Computing with Circuits
Motivation

One might imagine that P $\neq$ 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
Motivation

One might imagine that $P \neq NP$, but $Sat$ 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?
One might imagine that $P \neq NP$, 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?

$\sim$ circuit complexity provides some answers

**Intuition:** use circuits with logical gates to model computation
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.

\[ \text{circuits with } k \text{ inputs and } \ell \text{ outputs represent functions } \{0, 1\}^k \rightarrow \{0, 1\}^\ell \]

We often consider circuits with only one output.
Example 1

XOR function:

\[
\begin{align*}
&x_1 \\
&\wedge \\
&\wedge \\
&\neg \\
&\neg
\end{align*}
\]
Example 1

**XOR function:**

![XOR function diagram]

Markus Krötzsch, 6th Jan 2020  Complexity Theory  slide 5 of 23
Example 2

Parity function with four inputs:

\( x_1 \land x_2 \land x_3 \land x_4 \)

Markus Krötzsch, 6th Jan 2020 Complexity Theory slide 6 of 23
Example 2

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

\[((\neg x_1 \land x_2) \lor (x_1 \land \neg x_2))\]
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

$n$-line programs correspond to $n$-gate circuits

\[ \begin{align*}
01 & \quad z_1 := \neg x_1 \\
02 & \quad z_2 := \neg x_2 \\
03 & \quad z_3 := z_1 \land x_2 \\
04 & \quad z_4 := z_2 \land x_1 \\
05 & \quad \text{return } z_3 \lor z_4
\end{align*} \]
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)
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 $w \in L$ if and only if $C_n(w) = 1$ for $n = |w|$. 

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

$$w \in L \quad \text{if and only if} \quad C_n(w) = 1 \text{ for } n = |w|.$$

**Example 19.3:** The circuits we gave for generalised AND are a circuit family that decides the language $\{1^n \mid n \geq 1\}$. 
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 $f : \mathbb{N} \to \mathbb{R}^+$ 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
Polynomial Circuits
A natural class of problems to consider are those that have polynomial circuit families:

**Definition 19.6:** $P_{/\text{poly}} = \bigcup_{d \geq 1} \text{Size}(n^d)$.

**Note:** A language is in $P_{/\text{poly}}$ 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 $P_{/\text{poly}}$ relate to other classes?
Theorem 19.7: For $f(n) \geq n$, we have $\text{DTime}(f) \subseteq \text{Size}(f^2)$.
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

\[ * \sigma_1 \cdots \sigma_{i-1} \langle q, \sigma_i \rangle \sigma_{i+1} \cdots \sigma_m * \]

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

\[ \sim \text{Tableau (i.e., grid) with } O(f^2) \text{ cells.} \]

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

**Corollary 19.8:** $\mathsf{P} \subseteq \mathsf{P/poly}$. 
From $\text{DTime}(f) \subseteq \text{Size}(f^2)$ we get:

**Corollary 19.8:** $P \subseteq P_{/\text{poly}}$.

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

If any language in NP is not in $P_{/\text{poly}}$, then $P \neq NP$.

(but nobody has found any such language yet)
Circuit-Sat

Input: A Boolean Circuit $C$ with one output.
Problem: Is there any input for which $C$ returns 1?

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.
**Circuit-Sat**

Input: A Boolean Circuit $C$ with one output.

Problem: Is there any input for which $C$ returns 1?

**Theorem 19.9:** Circuit-Sat is NP-complete.
**Theorem 19.9:** \textsc{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 \(w\) \(\square\)

**Note:** It would also be easy to reduce \textsc{Sat} to \textsc{Circuit-Sat}, but the above yields a proof from first principles.
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 $(X_1 \land X_2) \leftrightarrow X_3$ 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 $X$. 

Markus Krötzsch, 6th Jan 2020
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 $(X_1 \land X_2) \leftrightarrow X_3$ 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 $X$
The Power of Circuits
Is $P = P_{\text{poly}}$?

We showed $P \subseteq P_{\text{poly}}$. Does the converse also hold?

Theorem 19.11: $P_{\text{poly}}$ contains undecidable problems.

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

$$U_{\text{Halting}} = \{1^n | \text{the binary encoding of } n \text{ encodes a pair } \langle M, w \rangle \text{ where } M \text{ is a TM that halts on word } w\}$$

For a number $1^n \in U_{\text{Halting}}$, 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 $C_1, C_2, C_3, ...$ accepts $U_{\text{Halting}}$. □
Is $P = P_{\text{poly}}$?

We showed $P \subseteq P_{\text{poly}}$. Does the converse also hold?

No!

**Theorem 19.11:** $P_{\text{poly}}$ contains undecidable problems.
Is $P = P_{\text{poly}}$?

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:

$$U_{\text{Halting}} := \{1^n \mid \text{the binary encoding of } n \text{ encodes a pair } \langle M, w \rangle \text{ where } M \text{ is a TM that halts on word } w\}$$

For a number $1^n \in U_{\text{Halting}}$, 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 $U_{\text{Halting}}$. □
P\text{\textsubscript{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.
$P_{/\text{poly}}$ 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 DTMs 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). □
One can also describe \( P_{/poly} \) 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 \( (w \# |w|) \) if and only if \( w \in L \).

\( P_{/poly} \) 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 \( P_{/poly} \) comes from.)
Summary and Outlook

Circuits provide an alternative model of computation

\[ P \subseteq P_{/\text{poly}} \]

\textsc{Circuit-Sat} is NP-complete.

\[ P_{/\text{poly}} \] is very powerful – uniform circuit families help to restrict it

\textbf{What's next?}

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