Motivation

One might imagine that $P \neq \text{NP}$, but $\text{Sat}$ is tractable in the following sense: for every $\ell$ there is a very short program that runs in time $\ell^2$ and correctly treats all instances of size $\ell$. – Karp and Lipton, 1982

Some questions:
- Even if it is hard to find a universal algorithm for solving all instances of a problem, couldn't it still be that there is a simple algorithm for every fixed problem size?
- What can complexity theory tell us about parallel computation?
- Are there any meaningful complexity classes below LogSpace? Do they contain relevant problems?

$\leadsto$ circuit complexity provides some answers

Intuition: use circuits with logical gates to model computation

Boolean Circuits

**Definition 19.1:** A Boolean circuit is a finite, directed, acyclic graph where
- each node that has no predecessor is an input node
- each node that is not an input node is one of the following types of logical gate:
  - AND with two input wires
  - OR with two input wires
  - NOT with one input wire
- one or more nodes are designated output nodes

The outputs of a Boolean circuit are computed in the obvious way from the inputs. $\leadsto$ circuits with $k$ inputs and $\ell$ outputs represent functions $\{0, 1\}^k \rightarrow \{0, 1\}^\ell$

We often consider circuits with only one output.
Example 1

XOR function:

```
   X
  /|
 /  \\
A   A
x1  x2
```

Example 2

Parity function with four inputs:
(true for odd number of 1s)

```
   X
  /|
 /  \\
A   A
x1  x2  x3  x4
```

Alternative Ways of Viewing Circuits (1)

Propositional formulae
- propositional formulae are special circuits:
  - each non-input node has only one outgoing wire
  - each variable corresponds to one input node
  - each logical operator corresponds to a gate
  - each sub-formula corresponds to a wire

```
((¬x1 ∧ x2) ∨ (x1 ∧ ¬x2))
```

Alternative Ways of Viewing Circuits (2)

Straight-line programs
- are programs without loops and branching (if, goto, for, while, etc.)
- that only have Boolean variables
- and where each line can only be an assignment with a single Boolean operator

```
01 z1 := ¬x1
02 z2 := ¬x2
03 z3 := z1 ∧ x2
04 z4 := z2 ∧ x1
05 return z3 ∨ z4
```
Example: Generalised AND
The function that tests if all inputs are 1 can be encoded by combining binary AND gates:

- works similarly for OR gates
- number of gates: \( n - 1 \)
- we can use \( n \)-way AND and OR (keeping the real size in mind)

\[
\prod_{i=1}^{n} x_i
\]

Solving Problems with Circuits
Circuits are not universal: they have a fixed number of inputs!
How can they solve arbitrary problems?

**Definition 19.2:** A circuit family is an infinite list \( C = C_1, C_2, C_3, \ldots \) where each \( C_i \) is a Boolean circuit with \( i \) inputs and one output. We say that \( C \) decides a language \( L \) (over \( \{0, 1\} \)) if
\[
w \in L \quad \text{if and only if} \quad C_n(w) = 1 \quad \text{for} \quad n = |w|.
\]

**Example 19.3:** The circuits we gave for generalised AND are a circuit family that decides the language \( \{1^n \mid n \geq 1\} \).

Circuit Complexity
To measure difficulty of problems solved by circuits, we can count the number of gates needed:

**Definition 19.4:** The size of a circuit is its number of gates.
Let \( f : \mathbb{N} \rightarrow \mathbb{R}^+ \) be a function. A circuit family \( C \) is \( f \)-size bounded if each of its circuits \( C_n \) is of size at most \( f(n) \).

\( \text{Size}(f(n)) \) is the class of all languages that can be decided by an \( O(f(n)) \)-size bounded circuit family.

**Example 19.5:** Our circuits for generalised AND show that \( \{1^n \mid n \geq 1\} \in \text{Size}(n) \).

Examples
Many simple operations can be performed by circuits of polynomial size:
- Boolean functions such as parity (=sum modulo 2), sum modulo \( n \), or majority
- Arithmetic operations such as addition, subtraction, multiplication, division (taking two fixed-arity binary numbers as inputs)
- Many matrix operations
See exercise for some more examples
A natural class of problems to consider are those that have polynomial circuit families:

**Definition 19.6:** \( \mathcal{P}_{\text{poly}} = \bigcup_{d \geq 1} \text{Size}(n^d). \)

**Note:** A language is in \( \mathcal{P}_{\text{poly}} \) if it is solved by some polynomial-sized circuit family. There may not be a way to compute (or even finitely represent) this family.

How does \( \mathcal{P}_{\text{poly}} \) relate to other classes?

---

**Theorem 19.7:** For \( f(n) \geq n \), we have \( \text{DTime}(f) \subseteq \text{Size}(f^2) \).

**Proof sketch (see also Sipser, Theorem 9.30)**

- We can represent the DTime computation as in the proof of Theorem 16.10: as a list of configurations encoded as words
  
  \[ * \sigma_1 \cdots \sigma_{m-1} \langle q, \sigma_i \rangle \sigma_{i+1} \cdots \sigma_m * \]

  of symbols from the set \( \Omega = \{ * \} \cup \Gamma \cup (Q \times \Gamma) \).

- Tableau (i.e., grid) with \( O(f^2) \) cells.

- We can describe each cell with a list of bits (wires in a circuit).

- We can compute one configuration from its predecessor by \( O(f) \) circuits (idea: compute the value of each cell from its three upper neighbours as in Theorem 16.10)

- Acceptance can be checked by assuming that the TM returns to a unique configuration position/state when accepting.

---

**From Polynomial Time to Polynomial Size**

From \( \text{DTime}(f) \subseteq \text{Size}(f^2) \) we get:

**Corollary 19.8:** \( \mathcal{P} \subseteq \mathcal{P}_{\text{poly}}. \)

This suggests another way of approaching the P vs. NP question:

If any language in \( \mathcal{NP} \) is not in \( \mathcal{P}_{\text{poly}} \), then \( \mathcal{P} \neq \mathcal{NP} \).

(but nobody has found any such language yet)
**Theorem 19.9**: Circuit-Sat is NP-complete.

**Proof**: Inclusion in NP is easy (just guess the input).

For NP-hardness, we use that NP problems are those with a P-verifier:

- The DTM simulation of Theorem 19.7 can be used to implement a verifier (input: \( w \# c \) in binary)
- We can hard-wire the \( w \)-inputs to use a fixed word instead (remaining inputs: \( c \))
- The circuit is satisfiable iff there is a certificate for which the verifier accepts \( w \) □

Note: It would also be easy to reduce Sat to Circuit-Sat, but the above yields a proof from first principles.

---

**A New Proof for Cook-Levin**

**Theorem 19.10**: 3Sat is NP-complete.

**Proof**: Membership in NP is again easy (as before).

For NP-hardness, we express the circuit that was used to implement the verifier in Theorem 19.9 as propositional logic formula in 3-CNF:

- Create a propositional variable \( X \) for every wire in the circuit
- Add clauses to relate input wires to output wires, e.g., for AND gate with inputs \( X_1 \) and \( X_2 \) and output \( X_3 \), we encode \((X_1 \land X_2) \iff X_3\) as:
  \[
  (\neg X_1 \lor \neg X_2 \lor X_3) \land (X_1 \lor \neg X_3) \land (X_2 \lor \neg X_3)
  \]
- Fixed number of clauses per gate = constant factor size increase
- Add a clause \((X)\) for the output wire \( X \) □

---

**Is P = P/poly?**

We showed \( P \subseteq P/\text{poly} \). Does the converse also hold?

No!

**Theorem 19.11**: \( P/\text{poly} \) contains undecidable problems.

**Proof**: We define the unary Halting problem as the (undecidable) language:

\[
\text{UHal} := \{1^n \mid \text{the binary encoding of } n \text{ encodes a pair } (M, w) \text{ where } M \text{ is a TM that halts on word } w\}
\]

For a number \( 1^n \in \text{UHal} \), let \( C_n \) be the circuit that computes a generalised AND of all inputs. For all other numbers, let \( C_n \) be a circuit that always returns 0. The circuit family \( C_1, C_2, C_3, \ldots \) accepts UHal. □
Uniform Circuit Families

$P/\text{poly}$ is too powerful, since we do not require the circuits to be computable. We can add this requirement:

**Definition 19.12:** A circuit family $C_1, C_2, C_3, \ldots$ is log-space-uniform if there is a log-space computable function that maps words $1^n$ to (an encoding of) $C_n$.

**Note:** We could also define similar notions of uniformity for other complexity classes.

**Theorem 19.13:** The class of all languages that are accepted by a log-space-uniform circuit family of polynomial size is exactly $P$.

**Proof sketch:** A detailed analysis shows that our earlier reduction of polytime DTM to circuits is log-space-uniform. Conversely, a polynomial-time procedure can be obtained by first computing a suitable circuit (in log-space) and then evaluating it (in polynomial time). □

Summary and Outlook

Circuits provide an alternative model of computation

$P \subseteq P/\text{poly}$

**Circuit-Sat** is NP-complete.

$P/\text{poly}$ is very powerful – uniform circuit families help to restrict it

What’s next?

- Circuits for parallelism
- Complexity classes (strictly!) below $P$
- Randomness

Turing Machines That Take Advice

One can also describe $P/\text{poly}$ using TMs that take “advice”:

**Definition 19.14:** Consider a function $a: \mathbb{N} \rightarrow \mathbb{N}$. A language $L$ is accepted by a Turing Machine $M$ with a bits of advice if there is a sequence of advice strings $\alpha_0, \alpha_1, \alpha_2, \ldots$ of length $|\alpha_i| = a(i)$ and $M$ accepts inputs of the form $(w \# \alpha_0)$ if and only if $w \in L$.

$P/\text{poly}$ is equivalent to the class of problems that can be solved by a PTime TM that takes a polynomial amount of “advice” (where the advice can be a description of a suitable circuit).

(This is where the notation $P/\text{poly}$ comes from.)