Hardness Theory of Parameterized Complexity

Simon Rey and Ronald de Haan

June Project

Institute for Logic, Language and Computation
University of Amsterdam

From Tractability to Intractability

During the first two lectures, we studied the class FPT that contains all the *fixed-parameter tractable* problems. These problems are tractable—solvable in polynomial time—when the value of the parameter is fixed. In that sense, FPT can be thought as an equivalent of P in classical complexity theory.

From Tractability to Intractability

During the first two lectures, we studied the class `FPT` that contains all the *fixed-parameter tractable* problems. These problems are tractable—solvable in polynomial time—when the value of the parameter is fixed. In that sense, `FPT` can be thought as an equivalent of `P` in classical complexity theory.

Today, we will study problems that are *fixed-parameter intractable*, that is, problems for which no `FPT` algorithm can be devised. We will see complexity classes that can be seen as equivalent of the class `NP` in classical complexity theory. And some other classes.

From Tractability to Intractability

During the first two lectures, we studied the class `FPT` that contains all the *fixed-parameter tractable* problems. These problems are tractable—solvable in polynomial time—when the value of the parameter is fixed. In that sense, `FPT` can be thought as an equivalent of `P` in classical complexity theory.

Today, we will study problems that are *fixed-parameter intractable*, that is, problems for which no `FPT` algorithm can be devised. We will see complexity classes that can be seen as equivalent of the class `NP` in classical complexity theory. And some other classes.

Before going further into parameterized complexity, let's start with a remainder about intractability in classical complexity theory.

1. A Detour to Classical Complexity Theory

Definition: Complexity Class NP

NP is the class of all the languages $L \subseteq \Sigma^*$ for which there exists a Turing machine $\mathbb{M}$ (the verifier), a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a constant $c \in \mathbb{N}$ such that:

- For all $x \in \Sigma^*$, we have:

$$x \in L \quad \text{if and only if} \quad \exists u \in \{0,1\}^{p(|x|)} \text{ (the certificate) such that } \mathbb{M}(x,u) = 1;$$

- $\mathbb{M}$ runs in time $\mathcal{O}(|x|^c)$ on every input $x \in \Sigma^*$.

While P was the class of all the problems decidable in polynomial time, NP is the class of all the problems for which we can *verify* a (potential) solution in polynomial time.

Non-Deterministic Turing Machines

<u>Definition</u>: Non-Deterministic Turing Machines

A *non-deterministic Turing machine* $\mathbb{M}$ is a of a Turing machine such that:

- It has two transition functions δ_1 and δ_2 (instead of only one);
- At each step, the one to be used is chosen non-deterministically;
- $\mathbb{M}$ accepts an input if there exists a sequence of non-deterministic choices that lead to an accepting state.

Non-Deterministic Turing Machines

Definition: Non-Deterministic Turing Machines

A *non-deterministic Turing machine* $\mathbb{M}$ is a of a Turing machine such that:

- It has two transition functions δ_1 and δ_2 (instead of only one);
- At each step, the one to be used is chosen non-deterministically;
- $\mathbb{M}$ accepts an input if there exists a sequence of non-deterministic choices that lead to an accepting state.

$\texttt{NP}$ can be equivalently defined through non-deterministic Turing machines.

Proposition: Another Characterization of $\texttt{NP}$

$\texttt{NP}$ is the class of all the languages $L \subseteq \Sigma^*$ for which there exists a *non-deterministic* Turing machine $\mathbb{M}$ running in *polynomial* time and *deciding* L.

The Class coNP

Definition: Complexity Class coNP

coNP is the class of all the languages $L \subseteq \Sigma^*$ for which there exists a Turing machine $\mathbb{M}$ (the verifier), a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a constant $c \in \mathbb{N}$ such that:

- For all $x \in \Sigma^*$, we have:

$$x \in L \quad \text{if and only if} \quad \forall u \in \{0,1\}^{p(|x|)} \text{ (the certificate)}, \mathbb{M}(x, u) = 1;$$

- $\mathbb{M}$ runs in time $\mathcal{O}(|x|^c)$ on every input $x \in \Sigma^*$.

The Class coNP

Definition: Complexity Class coNP

coNP is the class of all the languages $L \subseteq \Sigma^*$ for which there exists a Turing machine $\mathbb{M}$ (the verifier), a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a constant $c \in \mathbb{N}$ such that:

- For all $x \in \Sigma^*$, we have:

$$x \in L \quad \text{if and only if} \quad \forall u \in \{0,1\}^{p(|x|)} \text{ (the certificate)}, \mathbb{M}(x, u) = 1;$$

- $\mathbb{M}$ runs in time $\mathcal{O}(|x|^c)$ on every input $x \in \Sigma^*$.

coNP can also be seen as the class of all the problems for which checking whether something is *not* a solution is an NP problem.

Proposition: Complexity Class coNP

A language $L \subseteq \Sigma^*$ is in coNP if and only if $\overline{L} = \{x \in \Sigma^* \mid x \notin L\}$ is in NP.

Some Examples

Is Pareto-Optimal	
Instance:	A set of items $\mathcal{I}$, a set of n agents $\mathcal{N}$, n utility functions $u_i : \mathcal{I} \to \mathbb{N}$, and an allocation $\pi : \mathcal{N} \to 2^{\mathcal{I}}$ such that all items are allocated and no item is allocated to several agents (a partition of the items)
Question:	Is the allocation π Pareto-optimal, i.e., there is no other allocation π' such that all agents are better off in π' and at least one agent is strictly better off?

Max-Approval Participatory Budgeting	
Instance:	A set of projects $\mathcal{P}$, a cost function $c : \mathcal{P} \to \mathbb{N}$, a budget limit $B \in \mathbb{N}$, a set of agents $\mathcal{N} = \{1, \ldots, n\}$, n approval ballots $A_i \subseteq \mathcal{P}$ and a parameter $k \in \mathbb{N}$
Question:	Is there a budget allocation $\pi \subseteq \mathcal{P}$ with $\sum_{p \in \pi} c(p) \leq B$ and such that $\sum_{i \in \mathcal{N}} \lvert \pi \cap A_i \rvert \geq k$?

↳ Who is where?

NP-completeness

The hardness theory in classical complexity theory is based on the idea that several problems are *equivalent* in terms of how hard they are to solve. The formalization of this idea is based on *polynomial time reductions*.

A language $L_1 \subseteq \Sigma^*$ is *polynomial time reducible* to another language $L_2 \subseteq \Sigma^*$ if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ (the reduction) such that for all $x \in \Sigma^*$, we have: $x \in L_1 \iff f(x) \in L_2$.

NP-completeness

The hardness theory in classical complexity theory is based on the idea that several problems are *equivalent* in terms of how hard they are to solve. The formalization of this idea is based on *polynomial time reductions*.

A language $L_1 \subseteq \Sigma^*$ is *polynomial time reducible* to another language $L_2 \subseteq \Sigma^*$ if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ (the reduction) such that for all $x \in \Sigma^*$, we have: $x \in L_1 \iff f(x) \in L_2$.

Definition: NP-completeness

A language $L \subseteq \Sigma^*$ is NP-*hard* if every L' in NP is polynomial time reducible to L.
A language $L \subseteq \Sigma^*$ is NP-*complete* if it NP-hard and it is in NP.

NP-completeness

The hardness theory in classical complexity theory is based on the idea that several problems are *equivalent* in terms of how hard they are to solve. The formalization of this idea is based on *polynomial time reductions*.

A language $L_1 \subseteq \Sigma^*$ is *polynomial time reducible* to another language $L_2 \subseteq \Sigma^*$ if there exists a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ (the reduction) such that for all $x \in \Sigma^*$, we have: $x \in L_1 \iff f(x) \in L_2$.

Definition: NP-completeness

A language $L \subseteq \Sigma^*$ is NP-*hard* if every L' in NP is polynomial time reducible to L.
A language $L \subseteq \Sigma^*$ is NP-*complete* if it NP-hard and it is in NP.

To show that a language L is NP-hard we start from a language L_0 in NP (Max-Approval Participatory Budgeting for instance, or, historically, SAT) and provide a polynomial time reduction showing how to *embed* L_0 in L so that if one can decide L, one would also decide L_0. In that sense, all NP-complete languages are *equally hard* to solve: solving one implies solving all of them (since reductions are transitive).

2. Fixed-Parameter Tractable Reduction

Fixed-Parameter Tractable Reduction

Definition: FPT-Reduction

An *FPT-reduction* from a parameterized language $L_1 \subseteq \Sigma^* \times \mathbb{N}$ and to another $L_2 \subseteq \Gamma^* \times \mathbb{N}$ is a mapping $f : \Sigma^* \times \mathbb{N} \to \Gamma^* \times \mathbb{N}$ such that:

1. $\langle x, k \rangle \in L_1$ if and only if $f(x, k) \in L_2$;
2. $k' \leq g(k)$ for some computable function g for k' such that $\langle x', k' \rangle = f(x, k)$;
3. f is computable by an FPT algorithm (with respect to k).

Fixed-Parameter Tractable Reduction

Definition: FPT-Reduction

An *FPT-reduction* from a parameterized language $L_1 \subseteq \Sigma^* \times \mathbb{N}$ and to another $L_2 \subseteq \Gamma^* \times \mathbb{N}$ is a mapping $f : \Sigma^* \times \mathbb{N} \to \Gamma^* \times \mathbb{N}$ such that:

1. $\langle x, k \rangle \in L_1$ if and only if $f(x, k) \in L_2$;
2. $k' \leq g(k)$ for some computable function g for k' such that $\langle x', k' \rangle = f(x, k)$;
3. f is computable by an FPT algorithm (with respect to k).

Condition 2 ensures that the class FPT is closed under FPT-reduction.

Proposition: Two Facts About FPT-Reduction

The relation between languages derived from FPT-reductions is transitive.

If there is an FPT-reduction from a parameterized language $L_1 \subseteq \Sigma^* \times \mathbb{N}$ to another parameterized language $L_2 \in \Gamma^* \times \mathbb{N}$ that is in FPT, then L_1 is in FPT.

A Parameterized Equivalent of NP?

FPT can be interpreted as the parameterized counterpart of P (recall that for a language L in P, all of its parameterizations are in FPT). Now that we have a suitable notion of parameterized reduction, can we can try to define an equivalent of NP in parameterized complexity theory.

A Parameterized Equivalent of NP?

FPT can be interpreted as the parameterized counterpart of P (recall that for a language L in P, all of its parameterizations are in FPT). Now that we have a suitable notion of parameterized reduction, can we can try to define an equivalent of NP in parameterized complexity theory.

↪ We will see there is not a single class that resemble NP in parameterized complexity theory but rather a whole *hierarchy* of them. Let's begin!

3. Parameterized NP?

Parameterized NP?

First Attempt: paraNP

Non-Deterministic Parameterized Classes

To move from $\texttt{P}$ to $\texttt{NP}$, we plugged in non-deterministic Turing machines. Let's do the same with $\texttt{FPT}$, we obtain the class $\texttt{paraNP}$.

DEFINITION: PARAMETERIZED COMPLEXITY CLASS $\texttt{paraNP}$

$\texttt{paraNP}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a *non-deterministic* Turing machine $\mathbb{M}$, a constant $c \in \mathbb{N}$ and a computable function f such that, for all pairs $\langle x, k \rangle \in L$:

- $\mathbb{M}$ runs in time $f(k)|x|^c$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

Non-Deterministic Parameterized Classes

To move from $\mathtt{P}$ to $\mathtt{NP}$, we plugged in non-deterministic Turing machines. Let's do the same with $\mathtt{FPT}$, we obtain the class paraNP.

Definition: Parameterized Complexity Class paraNP

paraNP is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a *non-deterministic* Turing machine $\mathbb{M}$, a constant $c \in \mathbb{N}$ and a computable function f such that, for all pairs $\langle x, k \rangle \in L$:

- $\mathbb{M}$ runs in time $f(k)|x|^c$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

It is clear that for all languages L in NP, all the parameterized languages $L' \subseteq L \times \mathbb{N}$ are in paraNP. In that sense paraNP is an equivalent to NP. Similarities are even stronger:

Non-Deterministic Parameterized Classes

To move from $\texttt{P}$ to $\texttt{NP}$, we plugged in non-deterministic Turing machines. Let's do the same with $\texttt{FPT}$, we obtain the class $\texttt{paraNP}$.

Definition: Parameterized Complexity Class $\texttt{paraNP}$

$\texttt{paraNP}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a *non-deterministic* Turing machine $\mathbb{M}$, a constant $c \in \mathbb{N}$ and a computable function f such that, for all pairs $\langle x, k \rangle \in L$:

- $\mathbb{M}$ runs in time $f(k)|x|^c$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

It is clear that for all languages L in $\texttt{NP}$, all the parameterized languages $L' \subseteq L \times \mathbb{N}$ are in $\texttt{paraNP}$. In that sense $\texttt{paraNP}$ is an equivalent to $\texttt{NP}$. Similarities are even stronger:

Proposition:

$\texttt{FPT} = \texttt{paraNP}$ if and only $\texttt{P} = \texttt{NP}$.

paraNP-Complete Parameterized Languages

Before giving the main result about paraNP, we need two more things:

- The fact that paraNP is closed under FPT-reductions;
- The k-slice of a parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$ defined as:

$$L_k = \{x \in \Sigma^* \mid \langle x, k \rangle \in L\}.$$

paraNP-Complete Parameterized Languages

Before giving the main result about paraNP, we need two more things:

- The fact that paraNP is closed under FPT-reductions;
- The k-slice of a parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$ defined as:

$$L_k = \{x \in \Sigma^* \mid \langle x, k \rangle \in L\}.$$

Now the main result:

THEOREM:

Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a non-trivial parameterized language in paraNP. The following statements are equivalent:

- L is paraNP-complete under FPT-reduction;
- There exist $\ell \in \mathbb{N}_{>0}$ and $k_1, \ldots, k_\ell \in \mathbb{N}$ such that $L_{k_1} \cup \ldots \cup L_{k_\ell}$ is NP-complete (under polynomial time reductions).

Implications and Interpretations

COROLLARY:

A non-trivial parameterized language in paraNP with at least one NP-complete slice is paraNP-complete under FPT-reduction.

<u>COROLLARY</u>:

A non-trivial parameterized language in paraNP with at least one NP-complete slice is paraNP-complete under FPT-reduction.

↳ This, unfortunately, implies that paraNP-complete problems are *not super interesting* from a parameterized point of view since their hardness is already present for finitely many values of the parameters.

Corollary:

A non-trivial parameterized language in paraNP with at least one NP-complete slice is paraNP-complete under FPT-reduction.

↳ This, unfortunately, implies that paraNP-complete problems are *not super interesting* from a parameterized point of view since their hardness is already present for finitely many values of the parameters.

Let's try to find a better counterpart for NP in the parameterized world!

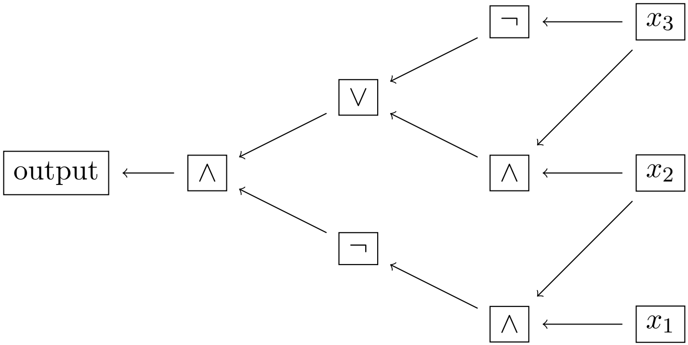
Parameterized NP?

Another try: the W Hierarchy

Boolean Circuit

Definition: Boolean Circuit

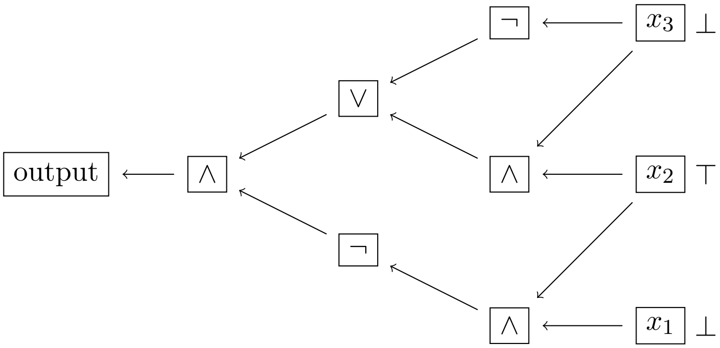
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

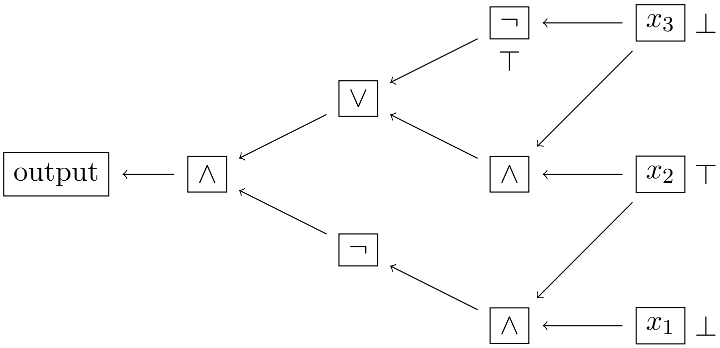
Definition: Boolean Circuit

A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\land$, $\lor$, $\neg$). There is also a specified output node with outdegree 0.



DEFINITION: BOOLEAN CIRCUIT

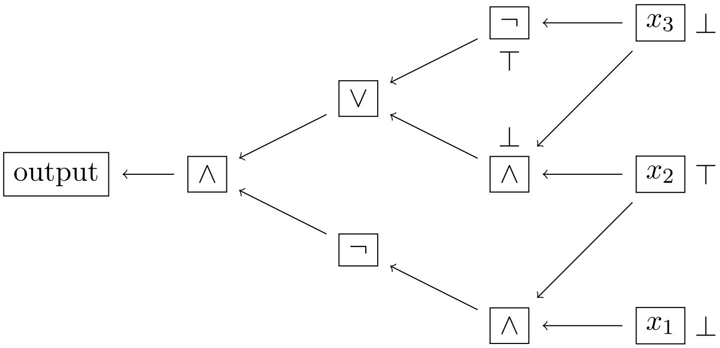
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

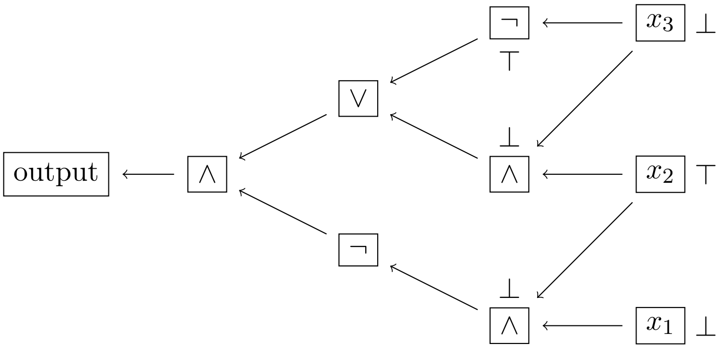
Definition: Boolean Circuit

A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Definition: Boolean Circuit

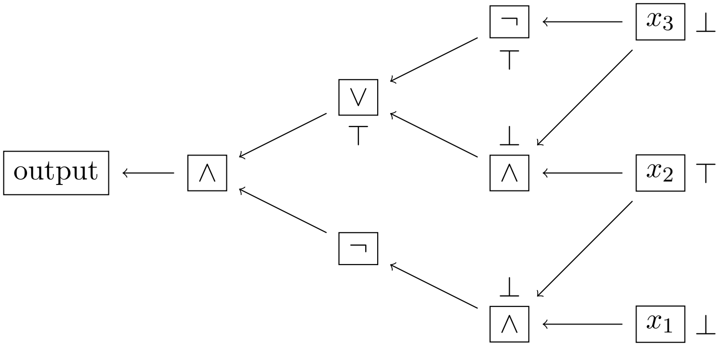
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

DEFINITION: BOOLEAN CIRCUIT

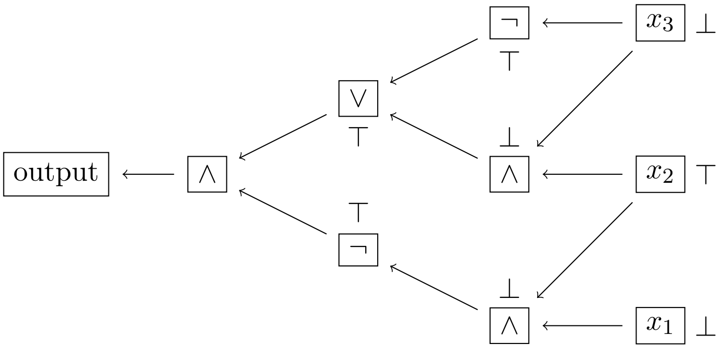
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

<u>Definition</u>: Boolean Circuit

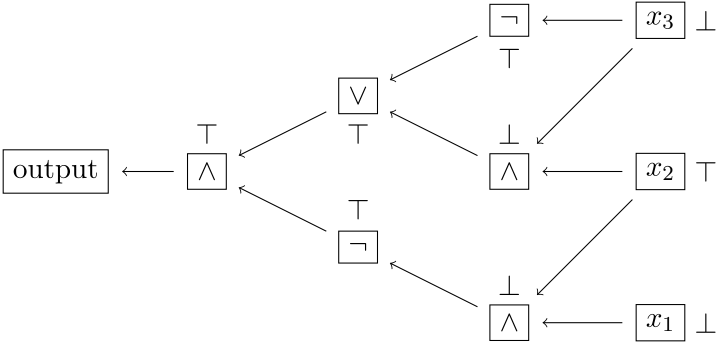
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

Definition: Boolean Circuit

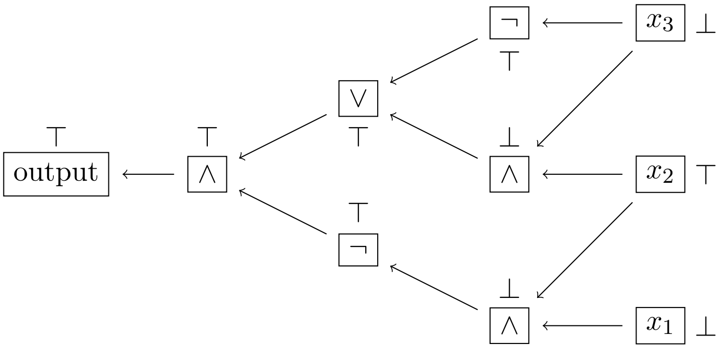
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Boolean Circuit

Definition: Boolean Circuit

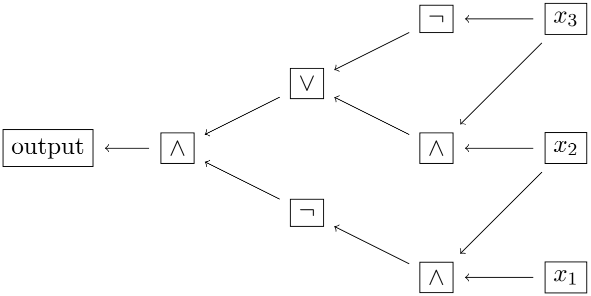
A *boolean circuit* is a directed acyclic graph whose nodes are labeled either with a Boolean constant ($\top$ or $\bot$), a propositional variable or a Boolean operator ($\wedge$, $\vee$, $\neg$). There is also a specified output node with outdegree 0.



Properties of Boolean Circuits

We will need the following definitions about boolean circuits.

- A *formula* is a Boolean circuit in which all gates have outdegree at most 1.
- The *depth* of a Boolean circuit is the length of a longest path from a variable node to the output node.
- The *weft* of a Boolean circuit is the largest number of large gates (with indegree at least 3) on any path from a variable node to the output node.



↪ Is the previous Boolean circuit a formula? What is its depth? And its weft?

Other Properties of Boolean Circuits

We will also need the following classes of Boolean circuits.

Other Properties of Boolean Circuits

We will also need the following classes of Boolean circuits.

- The class of all Boolean circuits of depth u and weft t: $CIRC_{t,u}$.

Other Properties of Boolean Circuits

We will also need the following classes of Boolean circuits.

- The class of all Boolean circuits of depth u and weft t: $CIRC_{t,u}$.
- The class of all Boolean circuits: $CIRC$.

We will also need the following classes of Boolean circuits.

- The class of all Boolean circuits of depth u and weft t: $CIRC_{t,u}$.
- The class of all Boolean circuits: $CIRC$.
- The class of all Boolean formulas: $FORM$.

We will also need the following classes of Boolean circuits.

- The class of all Boolean circuits of depth u and weft t: $CIRC_{t,u}$.
- The class of all Boolean circuits: $CIRC$.
- The class of all Boolean formulas: $FORM$.

Finally, an assignment maps each variable to a truth value ($\bot$ or $\top$). A assignment *satisfies* the Boolean circuit C if after propagating the truth values, the output node is set to $\top$. An assignment has *weight k* if it sets exactly k variables to $\top$.

A Family of Classes: $\mathtt{W}[t]$

The family of classes $\mathtt{W}[t]$ is defined with respect to the following parameterized language.

WSat($\mathcal{C}$)	
Instance:	A Boolean circuit $C \in \mathcal{C}$ and $k \in \mathbb{N}$
Parameter:	k
Question:	Does there exists an assignment of weight k that satisfies C?

A Family of Classes: $\texttt{W}[t]$

The family of classes $\texttt{W}[t]$ is defined with respect to the following parameterized language.

WSAT($\mathcal{C}$)	
Instance:	A Boolean circuit $C \in \mathcal{C}$ and $k \in \mathbb{N}$
Parameter:	k
Question:	Does there exists an assignment of weight k that satisfies C?

DEFINITION: PARAMETERIZED COMPLEXITY CLASS $\texttt{W}[t]$

The parameterized complexity class $\texttt{W}[t]$, for $t \in \mathbb{N}_{>0} \cup \{SAT, \texttt{P}\}$, is defined as:

$$\begin{aligned} \texttt{W}[t] &= [\{\text{WSAT}(CIRC_{t,u}) \mid u \geq 1\}]^{\texttt{FPT}}, \\ \texttt{W}[SAT] &= [\text{WSAT}(FORM)]^{\texttt{FPT}}, \\ \texttt{W}[\texttt{P}] &= [\text{WSAT}(CIRC)]^{\texttt{FPT}}, \end{aligned}$$

where $[\mathcal{S}]^{\texttt{FPT}}$ is the transitive closure of $\mathcal{S} \subseteq 2^{\Sigma^* \times \mathbb{N}}$ under $\texttt{FPT}$-reductions.

Restricted Non-Determinism and $\mathtt{W[P]}$

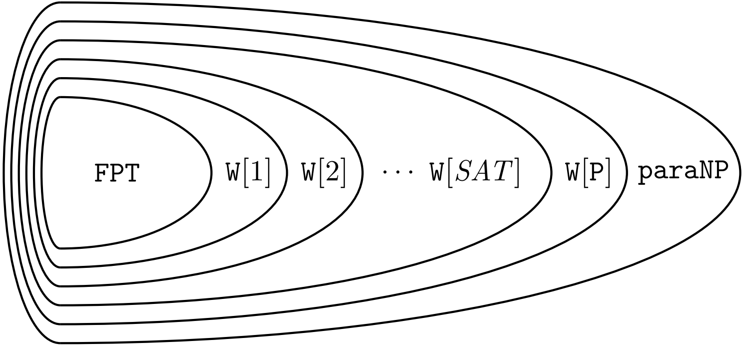
$\mathtt{W[P]}$ can be alternatively characterized with non-deterministic Turing machines that have only limited access to non-determinism.

A non-deterministic Turing machine $\mathbb{M}$ is *k-restricted* if there are two functions $f, g : \mathbb{N} \to \mathbb{N}$ and a constant $c \in \mathbb{N}$ such that on every input $\langle x, k \rangle \in \Sigma^* \times \mathbb{N}$, the machine $\mathbb{M}$ runs in at most $f(k)|x|^c$ steps of which at most $g(k) \log(|x|)$ are non-deterministic.

Proposition:

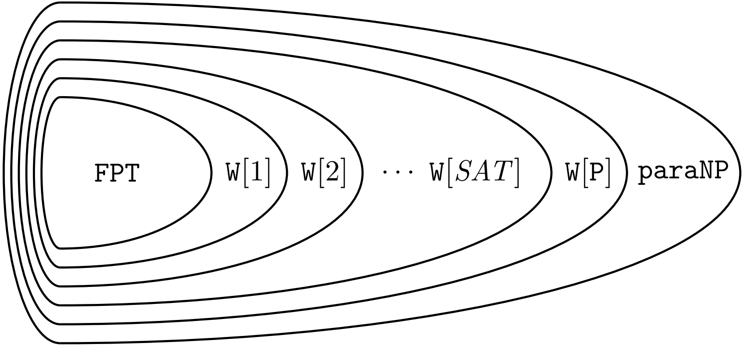
$\mathtt{W[P]}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ that can be decided by a non-deterministic *k-restricted* Turing machine $\mathbb{M}$.

Current Taxonomy of Parameterized Complexity Classes



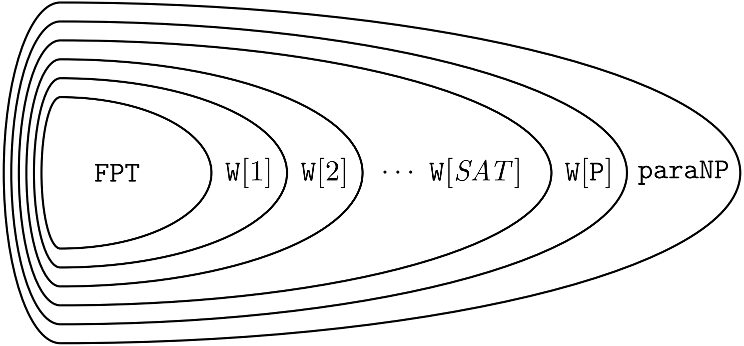
<u>Proposition</u>:

$\mathtt{FPT} \subseteq \mathtt{W}[1] \subseteq \mathtt{W}[2] \subseteq \cdots \subseteq \mathtt{W}[SAT] \subseteq \mathtt{W}[\mathrm{P}] \subseteq \mathtt{paraNP}$.



All the inclusions in the figure above are *believed* to be strict. However, it seems really hard to show this. In particular, for any $t \in \mathbb{N} \cup \{SAT, \mathtt{P}\}$ if $\mathtt{FPT} \neq \mathtt{W}[t]$, then also $\mathtt{P} \neq \mathtt{NP}$ (since $\mathtt{P} = \mathtt{NP}$ if and only $\mathtt{FPT} = \mathtt{paraNP}$). Whether the converse also holds is an open problem.

Towards Intractability



All the inclusions in the figure above are *believed* to be strict. However, it seems really hard to show this. In particular, for any $t \in \mathbb{N} \cup \{SAT, \texttt{P}\}$ if $\texttt{FPT} \neq \texttt{W}[t]$, then also $\texttt{P} \neq \texttt{NP}$ (since $\texttt{P} = \texttt{NP}$ if and only $\texttt{FPT} = \texttt{paraNP}$). Whether the converse also holds is an open problem.

↪ Let's now look at a class of parameterized languages that we can *prove* is *strictly larger* than FPT (diagonalization is a great tool!).

4. Provable Fixed-Parameter Intractability

Non-Uniform XP

Let's come back to the idea of having polynomial slices that we mentioned in paraNP.

DEFINITION: PARAMETERIZED COMPLEXITY CLASS $\mathtt{XP}_{nu}$

$\mathtt{XP}_{nu}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ whose slices L_k for $k \in \mathbb{N}_{>0}$ are *all in* P.

Non-Uniform XP

Let's come back to the idea of having polynomial slices that we mentioned in paraNP.

DEFINITION: PARAMETERIZED COMPLEXITY CLASS $\mathtt{XP}_{nu}$

$\mathtt{XP}_{nu}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ whose slices L_k for $k \in \mathbb{N}_{>0}$ are *all in* P.

This class is *non-uniform* in the sense that it contains undecidable problems.

PROOF: Consider an undecidable language $Q \subseteq 1^*$ and its parameterized version $Q^{para} = \{\langle x, k\rangle \mid x \in Q, k = \max\{1, |x|\}\}$. The k-slide of Q^{para} is then $Q_k^{para} = \{x \in Q \mid |x| = k\}$. That is, $Q_k^{para} = \emptyset$ if $1^k \notin Q$ and $Q_k^{para} = \{1^k\}$ otherwise. This is trivially decidable in polynomial time. Q^{para} is thus in $\mathtt{XP}_{nu}$.

Non-Uniform XP

Let's come back to the idea of having polynomial slices that we mentioned in paraNP.

Definition: Parameterized Complexity Class $\mathtt{XP}_{nu}$

$\mathtt{XP}_{nu}$ is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ whose slices L_k for $k \in \mathbb{N}_{>0}$ are *all in* P.

This class is *non-uniform* in the sense that it contains undecidable problems.

Proof: Consider an undecidable language $Q \subseteq 1^*$ and its parameterized version $Q^{para} = \{\langle x, k\rangle \mid x \in Q, k = \max\{1, |x|\}\}$. The k-slide of Q^{para} is then $Q_k^{para} = \{x \in Q \mid |x| = k\}$. That is, $Q_k^{para} = \emptyset$ if $1^k \notin Q$ and $Q_k^{para} = \{1^k\}$ otherwise. This is trivially decidable in polynomial time. Q^{para} is thus in $\mathtt{XP}_{nu}$.

↳ This is not so nice, let's get rid of the non-uniformity!

(Uniform) XP

Definition: Parameterized Complexity Class XP

XP is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ and a Turing machine $\mathbb{M}$ such that:

- $\mathbb{M}$ runs in time $|x|^{f(k)} + f(k)$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

(Uniform) XP

Definition: Parameterized Complexity Class XP

XP is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ and a Turing machine $\mathbb{M}$ such that:

- $\mathbb{M}$ runs in time $|x|^{f(k)} + f(k)$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

↳ Sometimes the running time is only required to be $|x|^{f(k)}$, it is roughly equivalent but less precise (for when $|x| = 1$).

(Uniform) XP

Definition: Parameterized Complexity Class XP

XP is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ and a Turing machine $\mathbb{M}$ such that:

- $\mathbb{M}$ runs in time $|x|^{f(k)} + f(k)$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

↪ Sometimes the running time is only required to be $|x|^{f(k)}$, it is roughly equivalent but less precise (for when $|x| = 1$).

Once again, it is easy to see that XP is close under FPT-reductions.

(Uniform) XP

Definition: Parameterized Complexity Class XP

XP is the class of all the parameterized languages $L \subseteq \Sigma^* \times \mathbb{N}$ for which there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ and a Turing machine $\mathbb{M}$ such that:

- $\mathbb{M}$ runs in time $|x|^{f(k)} + f(k)$ on $\langle x, k \rangle$;
- $\langle x, k \rangle \in L$ if and only if $\mathbb{M}(\langle x, k \rangle) = 1$.

↳ Sometimes the running time is only required to be $|x|^{f(k)}$, it is roughly equivalent but less precise (for when $|x| = 1$).

Once again, it is easy to see that XP is close under FPT-reductions.

While we have seen parameterized classes that resembles NP, XP plays the role of EXP in parameterized complexity theory. In the following, we will show that XP *is a strict superset of* FPT, mimicking the fact that $\mathtt{P} \subsetneq \mathtt{EXP}$.

An XP-Complete Language

We first show that the following parameterized language is XP-complete under FPT-reductions.

Exp-DTM-Halt	
Instance:	A Turing machine $\mathbb{M}$, $n \in \mathbb{N}$ in unary and $k \in \mathbb{N}$
Parameter:	k
Question:	Does $\mathbb{M}$ accept the empty string ϵ in at most n^k steps?

To show XP-completeness, we will first show that the language *Exp-DTM-Halt is in* XP by presenting an algorithm solving it in time $|x|^{f(k)} + f(k)$. In a second time, we will show that the language is XP-*hard* by presenting an FPT-reduction from an arbitrary problem in XP to Exp-DTM-Halt.

An XP-Complete Language

We first show that the following parameterized language is XP-complete under FPT-reductions.

Exp-DTM-Halt	
Instance:	A Turing machine $\mathbb{M}$, $n \in \mathbb{N}$ in unary and $k \in \mathbb{N}$
Parameter:	k
Question:	Does $\mathbb{M}$ accept the empty string ϵ in at most n^k steps?

To show XP-completeness, we will first show that the language *Exp-DTM-Halt is in XP* by presenting an algorithm solving it in time $|x|^{f(k)} + f(k)$. In a second time, we will show that the language is *XP-hard* by presenting an FPT-reduction from an arbitrary problem in XP to Exp-DTM-Halt.

Proof: An algorithm witnessing membership in XP simply *simulates* $\mathbb{M}$ on ϵ for n^k steps and output the result of the simulation. This is possible since an *efficient universal Turing machine* exists.

Exp-DTM-Halt	
Instance:	A Turing machine $\mathbb{M}$, $n \in \mathbb{N}$ in unary and $k \in \mathbb{N}$
Parameter:	k
Question:	Does $\mathbb{M}$ accept the empty string ϵ in at most n^k steps?

Proof: Let's show XP-*hardness* now. Take any parameterized language $L \subseteq \Sigma^* \times \mathbb{N}$ in XP. Let $f : \mathbb{N} \to \mathbb{N}$ be a computable function and $\mathbb{M}$ a Turing machine *deciding* if $\langle x, k\rangle \in L$ in time $|x|^{f(k)} + f(k)$.

Consider another Turing machine $\mathbb{M}'$ which first *writes* $\langle x, k\rangle$ on its input tape and then *simulates* $\mathbb{M}$ *on* $\langle x, k\rangle$.

Assume without loss of generality that for some computable $g : \mathbb{N} \to \mathbb{N}$, $\mathbb{M}'$ needs at most $(|x| + 2)^{g(k)}$ steps on input $\langle x, k\rangle$.

The function that maps any $\langle x, k\rangle \in L$ to an instance $\langle \mathbb{M}'(\langle x, k\rangle), |x| + 2, g(k)\rangle$ of Exp-DTM-Halt is an FPT-*reduction from* L *to* Exp-DTM-Halt.

The previous result is particularly interesting for us as it allows to show that FPT is a strict subset of XP. This entails that XP-hard languages *cannot* be solved in FPT time.

Theorem:

$\mathtt{FPT} \subsetneq \mathtt{XP}$.

Provable Fixed-Parameter Intractability

The previous result is particularly interesting for us as it allows to show that FPT is a strict subset of XP. This entails that XP-hard languages *cannot* be solved in FPT time.

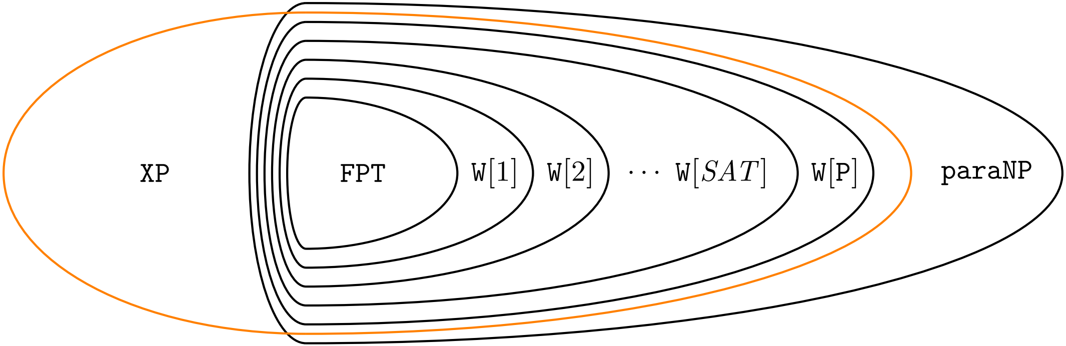
<u>THEOREM</u>:

$\texttt{FPT} \subsetneq \texttt{XP}$.

<u>PROOF</u>: Obviously $\texttt{FPT} \subseteq \texttt{XP}$. Assume now that EXP-DTM-HALT is in FPT. Then, for some $c \in \mathbb{N}$ all the slices of EXP-DTM-HALT are solvable in $\texttt{DTIME}[n^c]$. In particular, the $(c+1)$-slice also is. It implies that $\texttt{DTIME}[n^{c+1}] \subseteq \texttt{DTIME}[n^c]$ which contradicts the *time hierarchy theorem* (see below).

The time hierarchy theorem states that for two time-constructible functions $f, g : \mathbb{N} \to \mathbb{N}$, if $f(n)\log(f(n))$ is $o(g(n))$, then $\texttt{DTIME}[f(n)] \subsetneq \texttt{DTIME}[g(n)]$. Roughly speaking, this says that we can solve strictly more problems when allowing extra running time.

Final Taxonomy of What we Have Seen

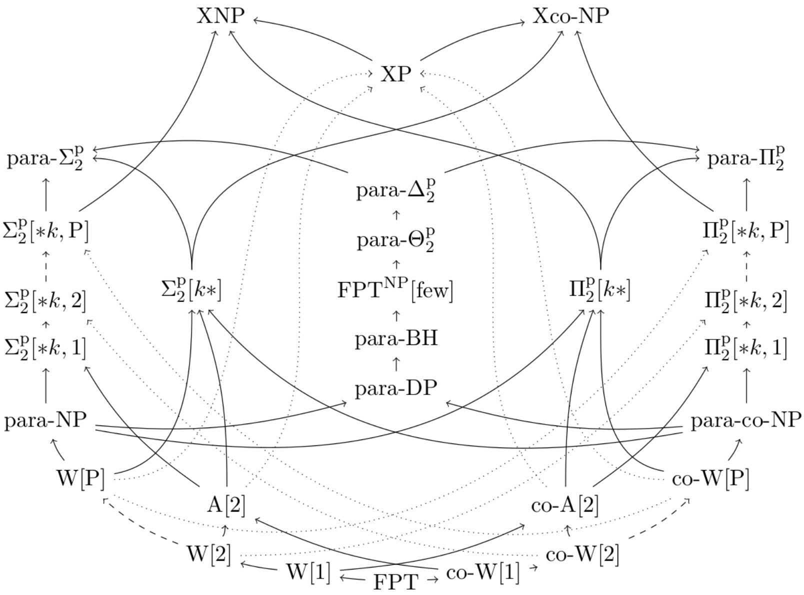


Proposition:

$\mathtt{FPT} \subseteq \mathtt{W}[1] \subseteq \mathtt{W}[2] \subseteq \cdots \subseteq \mathtt{W}[SAT] \subseteq \mathtt{W}[\mathtt{P}] \subseteq \mathtt{paraNP} \cap \mathtt{XP}$.

5. Conclusion

Want More?


XNP
Xco-NP
XP
para-Σ_2^p
para-Π_2^p
para-Δ_2^p
$\Sigma_2^p[*k, P]$
$\Pi_2^p[*k, P]$
para-Θ_2^p
$\Sigma_2^p[*k, 2]$
$\Sigma_2^p[k*]$
FPT^{NP}[few]
$\Pi_2^p[k*]$
$\Pi_2^p[*k, 2]$
$\Sigma_2^p[*k, 1]$
para-BH
$\Pi_2^p[*k, 1]$
para-DP
para-NP
para-co-NP
W[P]
co-W[P]
A[2]
co-A[2]
W[2]
co-W[2]
W[1]
FPT
co-W[1]

Summary

Today, we have:

Summary

Today, we have:

- Recalled the idea behind NP;

Summary

Today, we have:

- Recalled the idea behind NP;
- Tried to define parameterized counterparts of NP first via paraNP and then through the W-hierarchy;

Summary

Today, we have:

- Recalled the idea behind NP;
- Tried to define parameterized counterparts of NP first via paraNP and then through the W-hierarchy;
- Explored the class XP and proved that it contains fixed-parameter intractable problems.

Summary

Today, we have:

- Recalled the idea behind NP;
- Tried to define parameterized counterparts of NP first via paraNP and then through the W-hierarchy;
- Explored the class XP and proved that it contains fixed-parameter intractable problems.

Summary

Today, we have:

- Recalled the idea behind NP;
- Tried to define parameterized counterparts of NP first via paraNP and then through the W-hierarchy;
- Explored the class XP and proved that it contains fixed-parameter intractable problems.

In the last lecture of the week, Ronald will tell you about some more advanced techniques about lower bounds for kernelization.