly k. To tackle reachability it suffices to compute

$$(A+I)^{n-1} = I + A + A^2 + \ldots + A^{n-1}.$$

Using fast exponentiation, this can be handled in $O(\log n)$ Boolean matrix multiplications (BMM).

As written, matrix multiplication is cubic time. Alas, the only obvious lower bound is $\Omega(n^2)$. A lot of effort has gone into devising algorithms that multiply integer matrices in time $O(n^\omega)$ for some $\omega < 3$.



1990: 2.3755 $\rightsquigarrow$ 2023: 2.3716

One can exploit fast integer MM for BMM: think of a Boolean matrix as an integer matrix. Then multiply (using only addition and multiplication, but no subtraction; we are really working over $\mathbb{N}$) and get back to Boolean by applying the sign function everywhere.

This raises the thorny question of what can be done with "genuine Boolean" algorithms.

How about circuits? We can compute $A^\star = (A + I)^{n-1}$ by $\log n$ many matrix-squaring operations.

But for unbounded fan-in we can square a Boolean matrix in depth 2. Hence we can compute $A^\star$ in uniform AC^1 and thus in uniform NC^2.

Lemma (Uniform)

$\mathsf{NC}^1 \subseteq \mathbb{L} \subseteq \mathbb{NL} \subseteq \mathsf{NC}^2 \subseteq \mathsf{NC} \subseteq \mathbb{P}$

Proof.

To show $\mathsf{NC}^1 \subseteq \mathbb{L}$, consider some input $\boldsymbol{x} \in \mathbf{2}^n$. Construct the log-depth circuit C_n, but note that we cannot simply evaluate in the customary fashion: we need to stay in $\mathbb{L}$. Instead we evaluate the circuit by recursion, essentially performing DFS from the terminal node in a virtual graph (create new nodes/gates as needed).

The recursion stack has depth $\log n$, and each stack frame contains only a constant number of bits since the circuit is in NC^1 (rather than AC^1). So the whole computation runs in $\mathbb{L}$.

Performing DFS in the virtual circuit really requires some attention to the details of the representation of the uniform NC^1 circuit. The customary assumptions are that we can compute all of the following in logarithmic space:

- size(0^n) the size of the circuit
- gate$(0^n, i)$ the type of the gate at vertex i
- wire$(0^n, i, j)$ checks whether the is a wire from i to j

The log-space machine can perform all these calculations and that is enough to handle graph exploration.

For $\mathbb{NL} \subseteq \mathsf{NC}^2$, consider a $\mathbb{NL}$ Turing machine $\mathcal{M}$.

For each length n, consider the digraph $\mathfrak{C}_{\mathcal{M},n}$ whose nodes represent the configurations of $\mathcal{M}$ but without the actual input $\boldsymbol{x} = x_1 \ldots x_n$. In other words, we keep track of

- the state,
- the contents of the $\log n$ worktape,
- the position of the read head on the input tape.

Since $\mathcal{M}$ is $\mathbb{NL}$, our truncated configurations have logarithmic size and the whole graph has size N which is polynomial in n. This part depends only on the length of $\boldsymbol{x}$, not the actual bits.

Now suppose we are given a concrete input $\boldsymbol{a} \in \mathbf{2}^n$. The edges in $\mathfrak{C}_{\mathcal{M},\boldsymbol{a}}$, the computation graph corresponding to a particular input, represent single transitions of the Turing machine $\mathcal{M}$ and will thus in general depend on particular input bits.

We will construct the $N \times N$ adjacency matrix of $\mathfrak{C}_{\mathcal{M},\boldsymbol{a}}$ using a constant depth circuit: each bit is either independent of $\boldsymbol{a}$, or it depends on just a single bit in $\boldsymbol{a}$: we keep track of the position of the read head.

To check acceptance by $\mathcal{M}$, it then suffices to compute the transitive closure of this graph with edges controlled by $\boldsymbol{a}$. Using the repeated squaring trick from above, we can find the transitive closure by a NC^2 circuit. Everything is uniform in $\mathcal{M}$ and n, so we get an uniform circuit family.

□

Counting the number of 1 bits in the input turns out to be very useful.

Definition

A threshold function thr^n_k, $0 \leq m \leq n$, is an n-ary Boolean function defined by

$$\mathrm{thr}^n_m(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \#(i \mid x_i = 1) \geq k, \\ 0 & \text{otherwise.} \end{cases}$$

thr^n_k is nicely symmetric:

$$\mathrm{thr}^n_k(\boldsymbol{x}) = \mathrm{thr}^n_k(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

These are the kinds of functions used in neural nets.

Lots of useful Boolean functions can be defined in terms of threshold functions.

- thr^n_0 is the constant tt
- thr^n_1 is n-ary disjunction
- thr^n_n is n-ary conjunction
- $\mathrm{thr}^n_k(\boldsymbol{x}) \wedge \neg \mathrm{thr}^n_{k+1}(\boldsymbol{x})$ is the counting function: "exactly k out of n"

We can express threshold functions in terms of majority, a special case of a threshold function: $\mathrm{maj}(\boldsymbol{x}) = \mathrm{thr}^n_{n/2}$.

$$\mathrm{thr}^n_k(\boldsymbol{x}) = \begin{cases} \mathrm{maj}\big(\boldsymbol{x}, 1^{n-2k}\big) & \text{if } k \leq n/2 \\ \mathrm{maj}\big(\boldsymbol{x}, 0^{n-2k}\big) & \text{otherwise.} \end{cases}$$

Consider a slightly more complicated problem than plain addition: we are given n numbers, each n bits, and we want to compute their sum. Let's call this multi-addition.

Lemma

Multi-addition is in NC^1.

Proof.

We assume for simplicity that the sum is still an n-bit number.

The idea is to reduce the problem of adding 3 n-bit numbers to adding just 2 numbers with $n+1$ bits each.

Suppose $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are n-bit numbers, MSD first, 0-indexed. We define two $(n+1)$-bit numbers $\boldsymbol{d}$ and $\boldsymbol{e}$ as follows:

$$\boldsymbol{d} = d_{n-1} d_{n-2} \dots d_0\, 0 \qquad \boldsymbol{e} = 0\, e_{n_1} e_{n-2} \dots e_0$$

$$d_i = \mathrm{thr}^3_2(a_{i-1}, b_{i-1}, c_{i-1}) \qquad e_i = a_i \oplus b_i \oplus c_i$$

We can compute $\boldsymbol{d}$ and $\boldsymbol{e}$ in bounded fan-in and constant depth.

Hence we reduce the problem of adding n-many n-bit numbers to adding $2n/3$-many $n+1$-bit numbers. Repeating this step $\log n$ times we wind up with 2 numbers of size $n + \log n$ and the whole maneuver requires only logarithmic depth.

We already know how to add these two numbers and we get a circuit in NC^1, done.

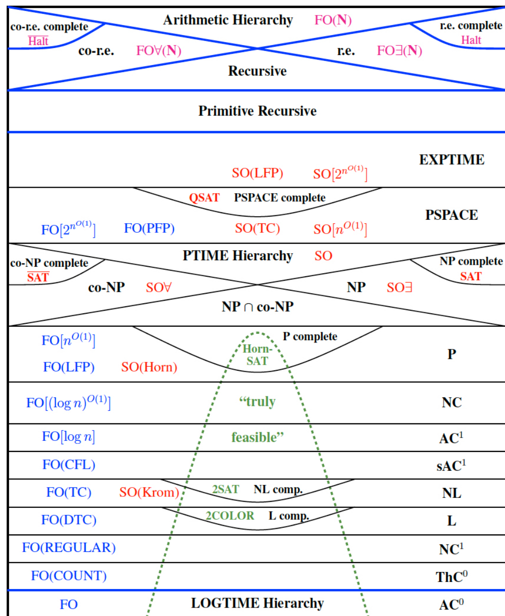
□

Intuitively, multiplication of n-bit numbers “reduces” to multi-addition: to compute

$$(x_{n-1}x_{n-2}\dots x_1x_0)\cdot(y_{n-1}y_{n-2}\dots y_1y_0)$$

we just have to add the numbers $2^i x$ for all $y_i = 1$.

This could be made precise by introducing yet another notion of reducibility suitable for circuits, but we won’t go there.



abstract computation

civilized computation

feasible computation

1 **Parallelism and Non-Uniformity**

2 **Small Circuits**

3 NC **and** AC

4 **Branching Programs**

There is very little difference between a circuit and a straight line program, a sequence of instructions to compute certain values, without any branching (no Boolean tests, no loops, no nothing).

For example, over $\mathbb{N}$, an arithmetic straight-line program of length n is a sequence of n assignments of the form

$$
\begin{aligned}
v_1 &= 1 && \text{constant input} \\
v_2 &= x && \text{variable input} \\
v_i &= v_l \ \mathsf{op}\ v_r && \text{where } 0 \leq l, r < i \leq n.
\end{aligned}
$$

The only allowed operations are op $= +, -, \times$. The output of the program is the value of v_n.

It is clear that any SLP computes a polynomial function: just substitute v_l op v_r for v_i everywhere, and the resulting expression is a polynomial in x.

Also, any polynomial can be computed by a SLP.

$$
\begin{array}{ll}
v_1 = 1 & 1 \\
v_2 = x & x \\
v_3 = v_1 + v_1 & 2 \\
v_4 = v_2 * v_2 & x^2 \\
v_5 = v_4 * v_3 & 2x^2 \\
v_6 = v_5 - v_1 & 2x^2 - 1 \\
v_7 = v_6 * v_2 & 2x^3 - x \\
v_8 = v_7 + v_1 & 2x^3 - x + 1
\end{array}
$$

Sometimes a very short program can compute long polynomials (if they have a lot of structure that can be exploited).

$$1 + 8\,x + 28\,x^2 + 56\,x^3 + 70\,x^4 + 56\,x^5 + 28\,x^6 + 8\,x^7 + x^8$$

$v_1 = 1$	1
$v_2 = x$	x
$v_3 = v_2 + v_1$	$1 + x$
$v_4 = v_3 * v_3$	$(1 + x)^2$
$v_5 = v_4 * v_4$	$(1 + x)^4$
$v_6 = v_5 * v_5$	$(1 + x)^8$

Remember addition chains from 15-251?

$$
\begin{array}{ll}
v_1 = 1 & 1 \\
v_2 = v_1 + v_1 & 2 \\
v_3 = v_2 + v_2 & 4 \\
v_4 = v_3 + v_3 & 8 \\
v_5 = v_4 + v_4 & 16 \\
v_6 = v_5 + v_4 & 24 \\
v_7 = v_6 + v_3 & 28 \\
v_8 = v_7 + v_2 & 30
\end{array}
$$

This is the obvious SLP based on the binary representation of n.

Is this really the shortest program for 30?

$$
\begin{aligned}
v_1 &= 1 && 1 \\
v_2 &= v_1 + v_1 && 2 \\
v_3 &= v_2 + v_2 && 4 \\
v_4 &= v_3 + v_3 && 8 \\
v_5 &= v_4 + v_2 && 10 \\
v_6 &= v_5 + v_5 && 20 \\
v_7 &= v_6 + v_5 && 30
\end{aligned}
$$

Length 7 is enough. Is that it? Is this SLP unique?

$$
\begin{aligned}
v_1 &= 1 & 1 \\
v_2 &= v_1 + v_1 & 2 \\
v_3 &= v_1 + v_2 & 3 \\
v_4 &= v_2 + v_3 & 5 \\
v_5 &= v_4 + v_4 & 10 \\
v_6 &= v_4 + v_5 & 15 \\
v_7 &= v_6 + v_6 & 30
\end{aligned}
$$

It is true that 7 is best possible, but it takes a bit of effort to prove this. Also note that the solution is not unique.

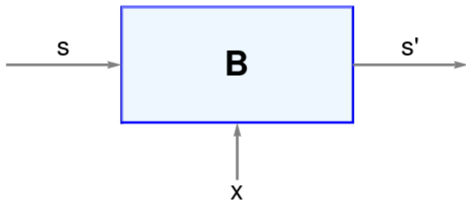
Straight-line programs are already surprisingly complicated, in particular when one starts to ask questions about the minimal SLP to accomplish some particular task (a much reduced version of Kolmogorov-Chaitin complexity).

Wild Idea: What if we make the programs a little more complicated by allowing a rather feeble if-then-else construct?

Instead of just blindly running through a sequence of simple steps, we ask whether some condition holds and then apply one operation or another. On the other hand, we will only handle Boolean functions to keep things from spinning out of control.

We want to compute a Boolean function $\mathbf{2}^n \to \mathbf{2}$, so the input is a sequence of bits $\boldsymbol{x} = x_1, x_2, \ldots, x_n$.

Our devices are composed of several boxes:



x is the input bit, s and s' are signals being sent from one box to the next. We will describe the set S of signals in a moment. A box can compute arbitrary Boolean functions, no questions asked. However, it has access only to s and bit x, rather than the whole input.

A chain of boxes: the signals propagate from the left to the right; the output appears all the way on the right. The input signal to the leftmost box is fixed. The input bits are provided at the bottom of the boxes.

The number ℓ of boxes is the length of the device.

We want to understand what sort of Boolean function can be computed by these box chains.

Recall that we allow the boxes to be all powerful, they can compute any Boolean function of their inputs.

Without restrictions, this model is useless: box 1 sends x_1 to box 2, which sends $x_1 x_2$ to box 3, ... Then box $n+1$ has all of $\boldsymbol{x}$ and can compute anything at all.

Constraint I: The signal set must be finite.

In other words, each box can send only a bounded number of bits to the next.

If $\ell = n$, this renders the model essentially useless: we really wind a up with a finite state machine on a binary alphabet. We could compute parity this way, but cannot even handle majority.

The trick is to allow $\ell > n$: the same bit may be read repeatedly in multiple boxes. This is very similar to a Boolean variable appearing multiple times in a formula.

We will call these gizmos bounded width branching programs (BWBPs). Each box is an if-then-else instruction where the input bit determines which branch is taken; the whole program is a sort of binary decision tree.

This may all sound rather bizarre, but think about a DFA $\mathcal{A}$ over the binary alphabet $\mathbf{2}$.

The automaton consists essentially of two transition functions $\delta_s : Q \to Q$ where Q is the state set of $\mathcal{A}$, $s \in \mathbf{2}$.

Letting $\ell = n = |x|$ of some binary input word, each box receives as input signal the previous state, and returns as output signal the next state. The initial signal is the initial state of $\mathcal{A}$, acceptance depends on the last signal being a final state.

We could easily build a Turing machine that constructs this BWBP from 0^n, everything is nicely uniform.

Constraint II: The length must be polynomial.

These devices are called bounded width poly length branching programs (BWPLBPs).

Why are these constraints particularly interesting?

- If we allow exponential length, then a fixed size signal set suffices for all languages $L \subseteq \mathbf{2}^\star$.
- On the other hand, we can achieve linear length at the cost of exponentially many signals.

What should we use as signal set to make a BWPLBPs powerful?

David Barrington come up with an utterly amazing answer: groups, in the algebraic sense of the word.

It's utterly unclear why an algebraic structure would help, one might think more of something combinatorial, some clever code or some such.

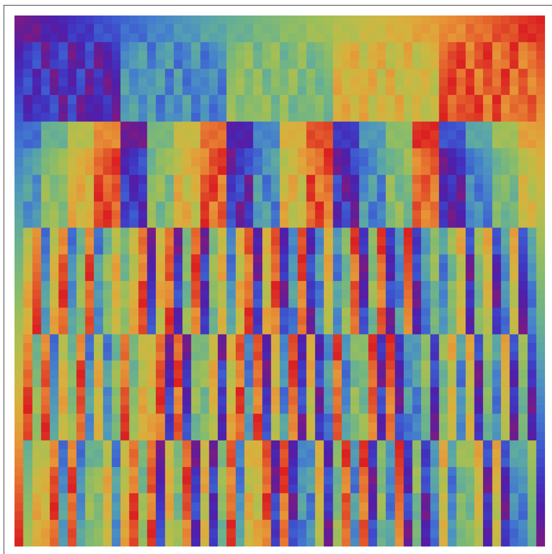
More precisely, one uses a non-commutative, simple group G.

A simple group is a group that has no nontrivial normal subgroups. Finite simple groups are hugely important, they form the basic building blocks of all finite groups.

A commutator in a group is an element $[a, b] = aba^{-1}b^{-1}$. Note that $ab = ba$ iff $[a, b] = 1$; more generally $ab = [a, b]\, ba$. The commutators in a group generate a normal subgroup, the commutator subgroup.

Hence, in a simple group, every element must be a product of commutators.

As a concrete example, think about the alternating group $\mathfrak{A}_5$, all even permutations on 5 letters, a group of order 60 and the smallest non-commutative simple group.



Since signals are group elements, lets agree that if the output is the identity element $1 = 1_G$ we interpret that as the function value `true`; any other element $g \neq 1$ means `false`. The initial signal on the leftmost box is 1.

Each box takes as input a group element g and depends on two parameters a and b, both in G. The box $B_{a,b}$ returns

$$\textbf{if } x \textbf{ then } g \cdot a \textbf{ else } g \cdot b$$

By choosing the multipliers a and b in the right manner, we can build a BWBP that uses any specific non-identity element g to represent `false`. So we can concoct a $(1, g)$-representation for any $g \neq 1$.

We are given a Boolean formula $\Phi(\boldsymbol{x})$ and want to construct a box chain that computes the corresponding Boolean function.

It suffices to handle just the logical connectives $\neg$ and $\vee$. The construction is by structural induction on Φ.

Variables are straightforward: those are just the input bits to our boxes. Note that a variable appearing multiple times in a formula is not a problem: we can read the same bit repeatedly.

For negation, suppose we already have a $(1, g)$ representation of φ. We multiply by g^{-1}, which turns $(1, g)$ into $(g^{-1}, 1)$, a representation of $\neg\varphi$.

Disjunctions are a bit more complicated.

Suppose we want a representation of $\phi \vee \psi$ via g.

Pick $a \neq b$ such that $[g_1, g_2] = g$. This is a white lie, we really should deal with a product of commutators, but hey ...

By induction, we can already represent ϕ and ψ via g_1 and g_2, respectively. Say, the BWBPs are P_ϕ and P_ψ. Now we form a new chain

$$P_\phi \to P_\psi \to P_\phi^{-1} \to P_\psi^{-1}$$

If at least one of these BWBPs produces 1, the final output is also 1. Otherwise, we get g.

Theorem (Barrington 1989)

The class of languages recognized by BWPLBP is exactly non-uniform NC^1.

Converting a BWPLBP into an NC^1 circuit is fairly straightforward.

But the opposite direction is a small miracle: the proof relies heavily on group theory and linear algebra over finite fields.