Lower bounds for polynomial kernelization

Part 1

Michał Pilipczuk

Institutt for Informatikk, Universitetet i Bergen

August 19th, 2014
Outline

**Goal**: how to prove that some problems do not admit polynomial kernelization algorithms?

**Part 1**:  
- Introduction of the (cross)-composition framework.  
- Basic examples.

**Part 2**:  
- PPT reductions.  
- Case study of several cross-compositions.  
- Weak compositions.
This will be a complexity theory lecture.
- This will be a complexity theory lecture.
- Unparameterized problems = languages over $\Sigma = \text{subsets of } \Sigma^*$, for a constant size alphabet $\Sigma$. 
Disclaimer

- This will be a complexity theory lecture.
- Unparameterized problems = languages over $\Sigma = \text{subsets of } \Sigma^*$, for a constant size alphabet $\Sigma$.
- Parameterized problems are sets of pairs $(x, k)$, where $x \in \Sigma^*$ and $k$ is a nonnegative integer.
Disclaimer

- This will be a complexity theory lecture.
- Unparameterized problems = languages over $\Sigma = \text{subsets of } \Sigma^*$, for a constant size alphabet $\Sigma$.
- Parameterized problems are sets of pairs $(x, k)$, where $x \in \Sigma^*$ and $k$ is a nonnegative integer.
- *Unparameterized variant:* $k$ is appended to $x$ in unary.
Kernelization — recap
Kernelization — recap

instance of $L$
Kernelization — recap

instance of $L$
Kernelization — recap

instance of $L$

$P$-time

instance of $L$
Kernelization — recap

instance of $L_k$ 

$P$-time

instance of $L$

size $\leq f(k)$
If a decidable problem admits a kernelization algorithm, then it is FPT.
If a decidable problem admits a kernelization algorithm, then it is FPT.

Any FPT problem admits a kernelization algorithm:
Kernelization and FPT

- If a decidable problem admits a kernelization algorithm, then it is FPT.
- Any FPT problem admits a kernelization algorithm:
  - Let \((x, k)\) be the input instance.
Kernelization and FPT

- If a decidable problem admits a kernelization algorithm, then it is FPT.
- Any FPT problem admits a kernelization algorithm:
  - Let \((x, k)\) be the input instance.
  - If \(|x| \leq f(k)\), then we already have a kernel.
Kernelization and FPT

- If a decidable problem admits a kernelization algorithm, then it is FPT.
- Any FPT problem admits a kernelization algorithm:
  - Let \((x, k)\) be the input instance.
  - If \(|x| \leq f(k)\), then we already have a kernel.
  - Otherwise \(f(k) \cdot |x|^c = O(|x|^{c+1})\).
If a decidable problem admits a kernelization algorithm, then it is FPT.

Any FPT problem admits a kernelization algorithm:

- Let \((x, k)\) be the input instance.
- If \(|x| \leq f(k)\), then we already have a kernel.
- Otherwise \(f(k) \cdot |x|^c = \mathcal{O}(|x|^{c+1})\).

Question of existence of any kernel is equivalent to being FPT.
Kernelization and FPT

- If a decidable problem admits a kernelization algorithm, then it is FPT.
- Any FPT problem admits a kernelization algorithm:
  - Let \((x, k)\) be the input instance.
  - If \(|x| \leq f(k)\), then we already have a kernel.
  - Otherwise \(f(k) \cdot |x|^c = O(|x|^{c+1})\).
- Question of existence of any kernel is equivalent to being FPT.
- We are interested in polynomial kernels, where \(f\) is a polynomial.
If a decidable problem admits a kernelization algorithm, then it is FPT.

Any FPT problem admits a kernelization algorithm:
- Let \((x, k)\) be the input instance.
- If \(|x| \leq f(k)\), then we already have a kernel.
- Otherwise \(f(k) \cdot |x|^c = O(|x|^{c+1})\).

Question of existence of any kernel is equivalent to being FPT.

We are interested in polynomial kernels, where \(f\) is a polynomial.

Before 2008, no tool to classify FPT problems wrt. whether they have polykernels or not.
Motivating intuition

Consider the \textit{k-PATH} problem: verify whether the input graph contains a simple path on \textit{k} vertices.
Consider the *k*-PATH problem: verify whether the input graph contains a simple path on *k* vertices.

Suppose for a moment that *k*-PATH admits a kernel that has always, say, at most *k*³ vertices.
Motivating intuition

- Consider the \(k\)-PATH problem: verify whether the input graph contains a simple path on \(k\) vertices.
- Suppose for a moment that \(k\)-PATH admits a kernel that has always, say, at most \(k^3\) vertices.
- Take \(t = k^7\) instances \((G_1, k), (G_2, k), \ldots, (G_t, k)\).
Consider the $k$-PATH problem: verify whether the input graph contains a simple path on $k$ vertices.

Suppose for a moment that $k$-PATH admits a kernel that has always, say, at most $k^3$ vertices.

Take $t = k^7$ instances $(G_1, k), (G_2, k), \ldots, (G_t, k)$.

Let $H$ be a disjoint union of $G_1, G_2, \ldots, G_t$. Then the answer to $(H, k)$ is YES if and only if the answer to any $(G_i, k)$ is YES.
Motivating intuition

- Consider the $k$-PATH problem: verify whether the input graph contains a simple path on $k$ vertices.
- Suppose for a moment that $k$-PATH admits a kernel that has always, say, at most $k^3$ vertices.
- Take $t = k^7$ instances $(G_1, k)$, $(G_2, k)$, $\ldots$, $(G_t, k)$.
- Let $H$ be a disjoint union of $G_1$, $G_2$, $\ldots$, $G_t$. Then the answer to $(H, k)$ is YES if and only if the answer to any $(G_i, k)$ is YES.
- Apply kernelization to $(H, k)$ obtaining an instance with $k^3$ vertices, encodable in $k^6$ bits.
**Motivating intuition**

- **Intuition**: The final number of bits is much less than the number input instances. Most of the instances must have been **discarded completely**.
Kernelization and Compression

**Kernelization**

Instance of $L$ \( k \)

\[ P\text{-time} \]

Instance of $L$

\[ \text{size } \leq p(k) \]
**KERNELIZATION**

\[ \text{instance of } L \xrightarrow{P\text{-time}} \text{instance of } L \quad \text{size } \leq p(k) \]

**COMPRESION**

\[ \text{instance of } L \xrightarrow{P\text{-time}} \text{instance of } R \quad \text{any} \quad \text{size } \leq p(k) \]
Kernelization and Compression

**KERNELIZATION**

Instance of $L \xrightarrow{P\text{-time}} \text{instance of } L$

Instance of $L$ size $\leq p(k)$

**COMPRESSION**

Instance of $L \xrightarrow{P\text{-time}} \text{instance of } R$ (any)

Instance of $R$ bitsize $\leq p(k)$
A polynomial kernelization is always a polynomial compression.
Kernelization and Compression

- A polynomial kernelization is always a polynomial compression.
- A polynomial compression can be turned into a polynomial kernelization provided that there is a $\mathbb{P}$-reduction from $R$ to $L$. 

Note: There are examples when a poly-compression is known but a poly-kernel is not known, because it is unclear whether $R$ is in $\mathbb{NP}$.
A polynomial kernelization is always a polynomial compression.

A polynomial compression can be turned into a polynomial kernelization provided that there is a $\mathbf{P}$-reduction from $R$ to $L$.

- For instance, when $R \in \mathbf{NP}$ and $L$ is $\mathbf{NP}$-hard.
Kernelization and Compression

- A polynomial kernelization is always a polynomial compression.
- A polynomial compression can be turned into a polynomial kernelization provided that there is a P-reduction from $R$ to $L$.
  - For instance, when $R \in \text{NP}$ and $L$ is $\text{NP}$-hard.
- **Note**: There are examples when a poly-compression is known but a poly-kernel is not known, because it is unclear whether $R$ is in $\text{NP}$.
Let $L, R$ be unparameterized languages.
Let $L, R$ be unparameterized languages.

**OR-distillation of $L$ into $R$**

- **Input:** Strings $x_1, x_2, \ldots, x_t$, each of length at most $k$.
- **Time:** $\text{poly}(t + \sum_{i=1}^{t} |x_i|)$.
- **Output:** One string $y$ such that
  
  (a) $|y| = \text{poly}(k)$, and
  
  (b) $y \in R$ if and only if $x_i \in L$ for at least one $i$. 
OR-distillation on picture
OR-distillation on picture

\[ \leq k \quad \leq k \quad \leq k \quad \leq k \quad \leq k \quad \leq k \quad \leq k \quad \leq k \quad \leq k \]

\( t \) instances
OR-distillation on picture

\[ P\text{-time} \leq \text{poly}(k) \]

Michał Pilipczuk
Kernelization lower bounds, part 1
OR-distillation on picture

\[ \leq k \leq k \leq k \leq k \leq k \leq k \leq k \leq k \leq k \]

\[ P\text{-time} \leq \text{poly}(k) \]
OR-distillation

- **Grocery intuition:**
OR-distillation

**Grocery intuition:**

- Suppose input instances are apples, and the OR-distillation algorithm is a blender.
**OR-distillation**

**Grocery intuition:**
- Suppose input instances are apples, and the OR-distillation algorithm is a blender.
- If one of the apples was rotten, then the blend must be untasty.
**OR-distillation**

- **Grocery intuition:**
  - Suppose input instances are apples, and the OR-distillation algorithm is a blender.
  - If one of the apples was rotten, then the blend must be untasty.
  - If the blend is much smaller than the total input fruit mass, then it will be possible that a computationally too weak blender will lose the rotten apple.
**OR-distillation**

- **Grocery intuition:**
  - Suppose input instances are apples, and the OR-distillation algorithm is a blender.
  - If one of the apples was rotten, then the blend must be untasty.
  - If the blend is much smaller than the total input fruit mass, then it will be possible that a computationally too weak blender will lose the rotten apple.

- **OR-$L$: language of strings** $x_1 \# x_2 \# \ldots \# x_t$ such that $x_i \in L$ for at least one $i$. 

---

Michał Pilipczuk

Kernelization lower bounds, part 1
**OR-distillation**

- **Grocery intuition:**
  - Suppose input instances are apples, and the OR-distillation algorithm is a blender.
  - If one of the apples was rotten, then the blend must be untasty.
  - If the blend is much smaller than the total input fruit mass, then it will be possible that a computationally too weak blender will lose the rotten apple.

- **OR-\(L\):** language of strings \(x_1 \# x_2 \# \ldots \# x_t\) such that \(x_i \in L\) for at least one \(i\).

- **OR-distillation of \(L\) into \(R\)** is a polynomial compression of OR-\(L\) into \(R\), where OR-\(L\) is parameterized by \(\max |x_i|\).
Backbone theorem

**OR-distillation theorem**

SAT does not admit an OR-distillation algorithm into any language $R$, unless $\textbf{NP} \subseteq \textbf{coNP}/\text{poly}$.
Backbone theorem

**OR-distillation theorem**

SAT does not admit an OR-distillation algorithm into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.

**Corollary**

No $\text{NP}$-hard problem admits an OR-distillation algorithm into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
The assumption

- Assumption $\textbf{NP} \subseteq \textit{coNP/poly}$ may seem mysterious.
The assumption

- Assumption $\text{NP} \subseteq \text{coNP}/\text{poly}$ may seem mysterious.
- More known variant: $\text{NP} \neq \text{coNP}$. 
The assumption

- Assumption $\text{NP} \subseteq \text{coNP}/\text{poly}$ may seem mysterious.
- More known variant: $\text{NP} \neq \text{coNP}$.
  - Verifying proofs in $\text{P}$-time is not equivalent to verifying counterexamples in $\text{P}$-time.
The assumption

- Assumption $\text{NP} \subseteq \text{coNP}/\text{poly}$ may seem mysterious.
- More known variant: $\text{NP} \neq \text{coNP}$.
  - Verifying proofs in $\text{P}$-time is not equivalent to verifying counterexamples in $\text{P}$-time.
- $\text{NP} \subseteq \text{coNP}/\text{poly}$ is strengthening of this by saying that verifying proofs cannot be simulated by verifying counterexamples even if we allow polynomial advice.
The assumption

- Assumption $\mathbf{NP} \subseteq \mathbf{coNP}/\text{poly}$ may seem mysterious.
- More known variant: $\mathbf{NP} \neq \mathbf{coNP}$.
  - Verifying proofs in $\mathbf{P}$-time is not equivalent to verifying counterexamples in $\mathbf{P}$-time.
- $\mathbf{NP} \subseteq \mathbf{coNP}/\text{poly}$ is strengthening of this by saying that verifying proofs cannot be simulated by verifying counterexamples even if we allow polynomial advice.
- It is known that $\mathbf{NP} \subseteq \mathbf{coNP}/\text{poly}$ implies that $\mathbf{PH} = \Sigma_3^P$. 

The assumption

- Assumption $\text{NP} \subseteq \text{coNP}/\text{poly}$ may seem mysterious.
- More known variant: $\text{NP} \neq \text{coNP}$.
  - Verifying proofs in $\text{P}$-time is not equivalent to verifying counterexamples in $\text{P}$-time.
- $\text{NP} \subseteq \text{coNP}/\text{poly}$ is strengthening of this by saying that verifying proofs cannot be simulated by verifying counterexamples even if we allow polynomial advice.
- It is known that $\text{NP} \subseteq \text{coNP}/\text{poly}$ implies that $\text{PH} = \Sigma_3^\text{P}$.
- Not as bad as $\text{P} = \text{NP}$, but pretty severe.
A glimpse into the proof

- The proof is purely information-theoretical.
A glimpse into the proof

- The proof is purely information-theoretical.
- Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
A glimpse into the proof

- The proof is purely information-theoretical.
- Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
- An algorithm in \( \mathbf{P} \) cannot guess, which instance is more prone to have a positive answer, so we need to store information about all of them.
A glimpse into the proof

- The proof is purely information-theoretical.
- Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
- An algorithm in $\mathcal{P}$ cannot guess, which instance is more prone to have a positive answer, so we need to store information about all of them.
- **Main trick:**
A glimpse into the proof

- The proof is purely information-theoretical.
- Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
- An algorithm in \( \mathbf{P} \) cannot guess, which instance is more prone to have a positive answer, so we need to store information about all of them.
- Main trick:
  - show that the space for kernels is so small that one can find a linear number of representative kernels;
A glimpse into the proof

- The proof is purely information-theoretical.
- Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
- An algorithm in \( \mathbf{P} \) cannot guess, which instance is more prone to have a positive answer, so we need to store information about all of them.
- **Main trick:**
  - show that the space for kernels is so small that one can find a linear number of \textit{representative} kernels;
  - plug these kernels as the advice to a \textbf{coNP}-algorithm for SAT.
The proof is purely information-theoretical.
Intuitively, the space for possible kernels is too small to store information about very long sequences of instances.
An algorithm in $\mathbf{P}$ cannot guess, which instance is more prone to have a positive answer, so we need to store information about all of them.

**Main trick:**
- show that the space for kernels is so small that one can find a linear number of representative kernels;
- plug these kernels as the advice to a $\text{coNP}$-algorithm for $\text{SAT}$.

**Look into the book.**
Let $L$ be a *parameterized* language.
Let $L$ be a *parameterized* language.

**OR-composition algorithm for $L$**

**Input:** Instances $(x_1, k), (x_2, k), \ldots, (x_t, k)$.

**Time:** $\text{poly}(t + \sum_{i=1}^{t} |x_i| + k)$.

**Output:** One instance $(y, k^*)$ such that

(a) $k^* = \text{poly}(k)$, and
(b) $(y, k^*) \in L$ iff $(x_i, k) \in L$ for at least one $i$. 
OR-composition on picture
OR-composition on picture

$t$ instances
OR-composition on picture

$t$ instances

$P$-time
OR-composition on picture

$t$ instances

$P$-time

poly($k$)
If a parameterized problem $L$ admits an OR-composition algorithm, and the unparameterized version of $L$ is $\text{NP}$-hard, then $L$ does not admit a polynomial kernel unless $\text{NP} \subseteq \text{coNP}/\text{poly}$. 
Proof
Proof
Proof
Proof

Michał Pilipczuk  Kernelization lower bounds, part 1  19/32
Proof

\[
\begin{align*}
\text{OR-SAT} & \xrightarrow{\text{NP-hrd}} \text{1} & \xrightarrow{\text{NP-hrd}} \text{1} & \xrightarrow{\text{NP-hrd}} \text{2} & \xrightarrow{\text{NP-hrd}} \text{2} & \xrightarrow{\text{NP-hrd}} k' & \xrightarrow{\text{NP-hrd}} k'
\end{align*}
\]

\[
\begin{align*}
L & \xrightarrow{\text{cmp}} \text{poly}(k) & \xrightarrow{\text{cmp}} \text{poly}(k) & \xrightarrow{\text{cmp}} \text{poly}(k)
\end{align*}
\]
Proof
Proof
Proof

\[
\text{OR-SAT} \quad \text{NP-hrd} \quad \text{NP-hrd} \quad \text{NP-hrd} \quad \text{NP-hrd} \quad \text{NP-hrd} \quad \text{NP-hrd} \quad \text{NP-hrd} \\
\text{L} \quad \text{L} \quad \text{L} \quad \text{L} \quad \text{L} \quad \text{L} \quad \text{L} \\
\text{OR-L} \quad \text{poly}(k) \quad \text{poly}(k) \quad \text{poly}(k) \\
\text{kern} \quad \text{kern} \quad \text{kern} \\
\text{Michał Pilipczuk} 
\]
Corollaries

- $k$-Path does not admit poly-kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$. 
Corollaries

- **k-Path** does not admit poly-kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
- **Composition**: Take disjoint union of graphs and the same parameter.
**Corollaries**

- **$k$-PATH** does not admit poly-kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
- **Composition**: Take disjoint union of graphs and the same parameter.
  - A graph admits a $k$-path iff any of its connected components does.
Corollaries

- **$k$-Path** does not admit poly-kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
- **Composition**: Take disjoint union of graphs and the same parameter.
  - A graph admits a $k$-path iff any of its connected components does.
- **Same for $k$-Cycle**; this opens a bag of results.
Corollaries

- **k-Path** does not admit poly-kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
- **Composition**: Take disjoint union of graphs and the same parameter.
  - A graph admits a $k$-path iff any of its connected components does.
- Same for $k$-Cycle; this opens a bag of results.
- Today, investigating the existence of a polynomial kernel is an immediate second goal after showing that a problem is FPT.
Does the proof actually exclude even polynomial compression, not just kernelization?
Does the proof actually exclude even polynomial compression, not just kernelization?

- Sure, we will just end up with an instance of OR-$R$. 

Can we add more refined bucket sorting? For instance, also by the number of vertices in the graph?

- Yes, as long as we have polynomial number of buckets.
Does the proof actually exclude even polynomial compression, not just kernelization?
  - Sure, we will just end up with an instance of OR-$R$.

Do we need to start the composition with the same language $L$ as we apply the compression to?

Yes, as long as we have polynomial number of buckets.
Adding features

- Does the proof actually exclude even polynomial compression, not just kernelization?
  - Sure, we will just end up with an instance of OR-\( R \).

- Do we need to start the composition with the same language \( L \) as we apply the compression to?
  - No, the composition algorithm can compose instances of any language \( Q \) into one instance of \( L \).
Adding features

- Does the proof actually exclude even polynomial compression, not just kernelization?
  - Sure, we will just end up with an instance of OR-$R$.

- Do we need to start the composition with the same language $L$ as we apply the compression to?
  - No, the composition algorithm can compose instances of any language $Q$ into one instance of $L$.

- Can we add more refined bucket sorting? For instance, also by the number of vertices in the graph?
Adding features

- Does the proof actually exclude even polynomial compression, not just kernelization?
  - Sure, we will just end up with an instance of OR-$R$.
- Do we need to start the composition with the same language $L$ as we apply the compression to?
  - No, the composition algorithm can compose instances of any language $Q$ into one instance of $L$.
- Can we add more refined bucket sorting? For instance, also by the number of vertices in the graph?
  - Yes, as long as we have polynomial number of buckets.
Adding more features

- How large can $t$ be?

Well, not larger than $(|\Sigma| + 1)^k$, as we may remove duplicates of the input instances. Hence, we may assume that $\log t = O(k)$, which means that the parameter of the composed instance may depend polynomially on both $k$ and $\log t$.

Observed also earlier via different arguments (Dom, Lokshtanov, and Saurabh; ICALP 2009).
Adding more features

- How large can $t$ be?
- Well, not larger than $(|\Sigma| + 1)^k$, as we may remove duplicates of the input instances.
Adding more features

- How large can $t$ be?
- Well, not larger than $(\sum + 1)^k$, as we may remove duplicates of the input instances.
- Hence, we may assume that $\log t = \mathcal{O}(k)$,
Adding more features

- How large can $t$ be?
- Well, not larger than $(|\Sigma| + 1)^k$, as we may remove duplicates of the input instances.
- Hence, we may assume that $\log t = O(k)$,
- which means that the parameter of the composed instance may depend polynomially on both $k$ and $\log t$. 

Observed also earlier via different arguments (Dom, Lokshtanov, and Saurabh; ICALP 2009).
How large can $t$ be?
Well, not larger than $(|\Sigma| + 1)^k$, as we may remove duplicates of the input instances.
Hence, we may assume that $\log t = \mathcal{O}(k)$,
which means that the parameter of the composed instance may depend polynomially on both $k$ and $\log t$.

Observed also earlier via different arguments (Dom, Lokshtanov, and Saurabh; ICALP 2009).
After invention of the composition framework

- A huge amount of no-poly-kernel results.
After invention of the composition framework

- A huge amount of no-poly-kernel results.
- Most of the works use a subset of mentioned features.
A huge amount of no-poly-kernel results.

Most of the works use a subset of mentioned features.

**STACS 2011:** Bodlaender, Jansen, and Kratsch propose a new formalism, dubbed *cross-composition*, that gathers all these features.
An equivalence relation $\mathcal{R}$ on $\Sigma^*$ is called a polynomial equivalence relation if the following two conditions hold:

- Checking whether two strings $x, y \in \Sigma^*$ are $\mathcal{R}$-equivalent can be done in $\text{poly}(|x| + |y|)$ time.
- $\mathcal{R}$ partitions strings of length at most $n$ into $\text{poly}(n)$ equivalence classes.
An equivalence relation $\mathcal{R}$ on $\Sigma^*$ is called a polynomial equivalence relation if the following two conditions hold:

- Checking whether two strings $x, y \in \Sigma^*$ are $\mathcal{R}$-equivalent can be done in $\text{poly}(|x| + |y|)$ time.
- $\mathcal{R}$ partitions strings of length at most $n$ into $\text{poly}(n)$ equivalence classes.

**Examples:**
Polynomial equivalence relation

An equivalence relation $\mathcal{R}$ on $\Sigma^*$ is called a polynomial equivalence relation if the following two conditions hold:

- Checking whether two strings $x, y \in \Sigma^*$ are $\mathcal{R}$-equivalent can be done in $\text{poly}(|x| + |y|)$ time.
- $\mathcal{R}$ partitions strings of length at most $n$ into $\text{poly}(n)$ equivalence classes.

**Examples:**
- partitioning with respect to the number of vertices of the graph;
An equivalence relation $\mathcal{R}$ on $\Sigma^*$ is called a polynomial equivalence relation if the following two conditions hold:

- Checking whether two strings $x, y \in \Sigma^*$ are $\mathcal{R}$-equivalent can be done in $\text{poly}(|x| + |y|)$ time.
- $\mathcal{R}$ partitions strings of length at most $n$ into $\text{poly}(n)$ equivalence classes.

**Examples:**
- partitioning with respect to the number of vertices of the graph;
- or with respect to (i) the number of vertices, (ii) the number of edges, (iii) size of the maximum matching, (iv) budget.
An unparameterized problem $Q$ \textit{cross-composes} into a parameterized problem $L$, if there exists a polynomial equivalence relation $\mathcal{R}$ and an algorithm that, given $\mathcal{R}$-equivalent strings $x_1, x_2, \ldots, x_t$, in time $\text{poly} \left( t + \sum_{i=1}^{t} |x_i| \right)$ produces one instance $(y, k^*)$ such that

- $(y, k^*) \in L$ iff $x_i \in Q$ for at least one $i = 1, 2, \ldots, t$,
- $k^* = \text{poly} \left( \log t + \max_{i=1}^{t} |x_i| \right)$. 

Cross-composition theorem

Bodlaender et al.; STACS 2011, SIDMA 2014

If some $\mathsf{NP}$-hard problem $Q$ cross-composes into $L$, then $L$ does not admit a polynomial compression into any language $\mathcal{R}$, unless $\mathsf{NP} \subseteq \mathsf{coNP}/\text{poly}$. 

Michał Pilipczuk

Kernelization lower bounds, part 1
An unparameterized problem $Q$ **cross-composes** into a parameterized problem $L$, if there exists a polynomial equivalence relation $R$ and an algorithm that, given $R$-equivalent strings $x_1, x_2, \ldots, x_t$, in time $\text{poly} \left( t + \sum_{i=1}^{t} |x_i| \right)$ produces one instance $(y, k^*)$ such that

- $(y, k^*) \in L$ iff $x_i \in Q$ for at least one $i = 1, 2, \ldots, t$,
- $k^* = \text{poly} \left( \log t + \max_{i=1}^{t} |x_i| \right)$.

**Cross-composition theorem**

If some $\text{NP}$-hard problem $Q$ cross-composes into $L$, then $L$ does not admit a polynomial compression into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
Proof

$k = \max |x_i|, \quad \log t = O(k)$
Proof

\( k = \max |x_i|, \quad \log t = \mathcal{O}(k) \)
Proof

\[ k = \max |x_i|, \quad \log t = \mathcal{O}(k) \]
Proof

\[ k = \max |x_i|, \quad \log t = \mathcal{O}(k) \]
Proof

\[ k = \max |x_i|, \quad \log t = O(k) \]
Proof

\[ Q = \max |x_i|, \quad \log t = O(k) \]
Applications

- Original application of Bodlaender, Jansen and Kratsch was that of *structural parameters*.
Applications

- Original application of Bodlaender, Jansen and Kratsch was that of *structural parameters*.
- In fact, cross-composition is a good framework to express also all the previous results.
Applications

- Original application of Bodlaender, Jansen and Kratsch was that of *structural parameters*.
- In fact, cross-composition is a good framework to express also all the previous results.
- **Plan for now:** show a few cross-compositions and give intuition about basic tricks.
**Application 1: SET SPLITTING**

<table>
<thead>
<tr>
<th><strong>Input:</strong></th>
<th>Universe $U$ and family of subsets $\mathcal{F} \subseteq 2^U$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameter:</strong></td>
<td>$</td>
</tr>
<tr>
<td><strong>Question:</strong></td>
<td>Does there exist a colouring $C : U \rightarrow {B, W}$ such that every set $X \in \mathcal{F}$ is split, i.e., contains a black and a white element?</td>
</tr>
</tbody>
</table>
Application 1: **Set Splitting**

**Set Splitting**

**Input:** Universe $U$ and family of subsets $\mathcal{F} \subseteq 2^U$

**Parameter:** $|U|$

**Question:** Does there exist a colouring $C : U \rightarrow \{B, W\}$ such that every set $X \in \mathcal{F}$ is split, i.e., contains a black and a white element?

- We show a cross-composition of Set Splitting into itself.
Application 1: **Set Splitting**

**Set Splitting**

**Input:** Universe $U$ and family of subsets $\mathcal{F} \subseteq 2^U$

**Parameter:** $|U|$

**Question:** Does there exist a colouring $\mathcal{C} : U \rightarrow \{B, W\}$ such that every set $X \in \mathcal{F}$ is split, i.e., contains a black and a white element?

- We show a cross-composition of **Set Splitting** into itself.
- We may assume that the universes are of the same size, hence we think of them as of one, common universe.
Application 1: **Set Splitting**

**Set Splitting**

**Input:** Universe $U$ and family of subsets $\mathcal{F} \subseteq 2^U$

**Parameter:** $|U|$

**Question:** Does there exist a colouring $\mathcal{C} : U \to \{B, W\}$ such that every set $X \in \mathcal{F}$ is split, i.e., contains a black and a white element?

- We show a cross-composition of *Set Splitting* into itself.
- We may assume that the universes are of the same size, hence we think of them as of one, common universe.
- Assume that $t$ is a power of 2 (by copying the instances).
Cross-composing into **Set Splitting**

**Input**: Instances \((U, \mathcal{F}^i)\)

**Output**: Instance \((U^*, \mathcal{F}^*)\)
Cross-composing into **Set Splitting**

**Input:** Instances \((U, \mathcal{F}^i)\)

**Output:** Instance \((U^*, \mathcal{F}^*)\)

\[
|U^*| = |U| + 2 \log t + 2
\]

\[
|U^*| = |U| + 2 \log t + 2
\]

\(F^*\) consists of:

- \(1 + \log t\) 2-element sets for pairs,
- \(\forall X \in \mathcal{F}^i\), two sets \(X^*_0, X^*_1\),
- \(X^*_0\): left special guy, and binary encoding of \(i\) in IS
- \(X^*_1\): reverse \(X^*_0\) on IS

Take any solution \(C\) There is exactly one index \(i\) with monochromatic parts from IS. \((\Rightarrow)\): \(C\) on IS defines, which instance must be solved in PL

\((\Leftarrow)\): If \((U, \mathcal{F}^i)\) is solvable, we set IS accordingly, and solve this instance in PL. Remaining sets are split for free.
Cross-composing into \textsc{Set Splitting}

\begin{itemize}
\item \textbf{Input}: \text{Instances} $(U, \mathcal{F}^i)$
\item \textbf{Output}: \text{Instance} $(U^*, \mathcal{F}^*)$
\end{itemize}

\begin{center}
\textbf{INSTANCE SELECTOR}
\end{center}

\begin{center}
$1 + \log t$ pairs of vertices
\end{center}

\begin{center}
\textbf{PLAYGROUND}
\end{center}

\begin{center}
\text{joint universe} $U$
\end{center}
Cross-composing into Set Splitting

**INSTANCE SELECTOR**

*Input*: Instances \((U, \mathcal{F}^i)\)

*Output*: Instance \((U^*, \mathcal{F}^*)\)

\[|U^*| = |U| + 2 \log t + 2\]
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

Input: Instances \((U, \mathcal{F}^i)\)

Output: Instance \((U^*, \mathcal{F}^*)\)

\[|U^*| = |U| + 2 \log t + 2\]

\(\mathcal{F}^*\) consists of:

**PLAYGROUND**

joint universe \(U\)
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

*Input:* Instances \((U, F^i)\)

*Output:* Instance \((U^*, F^*)\)

\[|U^*| = |U| + 2\log t + 2\]

\(F^*\) consists of:
- \(1 + \log t\) 2-element sets for pairs,
- \(\forall X \in F^i,\) two sets \(X^0, X^1\), left special guy, and binary encoding of \(i\) in IS
- reverse \(X^0\) on IS

**PLAYGROUND**

joint universe \(U\)

Take any solution \(C\). There is exactly one index \(i\) with monochromatic parts from IS.\(\Rightarrow C\) on IS defines, which instance must be solved in PL.

\(\Leftarrow\) If \((U, F^i)\) is solvable, we set IS accordingly, and solve this instance in PL. Remaining sets are split for free.
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

1 + log \( t \) pairs of vertices

- **Input:** Instances \((U, \mathcal{F}^i)\)
- **Output:** Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

- \( \mathcal{F}^* \) consists of:
  - 1 + log \( t \) 2-element sets for pairs,
  - \( \forall X \in \mathcal{F}^i \), two sets \( X_0^*, X_1^* \)

**PLAYGROUND**

Joint universe \( U \)

Michał Pilipczuk: Kernelization lower bounds, part 1
Cross-composing into Set Splitting

**INSTANCE SELECTOR**

1 + \log t pairs of vertices

- **Input**: Instances \((U, \mathcal{F}^i)\)
- **Output**: Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

\(\mathcal{F}^*\) consists of:
- 1 + \log t 2-element sets for pairs,
- \(\forall X \in \mathcal{F}^i\), two sets \(X^*_0, X^*_1\)

- \(X^*_0\): \(X\), left special guy, and binary encoding of \(i\) in IS

**PLAYGROUND**

Joint universe \(U\)
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

1 + \( \log t \) pairs of vertices

Input: Instances \((U, \mathcal{F}^i)\)

Output: Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

\( \mathcal{F}^* \) consists of:
1 + \( \log t \) 2-element sets for pairs,
\( \forall X \in \mathcal{F}^i \), two sets \( X_0^*, X_1^* \)

\( X_0^* \): \( X \), left special guy, and binary encoding of \( i \) in IS

\( X_1^* \): reverse \( X_0^* \) on IS

**PLAYGROUND**

joint universe \( U \)
Cross-composing into \textbf{Set Splitting}

\section*{INSTANCE SELECTOR}
\begin{itemize}
\item \textbf{Input}: Instances \((U, \mathcal{F}^i)\)
\item \textbf{Output}: Instance \((U^*, \mathcal{F}^*)\)
\end{itemize}

\[|U^*| = |U| + 2\log t + 2\]

\(\mathcal{F}^*\) \textbf{consists of}:
1 + log \(t\) 2-element sets for pairs,
\(\forall X \in \mathcal{F}^i\), two sets \(X_0^*, X_1^*\)

\begin{itemize}
\item Take any solution \(C\)
\end{itemize}

\section*{PLAYGROUND}
\begin{itemize}
\item joint universe \(U\)
\end{itemize}
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

1 + log \( t \) pairs of vertices

**Input:** Instances \((U, \mathcal{F}^i)\)

**Output:** Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

\(\mathcal{F}^*\) **consists of:**

1 + log \( t \) 2-element sets for pairs,

\(\forall X \in \mathcal{F}^i\), two sets \(X^*_0, X^*_1\)

There is exactly one index \( i \) with monochromatic parts from \( IS \).

**PLAYGROUND**

**joint universe** \( U \)

Take any solution \( C \)

There is exactly one index \( i \) with monochromatic parts from \( IS \).
Cross-composing into \textbf{Set Splitting}

**INSTANCE SELECTOR**

\[ 1 + \log t \] pairs of vertices

\[
\begin{align*}
    \text{Input:} & \text{ Instances } (U, \mathcal{F}^i) \\
    \text{Output:} & \text{ Instance } (U^*, \mathcal{F}^*) \\
    |U^*| &= |U| + 2 \log t + 2 \\
    \mathcal{F}^* \text{ consists of:} \\
    &1 + \log t \text{ 2-element sets for pairs,} \\
    &\forall X \in \mathcal{F}^i, \text{ two sets } X_0^*, X_1^* \\
\end{align*}
\]

Take any solution \( C \)

There is exactly one index \( i \) with monochromatic parts from \( IS \).

**PLAYGROUND**

\[ \text{joint universe } U \]
Cross-composing into Set Splitting

**INSTANCE SELECTOR**

1 + log \( t \) pairs of vertices

**Input:** Instances \((U, \mathcal{F}^i)\)

**Output:** Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

\(\mathcal{F}^*\) consists of:

- \(1 + \log t\) 2-element sets for pairs,
- \(\forall X \in \mathcal{F}^i\), two sets \(X_0^*, X_1^*\)

Take any solution \(C\).

There is exactly one index \(i\) with monochromatic parts from IS.

PLAYGROUND

joint universe \(U\)
Cross-composing into **Set Splitting**

**INSTANCE SELECTOR**

1 + log t pairs of vertices

**Input:** Instances \((U, \mathcal{F}^i)\)

**Output:** Instance \((U^*, \mathcal{F}^*)\)

\[|U^*| = |U| + 2 \log t + 2\]

\(\mathcal{F}^*\) consists of:

1 + log t 2-element sets for pairs,

\(\forall X \in \mathcal{F}^i\), two sets \(X_0^*, X_1^*\)

Take any solution \(C\)

There is exactly one index \(i\) with monochromatic parts from IS.

\((\Rightarrow):\) \(C\) on IS defines, which instance must be solved in PL

**PLAYGROUND**

joint universe \(U\)
Cross-composing into Set Splitting

**INSTANCE SELECTOR**

1 + log \( t \) pairs of vertices

- **Input**: Instances \((U, \mathcal{F}^i)\)
- **Output**: Instance \((U^*, \mathcal{F}^*)\)

\[ |U^*| = |U| + 2 \log t + 2 \]

- \( \mathcal{F}^* \) consists of:
  - \( 1 + \log t \) 2-element sets for pairs,
  - \( \forall X \in \mathcal{F}^i \), two sets \( X_0^*, X_1^* \)

Take any solution \( C \)

There is exactly one index \( i \) with monochromatic parts from IS.

\((\Rightarrow)\): \( C \) on IS defines, which instance must be solved in PL

\((\Leftarrow)\): If \((U, \mathcal{F}^i)\) is solvable, we set IS accordingly, and solve this instance in PL. Remaining sets are split for free.
Wrap-up

- Unparameterized \textsc{Set Splitting} cross-composes into \textsc{Set Splitting} parameterized by $|U|$. 

Main lesson:
Model the choice of the instance to be solved. One strategy is to choose $\log t$ bits of its index on an appropriate gadget. Choice of the index makes the instance active, while the other instances are 'switched off'.

Tomorrow: More combinatorial examples.
Unparameterized Set Splitting cross-composes into Set Splitting parameterized by $|U|$.

Unparameterized Set Splitting is NP-hard.
Unparameterized \textsc{Set Splitting} cross-composes into \textsc{Set Splitting} parameterized by $|U|$.

Unparameterized \textsc{Set Splitting} is \textbf{NP}-hard.

Hence \textsc{Set Splitting} parameterized by $|U|$ does not admit a polynomial kernel, unless $\textbf{NP} \subseteq \textbf{coNP}/\text{poly}$.
Unparameterized \textsc{Set Splitting} cross-composes into \textsc{Set Splitting} parameterized by $|U|$.

Unparameterized \textsc{Set Splitting} is \textbf{NP}-hard.

Hence \textsc{Set Splitting} parameterized by $|U|$ does not admit a polynomial kernel, unless $\textbf{NP} \subseteq \textbf{coNP}/\text{poly}$.

\textbf{Main lesson:}
Unparameterized *Set Splitting* cross-composes into *Set Splitting* parameterized by $|U|$.

Unparameterized *Set Splitting* is NP-hard.

Hence *Set Splitting* parameterized by $|U|$ does not admit a polynomial kernel, unless NP ⊆ coNP/poly.

**Main lesson:**

- Model the choice of the instance to be solved.
Unparameterized Set Splitting cross-composes into Set Splitting parameterized by $|U|$. 

Unparameterized Set Splitting is \textbf{NP}-hard. 

Hence Set Splitting parameterized by $|U|$ does not admit a polynomial kernel, unless \textbf{NP} $\subseteq$ \textbf{coNP}/poly.

\textbf{Main lesson:}

- Model the \textbf{choice} of the instance to be solved.
- One strategy is to choose $\log t$ bits of its index on an appropriate gadget.
Unparameterized \textsc{Set Splitting} cross-composes into \textsc{Set Splitting} parameterized by $|U|$.

Unparameterized \textsc{Set Splitting} is \textsf{NP}-hard.

Hence \textsc{Set Splitting} parameterized by $|U|$ does not admit a polynomial kernel, unless $\textsf{NP} \subseteq \textsf{coNP}/\textsf{poly}$.

Main lesson:
- Model the \textit{choice} of the instance to be solved.
- One strategy is to choose $\log t$ bits of its index on an appropriate gadget.
- Choice of the index make the instance active, while the other instances are ‘switched off’.
Wrap-up

- Unparameterized Set Splitting cross-composes into Set Splitting parameterized by $|U|$.
- Unparameterized Set Splitting is NP-hard.
- Hence Set Splitting parameterized by $|U|$ does not admit a polynomial kernel, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.

**Main lesson:**

- Model the choice of the instance to be solved.
- One strategy is to choose $\log t$ bits of its index on an appropriate gadget.
- Choice of the index make the instance active, while the other instances are ‘switched off’.

**Tomorrow:** More combinatorial examples.
AND-compositions

- Everything we said so far would work in the same manner for AND function instead of OR.
AND-compositions

- Everything we said so far would work in the same manner for AND function instead of OR.
- **Problem**: The proof of Fortnow and Santhanam inherently breaks for AND.
AND-compositions

- Everything we said so far would work in the same manner for AND function instead of OR.
- **Problem**: The proof of Fortnow and Santhanam inherently breaks for AND.

**AND-distillation theorem**

SAT does not admit an AND-distillation algorithm into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.
AND-compositions

- Everything we said so far would work in the same manner for AND function instead of OR.
- **Problem**: The proof of Fortnow and Santhanam inherently breaks for AND.

**AND-distillation theorem**

SAT does not admit an AND-distillation algorithm into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.

- All the rest of the framework works the same (AND-cross-compositions, etc.).
AND-compositions

- Everything we said so far would work in the same manner for AND function instead of OR.
- **Problem**: The proof of Fortnow and Santhanam inherently breaks for AND.

**AND-distillation theorem**

Drucker; FOCS 2012

SAT does not admit an AND-distillation algorithm into any language $R$, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$.

- All the rest of the framework works the same (AND-cross-compositions, etc.).
- In particular, $\text{Treewidth}$, $\text{Pathwidth}$, etc. do not admit polykernels, unless $\text{NP} \subseteq \text{coNP}/\text{poly}$. 
Exercise 15.4, points 1, 2, 11, 12, 13.

Tikz faces based on a code by Raoul Kessels, http://www.texample.net/tikz/examples/emoticons/, under Creative Commons Attribution 2.5 license (CC BY 2.5)