Kernelization

Algorithmics, 186.814, VU 6.0

Jan Dreier

Outline

- In the previous lecture we were interested in exact algorithms.
- Particularly in algorithms for parameterized problems, running in time

$$f(k) \cdot \mathsf{poly}(n),$$

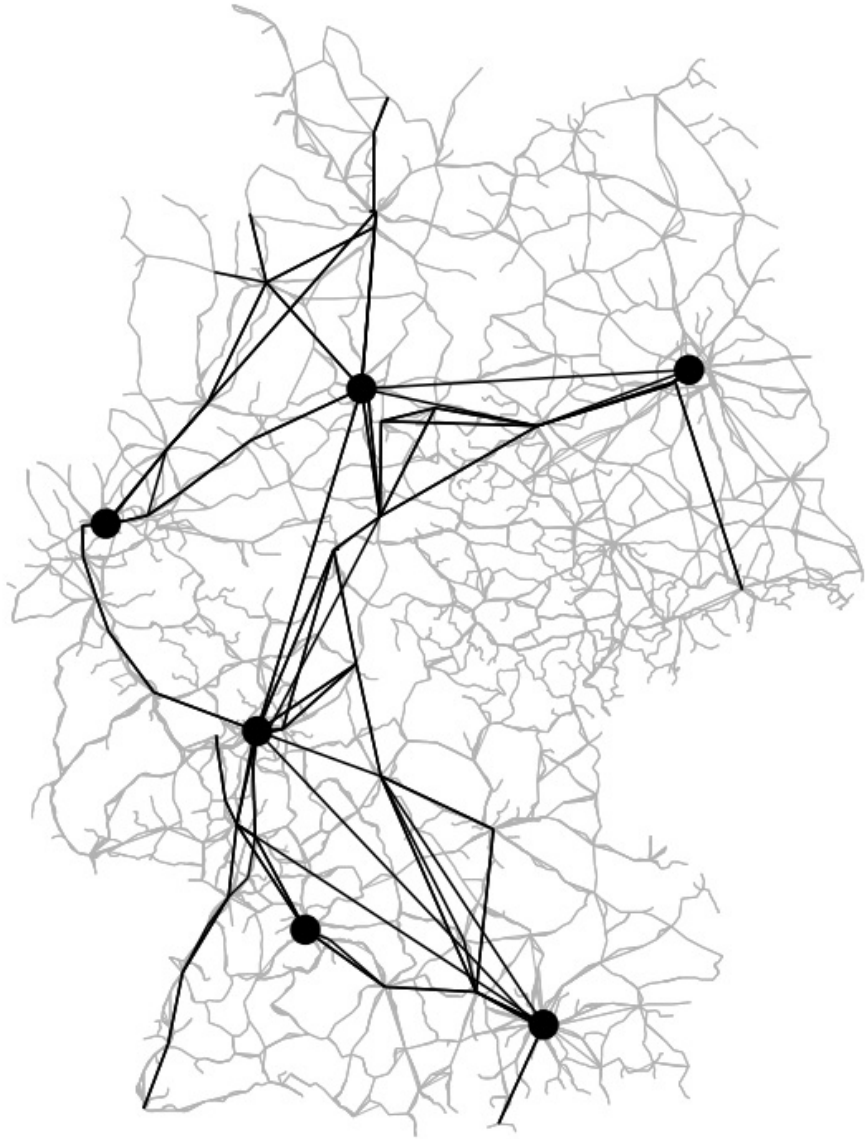
so-called FPT algorithms

- Today we will look at how parameterized complexity can also be helpful for understanding preprocessing/data reduction for hard problems.

The Power of Preprocessing

Case study: covering trains by stations [Weihe 98], real-world problem for Deutsche Bahn.

- For a given set of trains in some railroad network, the problem is to find a minimum set of stations such that every train stops at least at one station of the set (finding a minimum set of "home" stations for all trains).
- Problem known to be $\mathbf{NP}$-hard.
- Data sets contain 10.000s of trains and stations (from countries G,A,Ch,F,Cz,...). A brute-force approach is far from being feasible.



The rail network of the Deutsche Bahn.

Preprocessing rules

- In a preprocessing phase, two simple data reduction techniques were applied until no further application is possible:
  1. If all trains stopping at station S_1 also stop at station S_2, then S_1 can be safely removed.
     - Why set S_1 as a home station if S_2 is a better one?

Preprocessing rules

- In a preprocessing phase, two simple data reduction techniques were applied until no further application is possible:
  1. If all trains stopping at station S_1 also stop at station S_2, then S_1 can be safely removed.
     - Why set S_1 as a home station if S_2 is a better one?
  2. If a train T_1 only stops at stations where train T_2 also stops, then T_2 can be safely removed.
     - Any home station for T_1 can also be a home station for T_2.

Results

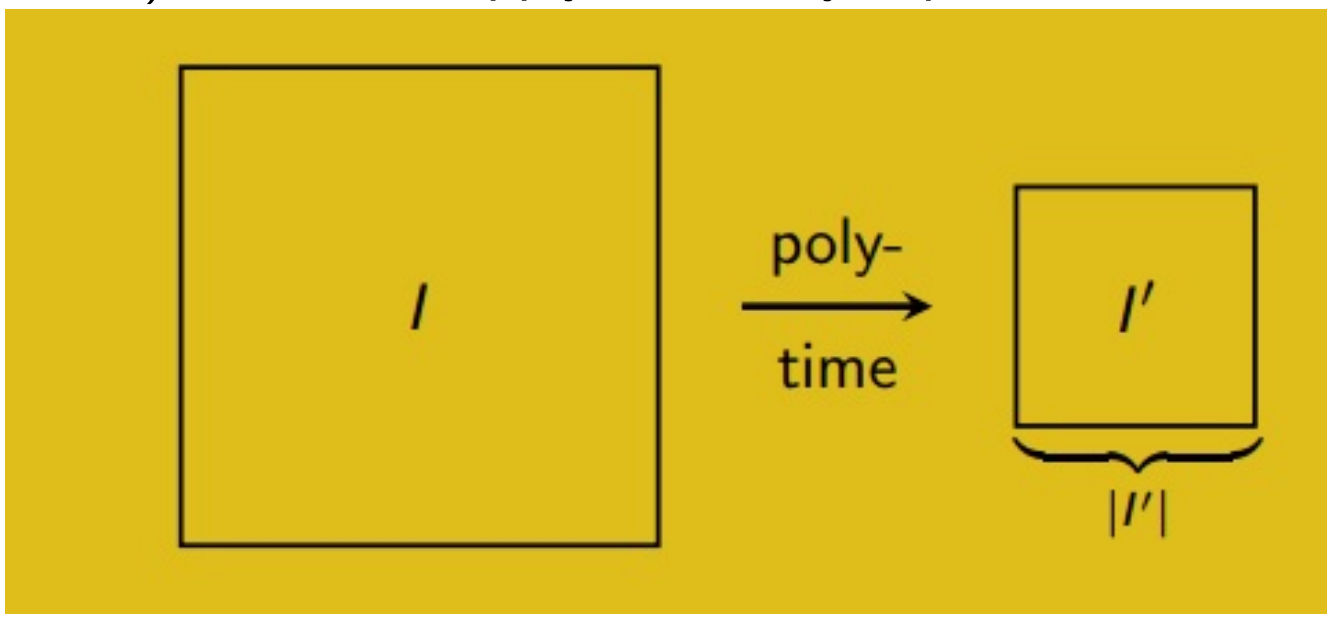
- Impressive results of this study: each of the real world instances (long-distance trains, all trains, etc.) was reduced to isolated stations and a few, very small non-trivial connected components.
- Clearly, all isolated stations plus an optimal selection from each non trivial connected component is an optimal solution to the input instance.
- A simple brute force approach was then sufficient.

Simple reduction rules can solve real-world NP-hard instances.

The Power of Preprocessing II

More complex reduction rules are used in many different areas of computer science.

Can such reduction rules provably shrink the input instance (in polynomial time), before we apply the costly exponential time algorithms?

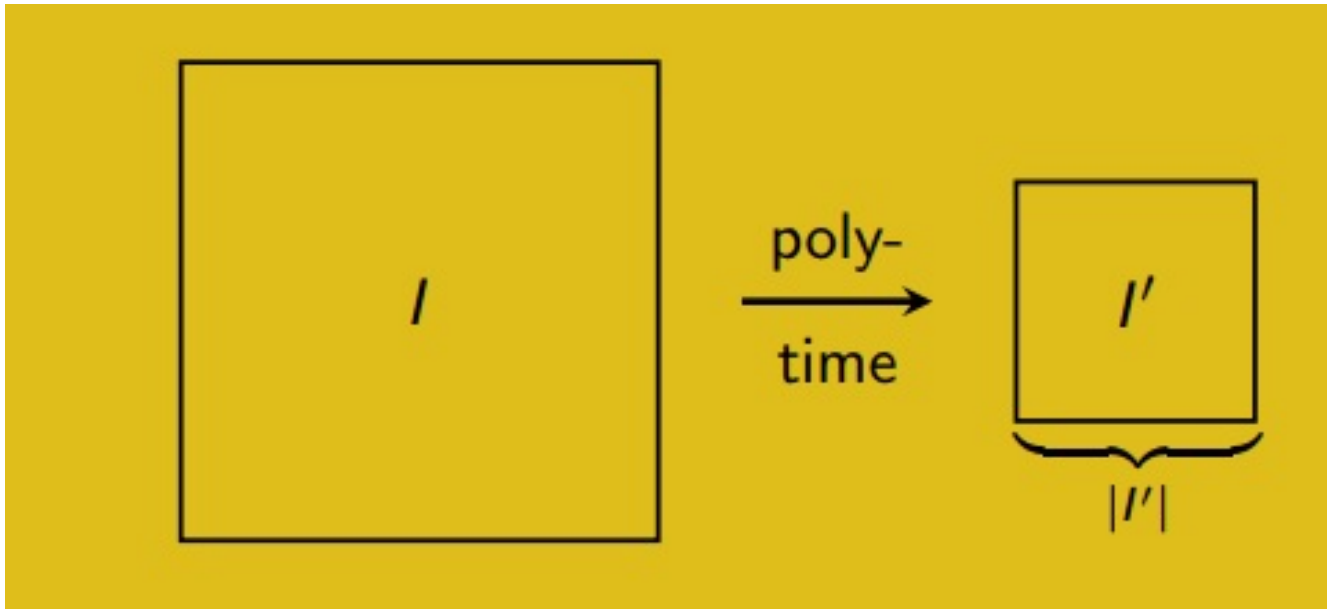


I and I' are both instances of the same $\mathbf{NP}$-hard problem, $|I'| < |I|$.

The Power of Preprocessing II

More complex reduction rules are used in many different areas of computer science.

Can such reduction rules provably shrink the input instance (in polynomial time), before we apply the costly exponential time algorithms?



I and I' are both instances of the same $\mathbf{NP}$-hard problem, $|I'| < |I|$.

Fact: No NP-complete problem admits such a preprocessing unless P = NP.

Fact: No NP-complete problem admits such a preprocessing unless P = NP.

Why?

Because we could run this preprocessing n times and end up with a constant-size instance, which can be brute-forced in constant time.

In other words...

- Unless $\mathbf{P} = \mathbf{NP}$ we cannot hope to shrink all instances of any NP-hard problem.
- For each polynomial-time preprocessing algorithm, there must be hard instances that it cannot shrink.

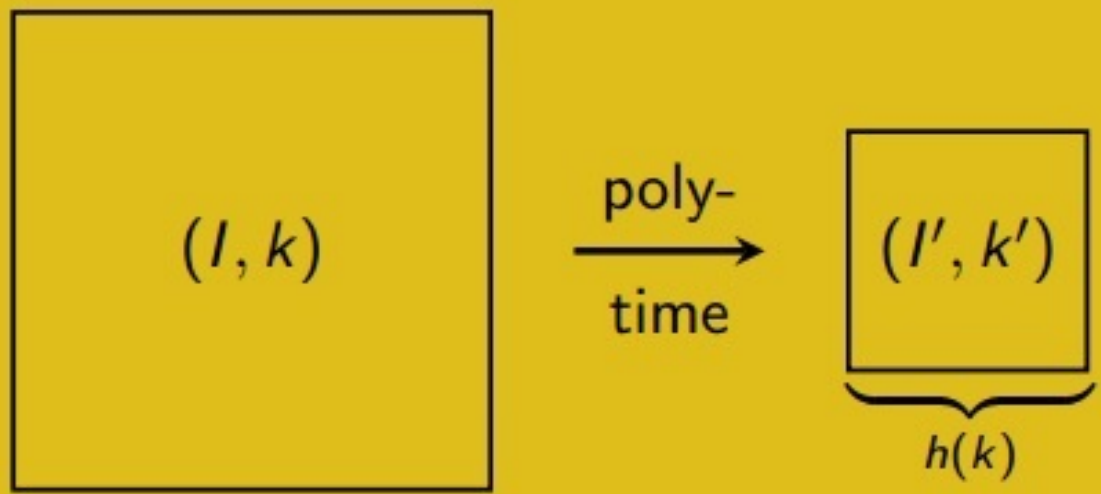
In other words...

- Unless $\mathbf{P} = \mathbf{NP}$ we cannot hope to shrink all instances of any NP-hard problem.
- For each polynomial-time preprocessing algorithm, there must be hard instances that it cannot shrink.
- We have seen in the last lecture that parameterized complexity allows a more fine-grained look at the complexity of problems.
- So, can we define a notion of preprocessing "with performance guarantees" by using a parameterized perspective?

Kernelization

Consider an instance (I, k) of a parameterized problem; I is the instance itself, and k is the parameter value.

Kernelization: a polynomial-time mapping:



such that (I, k) and (I', k') are equivalent.

polynomial kernelization: size $h(k) = poly(k)$

Kernelization – Formal Definition

Let $\mathcal{P}$ be a parameterized *decision* problem

- Decision problem = output is either "yes" or "no". This just makes things technically easier – all our algorithms will still be constructive (they'll compute a witness/solution).

Kernelization – Formal Definition

Let $\mathcal{P}$ be a parameterized *decision* problem

- Decision problem = output is either "yes" or "no". This just makes things technically easier – all our algorithms will still be constructive (they'll compute a witness/solution).

A *kernelization* or *kernelization algorithm* is a **polynomial-time algorithm** which takes as input an instance (I, k) of $\mathcal{P}$ and outputs an instance (I', k') of $\mathcal{P}$ such that:

1. (I, k) is a yes-instance of $\mathcal{P}$ iff (if and only if) (I', k') is a yes-instance of $\mathcal{P}$,
2. there exists a computable function h such that $|I'| \leq h(k)$ and $k' \leq h(k)$.

The instance (I', k') is then called the *kernel*; if h is a polynomial function, then we speak of *polynomial kernels*.

Kernelization – Formal Definition

Let $\mathcal{P}$ be a parameterized *decision* problem

- Decision problem = output is either "yes" or "no". This just makes things technically easier – all our algorithms will still be constructive (they'll compute a witness/solution).

A *kernelization* or *kernelization algorithm* is a **polynomial-time algorithm** which takes as input an instance (I, k) of $\mathcal{P}$ and outputs an instance (I', k') of $\mathcal{P}$ such that:

1. (I, k) is a yes-instance of $\mathcal{P}$ iff (if and only if) (I', k') is a yes-instance of $\mathcal{P}$,
2. there exists a computable function h such that $|I'| \leq h(k)$ and $k' \leq h(k)$.

The instance (I', k') is then called the *kernel*; if h is a polynomial function, then we speak of *polynomial kernels*.

Note: $|I|$ generally means the number of "elements" in I, for instance the number of vertices and edges for graphs.

Kernelization is equivalent to FPT

Lemma (Folklore)

Any decidable parameterized problem is FPT iff it admits a kernelization.

Recall: FPT means that there is a $f(k) \cdot \text{poly}(n)$ algorithm solving the problem.
Decidable is just a technical condition that says (roughly) that there exists an algorithm which always solves the problem.

Kernelization $\rightarrow$ FPT

- Let's start by applying the kernelization algorithm on the instance (I, k); this takes polynomial time and results in a kernel (I', k') of size $f(k)$ for some function f.
- Since $\mathcal{P}$ is decidable, there exists an algorithm that solves (I', k') in time that depends only on the size of (I', k'), i.e., in time $g(f(k))$ for some function g.
- So, the total runtime is $\text{poly}(n) + g(f(k)) \leq \text{poly}(n) \cdot g(f(k)) \rightarrow$ FPT.

FPT $\rightarrow$ Kernelization

- Since $\mathcal{P}$ is FPT, there exists an algorithm which solves any instance (I, k) in time $f(k) \cdot \text{poly}(n)$ for some function f.
- Since $f(k)$ and n are numbers, let's compare them.
- Consider $f(k) \geq n$. Then (I, k) itself is already a kernel (with respect to the function f).

FPT $\rightarrow$ Kernelization

- Since $\mathcal{P}$ is FPT, there exists an algorithm which solves any instance (I, k) in time $f(k) \cdot \text{poly}(n)$ for some function f.
- Since $f(k)$ and n are numbers, let's compare them.
- Consider $f(k) \geq n$. Then (I, k) itself is already a kernel (with respect to the function f).
- On the other hand, assume $f(k) < n$. Then the runtime of the FPT algorithm above, i.e., $f(k) \cdot \text{poly}(n)$, is upper-bounded by $n \cdot \text{poly}(n)$, and hence can be completed in polynomial time.

FPT $\rightarrow$ Kernelization

- Since $\mathcal{P}$ is FPT, there exists an algorithm which solves any instance (I, k) in time $f(k) \cdot \text{poly}(n)$ for some function f.
- Since $f(k)$ and n are numbers, let's compare them.
- Consider $f(k) \geq n$. Then (I, k) itself is already a kernel (with respect to the function f).
- On the other hand, assume $f(k) < n$. Then the runtime of the FPT algorithm above, i.e., $f(k) \cdot \text{poly}(n)$, is upper-bounded by $n \cdot \text{poly}(n)$, and hence can be completed in polynomial time.
  - Since we can solve (I, k) in polynomial time, we can use this to always output a trivial constant-size kernel. Specifically, if we detect that (I, k) is a yes-instance, then we output an arbitrary constant-size yes-instance, and otherwise we output an arbitrary constant-size no-instance.

Summing Up

A parameterized problem $\mathcal{P}$ admits a kernelization iff it is FPT.

- So what's there to study about kernelization?
- The kernels obtained in this way are typically not of practical use (way too big).
- The vast majority of work in kernelization focuses on *polynomial kernels* (and ideally *linear kernels*).
  - Not all FPT problems have polynomial kernels!

$\rightarrow$

Example: Vertex Cover

Let's revisit the (Parameterized) Vertex Cover problem.

(Parameterized) Vertex Cover
Instance: A graph G and an integer k.
Parameter: k.
Question: Does G admit a vertex cover of size at most k?

Does this problem admit a polynomial kernel?
Note: we drop the "parameterized" from now on when it's clear that we're speaking about a parameterized problem.

How to Get Polykernels

Most standard kernelization technique: *reduction rules*
Idea: formulate a set of reduction rules such that:

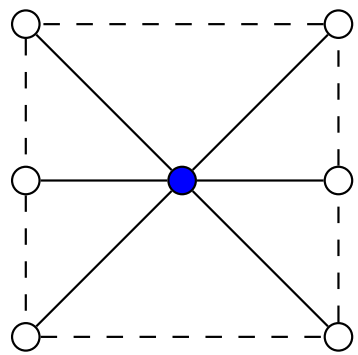
- each rule runs in polytime
- each rule slightly modifies the instance to obtain an equivalent instance
  - typically the new instance is smaller and the parameter value does not increase

To successfully obtain a kernel, it is necessary to show:

1. each rule is *safe*: it does not change yes-instances to no-instances and vice-versa
2. if no rule can be applied, then we already have a (polynomial) kernel

Reduction Rules for Vertex Cover

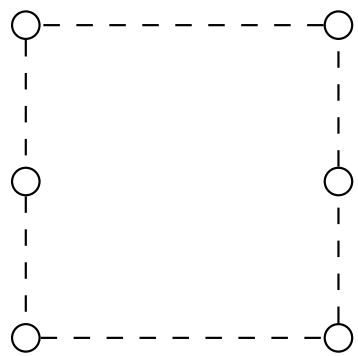
Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.



$k = 5$

Reduction Rules for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.



$k = 4$

Reduction Rules for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.
Why is this rule safe?

- Any vertex cover must contain either v or all of its neighbors, but v has more than k neighbors so the latter is not possible.

Reduction Rules for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.
Why is this rule safe?

- Any vertex cover must contain either v or all of its neighbors, but v has more than k neighbors so the latter is not possible.

Rule 2: If there is a vertex v with no neighbors, then delete v.
Why is this rule safe?

- "it doesn't make sense to pick v"
- equivalently: no minimum vertex cover contains such a vertex v

Kernel for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.
Rule 2: If there is a vertex v with no neighbors, then delete v.

Now assume we can no longer apply any of the rules. Let's count the number m of edges remaining in the graph after the reduction rules.
Idea:

- if m is much bigger than k, then we simply cannot cover all edges with just k vertices, **because the degree is now at most k**.
- otherwise we already have a small instance and we're done.

Kernel for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.
Rule 2: If there is a vertex v with no neighbors, then delete v.

Now assume we can no longer apply any of the rules. Let's count the number m of edges remaining in the graph after the reduction rules.
Idea:

- if m is much bigger than k, then we simply cannot cover all edges with just k vertices, **because the degree is now at most k**.
- otherwise we already have a small instance and we're done.

Running time: $\mathcal{O}(n^2)$

- We apply reduction rules at most n times, and each application can be done in linear time.

Kernel for Vertex Cover

Rule 1: If there is a vertex v of degree at least $k+1$, then delete v and set $k := k-1$.
Rule 2: If there is a vertex v with no neighbors, then delete v.

Now assume we can no longer apply any of the rules. Let's count the number m of edges remaining in the graph after the reduction rules.
Formalization:

- Assume $m > k^2$. Then we can correctly output "no". Indeed, a vertex cover of size k can only cover at most k^2 edges in a graph of maximum degree k.
- Assume $m \leq k^2$. Since each vertex is incident to at least 1 edge, this implies that the graph has at most $2k^2$ vertices $\rightarrow$ we have a kernel.

Theorem

Vertex Cover *admits a quadratic kernel.*

Clique

What about the following problem?

Clique
Instance: A graph G and an integer k.
Parameter: k.
Question: Do there exist at least k vertices in G which form a *clique*?

A *clique* is a complete subgraph, i.e., each pair of vertices in a clique are adjacent.
Does this problem admit a (polynomial) kernel?

From Cliques to Independent Sets

First question we need to ask: is Clique FPT?

Assume for a moment there exists an FPT algorithm $\mathbb{A}$ that solves Clique in time $f(k) \cdot \text{poly}(n)$.
Now, recall the Independent Set Problem from the last lecture:

> Independent Set
>
> *Instance*: A graph G and an integer k.
>
> *Parameter*: k.
>
> *Question*: Are there at least k vertices in G which are pairwise *independent*?

From Cliques to Independent Sets

First question we need to ask: is Clique FPT?

Assume for a moment there exists an FPT algorithm $\mathbb{A}$ that solves Clique in time $f(k) \cdot \text{poly}(n)$.
Now, recall the Independent Set Problem from the last lecture:

> Independent Set
>
> *Instance*: A graph G and an integer k.
>
> *Parameter*: k.
>
> *Question*: Are there at least k vertices in G which are pairwise *independent*?

Theorem (last lecture)

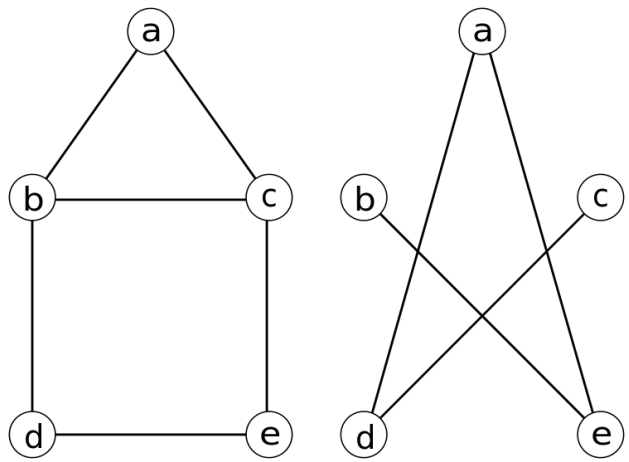
Independent *is (most likely) not FPT.*

An Argument via Complements

Let's assume we get an instance (G, k) of Independent Set.

Consider the following algorithm:

1. Construct the *complement* of G and call it $\bar{G}$.
   - A complement (of a graph G) is the graph obtained by complementing ("reversing") all edges, i.e., an edge is in the complement iff it is not there in the original graph.



An Argument via Complements

Let's assume we get an instance (G, k) of Independent Set.

Consider the following algorithm:

1. Construct the *complement* of G and call it $\bar{G}$.
2. Run the hypothetical algorithm $\mathbb{A}$ on $(\bar{G}, k)$.
3. Output the result of $\mathbb{A}$ on $(\bar{G}, k)$.

An Argument via Complements

Let's assume we get an instance (G, k) of Independent Set.

Consider the following algorithm:

1. Construct the *complement* of G and call it $\bar{G}$.
2. Run the hypothetical algorithm $\mathbb{A}$ on $(\bar{G}, k)$.
3. Output the result of $\mathbb{A}$ on $(\bar{G}, k)$.

Is this an FPT algorithm? Yes.
Does it solve Independent Set?

- Assume $\mathbb{A}$ outputs "yes" on $(\bar{G}, k)$. Then $\bar{G}$ contains a clique of size $k \to G$ must contain an independent set of size k.
- Assume $\mathbb{A}$ outputs "no" on $(\bar{G}, k)$. Then $\bar{G}$ does not contain a clique of size $k \to G$ cannot contain an independent set of size k.

An Argument via Complements

Let's assume we get an instance (G, k) of Independent Set.

Consider the following algorithm:

1. Construct the *complement* of G and call it $\bar{G}$.
2. Run the hypothetical algorithm $\mathbb{A}$ on $(\bar{G}, k)$.
3. Output the result of $\mathbb{A}$ on $(\bar{G}, k)$.

Is this an FPT algorithm? Yes.
Does it solve Independent Set?

- Assume $\mathbb{A}$ outputs "yes" on $(\bar{G}, k)$. Then $\bar{G}$ contains a clique of size $k \to G$ must contain an independent set of size k.
- Assume $\mathbb{A}$ outputs "no" on $(\bar{G}, k)$. Then $\bar{G}$ does not contain a clique of size $k \to G$ cannot contain an independent set of size k.

Theorem

Clique *is (most likely) not FPT, and hence does not admit a kernel.*

Hitting Set

This is a classical and important non-graph problem.

HITTING SET

Instance: A *ground set* U, a set $\mathcal{F}$ of subsets of U, and an integer k.

Parameter: k.

Question: Does there exist a set $X \subseteq U$ of cardinality at most k which *hits* each subset in $\mathcal{F}$?

hits: X hits a set $F \in \mathcal{F}$ iff $F \cap X \neq \emptyset$.

Hitting Set Example

Say we have 5 drugs, each effective against different strains of a disease.

Strain 1 is vulnerable to Drug 1
Strain 2 is vulnerable to Drugs $1, 3$
Strain 3 is vulnerable to Drugs $1, 2$
Strain 4 is vulnerable to Drugs $1, 3$
Strain 5 is vulnerable to Drugs $2, 4, 5$
Strain 6 is vulnerable to Drug 4

Can we protect against all strains with only 2 drugs?

Hitting Set Example

Say we have 5 drugs, each effective against different strains of a disease.

Strain 1 is vulnerable to Drug $1 \rightarrow F_1 = \{1\}$
Strain 2 is vulnerable to Drugs $1, 3 \rightarrow F_2 = \{1, 3\}$
Strain 3 is vulnerable to Drugs $1, 2 \rightarrow F_3 = \{1, 2\}$
Strain 4 is vulnerable to Drugs $1, 3 \rightarrow F_4 = \{1, 3\}$
Strain 5 is vulnerable to Drugs $2, 4, 5 \rightarrow F_5 = \{2, 4, 5\}$
Strain 6 is vulnerable to Drug $4 \rightarrow F_6 = \{4\}$

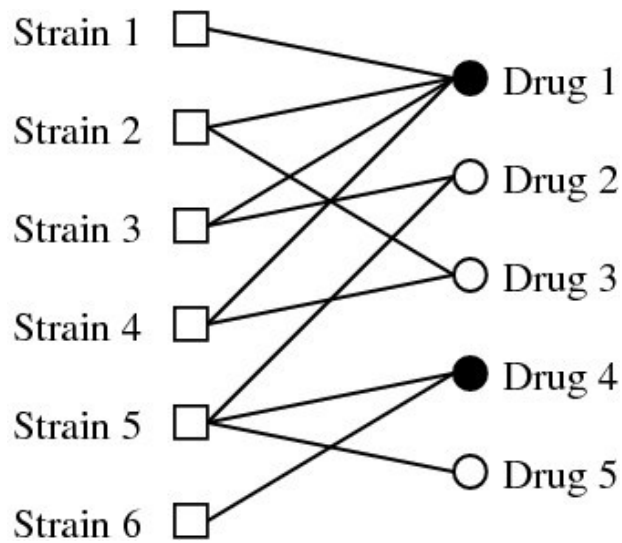
Can we find a hitting set of size 2?

Hitting Set Example

Say we have 5 drugs, each effective against different strains of a disease.

Strain 1 is vulnerable to Drug $1 \rightarrow F_1 = \{1\}$
Strain 2 is vulnerable to Drugs $1, 3 \rightarrow F_2 = \{1, 3\}$
Strain 3 is vulnerable to Drugs $1, 2 \rightarrow F_3 = \{1, 2\}$
Strain 4 is vulnerable to Drugs $1, 3 \rightarrow F_4 = \{1, 3\}$
Strain 5 is vulnerable to Drugs $2, 4, 5 \rightarrow F_5 = \{2, 4, 5\}$
Strain 6 is vulnerable to Drug $4 \rightarrow F_6 = \{4\}$

Can we find a hitting set of size 2?



Hitting Set vs d-Hitting Set

- Hitting Set is another problem which is almost certainly not FPT (it's W[2]-hard).
- But its variants, d-Hitting Set, are FPT (for any fixed constant d).

d-Hitting Set

Instance: A *ground set* U, a set $\mathcal{F}$ of subsets of U each of cardinality at most d, and an integer k.

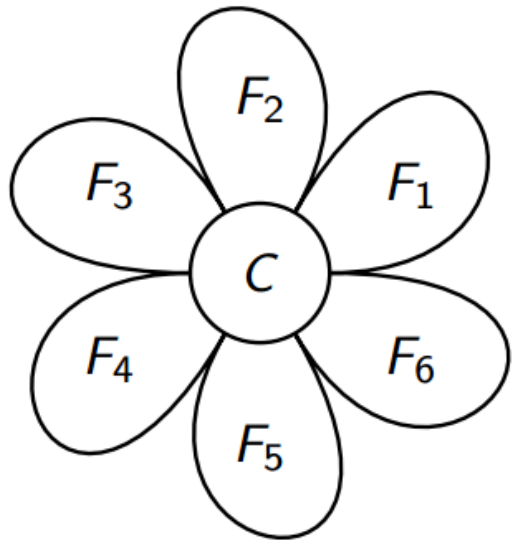
Parameter: k.

Question: Does there exist a set $X \subseteq U$ of cardinality at most k which *hits* each subset in $\mathcal{F}$?

Can we get a polynomial kernel for d-Hitting Set?

Sunflowers

Our kernelization algorithm needs one key tool: the Sunflower Lemma.



A sunflower of cardinality t consists of t sets $F_1, \ldots, F_t$ such that

$$F_i \cap F_j = C \text{ for all } i \neq j.$$

The set C is called the core of the sunflower.

note: t pairwise disjoint sets also form a sunflower

A sunflower of cardinality t is also called a t-sunflower.

Sunflower Lemma

Lemma (Erdös and Rado, 1960)

Let $\mathcal{F}$ be a set of subsets of a ground set U such that

1. *each $F \in \mathcal{F}$ has cardinality at most d, and*
2. *$|\mathcal{F}| > d! \cdot k^d$.*

Then there exists a $(k+1)$-sunflower.
Moreover, such a $(k+1)$-sunflower can be found in polynomial time.

Proof is non-trivial; we'll treat the lemma as a black box for this lecture.

Getting Sunflowers for d-Hitting Set

Let $(U, \mathcal{F}, k)$ be an instance of d-Hitting Set.
Assume we have already exhaustively applied the trivial rule that deletes elements of U which are not contained in any set of $\mathcal{F}$.

- Reasoning: It makes no sense to take such elements into the Hitting Set.

Getting Sunflowers for d-Hitting Set

Let $(U, \mathcal{F}, k)$ be an instance of d-Hitting Set.
Assume we have already exhaustively applied the trivial rule that deletes elements of U which are not contained in any set of $\mathcal{F}$.

- Reasoning: It makes no sense to take such elements into the Hitting Set.

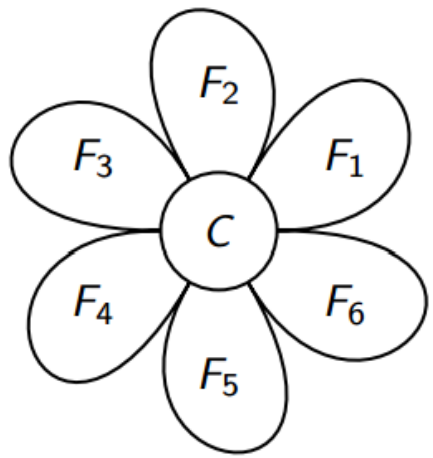
If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, since $|U| \leq d \cdot |\mathcal{F}|$.

Getting Sunflowers for d-Hitting Set

Let $(U, \mathcal{F}, k)$ be an instance of d-Hitting Set.
Assume we have already exhaustively applied the trivial rule that deletes elements of U which are not contained in any set of $\mathcal{F}$.

- Reasoning: It makes no sense to take such elements into the Hitting Set.

If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, since $|U| \leq d \cdot |\mathcal{F}|$.
Otherwise we can apply the Sunflower Lemma to find a $(k+1)$-sunflower.

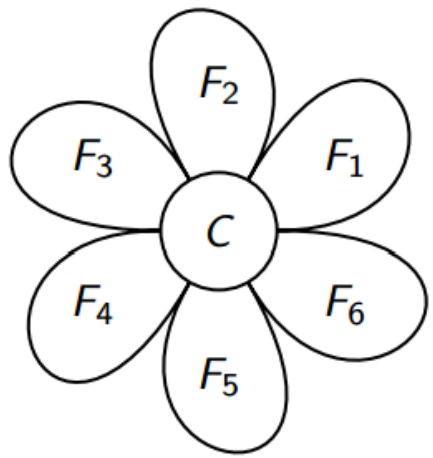


Getting Sunflowers for d-Hitting Set

Let $(U, \mathcal{F}, k)$ be an instance of d-Hitting Set.
Assume we have already exhaustively applied the trivial rule that deletes elements of U which are not contained in any set of $\mathcal{F}$.

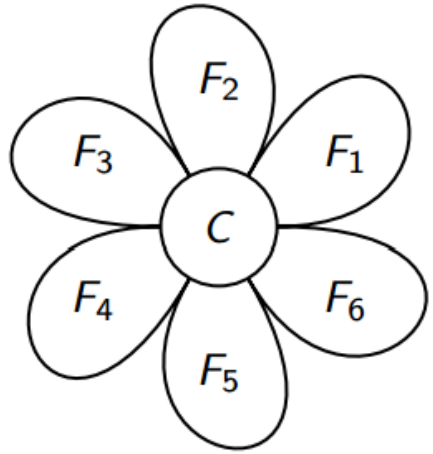
- Reasoning: It makes no sense to take such elements into the Hitting Set.

If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, since $|U| \leq d \cdot |\mathcal{F}|$.
Otherwise we can apply the Sunflower Lemma to find a $(k+1)$-sunflower.



There's no way to hit all these $k+1$ sets without hitting the core C.

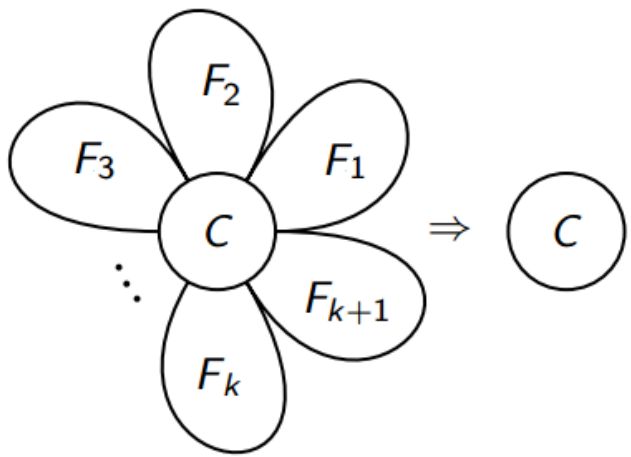
Using Sunflowers for d-Hitting Set



There's no way to hit all these $k+1$ sets without hitting the core C.

- So, we can add the core C into $\mathcal{F}$ as a new set that we need to hit.
- Whenever we hit C, we'll also hit all of $F_1, \ldots, F_{k+1}$ in the sunflower, and so these can be removed now.

Reduction Rule:



Kernelization Algorithms for d-Hitting Set

For an instance $(U, \mathcal{F}, k)$ of d-Hitting Set:

1. Remove elements of U that do not intersect anything in $\mathcal{F}$.
2. If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, otherwise we apply the Sunflower Lemma and find a $(k+1)$-sunflower.

Kernelization Algorithms for d-Hitting Set

For an instance $(U, \mathcal{F}, k)$ of d-Hitting Set:

1. Remove elements of U that do not intersect anything in $\mathcal{F}$.
2. If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, otherwise we apply the Sunflower Lemma and find a $(k+1)$-sunflower.
3. Apply sunflower reduction rule to remove $k+1$ elements of $\mathcal{F}$ and add the core C to $\mathcal{F}$; this reduces the size of $\mathcal{F}$ by k.
   - What if $C = \emptyset$?

Kernelization Algorithms for d-Hitting Set

For an instance $(U, \mathcal{F}, k)$ of d-Hitting Set:

1. Remove elements of U that do not intersect anything in $\mathcal{F}$.
2. If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, otherwise we apply the Sunflower Lemma and find a $(k+1)$-sunflower.
3. Apply sunflower reduction rule to remove $k+1$ elements of $\mathcal{F}$ and add the core C to $\mathcal{F}$; this reduces the size of $\mathcal{F}$ by k.
   - What if $C = \emptyset$? We'll be adding $\emptyset$ to $\mathcal{F}$, which correctly results in a clear no-instance (can't hit $k+1$ "petals" without using a core).
4. Restart.

Theorem

d-Hitting Set *admits a* $\mathcal{O}(k^d)$ *kernel.*

Kernelization Algorithms for d-Hitting Set

For an instance $(U, \mathcal{F}, k)$ of d-Hitting Set:

1. Remove elements of U that do not intersect anything in $\mathcal{F}$.
2. If $|\mathcal{F}| \leq d! \cdot k^d$, then we already have a $\mathcal{O}(k^d)$ kernel, otherwise we apply the Sunflower Lemma and find a $(k+1)$-sunflower.
3. Apply sunflower reduction rule to remove $k+1$ elements of $\mathcal{F}$ and add the core C to $\mathcal{F}$; this reduces the size of $\mathcal{F}$ by k.
   - What if $C = \emptyset$? We'll be adding $\emptyset$ to $\mathcal{F}$, which correctly results in a clear no-instance (can't hit $k+1$ "petals" without using a core).
4. Restart.

Theorem

d-Hitting Set *admits a* $\mathcal{O}(k^d)$ *kernel.*

Note: this also implies a quadratic kernel for Vertex Cover.

- Vertex Cover can be formulated as a 2-Hitting Set problem (edges are sets of cardinality 2 that we want to hit).

Kernelization: Summing Up

- Kernelization algorithms are polynomial-time preprocessing algorithms that can also be used to supplement other algorithmic methods.
- Kernelization can also be used to "compress" instances, e.g., for storage or if they need to be transmitted through a restricted transmission channel.
- Any parameterized problem that is FPT admits a kernel (and vice-versa), but only some admit a polynomial kernel.

One More Example: Point Line Cover

Consider the following problem:

Point Line Cover

Instance: A set of n points in the plane and an integer k.

Parameter: k.

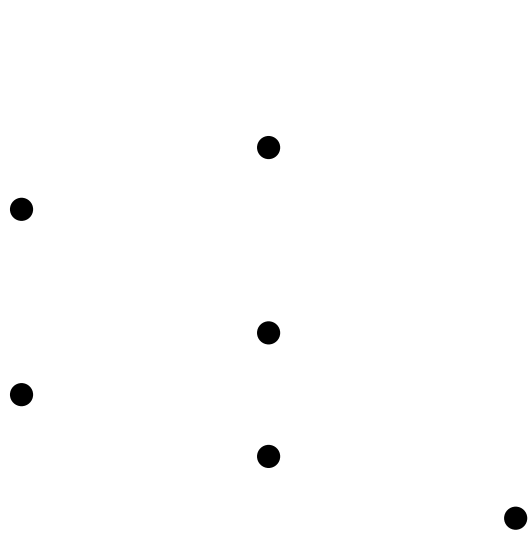
Question: Do there exist at most k lines on the plane that contain all the input points?

One More Example: Point Line Cover

Consider the following problem:

> Point Line Cover
>
> *Instance*: A set of n points in the plane and an integer k.
>
> *Parameter*: k.
>
> *Question*: Do there exist at most k lines on the plane that contain all the input points?

Example: $k = 3$

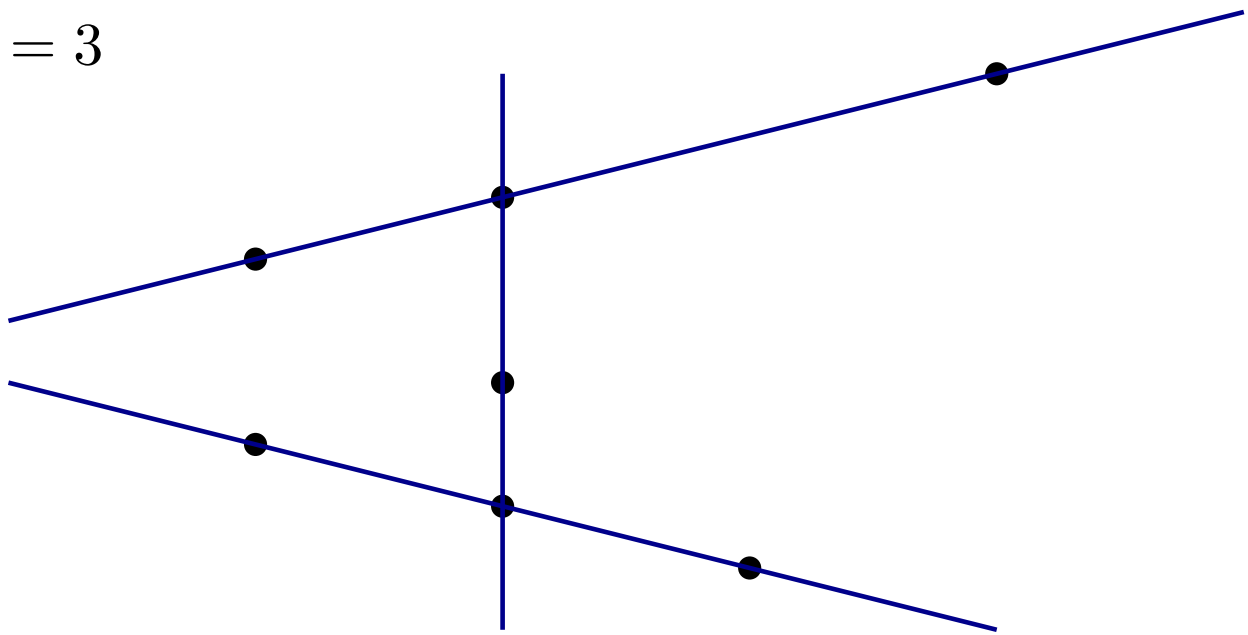


One More Example: Point Line Cover

Consider the following problem:

Point Line Cover
Instance: A set of n points in the plane and an integer k.
Parameter: k.
Question: Do there exist at most k lines on the plane that contain all the input points?

Example: $k = 3$



Polynomial Kernel for Point Line Cover

1. If there are at least 2 points, then one needs to consider only n^2 possible lines (between pairs of points).
   - Never helps to use a line that hits exactly 1 point.

Polynomial Kernel for Point Line Cover

1. If there are at least 2 points, then one needs to consider only n^2 possible lines (between pairs of points).
   - Never helps to use a line that hits exactly 1 point.
2. What if a line hits at least $k+1$ points?
   - Then this line **must** be in the solution. Otherwise, we'd need at least $k+1$ lines to hit these points.

   Rule 1: If there is a line that hits at least $k+1$ points, then delete these points and set $k := k-1$.

Polynomial Kernel for Point Line Cover

1. If there are at least 2 points, then one needs to consider only n^2 possible lines (between pairs of points).
   - Never helps to use a line that hits exactly 1 point.
2. What if a line hits at least $k+1$ points?
   - Then this line **must** be in the solution. Otherwise, we'd need at least $k+1$ lines to hit these points.

   Rule 1: If there is a line that hits at least $k+1$ points, then delete these points and set $k := k-1$.
3. What if we cannot apply **Rule 1**?
   - Each remaining possible line hits at most k points.
   - Then if there's more than k^2 points, we can never hit all of them with k lines.
   - Either we already have a k^2 kernel, or we have a no-instance.

Questions?

$\rightarrow$