Computational Complexity Theory

Lecture 9: Boolean circuits; Karp-Lipton theorem; class AC and NC

Department of Computer Science,
Indian Institute of Science

An algorithm for every input length?

- *"One might imagine that P ≠ NP, but SAT is tractable in the following sense: for every ℓ there is a very short program that runs in time ℓ^2 and correctly treats all instances of size ℓ."* — Karp and Lipton (1982).

An algorithm for every input length?

- *"One might imagine that P ≠ NP, but SAT is tractable in the following sense: for every ℓ there is a very short program that runs in time ℓ^2 and correctly treats all instances of size ℓ."* — Karp and Lipton (1982).

- P ≠ NP rules out the existence of a single efficient algorithm for SAT that handles all input lengths. But, it doesn't rule out the possibility of having a sequence of efficient SAT algorithms – one for each input length.

Lesson learnt from Cook-Levin

- Locality of computation implies that an algorithm A working on inputs of some fixed length n and running in time $T(n)$ can be viewed as a Boolean circuit ϕ of size $O(T(n)^2)$ s.t. $A(x) = \phi(x)$ for every $x \in \{0,1\}^n$.
- On the other hand, a circuit on inputs of length n and of size S can be viewed as an algorithm working on length n inputs and running in time S.

Lesson learnt from Cook-Levin

- Locality of computation implies that an algorithm A working on inputs of some fixed length n and running in time $T(n)$ can be viewed as a Boolean circuit ϕ of size $O(T(n)^2)$ s.t. $A(x) = \phi(x)$ for every $x \in \{0,1\}^n$.
- On the other hand, a circuit on inputs of length n and of size S can be viewed as an algorithm working on length n inputs and running in time S.

- To rule the existence of a sequence of algorithms – one for each input length – we need to rule out the existence of a sequence of (i.e., a family of) circuits.

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

Nodes with out-degree zero are the output gates.

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

  Nodes with out-degree zero are the output gates.

- Typically, we'll consider circuits with one output gate, and with nodes having in-degree at most two.

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

  Nodes with out-degree zero are the output gates.

- **Size** of circuit is the no. of edges in it. **Depth** is the length of the longest path from an i/p to o/p node.

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

  Nodes with out-degree zero are the output gates.

θ(no. of nodes)

- **Size** of circuit is the no. of edges in it. **Depth** is the length of the longest path from an i/p to o/p node.

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

  Nodes with out-degree zero are the output gates.

- **Size** corresponds to "sequential time complexity". **Depth** corresponds to "parallel time complexity".

Boolean circuits

- A Boolean circuit is a directed acyclic graph whose nodes/gates are labelled as follows:
  - A node with in-degree zero is labelled by an input variable, and it outputs the value of the variable.
  - Any other node is labelled by one of the three operations $\wedge$, $\vee$, $\neg$, and it outputs the value of the operation on its input.

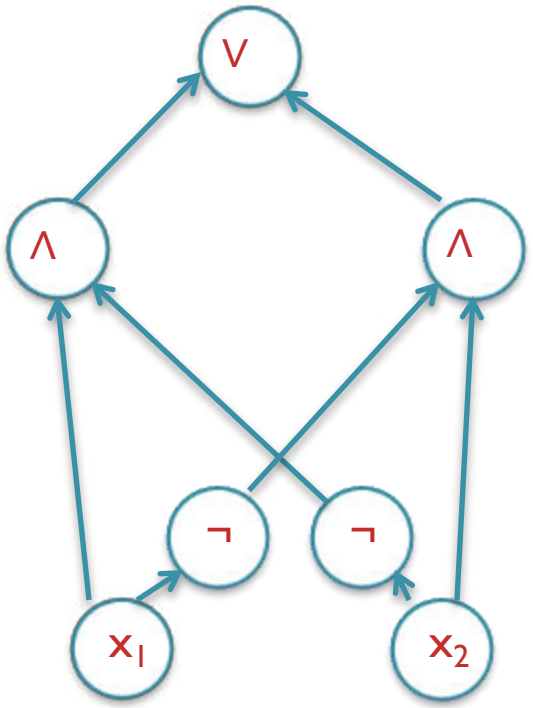
  Nodes with out-degree zero are the output gates.

- If every node in a circuit has out-degree at most one, then the circuit is called a **formula**.

A circuit for Parity

- PARITY$(x_1, x_2, \ldots, x_n) = x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

$x_1 \oplus x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$



ϕ

Size(ϕ) = $|\phi|$ = 8
Depth(ϕ) = 3

Circuit family

- Let $T: \mathbb{N} \to \mathbb{N}$ be some function.
- Definition: A $T(n)$-size circuit family is a set of circuits $\{C_n\}_{n \in \mathbb{N}}$ such that C_n has n inputs and $|C_n| \leq T(n)$.

Class P/poly

- Let $T: \mathbb{N} \to \mathbb{N}$ be some function.
- Definition: A $T(n)$-size circuit family is a set of circuits $\{C_n\}_{n \in \mathbb{N}}$ such that C_n has n inputs and $|C_n| \leq T(n)$.

- Definition: A language L is in $SIZE(T(n))$ if there's a $T(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$ such that

$$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

- Defintion: Class $P/poly = \bigcup_{c \geq 1} SIZE(n^c)$.

Class P/poly

- Let $T: \mathbb{N} \to \mathbb{N}$ be some function.
- Definition: A $T(n)$-size circuit family is a set of circuits $\{C_n\}_{n \in \mathbb{N}}$ such that C_n has n inputs and $|C_n| \leq T(n)$.

- Definition: A language L is in $SIZE(T(n))$ if there's a $T(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$ such that

$$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

- Defintion: Class $P/poly = \bigcup_{c \geq 1} SIZE(n^c)$.

The circuit family $\{C_n\}_{n \in \mathbb{N}}$ *decides* L, i.e., C_n *decides* $L \cap \{0,1\}^n$.

Class P/poly

- Let $T: \mathbb{N} \to \mathbb{N}$ be some function.
- Definition: A $T(n)$-size circuit family is a set of circuits $\{C_n\}_{n \in \mathbb{N}}$ such that C_n has n inputs and $|C_n| \leq T(n)$.

- Definition: A language L is in $SIZE(T(n))$ if there's a $T(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$ such that

$$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

- Defintion: Class $P/poly = \bigcup_{c \geq 1} SIZE(n^c)$.

Alternatively, we say C_n *computes* the characteristic function of $L \cap \{0,1\}^n$.

Class P/poly

- Observation: $P \subseteq P/poly$.
- Proof. If $L \in P$, then there's a n^c-time TM that decides L for some constant c. By Cook-Levin, there's a $O(n^{2c})$-size circuit family $\{C_n\}_{n \in N}$ such that

$$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

Class P/poly

- Observation: $P \subseteq P/poly$.
- Proof. If $L \in P$, then there's a n^c-time TM that decides L for some constant c. By Cook-Levin, there's a $O(n^{2c})$-size circuit family $\{C_n\}_{n \in N}$ such that

  $$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

  (Note: C_n is poly(n)-time computable from 1^n.)

- Is $P = P/poly$?

Class P/poly

- Observation: $P \subseteq P/poly$.
- Proof. If $L \in P$, then there's a n^c-time TM that decides L for some constant c. By Cook-Levin, there's a $O(n^{2c})$-size circuit family $\{C_n\}_{n \in N}$ such that

  $$x \in L \iff C_n(x) = 1, \text{ where } n = |x|.$$

  (Note: C_n is poly(n)-time computable from 1^n.)

- Is $P = P/poly$? No! P/poly contains undecidable languages.

Class P/poly

- Let HALT = {(M,y) : M halts on input y}. HALT is an undecidable language.
- Notation. #(M,y) = number corresponding to the binary string (M,y).
- Let UHALT = $\{1^{\#(M,y)} : (M,y) \in HALT\}$. Then, UHALT is also an undecidable language.

Class P/poly

- Let HALT = {(M,y) : M halts on input y}. HALT is an undecidable language.
- Notation. #(M,y) = number corresponding to the binary string (M,y).
- Let UHALT = $\{1^{\#(M,y)} : (M,y) \in \text{HALT}\}$. Then, UHALT is also an undecidable language.

- Obs. Any unary language is in P/poly. (*Homework*)
  Hence, P $\subsetneq$ P/poly .

Class P/poly

- What makes P/poly contain undecidable languages? *Ans:* $L \in$ P/poly implies that L is decided by a circuit family $\{C_n\}$, where $|C_n| = n^{O(1)}$. We don't require that C_n is poly-time computable from 1^n.

Class P/poly

- What makes P/poly contain undecidable languages? *Ans:* $L \in$ P/poly implies that L is decided by a circuit family $\{C_n\}$, where $|C_n| = n^{O(1)}$. We don't require that C_n is poly-time computable from 1^n.
- P/poly is a *non-uniform class* as a language in this class is allowed to have different algorithms/circuits for different input lengths.
- P is a *uniform class* as a language in this class has one algorithm for all inputs.

Class P/poly

- What makes P/poly contain undecidable languages? *Ans:* $L \in$ P/poly implies that L is decided by a circuit family $\{C_n\}$, where $|C_n| = n^{O(1)}$. We don't require that C_n is poly-time computable from 1^n.
- P/poly is a *non-uniform class* as a language in this class is allowed to have different algorithms/circuits for different input lengths.
- P is a *uniform class* as a language in this class has one algorithm for all inputs.

Hardware	Software
TM (uniform)	Algo/Enc. of TM
Circuits (non-uniform)	An algo per i/p length

Class P/poly

- What makes P/poly contain undecidable languages? *Ans:* $L \in P/poly$ implies that L is decided by a circuit family $\{C_n\}$, where $|C_n| = n^{O(1)}$. We don't require that C_n is poly-time computable from 1^n.
- P/poly is a *non-uniform class* as a language in this class is allowed to have different algorithms/circuits for different input lengths.
- P is a *uniform class* as a language in this class has one algorithm for all inputs.
- Is $SAT \in P/poly$? In other words, is $NP \subsetneq P/poly$?

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. We'll show that NP $\subsetneq$ P/poly implies $\prod_2 = \sum_2$. It's sufficient to show that $\prod_2 \subseteq \sum_2$.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)}\ \exists u_2 \in \{0,1\}^{q(|x|)}\ M(x,u_1,u_2) = 1$.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)}\ \exists u_2 \in \{0,1\}^{q(|x|)}\ M(x,u_1,u_2) = 1.$
- Goal. Come up with a polynomial function p(.) and a poly-time TM N s.t.

  $x \in L \iff \exists v_1 \in \{0,1\}^{p(|x|)}\ \forall v_2 \in \{0,1\}^{p(|x|)}\ N(x,u_1,u_2) = 1.$
- Think about designing such a TM N.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If $NP \subsetneq P/poly$ then $PH = \sum_2$.
- Proof. Let $L \in \prod_2$. There's a polynomial function $q(.)$ and a poly-time TM M s.t.

  by Cook-Levin

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)}\ \exists u_2 \in \{0,1\}^{q(|x|)}\ \phi(x,u_1,u_2) = 1.$

- If M runs in time $T(n) = n^{O(1)}$ on (x,u_1,u_2), where $|x| = n$, then $|\phi| = O(T(n)^2)$. Let m = #(bits to write ϕ).
- N can compute ϕ from M in $poly(|x|)$ time.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = Σ_2.
- Proof. Let L $\in \Pi_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  by Cook-Levin

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)}\ \exists u_2 \in \{0,1\}^{q(|x|)}\ \phi(x,u_1,u_2) = 1.$

- If M runs in time $T(n) = n^{O(1)}$ on (x,u_1,u_2), where $|x| = n$, then $|\phi| = O(T(n)^2)$. Let m = length of ϕ.
- N can compute ϕ from M in poly(|x|) time.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

$$x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \exists u_2 \in \{0,1\}^{q(|x|)} \phi(x,u_1,u_2) = 1.$$

$\phi(x,u_1,u_2)$ as a function of u_2 is satisfiable. Wlog ϕ is a CNF (why?).

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1,u_2) \in SAT$.
- By assumption, SAT $\in$ P/poly, i.e., there's a circuit C_m of size $p(m) = m^{O(1)}$ that correctly decides satifiability of all input circuits ϕ of length m.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1,u_2) \in SAT$.
- First attempt. A $\sum_2$ statement to capture membership of strings in L.

  $x \in L \iff \exists C_m \in \{0,1\}^{p(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ C_m(\phi(x,u_1,u_2))=1$.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1,u_2) \in SAT$.
- First attempt. A $\sum_2$ statement to capture membership of strings in L.

  $x \in L \iff \exists C_m \in \{0,1\}^{p(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ C_m(\phi(x,u_1,u_2))=1$.

- Wrong! Think about a C_m that always outputs 1.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1, u_2) \in SAT$.
- First attempt. A $\sum_2$ statement to capture membership of strings in L.

  $x \in L \iff \exists C_m \in \{0,1\}^{p(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ C_m(\phi(x,u_1, u_2))=1$.

- Need to be sure that C_m is the right circuit.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = Σ_2.
- Proof. Let L $\in \Pi_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)}\ \phi(x,u_1,u_2) \in SAT$.
- If there's a circuit C_m of size $m^{O(1)}$ that correctly decides satifiability of all input circuits ϕ of length m, then by self-reducibility of SAT, there's a multi-output circuit D_m of size $r(m) = m^{O(1)}$ that outputs a *satisfying assignment* for input ϕ if $\phi \in SAT$. *(Homework)*

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1,u_2) \in SAT$.

- A $\sum_2$ statement to capture membership in L.

  $x \in L \iff$

  $\exists D_m \in \{0,1\}^{r(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1, \underbrace{D_m(\phi(x,u_1,u_2))}) = 1$.

  assignment to the u_2 variables

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \iff \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1, u_2) \in SAT.$

- A $\sum_2$ statement to capture membership in L.

  $x \in L \iff$

  $\exists D_m \in \{0,1\}^{r(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ \underbrace{\phi(x,u_1, D_m(\phi(x,u_1, u_2)) = 1}.$

  Can be checked by a poly-time TM N.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If NP $\subsetneq$ P/poly then PH = $\sum_2$.
- Proof. Let L $\in \prod_2$. There's a polynomial function q(.) and a poly-time TM M s.t.

  $x \in L \Leftrightarrow \forall u_1 \in \{0,1\}^{q(|x|)} \ \phi(x,u_1, u_2) \in SAT$.
- A $\sum_2$ statement to capture membership in L.

  $x \in L \Leftrightarrow$

  $\exists D_m \in \{0,1\}^{r(m)} \ \forall u_1 \in \{0,1\}^{q(|x|)} \ N(x, D_m, u_1) = 1$.

Karp-Lipton theorem

- Theorem (*Karp & Lipton 1982*). If $NP \subsetneq P/poly$ then $PH = \sum_2$.

- If we can show $NP \not\subset P/poly$ assuming $P \neq NP$, then

  $NP \not\subset P/poly \iff P \neq NP$.

- Karp-Lipton theorem shows $NP \subsetneq P/poly$ assuming the stronger statement $PH \neq \sum_2$.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly?

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Follows from a counting argument.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.
- Number of bits required to write the adjacency lists it at most $s(\log s + 3) + 4(s + n) \leq 9s.\log s$.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.
- Number of circuits of size s is at most $3^s.2^{9s.\log s}$.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.
- Number of circuits of size s is at most $2^{11 s.\log s}$.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.
- Number of circuits of size s is at most $\exp(2^{n-1})$.
- Number of functions in n variables is $\exp(2^n)$.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many. Let $\exp(m) = 2^m$.
- Theorem. $1 - \exp(-2^{n-1})$ fraction of Boolean functions on n variables **do not** have circuits of size $2^n/(22n)$.
- Proof. Let $s = 2^n/(22n)$. A circuit of size s has at most s internal nodes. It can be specified by giving the labels of the internal nodes and the adjacency lists.
- So, circuits of size s can compute at most $\exp(-2^{n-1})$ fraction of all Boolean functions on n variables.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many.
- Is one out of so many functions outside P/poly in NP?

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many.
- Is one out of so many functions outside P/poly in NP? We don't know even after ~40 yrs of research!

- Theorem. *(Iwama, Lachish, Morizumi & Raz 2002)* There is a language $L \in NP$ such that any circuit C_n that decides $L \cap \{0,1\}^n$ requires $5n - o(n)$ many $\wedge$ and $\vee$ gates.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many.
- Is one out of so many functions outside P/poly in NP? We don't know even after ~40 yrs of research!

- Theorem. *(Iwama, Lachish, Morizumi & Raz 2002)* There is a language $L \in NP$ such that any circuit C_n that decides $L \cap \{0,1\}^n$ requires $5n - o(n)$ many $\wedge$ and $\vee$ gates.

Results of this kind are known as circuit lower bound.

Functions outside P/poly

- Are there Boolean functions (i.e., languages) outside P/poly? Yes! There are many.
- Is one out of so many functions outside P/poly in NP? We don't know even after ~40 yrs of research!

- Open problem. Prove that NEXP $\not\subset$ P/poly .

Lower bounds for restricted circuits

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

Lower bounds for restricted circuits

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

- Fact. PARITY(x_1, x_2, …, x_n) can be computed by a circuit of size $O(n)$ and a formula of size $O(n^2)$.

Homework

Lower bound for Boolean formulas

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

- Theorem. *(Khrapchenko 1971)* Any formula computing PARITY($x_1, x_2, \ldots, x_n$) has size $\Omega(n^2)$.

Lower bound for Boolean formulas

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

- Theorem. (*Andreev 1987, Hastad 1998*) There's a f that can be computed by a $O(n)$-size circuit such that any formula computing f has size $\Omega(n^{3-o(1)})$.

Technique: Shrinkage of formulas under random restrictions (*Subbotovskaya 1961*).

Lower bound for Boolean formulas

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

- Conjecture. There's a f that can be computed by a $O(n)$-size circuit such that any formula computing f has size $n^{\omega(1)}$.

An interesting approach was given by *Karchmer, Raz & Wigderson (1995)* .

LB for AC^0 and monotone circuits

- Nevertheless, the clean combinatorial structure of a circuit has been used to prove lower bounds for some natural classes of circuits.
- The proofs of these lower bounds introduced and developed some highly interesting techniques.

- We'll discuss lower bounds for constant depth circuits and monotone circuits in the next 2 lectures.

Non-uniform size hierarchy

- Shanon's result. There's a constant $c \geq 1$ such that every Boolean function in n variables has a circuit of size at most $c.(2^n/n)$.

- Theorem. There's a constant $d \geq 1$ s.t. if $T_1: N \to N$ & $T_2: N \to N$ and $T_1(n) \leq d^{-1}.T_2(n) \leq T_2(n) \leq c.(2^n/n)$ then

$$SIZE(T_1(n)) \subsetneq SIZE(T_2(n)).$$

Non-uniform size hierarchy

- Shanon's result. There's a constant $c \geq 1$ such that every Boolean function in n variables has a circuit of size at most $c.(2^n/n)$.

- Theorem. There's a constant $d \geq 1$ s.t. if $T_1: N \to N$ & $T_2: N \to N$ and $T_1(n) \leq d^{-1}.T_2(n) \leq T_2(n) \leq c.(2^n/n)$ then

$$SIZE(T_1(n)) \subsetneq SIZE(T_2(n)).$$

- Proof. Uses Shanon's result and a counting argument.

(*Homework*)

Class NC^i and AC^i

Class NC

- NC stands for Nick's Class – named after Nick Pippenger.
- Definition. For $i \in \mathbb{N}$, a language L is in NC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $NC = \bigcup_{i \in \mathbb{N}} NC^i$.

Class NC

- NC stands for Nick's Class – named after Nick Pippenger.
- Definition. For $i \in \mathbb{N}$, a language L is in NC^i if there is a polynomial function q(.) and a constant c s.t. L is decided by a q(n)-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $NC = \bigcup_{i \in \mathbb{N}} NC^i$.

- *Homework:* PARITY is in NC^1.

Class NC

- NC stands for Nick's Class – named after Nick Pippenger.
- Definition. For $i \in \mathbb{N}$, a language L is in NC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $NC = \bigcup_{i \in \mathbb{N}} NC^i$.

- NC^1 = *poly(n)*-size Boolean formulas. (*Assignment*)

Class NC

- NC stands for Nick's Class – named after Nick Pippenger.
- Definition. For $i \in \mathbb{N}$, a language L is in NC^i if there is a polynomial function q(.) and a constant c s.t. L is decided by a q(n)-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Further, L is in *log-space uniform* NC^i if C_n is implicitly log-space computable from 1^n.

Note: Sometimes NC^i is defined as log-space uniform NC^i.

Class NC

- NC stands for Nick's Class – named after Nick Pippenger.
- Definition. For $i \in \mathbb{N}$, a language L is in NC^i if there is a polynomial function q(.) and a constant c s.t. L is decided by a q(n)-size circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Further, L is in *log-space uniform* NC^i if C_n is implicitly log-space computable from 1^n.

log-space uniform $NC \subseteq P$.

NC ≡ Efficient parallel computation

- Definition. A language L can be decided *efficiently in parallel* if there's a polynomial function q(.) and constants c & i s.t. $L \cap \{0,1\}^n$ can be decided using q(n) many processors in $c.(\log n)^i$ time.

NC ≡ Efficient parallel computation

- Definition. A language L can be decided *efficiently in parallel* if there's a polynomial function q(.) and constants c & i s.t. $L \cap \{0,1\}^n$ can be decided using q(n) many processors in $c.(\log n)^i$ time.
- Assumptions on the parallel computation model:
  - A processor can deliver a message to any other processor in O(log n) time.
  - A processor has O(log n) bits of memory and performs a poly-time computation at every step.
  - Processor computation steps are synchronized.

NC ≡ Efficient parallel computation

- Definition. A language L can be decided *efficiently in parallel* if there's a polynomial function q(.) and constants c & i s.t. $L \cap \{0,1\}^n$ can be decided using q(n) many processors in $c.(\log n)^i$ time.

- Observation. A language L is in NC if and only if L can be decided efficiently in parallel.
- Proof. Almost immediate from the assumptions on the parallel computation model.

Class AC

- Definition. For $i \in \mathbb{N} \cup \{0\}$, a language L is in AC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size **unbounded fan-in** circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $AC = \bigcup_{i \geq 0} AC^i$. (stands for *Alternating Class*)

Class AC

- Definition. For $i \in \mathbb{N} \cup \{0\}$, a language L is in AC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size **unbounded fan-in** circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $AC = \bigcup_{i \geq 0} AC^i$.

- Observation. $AC^i \subseteq NC^{i+1} \subseteq AC^{i+1}$ for all $i \geq 0$.

Replace an unbounded fan-in gate by a binary tree of bounded fan-in gates.

Class AC

- Definition. For $i \in \mathbb{N} \cup \{0\}$, a language L is in AC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size **unbounded fan-in** circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $AC = \bigcup_{i \geq 0} AC^i$.

- Observation. $NC = AC$.

Class AC

- Definition. For $i \in \mathbb{N} \cup \{0\}$, a language L is in AC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size **unbounded fan-in** circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $AC = \bigcup_{i \geq 0} AC^i$.

- In the next lecture, we'll show that PARITY is not in AC^0, i.e., $AC^0 \subsetneq NC^1$.

Class AC

- Definition. For $i \in \mathbb{N} \cup \{0\}$, a language L is in AC^i if there is a polynomial function $q(.)$ and a constant c s.t. L is decided by a $q(n)$-size **unbounded fan-in** circuit family $\{C_n\}_{n \in \mathbb{N}}$, where depth of C_n is at most $c.(\log n)^i$ for every $n \in \mathbb{N}$.

- Definition. $AC = \bigcup_{i \geq 0} AC^i$.

- Further, L is in *log-space uniform* AC^i if C_n is implicitly log-space computable from 1^n.

log-space uniform $AC \subseteq P$.

P-completeness

P-completeness

- Recall, to define completeness of a complexity class, we need an appropriate notion of a *reduction*.
- What kind of reductions will be suitable is guided by *a complexity question*, like a comparison between the complexity class under consideration & another class.
- Is P = (uniform) NC? Is P = L?…use log-space reduction!

- Definition. A language $L \in P$ is P-*complete* if for every L' in P , $L' \leq_l L$.

P-complete problems

- Circuit value problem. Given a circuit and an input, compute the output of the circuit. (The reduction in the Cook-Levin theorem can be made a log-space reduction.)

- Linear programming. Check the feasibility of a system of linear inequality constraints over rationals. (Assignment problem)

- CFG membership. Given a context-free grammar and a string, decide if the string can be generated by the grammar.

No log-space algo for PC problems

- Theorem. Let L be a P-complete language. Then,

  L is in L $\Leftrightarrow$ P = L.

- Proof. Easy.

- Can't hope to get a log-space algorithm for a P-complete problem unless P = L.

No parallel algo for PC problems

- Theorem. Let L be a P-complete language. Then,

  L is in NC $\Leftrightarrow$ P $\subseteq$ NC.

- Proof. $\Leftarrow$ direction is straightforward.

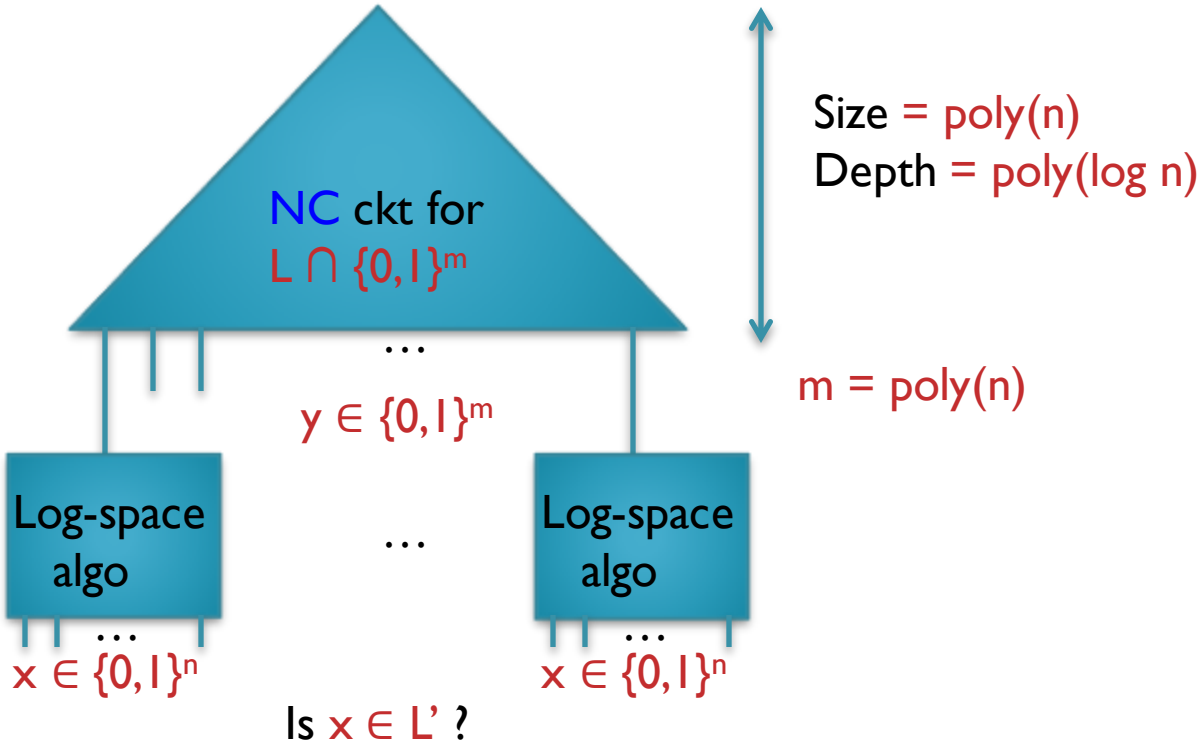
- Can't hope to get an efficient parallel algorithm for a P-complete problem unless P $\subseteq$ NC.

No parallel algo for PC problems

- Theorem. Let L be a P-complete language. Then,

  L is in NC $\iff$ P $\subseteq$ NC.

- Proof.($\Rightarrow$) Let L' $\in$ P. As L is P-complete, L' $\leq_l$ L.

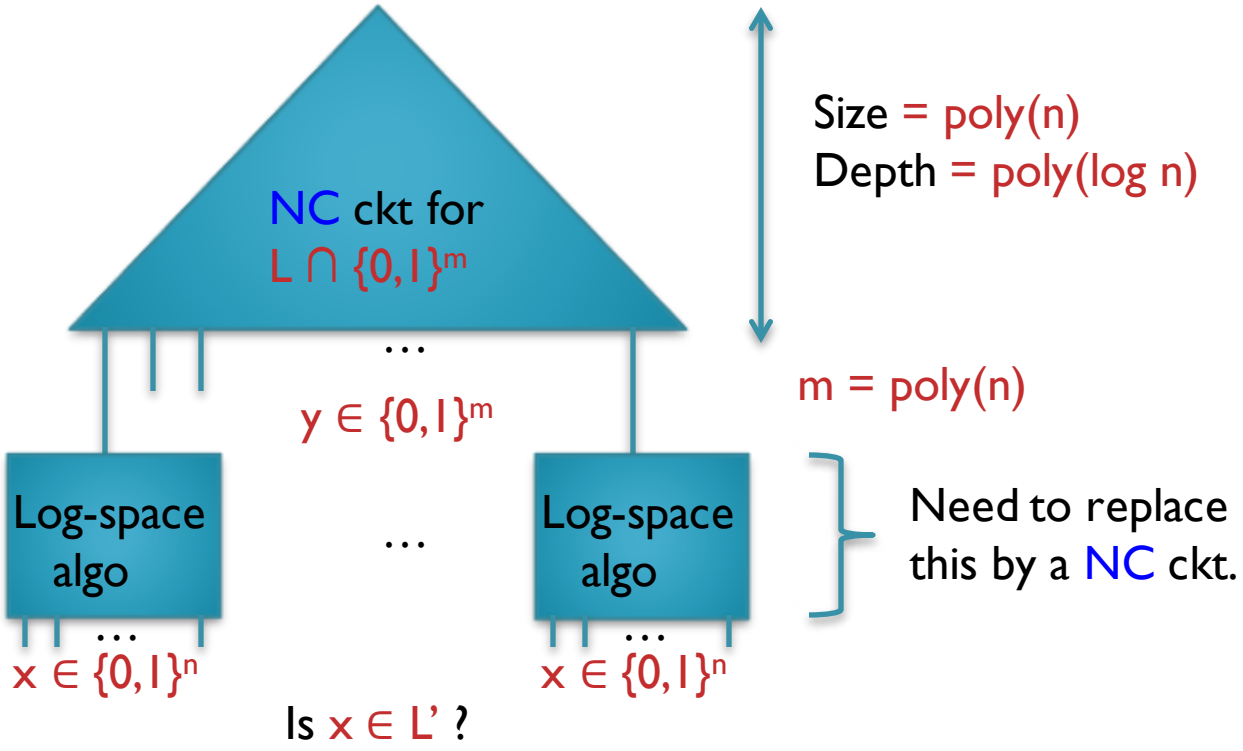


No parallel algo for PC problems

- Theorem. Let L be a P-complete language. Then,

  L is in NC $\iff$ P $\subseteq$ NC.

- Proof.($\Rightarrow$) Let L' $\in$ P. As L is P-complete, L' $\leq_l$ L.



Parallelization of Log-space

- Do problems in L have efficient parallel algorithms? Yes!

- Theorem. NL ⊆ (uniform) NC. *(Assignment problem)*

Parallelization of Log-space

- Do problems in L have efficient parallel algorithms? Yes!

- Theorem. NL ⊆ (uniform) NC. *(Assignment problem)*
- Proof sketch.
- 1. Construct the adjacency matrix A of the configuration graph.
- 2. Use repeated squaring of A to find out if there's a path from start to accept configurations.

Complexity zoo