Gentle Introduction to Computational Complexity

Francois Schwarzentruber

ENS Rennes, France

April 4, 2019
Motivation: naive methodology

Define a problem

Directly design an algo

- e.g. coverage by a connected multi-drone system

⚠️ Issues

- you may design an overly complicated algorithm... ...whereas the problem is intrinsically simpler
- you may try to design a simple algorithm... ...whereas the problem is intrinsically harder;
Motivation: computational complexity comes in

Define a problem

Study its intrinsic complexity

Theorem (Charrier et al., 2017)
The coverage by a connected multi-drone system is PSPACE-complete.

We learned:

- No polynomial-time algorithm exists, unless \( P = \text{PSPACE} \);
- Algorithmic techniques corresponding to \( \text{PSPACE} \) are suitable.

Design an algo
Motivation: algorithmic techniques

<table>
<thead>
<tr>
<th>Complexity class</th>
<th>Algorithmic techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Greedy algorithm, dynamic programming, algorithms on graphs</td>
</tr>
<tr>
<td>NP</td>
<td>Backtracking, Backjumping, Branch-and-bound, SAT, SAT modulo theories, Linear programming</td>
</tr>
<tr>
<td>PSPACE</td>
<td>Model checkers, planners, Monte-Carlo tree search, QBF solvers</td>
</tr>
</tbody>
</table>
Outline

1 Complexity classes defined with deterministic algorithms
   - Problems
   - Definition of an algorithm
   - Time and space (memory) as resources
   - Definition of complexity classes

2 Abstracting the combinatorics

3 Proving hardness

4 \( \text{PSPACE} \)

5 Big theorems in computational complexity
Complexity classes defined with deterministic algorithms

Abstracting the combinatorics

Proving hardness

PSPACE

Big theorems in computational complexity

Outline

1. Complexity classes defined with deterministic algorithms
   - Problems
     - Definition of an algorithm
     - Time and space (memory) as resources
     - Definition of complexity classes

2. Abstracting the combinatorics

3. Proving hardness

4. PSPACE

5. Big theorems in computational complexity
Example: traveler salesman problem (TSP)

Definition (Problem)
Defined in terms of input/output.

Example
TSP
- input: a weighted graph
  \[ G = (V, E, w); \]
  \( G \) described by an adjacency list and weights are written in binary
- output: a minimal tour;
Outline

1. Complexity classes defined with deterministic algorithms
   - Problems
   - Definition of an algorithm
     - Time and space (memory) as resources
     - Definition of complexity classes

2. Abstracting the combinatorics

3. Proving hardness

4. PSPACE

5. Big theorems in computational complexity
Algorithm

Definition (Deterministic algorithm)

Standard algorithm, but no random choices.

Example

```plaintext
function tsp(G)
    bestWeight := +∞
    bestTour = −
    for all tours t do
        if weight(t) < bestWeight then
            bestWeight := weight(t)
            bestTour := t
    return bestTour
```
Outline

1. **Complexity classes defined with deterministic algorithms**
   - Problems
   - Definition of an algorithm
   - Time and space (memory) as resources
   - Definition of complexity classes

2. **Abstracting the combinatorics**

3. **Proving hardness**

4. **PSPACE**

5. **Big theorems in computational complexity**
Cobham’s thesis

Polynomial in the size $n$ of the input = efficient.

Reasons:
- Many real algorithms are $O(n^3)$ at most;
- $algo$ efficient $\Rightarrow$ for $i := 1..n$ do $algo$ efficient;
- Do not depend so much on a computation model.

Very liberal concerning the complexity analysis

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
</tr>
<tr>
<td>$O(poly(n))$</td>
<td>polynomial</td>
</tr>
<tr>
<td>$2^{O(poly(n))}$</td>
<td>exponential</td>
</tr>
<tr>
<td>$2^{2^{O(poly(n))}}$</td>
<td>double-exponential</td>
</tr>
</tbody>
</table>
Membership in $\text{EXPTIME}$

**Proposition**

TSP is in $\text{EXPTIME}$.

**Proof.**

- tsp solves TSP.
- tsp runs in $2^{\text{poly}(|G|)}$.

```plaintext
function tsp(G)
    bestWeight := +∞
    bestTour = −
    for all tours t do
        if weight(t) < bestWeight then
            bestWeight := weight(t)
            bestTour := t
    return bestTour
```

Proposition

TSP is in EXPTIME.

Proof.

- tsp solves TSP.
- tsp runs in $2^{\text{poly}(|G|)}$.

```
function tsp(G)
    bestWeight := +∞
    bestTour = −
    for all tours t do
        if weight(t) < bestWeight then
            bestWeight := weight(t)
            bestTour := t
    return bestTour
```
Outline

1. Complexity classes defined with deterministic algorithms
   - Problems
   - Definition of an algorithm
   - Time and space (memory) as resources
   - Definition of complexity classes

2. Abstracting the combinatorics

3. Proving hardness

4. \( \text{PSPACE} \)

5. Big theorems in computational complexity
Definition of complexity classes

**Definition**

\( \text{EXPTIME} \) is the class of problems for which there is an algorithm that solves it in exponential-time.

\[
\begin{align*}
\text{P} & \quad \text{PSPACE} \\
\text{EXPTIME} & \quad \text{EXPSPACE} \\
2\text{EXPTIME} & \quad 2\text{EXPSPACE}
\end{align*}
\]
Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics
   - Decision problems
   - One-player algo
   - Definition of NP

3. Proving hardness

4. PSPACE

5. Big theorems in computational complexity
New fine-grained complexity classes

TSP is in Exptime...

Methodology
Define new fine-grained complexity classes by means of games.
Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics
   - Decision problems
   - One-player algo
   - Definition of NP

3. Proving hardness

4. PSPACE

5. Big theorems in computational complexity
Decision problems

Definition (decision problem)
Problem whose output is yes/no.

Example (⚠️)

TSP
- input: a weighted graph
  \( G = (V, E, w); \)
  
  \( G \) described by an adjacency list and weights are written in binary
- output: ⚠️a minimal tour.
TSP reformulated as a decision problem

Example

TSP

- **input:**
  - a weighted graph $G = (V, E, w)$;
  - a threshold $c \in \mathbb{N}$;

- **output:**
  - yes, if there is a tour in $G$ of weight $\leq c$;
  - no otherwise.
Examples

Example

GRAPH COLORING

- Input: an undirected graph $G = (V, E)$;
- Output: yes if there is a coloring of vertices using colors, than assigns different colors to adjacent vertices; no otherwise.
Examples

Example

GRAPH COLORING

- Input: an undirected graph \( G = (V, E) \);
- Output: yes if there is a coloring of vertices using \( \bullet \bullet \bullet \), than assigns different colors to adjacent vertices; no otherwise.
### Examples

**Example**

#### SAT

- **Input:** a Boolean formula \( \varphi \);
- **Output:** yes if there are values for Boolean variables that make \( \varphi \) true; no otherwise.

\[
(p \lor q) \land (r \rightarrow \neg p) \land r \land (r \rightarrow \neg s) \land (s \rightarrow \neg q)
\]
Examples

Example

HALT

- input: a program $\pi$;
- output:
  - yes, if the execution of $\pi$ halts;
  - no otherwise.
Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics
   - Decision problems
   - One-player algo
   - Definition of NP

3. Proving hardness

4. \( \text{PSPACE} \)

5. Big theorems in computational complexity
One-player algo

Definition (One-player algo)

A one-player algo is an algorithm that may use special instructions

\[
\text{choose } b \in \{0, 1\}
\]

and that ends with instruction \textbf{win} or instruction \textbf{loose}.
One-player algo for graph coloring

```
one-player-algo graphColoring (G)
    for all vertices v of G do
        choose a color among ••• for v
        if no adjacent vertices have the same color then
            win
        else
            loose
```

$G \xrightarrow{\text{one-player algo}} \text{win/loose}$
One-player game for graph coloring

```plaintext
one-player-algo graphColoring (G)
    for all vertices v of G do
        choose a color among \(\bullet\bullet\bullet\) for v
        if no adjacent vertices have the same color then
            win
        else
            loose
```

**Proposition**

\(G\) is a positive instance of GRAPH COLORING

iff

the player has a winning strategy in graphColoring (G)

\(\rightarrow\) we say that graphColoring decides GRAPH COLORING.
One-player game

```plaintext
one-player-algo tsp(G, c)
  t := empty tour
  for i = 1..nb vertices in G do
    choose a non-already chosen successor for t
    extend t with that successor
  if t is a tour and weight(t) \leq c then win else loose
```

Proposition

\[(G, c) \text{ is a positive instance of TSP} \iff \text{the player has a winning strategy at the game tsp}(G, c)\]

→ we say that tsp decides TSP.
Outline

1 Complexity classes defined with deterministic algorithms

2 Abstracting the combinatorics
   - Decision problems
   - One-player algo
   - Definition of NP

3 Proving hardness

4 PSPACE

5 Big theorems in computational complexity
Execution-time of a one-player algo

Definition (poly-time one-player algo)

A one-player algo $algo(x)$ is in polynomial-time in $|x|$ if the length of all runs is $poly(|x|)$.

```
one-player-algo graphColoring (G)
    for all vertices $v$ of $G$ do
        choose a color among $\bullet \bullet \bullet$ for $v$
        if no adjacent vertices have the same color then
            win
        else
            loose
```
Definition of NP

**Definition**

NP = class of decision problems such that there is a one-player game that decides it in polynomial-time.

**Example (of decision problems that are in NP)**

- TSP
- Graph coloring
- SAT
- Shortest path (is in P)
- Real linear programming (is in P)
- Integer linear programming
- Clique in graphs
Terminology

- one player game = non-deterministic algorithm
- $N = \text{non-deterministic}$;
- choice of a move = a non-deterministic choice/guess
- the list of moves = a certificate
Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics

3. Proving hardness
   - Easier than
   - Intrinsic hardness
   - NP-complete problems

4. PSPACE

5. Big theorems in computational complexity
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
   - Easier than
     - Intrinsic hardness
     - NP-complete problems
4. \( \text{PSPACE} \)
5. Big theorems in computational complexity
Easier than

Definition

Problem $Pb_1$ is easier than problem $Pb_2$ if there is a poly-time deterministic algorithm $tr$ such that:

$x$ is positive instance of $Pb_1$ iff $tr(x)$ is a positive instance of $Pb_2$

Terminology

- $Pb_1$ reduces to $Pb_2$ in poly-time
- $tr$ is called a poly-time reduction from $Pb_1$ to $Pb_2$
Example: Graph coloring is easier than SAT

\[ tr(G) := \text{a Boolean formula expressing that } G \text{ is colorable.} \]

http://people.irisa.fr/Francois.Schwarzentruber/reductioncatalog/
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
   - Easier than
   - Intrinsic hardness
   - NP-complete problems
4. PSPACE
5. Big theorems in computational complexity
SAT is NP-hard

**Terminology**

A problem is NP-hard if any NP-problem is easier than it.

**Theorem**

Cook’s theorem SAT is NP-hard.

Any NP-problem

\[ \text{reduction } tr \]

\[ tr(x) \]

SAT
Any \textbf{NP}-problem is easier than SAT
Any NP-problem is easier than SAT

\[ tr(x) := \text{Boolean formula saying 'the game run on } x \text{ is winning'} \]
NP-hard problems

Theorem

\[ \begin{align*}
Pb_1 & \text{ is NP-hard} \\
\text{Pb}_1 & \text{ easier than Pb}_2 \end{align*} \] implies Pb\textsubscript{2} is NP-hard.

any NP-problem \( \longrightarrow \) SAT \( \longrightarrow \) 3SAT

\[ \begin{align*}
\text{HAMILTONIAN CYCLE} & \text{ easier than} \\
\text{TSP} \end{align*} \]

\[ \begin{align*}
\text{COLORING} \end{align*} \]
Complexity classes defined with deterministic algorithms

Abstracting the combinatorics

Proving hardness

PSPACE

Big theorems in computational complexity

Easier than

Intrinsic hardness

NP-complete problems

Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics

3. Proving hardness
   - Easier than
   - Intrinsic hardness
   - NP-complete problems

4. PSPACE

5. Big theorems in computational complexity
NP-complete problems

Definition

A problem is NP-complete if:

- it is in NP;
- it is NP-hard.
Understand where the ‘complexity’ is

**Complexity of TSP restrictions**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NP-complete</strong></td>
<td>even for 2D grid graphs</td>
</tr>
<tr>
<td></td>
<td>[Itai, Papadimitriou, Szwarcfiter, 1982]</td>
</tr>
<tr>
<td><strong>in P</strong></td>
<td>for solid grid graphs</td>
</tr>
<tr>
<td></td>
<td>[Arkin, Bender, Demaine, Fekete, Mitchell, Sethia, 2001]</td>
</tr>
</tbody>
</table>

**Parameterized complexity**

identify a parameter (e.g. diameter of graphs, etc.) on inputs that sums up the complexity.
Outline

1. Complexity classes defined with deterministic algorithms

2. Abstracting the combinatorics

3. Proving hardness

4. **PSPACE**
   - Two-player poly-time games
   - One-player poly-space games
   - PSPACE-complete problems

5. Big theorems in computational complexity
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. **PSPACE**
   - Two-player poly-time games
   - One-player poly-space games
   - PSPACE-complete problems
5. Big theorems in computational complexity
Motivation

- Bounded planning versus the environment
- querying $(∃∀)^*\text{-properties}$ (e.g. SQL, etc.)

$$∃x, ∀y, (R(x, y) → ∃z, p(f(x, y, z)))$$

→ winning strategy for player one in a two-player poly-time game

[Arora, Barak, chap. 4.2.2 (« The essence of PSPACE: optimum strategies for game-playing »)]
Generalized Hex

Example (Even, Tarjan, 1976)

**HEX**

- **Input:** A graph $G$, a source $s$, a target $t$;
- **Output:** yes if player 1 has a winning strategy to the game:
  - by turn, player $i$ select a non-selected vertex in $G \setminus \{s, t\}$;
  - player 1 wins if there is a $s – t$-path made up of 1-vertices.
Two-player algo

Definition (Two-player algo)

A two-player algo is an algorithm that may use special instructions

player one chooses $b \in \{0, 1\}$
player two chooses $b \in \{0, 1\}$

and that ends with player one wins or player one looses.
Two-player poly-time algos

**two-player-algo** \texttt{hex}(G, s, t)

\begin{algorithmic}
  \While{some non-selected vertex in $G$}
    \textbf{player one chooses} a non-selected vertex in $G \setminus \{s, t\}$
    \textbf{player two chooses} a non-selected vertex in $G \setminus \{s, t\}$
    \If{there is a $s$ -- $t$-path made up of 1-vertices}
      \textbf{player one wins}
    \Else
      \textbf{player one looses}
  \EndIf
  \EndWhile
\end{algorithmic}

**Definition**

- A strategy for a player tells her/him which move to take at each time.
- A winning strategy for player one makes player one win, whatever the moves of the other player.
A two-player algo decides a decision problem

**Proposition**

\[(G, s, t) \text{ is a positive instance of HEX} \quad \text{iff} \quad \text{the first player has a winning strategy in } hex(G, s, t)\]

→ We say that hex decides HEX!
Alternating poly-time: AP

**Definition**

AP is the class of decision problems such that there is a two-player algo that decides it in poly-time.

**Theorem**

*HEX in AP.*

**Proof.**

hex decides HEX in poly-time.
Quantified binary formulas

Example

QBF-SAT

- Input: a closed quantified binary formula \( \varphi \);
- Output: yes if \( \varphi \) is true; no otherwise.

\[
\exists p, \forall q, \forall r, \exists s (p \rightarrow (q \land r \rightarrow s))
\]

Theorem

QBF-SAT in AP.
In \textbf{PSPACE}!

\textbf{Theorem}

\[ \text{AP} \subseteq \text{Pspace}. \]

Deterministic backtracking algorithm (min-max style)!
In \( \text{PSPACE!} \)

**Theorem**

\[ \text{AP} \subseteq \text{PSPACE}. \]

Deterministic backtracking algorithm (min-max style)!
In \textbf{PSPACE}!

**Theorem**

\[ \text{AP} \subseteq \text{PSPACE}. \]

Deterministic backtracking algorithm (min-max style)!
In \textbf{Pspace}!

\textbf{Theorem}

\[ \text{AP} \subseteq \text{PSPACE}. \]

Deterministic backtracking algorithm (min-max style)!
In \textbf{PSPACE}!

\textbf{Theorem}

\[ \text{AP} \subseteq \text{PSPACE}. \]

Deterministic backtracking algorithm (min-max style)!
In $\text{PSPACE}!$

**Theorem**

$\text{AP} \subseteq \text{PSPACE}$.

Deterministic backtracking algorithm (min-max style)!
In $\text{PSPACE}$!

**Theorem**

$$\text{AP} \subseteq \text{Pspace}.$$ 

Deterministic backtracking algorithm (min-max style)!
In \textbf{PSPACE}!

\textbf{Theorem}

\textbf{AP} \subseteq \textbf{PSPACE}.

Deterministic backtracking algorithm (min-max style)!
In $\text{PSPACE}$!

Theorem

$\text{AP} \subseteq \text{PSPACE}$.

Deterministic backtracking algorithm (min-max style)!
In \textbf{PSPACE}!

**Theorem**

\[ \text{AP} \subseteq \text{Pspace}. \]

Deterministic backtracking algorithm (min-max style)!
In $\text{PSPACE}$!

**Theorem**

$AP \subseteq \text{PSPACE}$. 

Deterministic backtracking algorithm (min-max style)!
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. PSPACE
   - Two-player poly-time games
   - One-player poly-space games
   - PSPACE-complete problems
5. Big theorems in computational complexity
Definition

**NPSPACE** is the class of decision problems such that there is a one-player algo that decides it in polynomial-space.

**Example**

**SOKOBAN:**
- Input: a Sokoban position;
- Output: yes if the player can win from that position; no otherwise.
One-player algo deciding SOKOBAN in poly-space

```plaintext
one-player-algo sokoban(position)

    while position is not winning do

        choose position := one of the next possible positions from position

    win
```

Big theorems in computational complexity
Savitch’s theorem

**Theorem**

\[ \text{AP} = \text{PSPACE} = \text{NPSPACE}. \]
Savitch’s theorem

**Theorem**

\[ \text{AP} = \text{PSPACE} = \text{NPSPACE}. \]

\[ \subseteq \subseteq \]

**Proof of NPSPACE \subseteq AP**

One player poly-space algo \[ \xrightarrow{\text{reformulation}} \] Two player poly-time algo
Two-player poly-time algo equivalent to Sokoban

- **Player one** chooses a mid-position of Sokoban
- **Player two** chooses which part to check

\[ \leq 2^{\text{poly}(n)} \text{ steps?} \]

\[ n = \text{size of the Sokoban position given in input} \]
Two-player poly-time algo equivalent to Sokoban

- **Player one** chooses a mid-position of Sokoban
- **Player two** chooses which part to check

\[ \leq 2^{\text{poly}(n) - 1} \]

\[ \leq 2^{\text{poly}(n) - 1} \]

\[ n = \text{size of the Sokoban position given in input} \]
Two-player poly-time algo equivalent to Sokoban

- Player one chooses a mid-position of Sokoban
- Player two chooses which part to check

\[ \leq 2^{\text{poly}(n) - 1} \]

\( n \) = size of the Sokoban position given in input
Two-player poly-time algo equivalent to Sokoban

- **Player one** chooses a mid-position of Sokoban
- **Player two** chooses which part to check

\[ \leq 2^{\text{poly}(n) - 2} \]

\[ \leq 2^{\text{poly}(n) - 2} \]

\[ \leq 2^{\text{poly}(n) - 1} \]

\[ n = \text{size of the Sokoban position given in input} \]
Two-player poly-time algo equivalent to Sokoban

- **Player one** chooses a mid-position of Sokoban
- **Player two** chooses which part to check

\[ \leq 2^{\text{poly}(n)-2} \]

\[ \leq 2^{\text{poly}(n)-2} \]

\[ \leq 2^{\text{poly}(n)-1} \]

\[ n = \text{size of the Sokoban position given in input} \]
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. \textbf{PSPACE}
   - Two-player poly-time games
   - One-player poly-space games
   - PSPACE-complete problems
5. Big theorems in computational complexity
### PSPACE-complete problems: some two-player poly-time games

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>QBF-SAT</strong></td>
<td>Input: a closed quantified binary formula ( \varphi ); Output: yes if ( \varphi ) is true; no otherwise.</td>
</tr>
</tbody>
</table>

\[
\exists p, \forall q, \forall r, \exists s (p \rightarrow (q \land r \rightarrow s))
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First-order query on a finite model</strong></td>
<td>Input: a finite model ( \mathcal{M} ), a first-order formula ( \varphi ); Output: yes if ( \mathcal{M} ) satisfies ( \varphi ), no otherwise.</td>
</tr>
</tbody>
</table>

\[
\exists x, \forall y, (R(x, y) \rightarrow \exists z, p(f(x, y, z)))
\]

also HEX!
PSPACE-complete problems: some one-player poly-space games

Universality of a regular expression

- input: a regular expression $e$;
- output: yes if the language denoted by $e$ is $\Sigma^*$, no otherwise.

Classical planning

- input: an initial state $\iota$, a final state $\gamma$, description of actions;
- output: yes if $\gamma$ is reachable from $\iota$ by executing some actions, no otherwise.

Also Sokoban!
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. \( \text{PSPACE} \)
5. Big theorems in computational complexity
   - About games
   - About \( \text{LOGSPACE} \) and \( \text{NLOGSPACE} \)
   - About non-uniform poly-time algorithm
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. PSPACE
5. Big theorems in computational complexity
   - About games
   - About LOGSPACE and NLOGSPACE
   - About non-uniform poly-time algorithm
Correspondence between alternating and usual classes

[Chandra, Stockmeyer, 1980, Alternations]

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{AP} = \text{Pspace}$</td>
</tr>
<tr>
<td>$\text{Aexptime} = \text{ExpSPACE}$</td>
</tr>
<tr>
<td>$\text{ALogspace} = \text{P}$</td>
</tr>
<tr>
<td>$\text{APspace} = \text{ExpTIME}$</td>
</tr>
</tbody>
</table>

[Chandra, Stockmeyer, 1980, Alternations]
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. $PSPACE$
5. Big theorems in computational complexity
   - About games
   - About $LOGSPACE$ and $NLOGSPACE$
   - About non-uniform poly-time algorithm
Definitions

**Definition**

**Logspace** is the class of decision problems decided by an algorithm in logarithmic space.

- The input is read-only
- $\sim$ a constant number of pointers

**Definition**

**NLogspace** is the class of decision problems decided by a one-player algo in logarithmic space.
Important results

**Theorem**

*Reachability in a directed graph is $\text{NLogspace}$-complete.*

**Theorem (Immerman-Szelepcsényi, 1988)**

$\text{NLogspace} = \text{coNLogspace}$

**Theorem (Reingold, 2005)**

*Reachability in an undirected graph is in $\text{Logspace}$.***
Outline

1. Complexity classes defined with deterministic algorithms
2. Abstracting the combinatorics
3. Proving hardness
4. PSPACE
5. Big theorems in computational complexity
   - About games
   - About LOGSPACE and NLOGSPACE
   - About non-uniform poly-time algorithm
We believe $\mathsf{P} \neq \mathsf{NP}$, even we believe $\mathsf{P}_{/\text{poly}} \neq \mathsf{NP}$

**Definition**

$\mathsf{P}_{/\text{poly}} = \text{class of decision problems s.th. there are deterministic algorithms } A_1, A_2, \ldots \text{ such that}$

$$A_n \text{ decides inputs of size } n \text{ in } \text{poly}(n).$$

**Theorem (Karp and Lipton, 1980)**

*If $\mathsf{NP} \subseteq \mathsf{P}_{/\text{poly}}$, the polynomial hierarchy collapses at level 2:*

![Transformation diagram](https://via.placeholder.com/150)

- Two-player poly-time algo with a fixed number of alternations
- Two-player poly-time algo with 2 alternations
This tutorial did not present...

- Formal definitions of complexity classes with Turing machines
- The obscure terminology (non-determinism, alternation, certificates, reduction, etc.)
- Other types of reduction: log-space, FO, etc.
- Probabilistic complexity classes
- Quantum complexity classes
- Counting and Toda’s theorem
- Descriptive complexity
- Circuit complexity
- Other computation models: RAM, etc.
- Interactive proofs
- Parametrized complexity
- ...

About games
About \textsc{Logspace} and \textsc{NLogspace}
About non-uniform poly-time algorithm
Complexity classes defined with deterministic algorithms
Abstracting the combinatorics
Proving hardness
**PSPACE**

Big theorems in computational complexity

About games
About **LOGSPACE** and **NLOGSPACE**
About non-uniform poly-time algorithm

Thank you for your attention!