UCT

Circuits

Klaus Sutner
Carnegie Mellon University
Spring 2022

Recall: Finite Problems 2

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

$$\begin{array}{cccccc} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ \hline b_1 & b_2 & b_3 & \dots & b_{n-1} & b_n \end{array}$$

Here b_i is a bit that encodes the answer.

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

Algorithms versus Constructivism 3

Think about SAT and all formulae of fixed size N. The corresponding table **exists**, but that is almost meaningless:

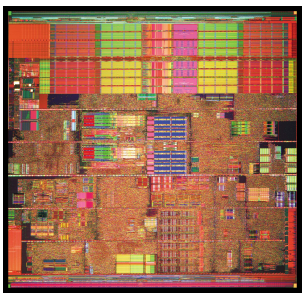
- The table would be gigantic, impossible to implement.
- We don't know how to determine the table entries efficiently.

The second problem might go away if something catastrophic happens in complexity theory, but that does not seem likely.

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

Fixed Input Size 4

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.

Circuits 5

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) operations on the given data.

Thus, there is an evaluation map eval so that $\text{eval}(C)(\boldsymbol{x})$ is the result of performing these operations on circuit C and input $\boldsymbol{x}$. 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 is linear time and, more importantly, can be handled in parallel.

Circuits, Defined

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 $D^{\text{indeg}(\nu)} \to D$.

Sometimes it is more natural to 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).

Fan-In/Fan-Out

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 the same as bounded fan-in: we can build a little logarithmic depth tree of in-degree 2 gates.

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).

Evaluation

It is clear how eval works for these circuits: we traverse the digraph from the sources to the terminal, propagating the values upward. Clearly, this traversal can be handled in parallel in layers starting at the sources. Given a reasonable representation of the circuit, and constant time functions at the gates, the whole evaluation is clearly linear in the size of the circuit.

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

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

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

Measuring Circuits

There are two essential measures for a circuit:

Size The total number of nodes in the circuit.

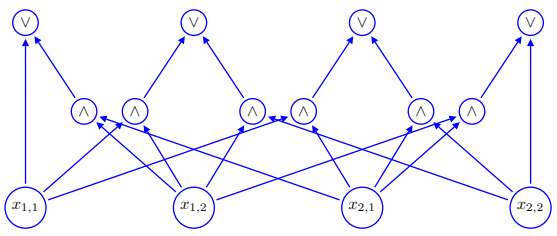
Depth The depth of the associated digraph (longest path).

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

We are particularly interested in small, shallow circuits: say, polynomial size and logarithmic depth: $\mathcal{C}_n(\text{poly}, \log)$.

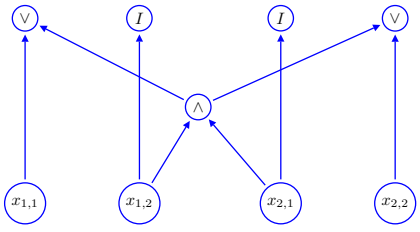
Such a circuit would admit a logarithmic time evaluation given parallel execution: we can work on several gates in parallel. This is perfectly practical as long as the number of gates at each level is reasonably small (so we can allocate one processor for each gate).

Squaring Circuit



This 4-terminal circuit in $\mathcal{C}_4$ computes the square of a Boolean matrix $\left(\begin{smallmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{smallmatrix}\right) \in \mathbf{2} \times \mathbf{2}$.

Transitive Closure Circuit



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.

Circuit SAT

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.*

Proof.

For hardness, transform ordinary SAT into a circuit problem using the exact same approach as in the last theorem.

□

Circuit Size

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.

Circuits and Languages

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

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

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 $O(t)$.

Proof Sketch

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.

Tableau to Circuit

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

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

Clearly this can be handled by a Boolean circuit.

□

Exercise

Figure out the details.

Dire Warning

Consider an arbitrary language $L \subseteq \mathbf{2}^*$. 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 $(C_n^L)_n$.

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

This may seem a bridge too far, but, as we will see, it is actually a useful notion.

Killing Monsters

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.

Uniform Circuits

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 constructs the circuits of all sizes.

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.

Simple Circuits

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 that maps 0^n to the circuit C_n.

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

Circuit Description

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), the first n nodes are input, the last node is output.

For each n, we need to be able to compute

- the size s of C_n, $s = O(n^c)$
- 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 logarithmic in n.

Nil Novis

Theorem

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

Theorem

Logspace-uniform circuit families of polynomial size recognize exactly $\mathbb{P}$.

P/poly

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.

Advice

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

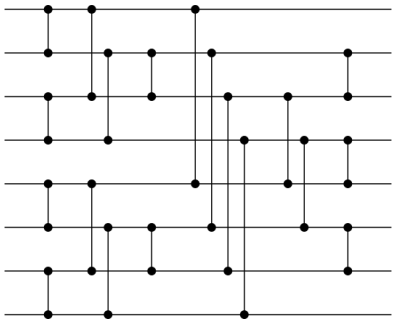
$$x \in L \cap \mathbf{2}^n \iff \mathcal{M}(x, a_n) \text{ accepts}$$

The advice strings come from La-La-Land, they don't have to be computed in any particular way.

Oblivious Turing Machines

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 $x \in \mathbf{2}^*$.

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.

Oblivious Simulations

Proposition

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

This is 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 $O(t \log t)$.

P/poly versus Polynomial Time

Theorem

An oblivious TM in time t 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 consists only of state and tape inscription, requires $O(t)$ bits (head position is not needed).

But then a constant size circuit is sufficient to update such an ID for one time step.

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 $O(t)$.

□

Randomness

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

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

□

But recall that the argument was non-constructive, we know the advice string exists, but we have to cheap way to construct it.

More Inclusions?

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.*

Nick's Class

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

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.

Note that evaluating a circuit can be handled in parallel: given the values at the leaves we can propagate them upwards level by level, using local information.

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

NC and Parallelism

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.

Unbounded Fan-In

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 realizability considerations (try to design an electronic circuit with unbounded fan-in).

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

Lemma

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

Uniform Versions

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 a_i and b_i be the input bits, a_n the MSD, and c_i the carries, $c_1 = 0$. Note that the standard kindergarten algorithm does not work: it is bounded fan-in, but linear depth.

We need a non-standard method to compute the carries.

Proof

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
- ...

In other words, the carries march down the line.

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

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

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

□

Basics

Lemma (Uniform)

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

Proof.

For $\mathsf{NC}^1 \subseteq \mathbb{L}$ suppose we have a log-depth circuit C and input $\boldsymbol{x} \in \mathbf{2}^n$. We can evaluate the circuit by recursion, essentially performing DFS from the terminal.

The recursion stack has depth $\log n$, and each stack frame contains only a constant number of bits. So this DFS runs in $\mathbb{L}$.

Proof Continued

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

For each length n, consider a digraph whose nodes represent the configurations of $\mathcal{M}$ sans actual input $\boldsymbol{x} = x_1 \ldots x_n$, coded up properly. So we have the state, the contents of the $\log n$ worktape, and the position of the read head, but not the actual input tape. Since $\mathcal{M}$ is $\mathbb{NL}$, the graph is polynomial in size.

Label the edges by x_i or $\overline{x}_i$ or blank to indicate possible compute steps given the corresponding input bit.

Given a concrete input $\boldsymbol{x} \in \mathbf{2}^n$, it now suffices to compute the transitive closure of this graph with appropriate edges, a task that can be handled by a NC^2 circuit.

□

Straight Line 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

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

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

SLP and Polynomials

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

Also, any polynomial can be computed by a SLP.

$v_1 = 1$	1
$v_2 = x$	x
$v_3 = v_1 + v_1$	2
$v_4 = v_2 * v_2$	x^2
$v_5 = v_4 * v_3$	$2x^2$
$v_6 = v_5 - v_1$	$2x^2 - 1$
$v_7 = v_6 * v_2$	$2x^3 - x$
$v_8 = v_7 + v_1$	$2x^3 - x + 1$

Short Programs

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

Addition Chains

Remember from 15-251?

$v_1 = 1$	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

This is the "obvious" SLP; it shows that $\text{slc}(30) \leq 8$.

Is this really the shortest program for 30?

No

$v_1 = 1$	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

So we have $\text{slc}(30) \leq 7$. Is that it?

Yes

$v_1 = 1$	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

It is true that $\text{slc}(30) = 7$, but it takes a bit of effort to prove this.

Also note that the solution is not unique.

Onward

Straight-line programs are already surprisingly complicated, in particular when one starts to ask questions about the minimal SLP to accomplish some particular task.

So here is a great idea: what if we make the programs a little more complicated by allowing a rather feeble if-then-else construct?

Instead of just running through a sequence of simple steps, we ask whether some condition holds and then apply one operation or another.

Branching Programs

Suppose we have a collection of functions $[w] \to [w]$ for some width w and Boolean variables $x_1, \ldots, x_n$.

We can construct bounded width branching programs (BPs) by fixing a sequence of instructions of the form

$$\langle j_i, f_i, g_i \rangle$$

where $1 \leq i \leq \ell$, $1 \leq j_i \leq n$ and $f_i, g_i : [w] \to [w]$. Here ℓ is the length of the program.

Given a BP P and Boolean values for the variables x_j, we define the semantics of $P(\boldsymbol{x})$ to be the function $[w] \to [w]$ obtained by composing the ℓ functions

$$\textbf{if } x_{j_i} \textbf{ then } f_i \textbf{ else } g_i$$

Quoi?

Write I for the identity function on $[2]$, and σ for the transposition.

$$P\colon \quad \langle x, \sigma, I \rangle, \langle y, I, \sigma \rangle, \langle z, \sigma, I \rangle$$

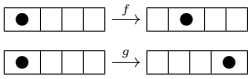
Then $P(x, y, z) = \sigma$ iff $x \oplus \overline{y} \oplus z = 1$.

Given a family of function $\mathcal{F} \subseteq [w] \to [w]$ we can say that P recognizes $\boldsymbol{x} \in \mathbf{2}^n$ if $P(\boldsymbol{x}) \in \mathcal{F}$.

Given a program P_n with n variables for each n allows us to recognize a whole language over $\mathbf{2}$ (family of BPs).

More Quoi

One can think of a width w, input n branching program as a game pushing a pebble around: we have a linear grid of length w, and one pebble initially sitting on square 1.



Executing an instruction means: depending on the input bit, and the current position of the pebble, move it to one place or another (according to some fixed functions on $[w]$).

In the end, accept if the pebble is no longer on square 1.

Intuitively, this seems to be pretty weak for small w like $2, 3, 4$. It's not clear how large w needs to be to get some mileage out of this.

Déjà Vu All Over

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

The automaton consists essentially of two functions $\delta_s : [w] \to [w]$ where w is the state complexity of $\mathcal{A}$, $s \in \mathbf{2}$.

Each binary word determines a composition f of these functions, and acceptance depends on whether $f(q_0) \in F$.

So any DFA is an example of a BP; in fact, we get a whole family of programs, one for each input length.

This is an example of a uniform family: there is one description that works for all n. We could easily build a Turing machine that constructs P_n from 0^n.

Bounds

Families of BPs are only really interesting if we impose a few bounds:

- the width of all the programs should be bounded, and
- the length should be polynomial in n.

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

The DFAs from above are an example of BWBP.

But why are these constraints interesting?

It is known that every language $L \subseteq \mathbf{2}^\star$ can be recognized by a family of BPs of width 4 but exponential length; alternatively, we can achieve linear length at the cost of exponential width.

Algebra to the Rescue

Theorem (Barrington 1989)

The class of languages recognized by BWBP is exactly non-uniform NC^1. *In fact, width 5 suffices.*

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

But the opposite direction relies heavily on group theory and linear algebra over finite fields.

The Message: Algebra can be critically important to study computation.