On the Infeasibility of Obtaining Polynomial Kernels

why some problems are incompressible

PROBLEM KERNELS

how why what

when

NO POLYNOMIAL KERNELS

EXAMPLES

satisfiability vertex-disjoint cycles disjoint factors

WORKAROUNDS

many kernels

PROBLEM KERNELS

how why what

when

NO POLYNOMIAL KERNELS

EXAMPLES

satisfiability vertex-disjoint cycles disjoint factors

WORKAROUNDS

many kernels

There are 25 people at a professor's party.

The professor would like to locate a group of *six* people who are popular.

i.e, everyone is a friend of at least one of the six.

Being a busy man, he asks two of his students to find such a group.

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There are 25 people at a professor's party.

The professor would like to locate a group of *six* people who are popular.

i.e, everyone is a friend of at least one of the six.

Being a busy man, he asks two of his students to find such a group.

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

The professor would like to locate a group of *six* people who are popular.

i.e, everyone is a friend of at least one of the six.

Being a busy man, he asks two of his students to find such a group.

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

Is there a dominating set of size at most k?

i.e, everyone is a friend of at least one of the six.

Being a busy man, he asks two of his students to find such a group.

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

Is there a dominating set of size at most k?

(Every vertex is a neighbor of at least one of the k vertices.)

Being a busy man, he asks two of his students to find such a group.

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

Is there a dominating set of size at most k?

(Every vertex is a neighbor of at least one of the k vertices.)

The problem is NP–complete,

The first grimaces and starts making a list of $\binom{25}{6}$ possibilities.

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

Is there a dominating set of size at most k?

(Every vertex is a neighbor of at least one of the k vertices.)

The problem is NP–complete,

but is trivial on “large” graphs of bounded degree,

The second knows that no one in the party has more than three friends[1].

[1] Academicians tend to be lonely.

There is a graph on n vertices.

Is there a dominating set of size at most k?

(Every vertex is a neighbor of at least one of the k vertices.)

The problem is NP–complete,

but is trivial on "large" graphs of bounded degree,

as you can say NO whenever $n > kb$.

Pre-processing is a humble strategy for coping with hard problems, almost universally employed.

Mike Fellows

We would like to talk about the “preprocessing complexity” of a problem.

To what extent may we simplify/compress a problem before we begin to solve it?

Note that any compression routine has to *run efficiently* to look attractive.

This makes it unreasonable for any NP–complete problem to admit compression algorithms.

However, (some) NP–complete problems can be compressed.

We extend our notion (and notation) to understand them.

Notation

We denote a parameterized problem as a pair (Q, κ) consisting of a classical problem $Q \subseteq \{0,1\}^*$ and a parameterization $\kappa : \{0,1\}^* \to \mathbb{N}$.

Data reduction

involves pruning down

a large input

into an equivalent,

significantly smaller object,

quickly.

Data reduction

is a function $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \mathbb{N}$, such that

a large input

into an equivalent,

significantly smaller object,

quickly.

Data reduction

is a function $f : \{0, 1\}^* \times \mathbb{N} \to \{0, 1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

into an equivalent,

significantly smaller object,

quickly.

Data reduction

is a function $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

$f(x), k' \in L$ iff $(x, k) \in L$,

significantly smaller object,

quickly.

Data reduction

is a function $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

$f(x), k' \ \in L$ iff $(x, k) \in L$,

$|f(x)| \ = g(k)$,

quickly.

Data reduction

is a function $f : \{0, 1\}^* \times \mathbb{N} \to \{0, 1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

$f(x), k' \in L$ iff $(x, k) \in L$,

$|f(x)| = g(k)$,

and f is polytime computable.

A *kernelization* procedure

is a function $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

$f(x), k' \in L$ iff $(x, k) \in L$,

$|f(x)| = g(k)$,

and f is polytime computable.

A *kernelization* procedure

is a function $f : \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

Kernel

$(f(x), k') \in L$ iff $(x, k) \in L$,

$|f(x)| = g(k)$,

and f is polytime computable.

A *kernelization* procedure

is a function $f : \{0,1\}^* \times \mathbb{N} \rightarrow \{0,1\}^* \times \mathbb{N}$, such that

for all (x, k), $|x| = n$

$f(x), k' \in L$ iff $(x, k) \in L$,

$|f(x)| = g(k)$,

Size of the Kernel

and f is polytime computable.

Theorem

Having a kernelization procedure implies, and is implied by, parameterized tractability.

Theorem

Having a kernelization procedure implies, and is implied by, parameterized tractability.

Definition

A parameterized problem L is *fixed-parameter tractable* if there exists an algorithm that decides in $f(k) \cdot n^{O(1)}$ time whether $(x, k) \in L$, where $n := |x|$, $k := \kappa(x)$, and f is a computable function that does not depend on n.

Theorem

A problem admits a kernel if, and only if, it is fixed–parameter tractable.

Definition

A parameterized problem L is *fixed-parameter tractable* if there exists an algorithm that decides in $f(k) \cdot n^{O(1)}$ time whether $(x, k) \in L$, where $n := |x|$, $k := \kappa(x)$, and f is a computable function that does not depend on n.

Theorem

A problem admits a kernel if, and only if,
it is fixed–parameter tractable.

Given a kernel, a FPT algorithm is immediate (even brute–force on the kernel will lead to such an algorithm).

Theorem

A problem admits a kernel if, and only if,
it is fixed–parameter tractable.

On the other hand, a FPT runtime of $f(k) \cdot n^c$ gives us a $f(k)$–sized kernel.

We run the algorithm for n^{c+1} steps and either have a trivial kernel if the algorithm stops, else:

$$n^{c+1} < f(k) \cdot n^c$$

Theorem

A problem admits a kernel if, and only if,
it is fixed–parameter tractable.

On the other hand, a FPT runtime of $f(k) \cdot n^c$ gives us a $f(k)$–sized kernel.

We run the algorithm for n^{c+1} steps and either have a trivial kernel if the algorithm stops, else:

$$n < f(k)$$

“Efficient” Kernelization

What is a reasonable notion of efficiency for kernelization?

“Efficient” Kernelization

What is a reasonable notion of efficiency for kernelization?

The smaller, the better.

“Efficient” Kernelization

What is a reasonable notion of efficiency for kernelization?

The smaller, the better.

In particular,

Polynomial–sized kernels $<_{\text{better than}}^{\text{are}}$ Exponential–sized Kernels

The problem of finding Dominating Set of size k on graphs where the degree is bounded by b, *parameterized by* k, has a linear kernel. This is an example of a polynomial–sized kernel.

PROBLEM KERNELS

how why what

when

NO POLYNOMIAL KERNELS

EXAMPLES

satisfiability vertex-disjoint cycles disjoint factors

WORKAROUNDS

many kernels

A composition algorithm $\mathcal{A}$ for a problem is designed to act as a fast Boolean OR of multiple problem-instances.

It receives as input a sequence of instances.

It produces as output a yes-instance with a small parameter if and only if at least one of the instances in the sequences is also a yes-instance.

A composition algorithm $\mathcal{A}$ for a problem is designed to act as a fast Boolean OR of multiple problem-instances.

It receives as input a sequence of instances.
$\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0, 1\}^*$ for $i \in [t]$, such that
$k_1 = k_2 = \cdots = k_t = k$

It produces as output a yes-instance with a small parameter if and only if at least one of the instances in the sequences is also a yes-instance.

A composition algorithm $\mathcal{A}$ for a problem is designed to act as a fast Boolean OR of multiple problem-instances.

It receives as input a sequence of instances.
$\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$, such that
$k_1 = k_2 = \cdots = k_t = k$

It produces as output a yes-instance with a small parameter if and only if at least one of the instances in the sequences is also a yes-instance.

$$k' = k^{O(1)}$$

A composition algorithm $\mathcal{A}$ for a problem is designed to act as a fast Boolean OR of multiple problem-instances.

It receives as input a sequence of instances.
$\overline{x} = (x_1, \dots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$, such that
$k_1 = k_2 = \cdots = k_t = k$

It produces as output a yes-instance with a small parameter if and only if at least one of the instances in the sequences is also a yes-instance.

$$k' = k^{O(1)}$$

Running time polynomial in $\Sigma_{i \in [t]} |x_i|$

The Recipe for Hardness

Composition Algorithm + Polynomial Kernel

$$\Downarrow$$

Distillation Algorithm

$$\Downarrow$$

$$\mathsf{PH} = \Sigma_3^{\mathsf{p}}$$

The Recipe for Hardness

Composition Algorithm + Polynomial Kernel

$$\Downarrow$$

Distillation Algorithm

$$\Downarrow$$

$$\mathsf{PH} = \Sigma_3^p$$

Theorem. Let (P, k) be a compositional parameterized problem such that P is NP-complete. If P has a polynomial kernel, then P also has a distillation algorithm.

Transformations

Let (P, κ) and (Q, γ) be parameterized problems.

We say that there is a polynomial parameter transformation from P to Q if there exists a polynomial time computable function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, and a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$, such that, if $f(x) = y$, we have:

$$x \in P \text{ if and only if } y \in Q,$$

Transformations

Let (P, κ) and (Q, γ) be parameterized problems.

We say that there is a polynomial parameter transformation from P to Q if there exists a polynomial time computable function $f : \{0, 1\}^* \longrightarrow \{0, 1\}^*$, and a polynomial $p : \mathbb{N} \to \mathbb{N}$, such that, if $f(x) = y$, we have:

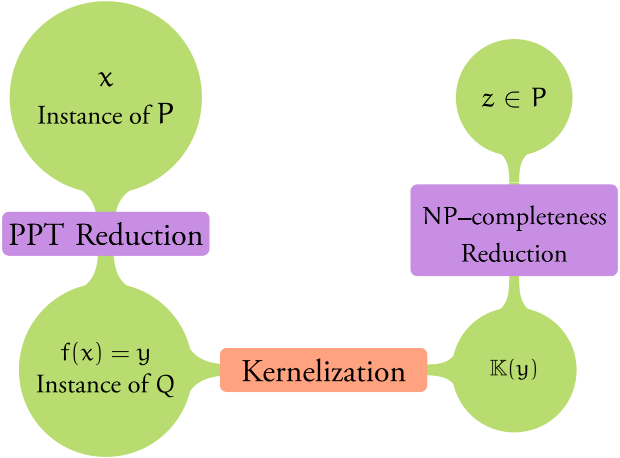
$$x \in P \text{ if and only if } y \in Q,$$

and

$$\gamma(y) \leqslant p(\kappa(x))$$

Theorem: Suppose P is NP-complete, and $\mathsf{Q} \in \mathsf{NP}$. If f is a polynomial time and parameter transformation from P to Q and Q has a polynomial kernel, then P has a polynomial kernel.

Theorem: Suppose P is NP-complete, and $Q \in NP$. If f is a polynomial time and parameter transformation from P to Q and Q has a polynomial kernel, then P has a polynomial kernel.



PROBLEM KERNELS

how why what

when

NO POLYNOMIAL KERNELS

EXAMPLES

satisfiability vertex-disjoint cycles disjoint factors

WORKAROUNDS

many kernels

Recall

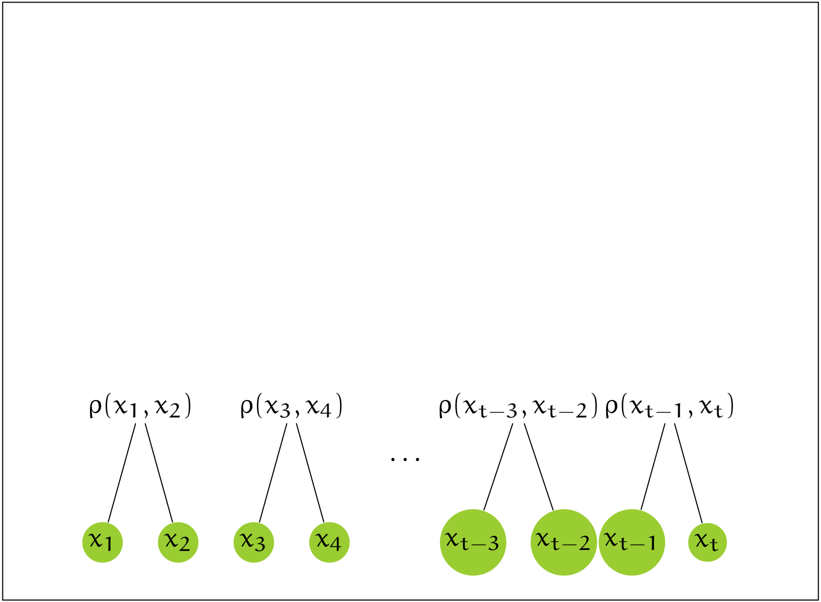
A composition for a parameterized language (Q, κ) is required to "merge" instances

$$x_1, x_2, \ldots, x_t,$$

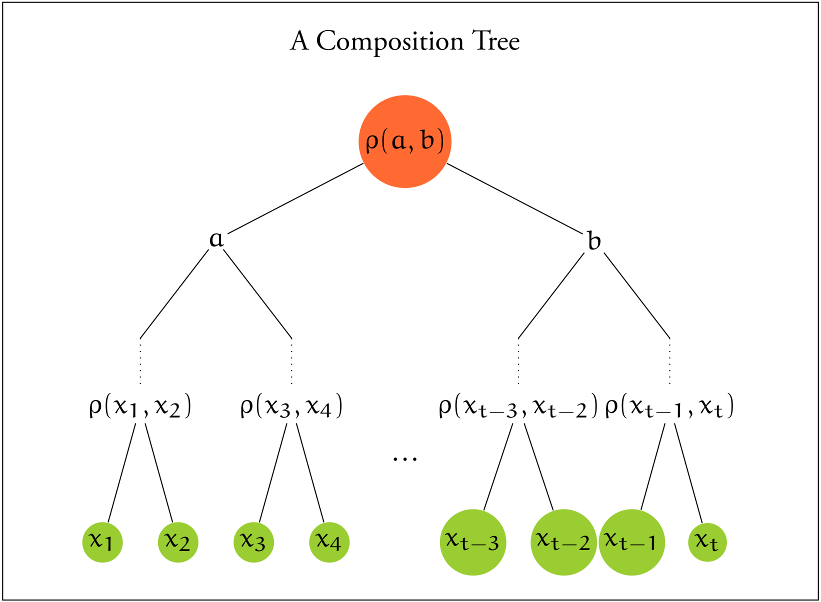
into a single instance x in polynomial time, such that $\kappa(x)$ is polynomial in $k := \kappa(x_i)$ for any i.

The output of the algorithm belongs to $(Q, \kappa(x))$
if, and only if
there exists at least one $i \in [t]$ for which $x_i \in (Q, \kappa)$.

General Framework For Composition

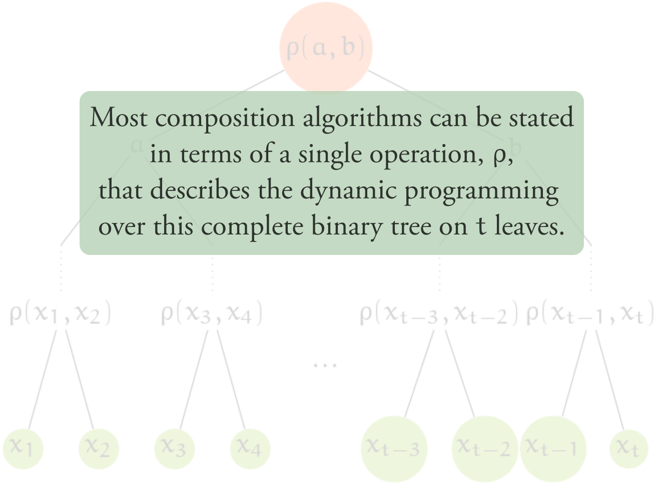


General Framework For Composition



General Framework For Composition

A Composition Tree



Most composition algorithms can be stated in terms of a single operation, ρ, that describes the dynamic programming over this complete binary tree on t leaves.

Weighted Satisfiability

Given a CNF formula ϕ,
Is there a satisfying assignment of weight at most k?

Weighted Satisfiability

Given a CNF formula ϕ,
Is there a satisfying assignment of weight at most k?

When the length of the longest clause is bounded by b,
there is an easy branching algorithm with runtime $\mathcal{O}(b^k \cdot p(n))$.

Weighted Satisfiability

Given a CNF formula ϕ,
Is there a satisfying assignment of weight at most k?

Parameter: $b+k$.

When the length of the longest clause is bounded by b,
there is an easy branching algorithm with runtime $\mathcal{O}(b^k \cdot p(n))$.

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.
$n := \max_{i \in [n]} n_i$

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.
$n := \max_{i \in [n]} n_i$

If $t > b^k$, then solve every α_i individually.

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.
$n := \max_{i \in [n]} n_i$

If $t > b^k$, then solve every α_i individually.

$$\text{Total time} = t \cdot b^k \cdot p(n)$$

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.
$n := \max_{i \in [n]} n_i$

If $t > b^k$, then solve every α_i individually.

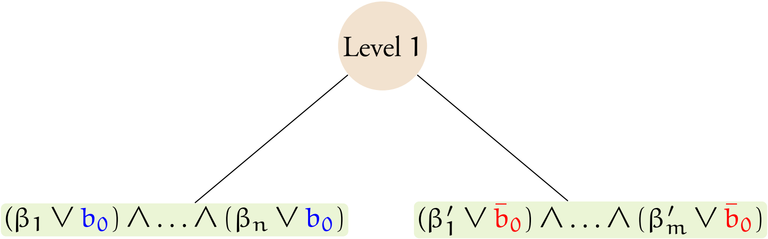
$$\text{Total time} < t \cdot t \cdot p(n)$$

Input: $\alpha_1, \alpha_2, \ldots, \alpha_t$.
$\kappa(\alpha_i) = b + k$.
$n := \max_{i \in [n]} n_i$

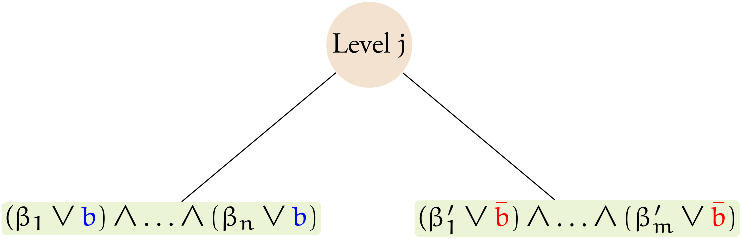
If $t > b^k$, then solve every α_i individually.

$$\text{Total time} < t \cdot t \cdot p(n)$$

If not, $t < b^k$ – this gives us a bound on the number of instances.



This is the scene at the leaves, where the β_js are the clauses in α_i for some i and the β'_js are clauses of α_{i+1}.



Level j
$(\beta_1 \vee b) \wedge \ldots \wedge (\beta_n \vee b)$
$(\beta'_1 \vee \bar{b}) \wedge \ldots \wedge (\beta'_m \vee \bar{b})$

$$(\beta_1 \vee b) \wedge (\beta_2 \vee b) \wedge \ldots \wedge (\beta_n \vee b) \;\; \wedge$$
$$(\beta'_1 \vee \bar{b}) \wedge (\beta'_2 \vee \bar{b}) \wedge \ldots \wedge (\beta'_m \vee \bar{b})$$

$$(\beta_1 \vee b) \wedge \ldots \wedge (\beta_n \vee b)$$

$$(\beta'_1 \vee \bar{b}) \wedge \ldots \wedge (\beta'_m \vee \bar{b})$$

Take the conjunction of the formulas stored at the child nodes.

$$(\beta_1 \lor b \lor b_j) \land (\beta_2 \lor b \lor b_j) \land \ldots \land (\beta_n \lor b \lor b_j) \land$$
$$(\beta'_1 \lor \bar{b} \lor b_j) \land (\beta'_2 \lor \bar{b} \lor b_j) \land \ldots \land (\beta'_m \lor \bar{b} \lor b_j)$$

$$(\beta_1 \lor b) \land \ldots \land (\beta_n \lor b) \qquad (\beta'_1 \lor \bar{b}) \land \ldots \land (\beta'_m \lor \bar{b})$$

if the parent is a "left child".

$$(\beta_1 \vee b \vee \bar{b}_j) \wedge (\beta_2 \vee b \vee \bar{b}_j) \wedge \ldots \wedge (\beta_n \vee b \vee \bar{b}_j) \wedge$$
$$(\beta'_1 \vee \bar{b} \vee \bar{b}_j) \wedge (\beta'_2 \vee \bar{b} \vee \bar{b}_j) \wedge \ldots \wedge (\beta'_m \vee \bar{b} \vee \bar{b}_j)$$

$$(\beta_1 \vee b) \wedge \ldots \wedge (\beta_n \vee b) \qquad (\beta'_1 \vee \bar{b}) \wedge \ldots \wedge (\beta'_m \vee \bar{b})$$

if the parent is a "right child".

adding a suffix to "control the weight"

$$\alpha$$
$$\wedge$$
$$(\bar{c}_0 \vee \bar{b}_0) \wedge (c_0 \vee b_0) \wedge$$
$$(\bar{c}_1 \vee \bar{b}_1) \wedge (c_1 \vee b_1) \wedge$$
$$\ldots$$
$$(\bar{c}_i \vee \bar{b}_i) \wedge (c_i \vee b_i) \wedge$$
$$\ldots$$
$$(\bar{c}_{l-1} \vee \bar{b}_{l-1}) \wedge (c_{l-1} \vee b_{l-1})$$

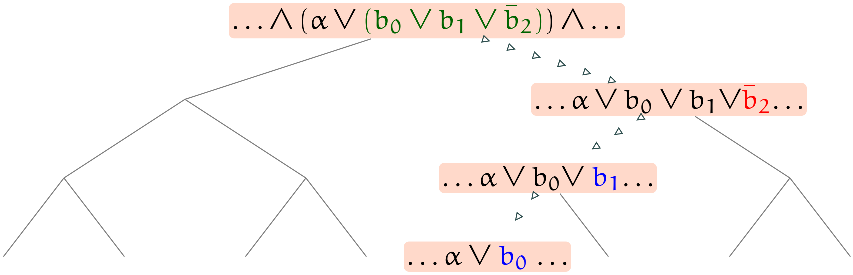
Claim

The composed instance α has a satisfying assignment of weight $2k$

$$\iff$$

at least one of the input instances admit a satisfying assignment of weight k.

Proof of Correctness



Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

1232235443513

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

12323544**3513**

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint factors do not overlap in the word.

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint factors do not overlap in the word.

Does the word have all the k factors, mutually disjoint?

Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$.

A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter.

123235443513

Disjoint factors do not overlap in the word.

Does the word have all the k factors, mutually disjoint?

Parameter: k

A $k! \cdot \mathcal{O}(n)$ algorithm is immediate.

A $k! \cdot \mathcal{O}(n)$ algorithm is immediate.
A $2^k \cdot p(n)$ algorithm can be obtained by Dynamic Programming.

A $k! \cdot \mathcal{O}(n)$ algorithm is immediate.
A $2^k \cdot p(n)$ algorithm can be obtained by Dynamic Programming.

Let t be the number of instances input to the composition algorithm.
Again, the non-trivial case is when $t < 2^k$.

A $k! \cdot \mathcal{O}(n)$ algorithm is immediate.
A $2^k \cdot p(n)$ algorithm can be obtained by Dynamic Programming.

Let t be the number of instances input to the composition algorithm.
Again, the non-trivial case is when $t < 2^k$.

Let $w_1, w_2, \ldots, w_t$ be words over L_k^*.

Leaf Nodes

Level j.

$$b_j\, b_{j-1}\, u\, b_{j-1}\, v\, b_{j-1}\, b_j$$

$$b_{j-1}\, u\, b_{j-1} \qquad\qquad b_{j-1}\, v\, b_{j-1}$$

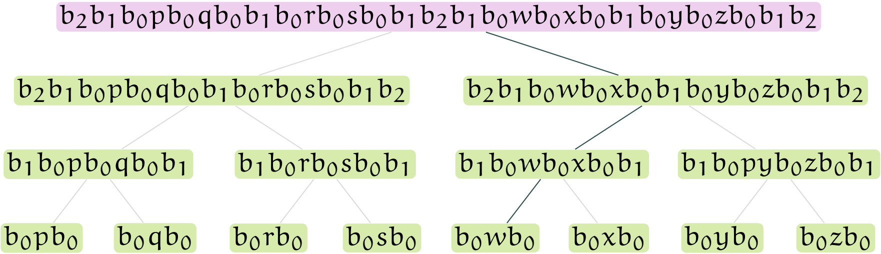
Claim

The composed word has all the $2k$ disjoint factors

$$\iff$$

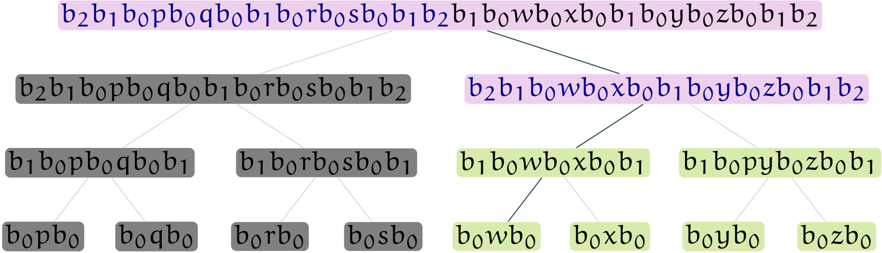
at least one of the input instances has all the k disjoint factors.

Proof of Correctness



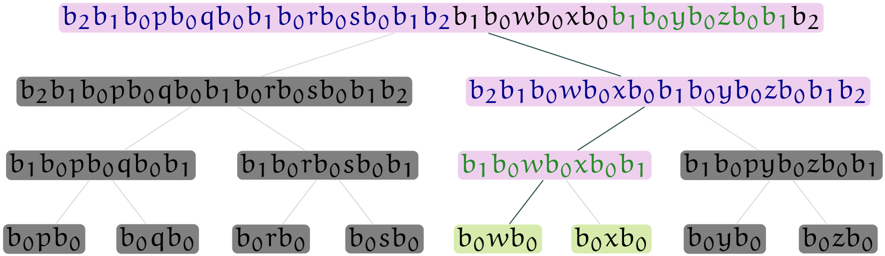
Input words: p, q, r, s, w, x, y, z.

Proof of Correctness



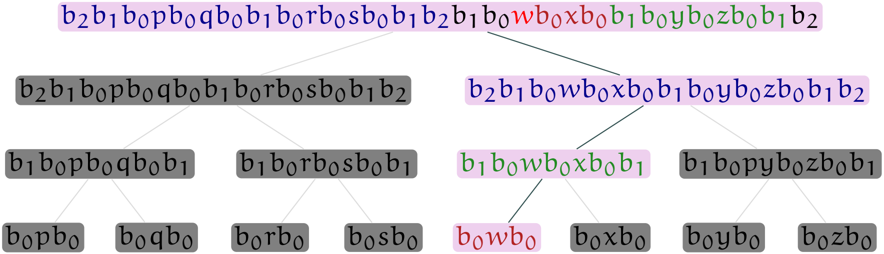
Input words: p, q, r, s, w, x, y, z.

Proof of Correctness



Input words: p, q, r, s, w, x, y, z.

Proof of Correctness



Input words: p, q, r, s, w, x, y, z.

Vertex-Disjoint Cycles

Input: $G = (V, E)$

Vertex-Disjoint Cycles

Input: $G = (V, E)$
Question: Are there k vertex–disjoint cycles?

Vertex-Disjoint Cycles

Input: $G = (V, E)$

Question: Are there k vertex–disjoint cycles?

Parameter: k.

Vertex-Disjoint Cycles

Input: $G = (V, E)$
Question: Are there k vertex–disjoint cycles?
Parameter: k.

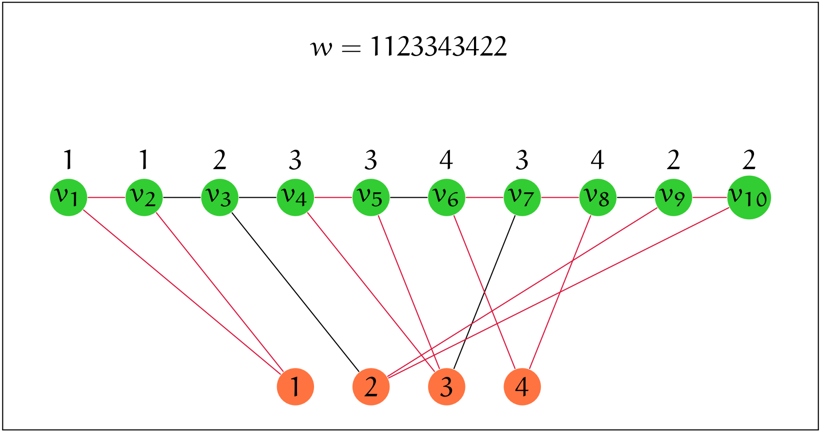
Related problems:
FVS, has a $\mathcal{O}(k^2)$ kernel.
Edge–Disjoint Cycles, has a $\mathcal{O}(k^2 \log^2 k)$ kernel.

Vertex-Disjoint Cycles

Input: $G = (V, E)$
Question: Are there k vertex–disjoint cycles?
Parameter: k.

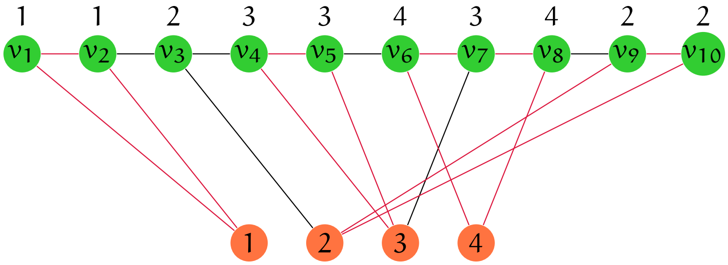
Related problems:
FVS, has a $\mathcal{O}(k^2)$ kernel.
Edge–Disjoint Cycles, has a $\mathcal{O}(k^2 \log^2 k)$ kernel.

In contrast, Disjoint Factors transforms into Vertex–Disjoint Cycles in polynomial time.



Disjoint Factors $\preccurlyeq_{ppt}$ Disjoint Cycles

$w = 1123343422$



Claim

w has all k disjoint factors $\iff G_w$ has k vertex–disjoint cycles.

PROBLEM KERNELS

how why what

when

NO POLYNOMIAL KERNELS

EXAMPLES

satisfiability vertex-disjoint cycles disjoint factors

WORKAROUNDS

many kernels

No polynomial kernel?

No polynomial kernel?

Look for the best exponential or subexponential kernels…

No polynomial kernel?

Look for the best exponential or subexponential kernels…

… or build *many* polynomial kernels.

Many polynomial kernels give us better algorithms:

$$\binom{k^2}{k}$$

$$2^{k \log k}$$

versus

$$p(n)c^k$$

$$p(n) \cdot \binom{ck}{k}$$

We say that a subdigraph T of a digraph D is an *out–tree* if T is an oriented tree with only one vertex r of in-degree zero (called the *root*).

We say that a subdigraph T of a digraph D is an *out-branching* if T is an oriented spanning tree with only one vertex r of in-degree zero.

We say that a subdigraph T of a digraph D is an *out-branching* if T is an oriented spanning tree with only one vertex r of in-degree zero.

The Directed Maximum Leaf Out-Branching problem is to find an out-branching in a given digraph with the maximum number of leaves.

We say that a subdigraph T of a digraph D is an *out-branching* if T is an oriented spanning tree with only one vertex r of in-degree zero.

The Directed k–Leaf Out-Branching problem is to find an out-branching in a given digraph with at least k leaves.

We say that a subdigraph T of a digraph D is an *out-branching* if T is an oriented spanning tree with only one vertex r of in-degree zero.

The Directed k–Leaf Out-Branching problem is to find an out-branching in a given digraph with at least k leaves.

Parameter: k

Rooted k–Leaf Out–Branching admits a kernel of size $O(k^3)$.

Rooted k–Leaf Out–Branching admits a kernel of size $O(k^3)$.

k–Leaf Out–Branching does not admit a polynomial kernel (proof deferred).

ROOTED k–LEAF OUT–BRANCHING admits a kernel of size $O(k^3)$.

k–LEAF OUT–BRANCHING does not admit a polynomial kernel (proof deferred).

However, it clearly admits n polynomial kernels (try all possible choices for root, and apply the kernel for ROOTED k–LEAF OUT–BRANCHING.

Rooted k–Leaf Out–Tree admits a kernel of size $O(k^3)$.

k–Leaf Out–Branching does not admit a polynomial kernel (proof deferred).

However, it clearly admits n polynomial kernels (try all possible choices for root, and apply the kernel for Rooted k–Leaf Out-Branching.

Rooted k–Leaf Out–Tree admits a kernel of size $O(k^3)$.

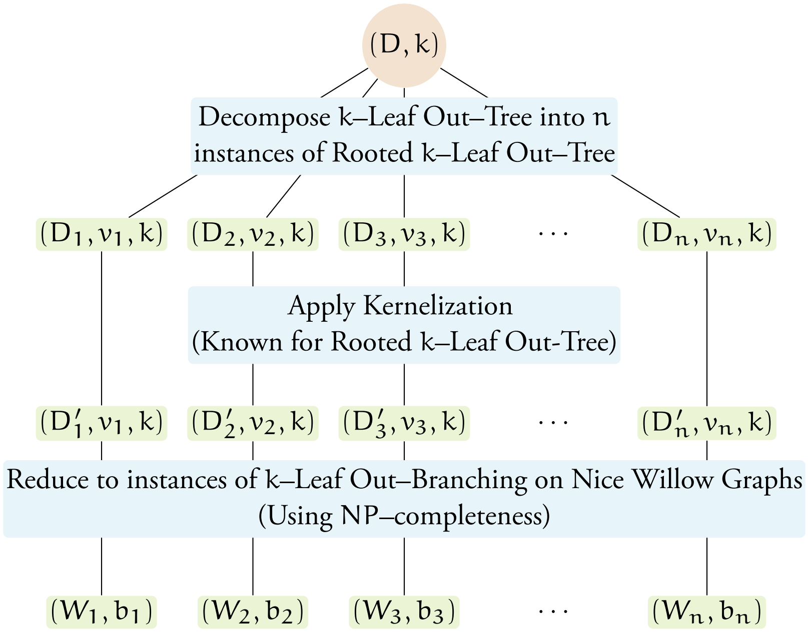
k–Leaf Out–Tree does not admit a polynomial kernel (composition via disjoint union).

However, it clearly admits n polynomial kernels (try all possible choices for root, and apply the kernel for Rooted k–Leaf Out-Branching.

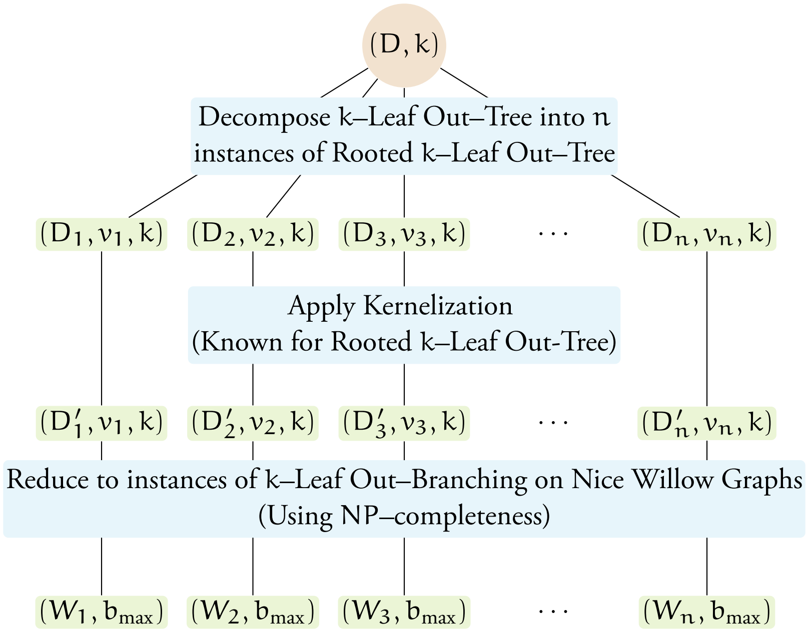
ROOTED k–LEAF OUT–TREE admits a kernel of size $O(k^3)$.

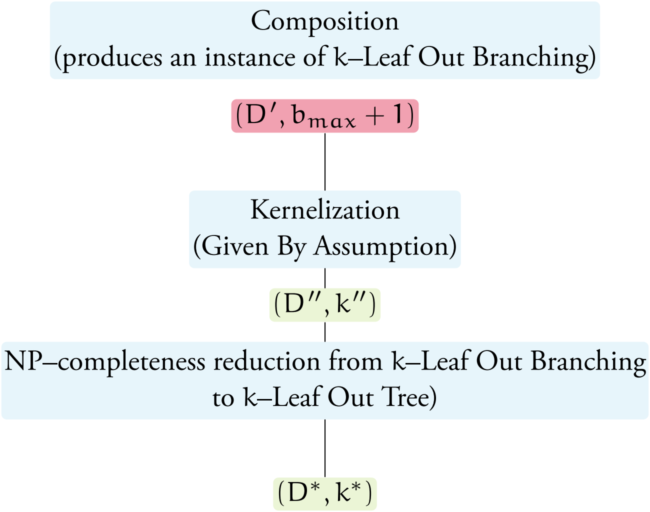
k–LEAF OUT–TREE does not admit a polynomial kernel (composition via disjoint union).

However, it clearly admits n polynomial kernels (try all possible choices for root, and apply the kernel for ROOTED k–LEAF OUT–TREE.



(D, k)
Decompose k–Leaf Out–Tree into n instances of Rooted k–Leaf Out–Tree
(D_1, v_1, k)
(D_2, v_2, k)
(D_3, v_3, k)
$\cdots$
(D_n, v_n, k)
Apply Kernelization (Known for Rooted k–Leaf Out-Tree)
(D'_1, v_1, k)
(D'_2, v_2, k)
(D'_3, v_3, k)
$\cdots$
(D'_n, v_n, k)
Reduce to instances of k–Leaf Out–Branching on Nice Willow Graphs (Using NP–completeness)
(W_1, b_1)
(W_2, b_2)
(W_3, b_3)
$\cdots$
(W_n, b_n)


(D, k)
Decompose k–Leaf Out–Tree into n instances of Rooted k–Leaf Out–Tree
(D_1, v_1, k)
(D_2, v_2, k)
(D_3, v_3, k)
· · ·
(D_n, v_n, k)
Apply Kernelization
(Known for Rooted k–Leaf Out-Tree)
(D'_1, v_1, k)
(D'_2, v_2, k)
(D'_3, v_3, k)
· · ·
(D'_n, v_n, k)
Reduce to instances of k–Leaf Out–Branching on Nice Willow Graphs
(Using NP–completeness)
(W_1, b_max)
(W_2, b_max)
(W_3, b_max)
· · ·
(W_n, b_max)


Composition
(produces an instance of k–Leaf Out Branching)
$(D', b_{max} + 1)$
Kernelization
(Given By Assumption)
(D'', k'')
NP–completeness reduction from k–Leaf Out Branching
to k–Leaf Out Tree)
(D^*, k^*)

A polynomial pseudo–kernelization K produces kernels whose size can be bounded by:

$$h(k) \cdot n^{1-\varepsilon}$$

where k is the parameter of the problem, and h is polynomial.

CONCLUDING REMARKS

Analogous to the composition framework, there are algorithms (called Linear OR) whose existence helps us rule out polynomial pseudo–kernels.

Analogous to the composition framework, there are algorithms (called Linear OR) whose existence helps us rule out polynomial pseudo–kernels.

Most compositional algorithms can be extended to fit the definition of Linear OR.

CONCLUDING REMARKS

A strong kernel is one where the parameter of the kernelized instance is at most the parameter of the original instance.

CONCLUDING REMARKS

A strong kernel is one where the parameter of the kernelized instance is at most the parameter of the original instance.

Mechanisms for ruling out strong kernels are simpler and rely on self–reductions and the hypothesis that $\mathsf{P} \neq \mathsf{NP}$.

A strong kernel is one where the parameter of the kernelized instance is at most the parameter of the original instance.

Mechanisms for ruling out strong kernels are simpler and rely on self–reductions and the hypothesis that $\mathsf{P} \neq \mathsf{NP}$.

Example: k–Path is not likely to admit a strong kernel.

CONCLUDING REMARKS

There are no known ways of inferring that a problem is unlikely to have a kernel of size k^c for some specific c.

Open Problems

Can lower bound theorems be proved under some "well-believed" conjectures of parameterized complexity - for instance - $\mathsf{FPT} \neq \mathsf{XP}$, or, $\mathsf{FPT} \neq W[\mathsf{t}]$ for some $\mathsf{t} \in \mathbb{N}^+$?

Open Problems

A lower bound framework for ruling out $p(k) \cdot f(l)$–sized kernels for problems with two parameters (k, l) would be useful.

Open Problems

“Many polynomial kernels” have been found only for the directed outbranching problem. It would be interesting to apply the technique to other problems that are not expected to have polynomial–sized kernels.

Open Problems

FIN