Parameterized Algorithms

Dániel Marx

Pranabendu Misra

Philip Wellnitz (tutorials)

Lecture #1
May 8, 2020

Can you read this line down here?

Classical complexity

A brief review:

- We usually aim for **polynomial-time** algorithms: the worst-case running time is $O(n^c)$, where n is the input size and c is a constant.
- Classical polynomial-time algorithms: shortest path, perfect matching, minimum spanning tree, 2SAT, convex hull, planar drawing, linear programming, etc.
- It is unlikely that polynomial-time algorithms exist for **NP-hard** problems.
- Unfortunately, many problems of interest are NP-hard: Hamiltonian Cycle, 3-Coloring, 3SAT, etc.
- We expect that these problems can be solved only in exponential time (i.e., $O(c^n)$).

Can we say anything nontrivial about NP-hard problems?

Parameterized problems

Main idea

Instead of expressing the running time as a function $T(n)$ of n, we express it as a function $T(n, k)$ of the input size n and some parameter k of the input.

In other words: we do not want to be efficient on all inputs of size n, only for those where k is small.

Parameterized problems

Main idea

Instead of expressing the running time as a function $T(n)$ of n, we express it as a function $T(n, k)$ of the input size n and some parameter k of the input.

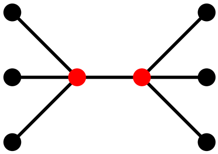
In other words: we do not want to be efficient on all inputs of size n, only for those where k is small.

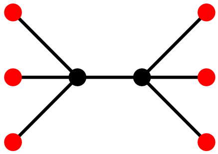
What can be the parameter k?

- The size k of the solution we are looking for.
- The maximum degree Δ of the input graph.
- The dimension d of the point set in the input.
- The length L of the strings in the input.
- The length ℓ of clauses in the input Boolean formula.
- $\dots$

Parameterized complexity

Problem:	VERTEX COVER	INDEPENDENT SET
Input:	Graph G, integer k	Graph G, integer k
Question:	Is it possible to cover the edges with k vertices?	Is it possible to find k independent vertices?

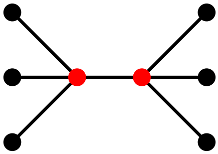


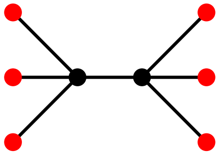


Complexity:	NP-complete	NP-complete

Parameterized complexity

	Vertex Cover	Independent Set
Problem:	VERTEX COVER	INDEPENDENT SET
Input:	Graph G, integer k	Graph G, integer k
Question:	Is it possible to cover the edges with k vertices?	Is it possible to find k independent vertices?

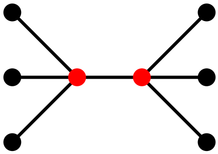


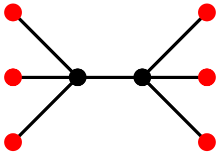


Complexity:	NP-complete	NP-complete
Brute force:	$O(n^k)$ possibilities	$O(n^k)$ possibilities

Parameterized complexity

Problem:	VERTEX COVER	INDEPENDENT SET
Input:	Graph G, integer k	Graph G, integer k
Question:	Is it possible to cover the edges with k vertices?	Is it possible to find k independent vertices?





Complexity:	NP-complete	NP-complete
Brute force:	$O(n^k)$ possibilities	$O(n^k)$ possibilities
	$O(2^k n^2)$ **algorithm exists**	**No** $n^{o(k)}$ **algorithm known**

Bounded search tree method

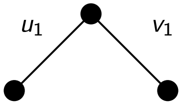
Algorithm for Vertex Cover:

$e_1 = u_1 v_1$

Bounded search tree method

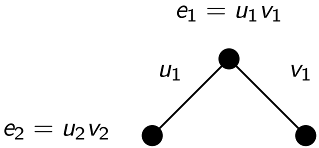
Algorithm for VERTEX COVER:

$e_1 = u_1 v_1$



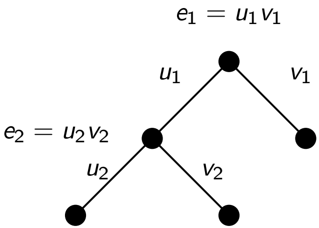
Bounded search tree method

Algorithm for VERTEX COVER:



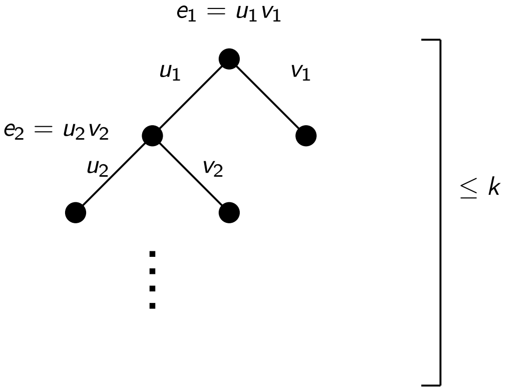
Bounded search tree method

Algorithm for Vertex Cover:



Bounded search tree method

Algorithm for VERTEX COVER:



Height of the search tree $\leq k \Rightarrow$ at most 2^k leaves $\Rightarrow 2^k \cdot n^{O(1)}$ time algorithm.

Fixed-parameter tractability

Main definition

A parameterized problem is **fixed-parameter tractable (FPT)** if there is an $f(k)n^c$ time algorithm for some constant c.

Fixed-parameter tractability

Main definition

A parameterized problem is **fixed-parameter tractable (FPT)** if there is an $f(k)n^c$ time algorithm for some constant c.

Examples of NP-hard problems that are FPT:

- Finding a vertex cover of size k.
- Finding a path of length k.
- Finding k disjoint triangles.
- Drawing the graph in the plane with k edge crossings.
- Finding disjoint paths that connect k pairs of points.
- ...

More formally

- We consider only **decision problems** here.
- Let Σ be a finite alphabet used to encode the inputs
  - ($\Sigma = \{0, 1\}$ for binary encodings)

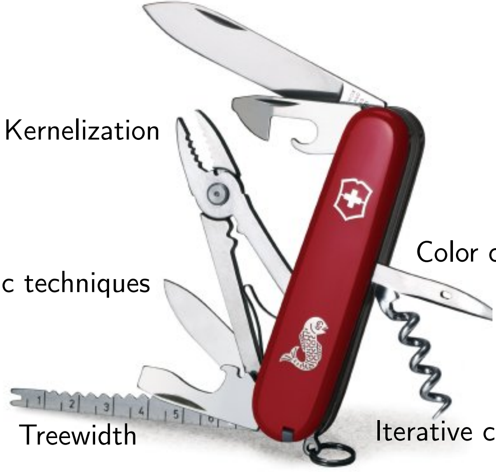
More formally

- We consider only **decision problems** here.
- Let Σ be a finite alphabet used to encode the inputs
  - ($\Sigma = \{0, 1\}$ for binary encodings)
- A **parameterized problem** is a set $P \subseteq \Sigma^* \times \mathbb{N}$
  - $P = \{(x_1, k_1), (x_2, k_2), \ldots\}$
- The set P contain the tuples (x, k) where the answer to the question encoded by (x, k) is yes; k is the **parameter**

More formally

- We consider only **decision problems** here.
- Let Σ be a finite alphabet used to encode the inputs
  - ($\Sigma = \{0, 1\}$ for binary encodings)
- A **parameterized problem** is a set $P \subseteq \Sigma^* \times \mathbb{N}$
  - $P = \{(x_1, k_1), (x_2, k_2), \dots\}$
- The set P contain the tuples (x, k) where the answer to the question encoded by (x, k) is yes; k is the **parameter**
- A parameterized problem P is **fixed-parameter tractable** if there is an algorithm that, given an input (x, k)
  - decides if (x, k) belongs to P or not, and
  - the running time is $f(k)n^c$ for some computable function f and constant c.

FPT techniques



W[1]-hardness

Negative evidence similar to NP-completeness. If a problem is **W[1]-hard,** then the problem is not FPT unless FPT=W[1].

Some W[1]-hard problems:

- Finding a clique/independent set of size k.
- Finding a dominating set of size k.
- Finding k pairwise disjoint sets.
- ...

W[1]-hardness

Negative evidence similar to NP-completeness. If a problem is **W[1]-hard,** then the problem is not FPT unless FPT=W[1].

Some W[1]-hard problems:

- Finding a clique/independent set of size k.
- Finding a dominating set of size k.
- Finding k pairwise disjoint sets.
- ...

General principle of hardness

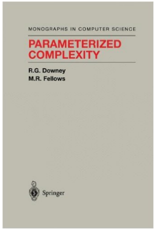
With an appropriate **reduction** from k-Clique to problem P, we show that if problem P is FPT, then k-Clique is also FPT.

Parameterized complexity

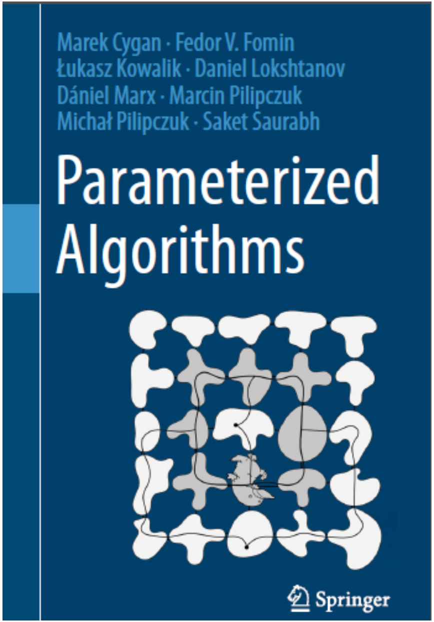
Rod G. Downey
Michael R. Fellows

Parameterized Complexity

Springer 1999



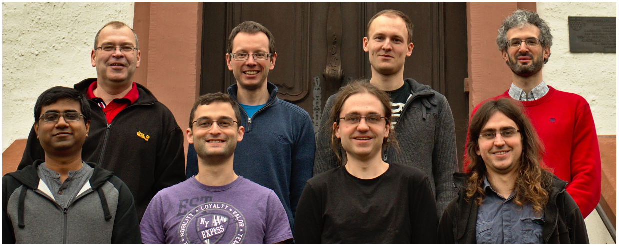
- The study of parameterized complexity was initiated by Downey and Fellows in the early 90s.
- First monograph in 1999.
- By now, strong presence in most algorithmic conferences.



Parameterized Algorithms

Marek Cygan, Fedor V. Fomin,
Lukasz Kowalik, Daniel Lokshtanov,
Dániel Marx, Marcin Pilipczuk,
Michał Pilipczuk, Saket Saurabh

Springer 2015



Course outline

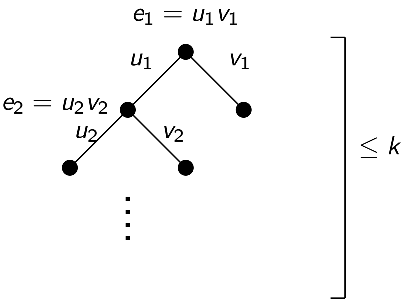
- Basic techniques
  - bounded search trees
  - color coding
  - dynamic programming
  - iterative compression
- Complexity
- Kernelization
- Treewidth
- Advanced topics:
  - cuts and separators
  - matroids
  - algebraic techniques

Bounded search tree method

Bounded search tree method

Algorithm for VERTEX COVER

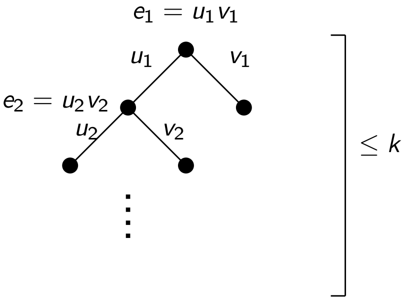
- **Main idea:** reduce problem instance (x, k) to solving a bounded number of instances with parameter $< k$.
- We should be able to solve instance (x, k) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most k.
- Size of the search tree:
  - If we branch into c directions: c^k.
  - If we branch into $O(k)$ directions: $k^{O(k)} = 2^{O(k \log k)}$.
  - (If we branch into $O(\log n)$ directions: $O(n) + 2^{O(k \log k)}$.)



Bounded search tree method

Algorithm for Vertex Cover

- **Main idea:** reduce problem instance (x, k) to solving a bounded number of instances with parameter $< k$.
- We should be able to solve instance (x, k) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most k.
- Size of the search tree:
  - If we branch into c directions: c^k.
  - If we branch into $O(k)$ directions: $k^{O(k)} = 2^{O(k \log k)}$.
  - (If we branch into $O(\log n)$ directions: $O(n) + 2^{O(k \log k)}$.)

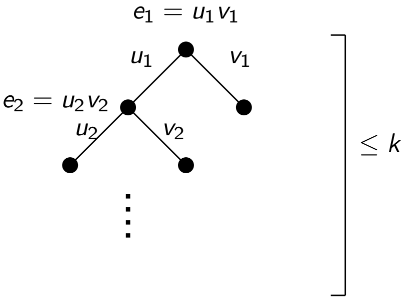


Next: A $1.41^k \cdot n^{O(1)}$ time algorithm for Vertex Cover.

Bounded search tree method

Algorithm for Vertex Cover

- **Main idea:** reduce problem instance (x, k) to solving a bounded number of instances with parameter $< k$.
- We should be able to solve instance (x, k) in polynomial time using the solutions of the new instances.
- If the parameter strictly decreases in every recursive call, then the depth is at most k.
- Size of the search tree:
  - If we branch into c directions: c^k.
  - If we branch into $O(k)$ directions: $k^{O(k)} = 2^{O(k \log k)}$.
  - (If we branch into $O(\log n)$ directions: $O(n) + 2^{O(k \log k)}$.)



Next: A $O^*(1.41^k)$ time algorithm for Vertex Cover.

Improved branching for VERTEX COVER

- If every vertex has degree ≤ 2, then the problem can be solved in polynomial time.
- **Branching rule:**
  If there is a vertex v with at least 3 neighbors, then
  - either v is in the solution,
  - or every neighbor of v is in the solution.

Crude upper bound: $O^*(2^k)$, since the branching rule decreases the parameter.

Improved branching for Vertex Cover

- If every vertex has degree ≤ 2, then the problem can be solved in polynomial time.
- **Branching rule:**
  If there is a vertex v with at least 3 neighbors, then
  - either v is in the solution, $\Rightarrow k$ decreases by 1
  - or every neighbor of v is in the solution. $\Rightarrow k$ decreases by at least 3

Crude upper bound: $O^*(2^k)$, since the branching rule decreases the parameter.

But it is somewhat better than that, since in the second branch, the parameter decreases by at least 3.

Better analysis

Let $T(k)$ be the maximum number of leaves of the search tree if the parameter is at most k (let $T(k) = 1$ for $k \leq 0$).

$$T(k) \leq T(k-1) + T(k-3)$$

There is a standard technique for bounding such functions asymptotically.

Better analysis

Let $T(k)$ be the maximum number of leaves of the search tree if the parameter is at most k (let $T(k) = 1$ for $k \leq 0$).

$$T(k) \leq T(k-1) + T(k-3)$$

There is a standard technique for bounding such functions asymptotically.

We prove by induction that $T(k) \leq c^k$ for some $c > 1$ as small as possible.

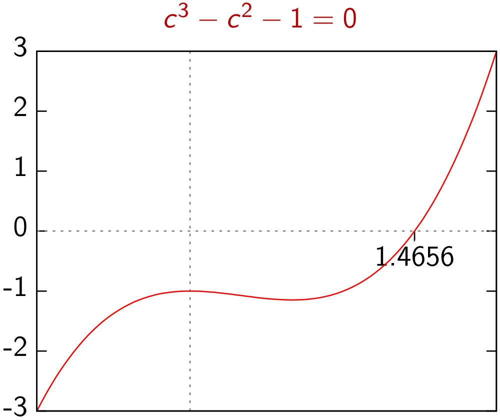
What values of c are good? We need:

$$c^k \geq c^{k-1} + c^{k-3}$$
$$c^3 - c^2 - 1 \geq 0$$

We need to find the roots of the **characteristic equation** $c^3 - c^2 - 1 = 0$.

Note: it is always true that such an equation has a unique positive root.

Better analysis



$c = 1.4656$ is a good value $\Rightarrow$ $T(k) \leq 1.4656^k$

$\Rightarrow$ We have a $O^*(1.4656^k)$ algorithm for VERTEX COVER.

Better analysis

We showed that if $T(k) \leq T(k-1) + T(k-3)$, then $T(k) \leq 1.4656^k$ holds.

Is this bound tight? There are two questions:

- Can the function $T(k)$ be that large?
  Yes (ignoring rounding problems).
- Can the search tree of the VERTEX COVER algorithm be that large?
  Difficult question, hard to answer in general.

Branching vectors

The **branching vector** of our $O^*(1.4656^k)$ Vertex Cover algorithm was $(1, 3)$.

Example: Let us bound the search tree for the branching vector $(2, 5, 6, 6, 7, 7)$.
(2 out of the 6 branches decrease the parameter by 7, etc.).

Branching vectors

The **branching vector** of our $O^*(1.4656^k)$ VERTEX COVER algorithm was $(1,3)$.

Example: Let us bound the search tree for the branching vector $(2,5,6,6,7,7)$.
(2 out of the 6 branches decrease the parameter by 7, etc.).

The value $c > 1$ has to satisfy:

$$c^k \geq c^{k-2} + c^{k-5} + 2c^{k-6} + 2c^{k-7}$$
$$c^7 - c^5 - c^2 - 2c - 2 \geq 0$$

Unique positive root of the characteristic equation: $1.4483 \Rightarrow T(k) \leq 1.4483^k$.

It is hard to compare branching vectors intuitively.

Branching vectors

Example: The roots for branching vector (i, j) $(1 \leq i, j \leq 6)$.

$$T(k) \leq T(k-i) + T(k-j) \Rightarrow c^k \geq c^{k-i} + c^{k-j}$$
$$c^j - c^{j-i} - 1 \geq 0$$

We compute the unique positive root.

	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2	1.6181	1.4143	1.3248	1.2721	1.2366	1.2107
3	1.4656	1.3248	1.2560	1.2208	1.1939	1.1740
4	1.3803	1.2721	1.2208	1.1893	1.1674	1.1510
5	1.3248	1.2366	1.1939	1.1674	1.1487	1.1348
6	1.2852	1.2107	1.1740	1.1510	1.1348	1.1225

Example: Triangle Free Deletion

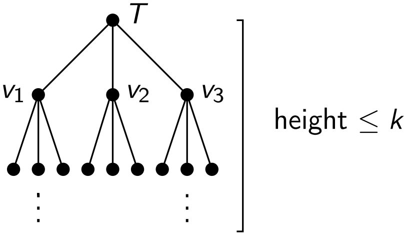
> **Triangle Free Deletion**
> Given (G, k), remove at most k vertices to make the graph triangle free.

What is the running time of a simple branching algorithm?

Example: Triangle Free Deletion

> **Triangle Free Deletion**
> Given (G, k), remove at most k vertices to make the graph triangle free.

What is the running time of a simple branching algorithm?



The search tree has at most 3^k leaves and the work to be done is polynomial at each step $\Rightarrow$ $O^*(3^k)$ time algorithm.

Note: If the answer is "NO", then the search tree has **exactly** 3^k leaves.

Graph modification problems

A general problem family containing tasks of the following type:

Given (G, k), do at most k allowed operations on G to make it have property $\mathcal{P}$.

- Allowed operations: vertex deletion, edge deletion, edge addition, ...
- Property $\mathcal{P}$: edgeless, no triangles, no cycles, planar, chordal, regular, disconnected, ...

Examples:

- Vertex Cover: Delete k vertices to make G edgeless.
- Triangle Free Deletion: Delete k vertices to make G triangle free.
- Feedback Vertex Set: Delete k vertices to make G acyclic (forest).

Hereditary properties

Definition

A graph property $\mathcal{P}$ is **hereditary** or **closed under induced subgraphs** if whenever $G \in P$, every induced subgraph of G is also in $\mathcal{P}$.

"removing a vertex does not ruin the property"
(e.g., triangle free, bipartite, planar)

Hereditary properties

Definition

A graph property $\mathcal{P}$ is **hereditary** or **closed under induced subgraphs** if whenever $G \in P$, every induced subgraph of G is also in $\mathcal{P}$.

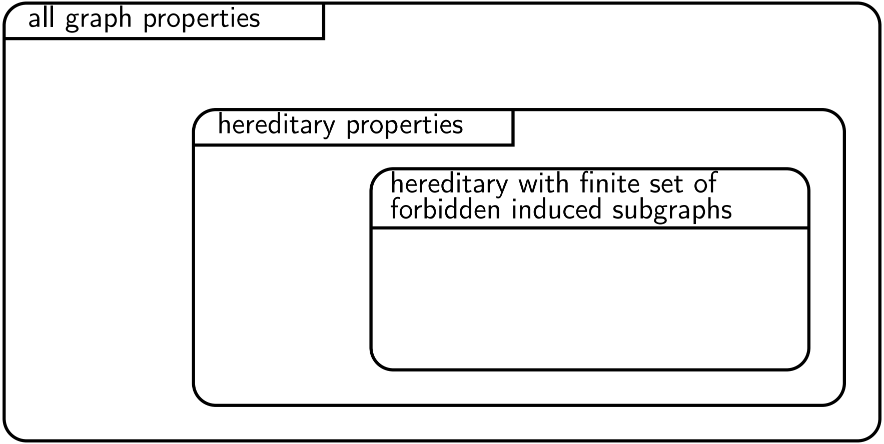
"removing a vertex does not ruin the property"
(e.g., triangle free, bipartite, planar)

Observation

Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of "minimal bad graphs" or "forbidden induced subgraphs": $G \in \mathcal{P}$ if and only if G **does not** have an induced subgraph isomorphic to a member of $\mathcal{F}$.

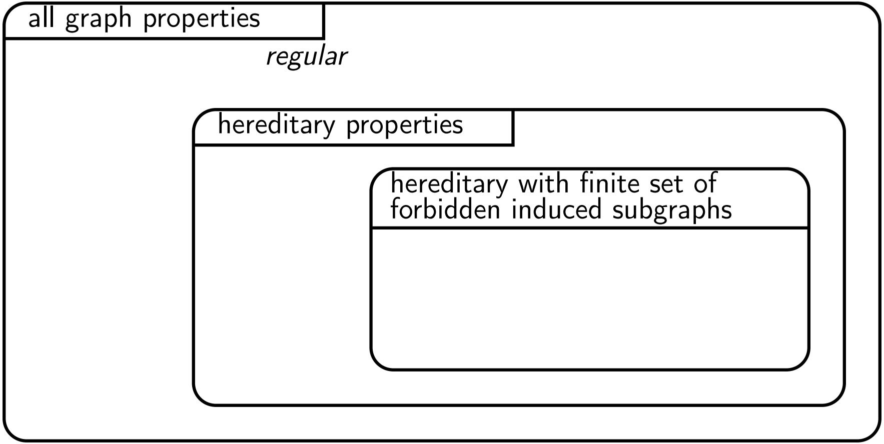
Example: a graph is bipartite if and only if it does not contain an odd cycle as an induced subgraph.

Graph properties



regular *bipartite* *triangle free* *connected*
planar *empty* *complete* *acyclic*

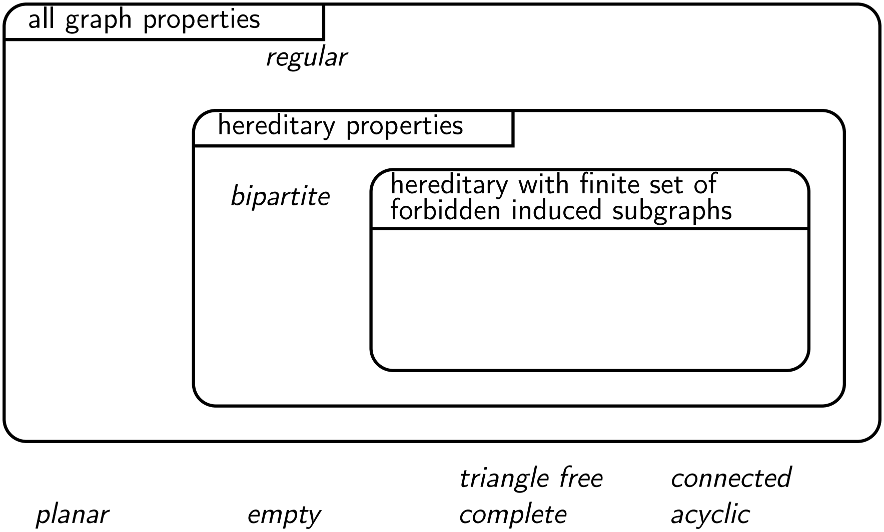
Graph properties



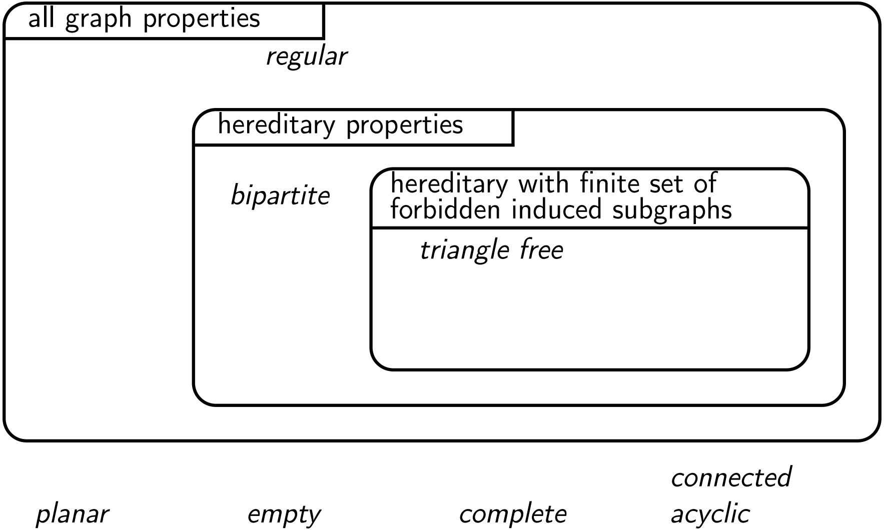
bipartite *triangle free* *connected*

planar *empty* *complete* *acyclic*

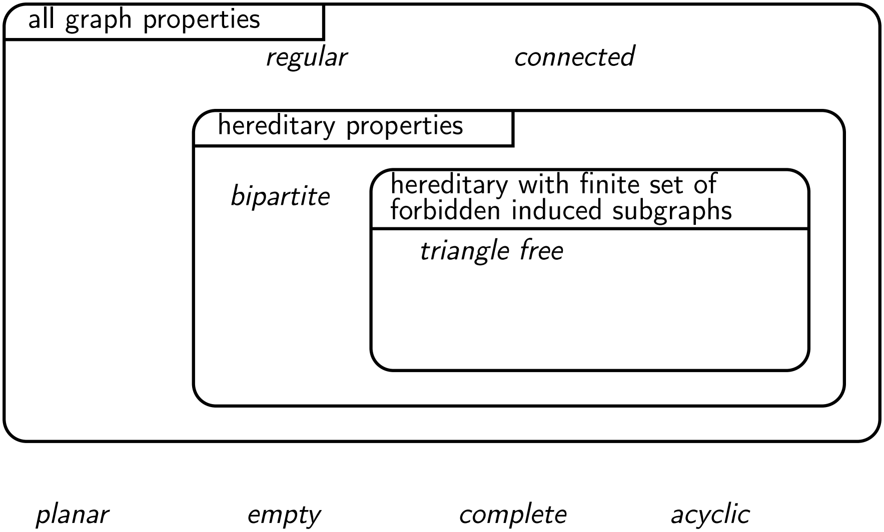
Graph properties



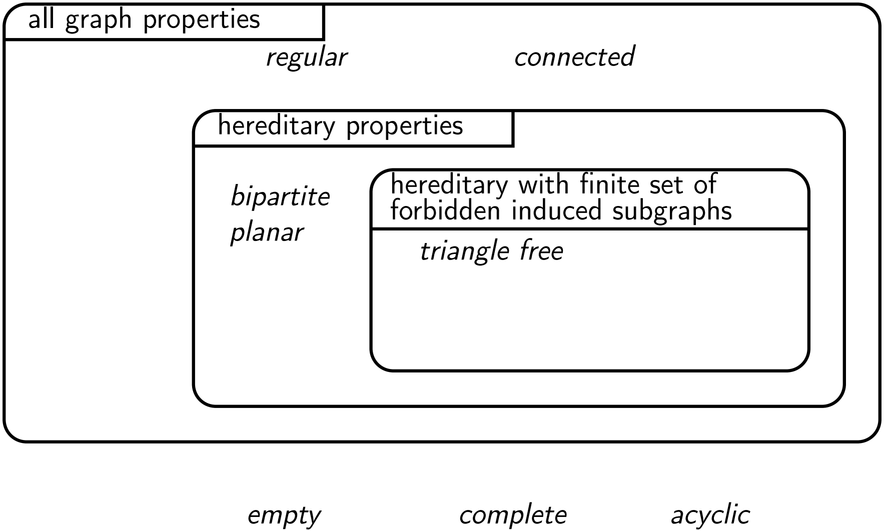
Graph properties



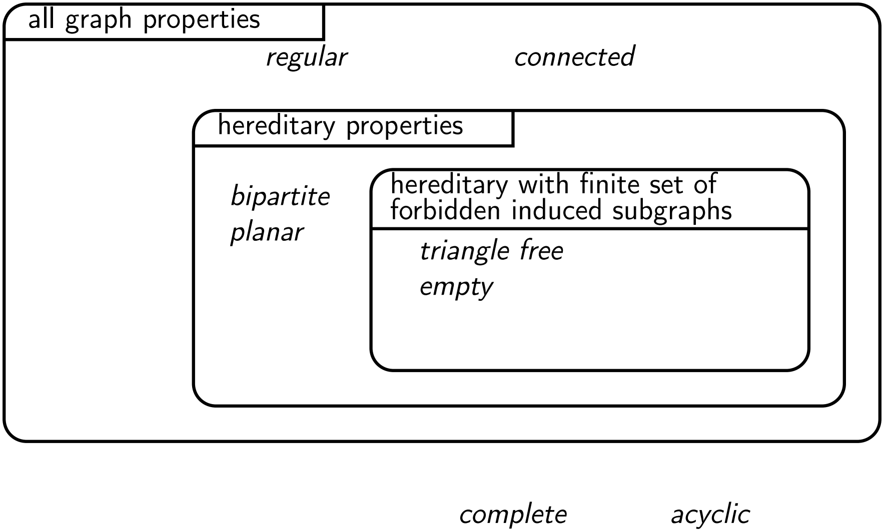
Graph properties



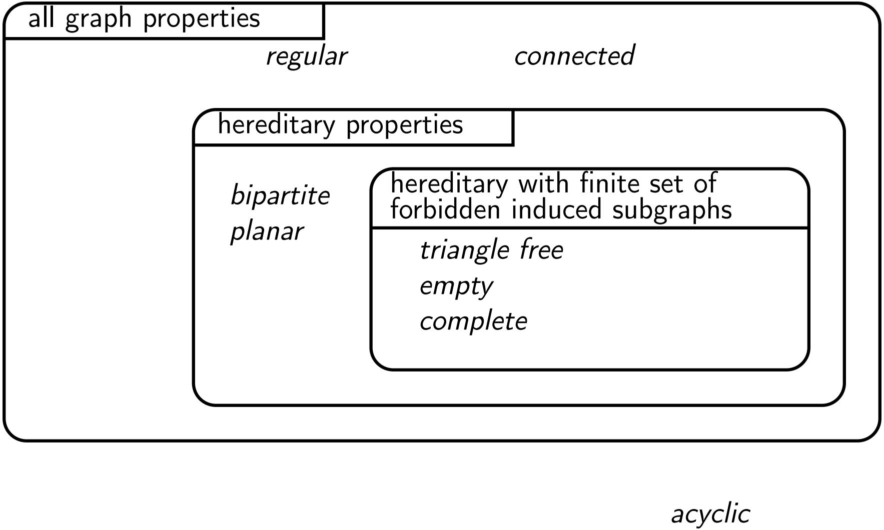
Graph properties



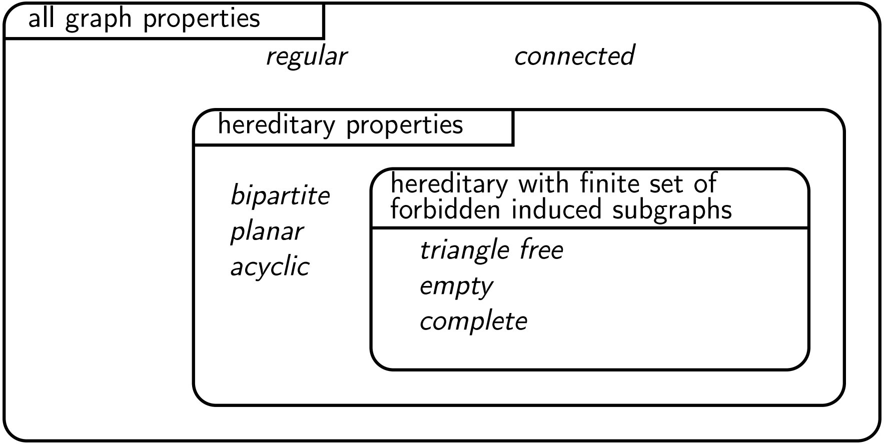
Graph properties



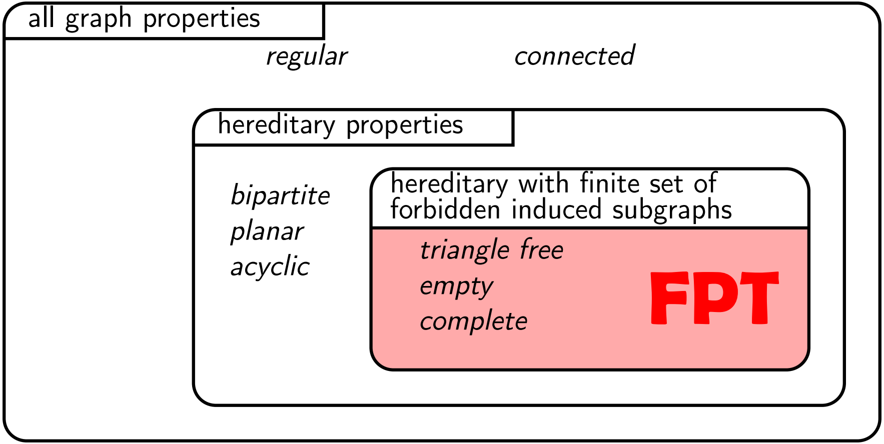
Graph properties



Graph properties



Graph properties



Using finite obstructions

Theorem

If $\mathcal{P}$ is hereditary and can be characterized by a **finite** set $\mathcal{F}$ of forbidden induced subgraphs, then the graph modification problems corresponding to $\mathcal{P}$ are FPT.

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Using brute force, we can find in time $O(n^r)$ a forbidden subgraph (if exists).
- If a forbidden subgraph exists, then we have to delete one of the at most r vertices or add/delete one of the at most $\binom{r}{2}$ edges
  $\Rightarrow$ Branching factor is a constant c depending on $\mathcal{F}$.
- The search tree has at most c^k leaves and the work to be done at each node is $O(n^r)$.

Graph modification problems

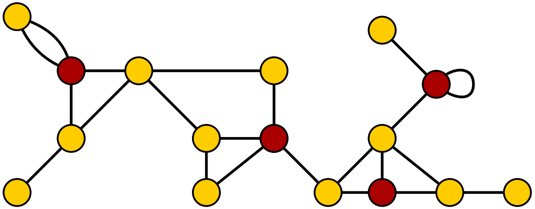
A very wide and active research area in parameterized algorithms.

- If the set of forbidden subgraphs is finite, then the problem is immediately FPT (e.g., Vertex Cover, Triangle Free Deletion). Here the challange is improving the naive running time.
- If the set of forbidden subgraphs is infinite, then very different techniques are needed to show that the problem is FPT (e.g., Feedback Vertex Set, Bipartite Deletion, Planar Deletion).

Feedback Vertex Set

Feedback Vertex Set:
Given (G, k), find a set S of at most k vertices such that $G - S$ has no cycles.

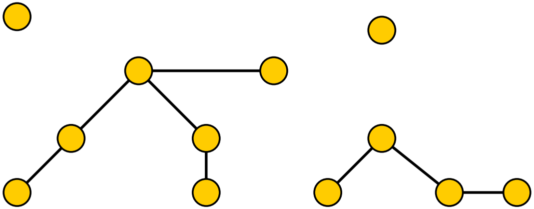
- We allow multiple parallel edges and self loops.
- A **feedback vertex set** is a set that hits every cycle in the graph.



Feedback Vertex Set

Feedback Vertex Set:
Given (G, k), find a set S of at most k vertices such that $G - S$ has no cycles.

- We allow multiple parallel edges and self loops.
- A **feedback vertex set** is a set that hits every cycle in the graph.



Feedback Vertex Set

- If we find a cycle, then we have to include at least one of its vertices into the solution. But the length of the cycle can be arbitrary large!
- **Main idea:** We identify a set of $O(k)$ vertices such that any size-k feedback vertex set has to contain one of these vertices.
- But first: some reductions to simplify the problem.

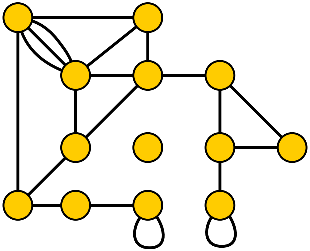
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



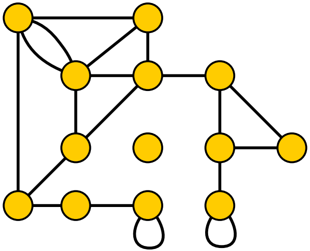
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



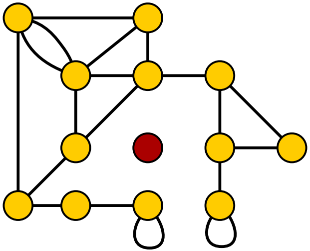
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



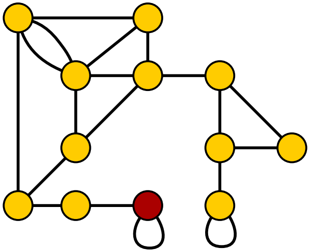
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



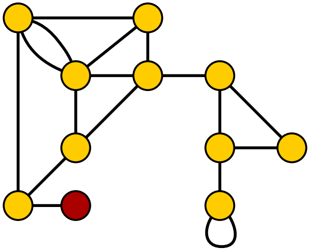
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



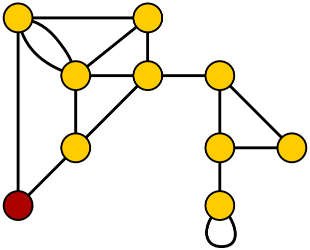
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



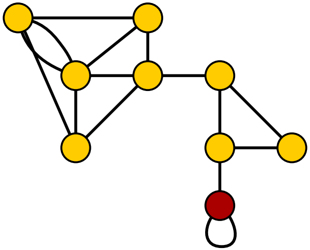
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



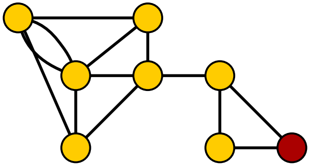
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



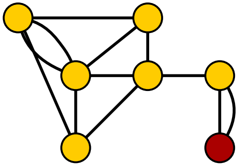
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



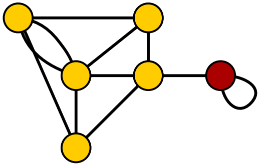
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



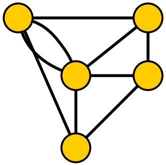
Reduction rules

(R1) If there is a loop at v, then delete v and decrease k by one.

(R2) If there is an edge of multiplicity larger than 2, then reduce its multiplicity to 2.

(R3) If there is a vertex v of degree at most 1, then delete v.

(R4) If there is a vertex v of degree 2, then delete v and add an edge between the neighbors of v.



If the reduction rules cannot be applied, then every vertex has degree at least 3.

Branching

Let G be a graph whose vertices have degree at least 3.

- Order the vertices as v_1, v_2, ..., v_n by **decreasing** degree (breaking ties arbitrarily).
- Let $V_{3k} = \{v_1, \dots, v_{3k}\}$ be the $3k$ largest-degree vertices.

Lemma

If G has minimum degree at least 3, then every feedback vertex set S of size at most k contains a vertex from V_{3k}.

Branching

Let G be a graph whose vertices have degree at least 3.

- Order the vertices as v_1, v_2, ..., v_n by **decreasing** degree (breaking ties arbitrarily).
- Let $V_{3k} = \{v_1, \ldots, v_{3k}\}$ be the $3k$ largest-degree vertices.

Lemma

If G has minimum degree at least 3, then every feedback vertex set S of size at most k contains a vertex from V_{3k}.

Algorithm:

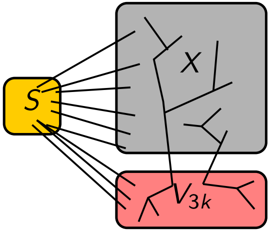
- Apply the reduction rules (poly time) $\Rightarrow$ graph has minimum degree 3.
- For each vertex $v \in V_{3k}$, recurse on the instance $(G - v, k - 1)$.
- Running time $(3k)^k \cdot n^{O(1)} = 2^{O(k \log k)} \cdot n^{O(1)}$.

Proof of the lemma

Lemma

If G has minimum degree at least 3, then every feedback vertex set S of size at most k contains a vertex from V_{3k}.

- $d :=$ minimum degree in V_{3k} ,
  $X = V(G) - (S \cup V_{3k})$.
- Total degree of $V_{3k} \cup X$: $\geq 3kd + 3|X|$
- Edges of $G[V_{3k} \cup X]$: $\leq 3k + |X| - 1$
- Total degree of these edges: $\leq 6k + 2|X| - 2$

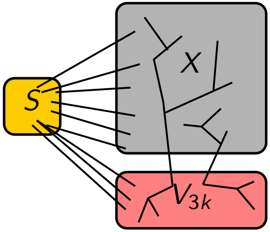


Proof of the lemma

Lemma

If G has minimum degree at least 3, then every feedback vertex set S of size at most k contains a vertex from V_{3k}.

- $d :=$ minimum degree in V_{3k} ,
  $X = V(G) - (S \cup V_{3k})$.
- Total degree of $V_{3k} \cup X$: $\geq 3kd + 3|X|$
- Edges of $G[V_{3k} \cup X]$: $\leq 3k + |X| - 1$
- Total degree of these edges: $\leq 6k + 2|X| - 2$
- Edges between S and $V_{3k} \cup X$:
  - $\leq dk$
  - $\geq 3kd + 3|X| - (6k + 2|X| - 2) > 3(d-2)k$
- As $d \geq 3$, we have $3(d-2) \geq d$, contradiction.



Branching: wrap up

- Branching into c directions: $O^*(c^k)$ algorithms.
- Branching into k directions: $O^*(k^k)$ algorithms.
- Branching vectors and analysis of recurrences of the form

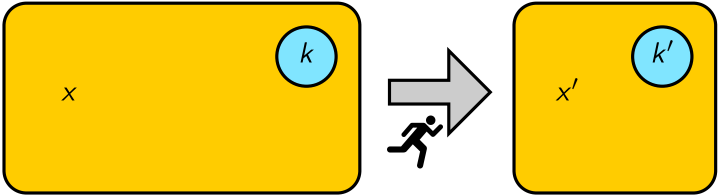
$$T(k) = T(k-1) + 2T(k-2) + T(k-3)$$

- Graph modification problems where the graph property can be characterized by a finite set of forbidden induced subgraphs is FPT.

Kernelization

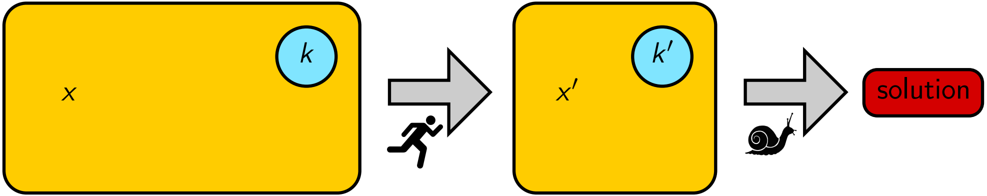
Data reductions—with a guarantee

- **Kernelization** is a method for parameterized preprocessing:
  - We want to efficiently reduce the size of the instance (x, k) to an equivalent instance with size bounded by $f(k)$.
- A basic way of obtaining FPT algorithms:
  - Reduce the size of the instance to $f(k)$ in polynomial time and then apply any brute force algorithm to the shrunk instance.
- Kernelization is also a rigorous mathematical analysis of efficient preprocessing.



Data reductions—with a guarantee

- **Kernelization** is a method for parameterized preprocessing:
  - We want to efficiently reduce the size of the instance (x, k) to an equivalent instance with size bounded by $f(k)$.
- A basic way of obtaining FPT algorithms:
  - Reduce the size of the instance to $f(k)$ in polynomial time and then apply any brute force algorithm to the shrunk instance.
- Kernelization is also a rigorous mathematical analysis of efficient preprocessing.



Kernel for VERTEX COVER

Reduction rules for instance (G, k):

(R1) If v is an isolated vertex, then reduce to $(G - v, k)$.

(R2) If v has degree more than k, then reduce to $(G - v, k - 1)$.

Kernel for VERTEX COVER

Reduction rules for instance (G, k):

(R1) If v is an isolated vertex, then reduce to $(G - v, k)$.

(R2) If v has degree more than k, then reduce to $(G - v, k - 1)$.

Lemma

If (G, k) is a yes-instance of VERTEX COVER such that (R1) and (R2) cannot be applied, then $|E(G)| \leq k^2$ and $|V(G)| \leq k^2 + k$.

Kernel for Vertex Cover

Reduction rules for instance (G, k):

(R1) If v is an isolated vertex, then reduce to $(G - v, k)$.

(R2) If v has degree more than k, then reduce to $(G - v, k - 1)$.

Lemma

If (G, k) is a yes-instance of Vertex Cover such that (R1) and (R2) cannot be applied, then $|E(G)| \leq k^2$ and $|V(G)| \leq k^2 + k$.

Proof:

- Each of the k vertices of the solution can cover at most k edges (by (R2)).
- Every vertex of G is either in the solution, or one of the $\leq k$ neighbors of a vertex in a solution (by (R1)+(R2)).

Kernel for Vertex Cover

Reduction rules for instance (G, k):

(R1) If v is an isolated vertex, then reduce to $(G - v, k)$.

(R2) If v has degree more than k, then reduce to $(G - v, k - 1)$.

Lemma

If (G, k) is a yes-instance of Vertex Cover such that (R1) and (R2) cannot be applied, then $|E(G)| \leq k^2$ and $|V(G)| \leq k^2 + k$.

Kernelization for Vertex Cover:

- Apply rules (R1) and (R2) exhaustively.
- If $|E(G)| > k^2$ or $|V(G)| > k^2 + k$, then we have a no-instance.
- Otherwise, we have a kernel of size $O(k^2)$.

Kernelization: formal definition

- Let $P \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized probem and $f : \mathbb{N} \to \mathbb{N}$ a computable function.
- A **kernel** for P of size f is an algorithm that, given (x, k), takes time polynomial in $|x| + k$ and outputs an instance (x', k') such that
    - $(x, k) \in P \iff (x', k') \in P$
    - $|x'| \leq f(k)$, $k' \leq f(k)$.
- A **polynomial kernel** is a kernel whose function f is polynomial.

Kernelization: formal definition

- Let $P \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized probem and $f : \mathbb{N} \to \mathbb{N}$ a computable function.
- A **kernel** for P of size f is an algorithm that, given (x, k), takes time polynomial in $|x| + k$ and outputs an instance (x', k') such that
  - $(x, k) \in P \iff (x', k') \in P$
  - $|x'| \leq f(k)$, $k' \leq f(k)$.
- A **polynomial kernel** is a kernel whose function f is polynomial.

Which parameterized problems have kernels?

A surprising equivalence

Theorem

A parameterized problem is FPT if and only if it is decidable and has a kernel (of arbitrary size).

A surprising equivalence

Theorem

A parameterized problem is FPT if and only if it is decidable and has a kernel (of arbitrary size).

Proof:

- If the problem has a kernel:
  Reducing the size of the instance to $f(k)$ in poly time + brute force
  $\Rightarrow$ problem is FPT.

A surprising equivalence

Theorem

A parameterized problem is FPT if and only if it is decidable and has a kernel (of arbitrary size).

Proof:

- If the problem has a kernel:
  Reducing the size of the instance to $f(k)$ in poly time + brute force
  $\Rightarrow$ problem is FPT.
- If the problem can be solved in time $f(k)|x|^{O(1)}$:
  - If $|x| \leq f(k)$, then we already have a kernel of size $f(k)$.
  - If $|x| \geq f(k)$, then we can solve the problem in time $f(k)|x|^{O(1)} \leq |x| \cdot |x|^{O(1)}$ (polynomial in $|x|$) and then output a trivial yes- or no-instance.

A surprising equivalence

Theorem

A parameterized problem is FPT if and only if it is decidable and has a kernel (of arbitrary size).

Proof:

- If the problem has a kernel:
  Reducing the size of the instance to $f(k)$ in poly time + brute force
  $\Rightarrow$ problem is FPT.
- If the problem can be solved in time $f(k)|x|^{O(1)}$:
  - If $|x| \leq f(k)$, then we already have a kernel of size $f(k)$.
  - If $|x| \geq f(k)$, then we can solve the problem in time $f(k)|x|^{O(1)} \leq |x| \cdot |x|^{O(1)}$ (polynomial in $|x|$) and then output a trivial yes- or no-instance.
- The existence of kernels is not a separate question...
- ...but the existence of **polynomial kernels** is a deep and nontrivial topic!

Color Coding

Why randomized?

- A guaranteed error probability of 10^{-100} is as good as a deterministic algorithm. (Probability of hardware failure is larger!)
- Randomized algorithms can be more efficient and/or conceptually simpler.
- Can be the first step towards a deterministic algorithm.

Polynomial-time vs. FPT randomization

Polynomial-time randomized algorithms

- Randomized selection to pick a **typical**, **unproblematic**, **average** element/subset.
- Success probability is constant or at most polynomially small.

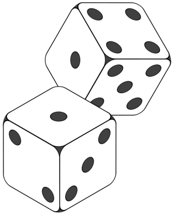
Randomized FPT algorithms

- Randomized selection to satisfy a **bounded number** of (unknown) constraints.
- Success probability might be exponentially small.

Randomization as reduction

Problem A
(what we want to solve)

Randomized magic



Problem B
(what we can solve)

Color Coding

k-Path

Input: A graph G, integer k.
Find: A simple path on k vertices.

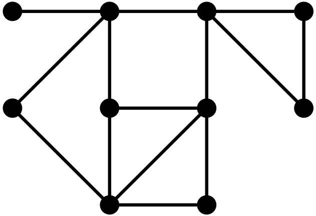
Note: The problem is clearly NP-hard, as it contains the Hamiltonian Path problem.

Theorem

k-Path can be solved in time $2^{O(k)} \cdot n^{O(1)}$.

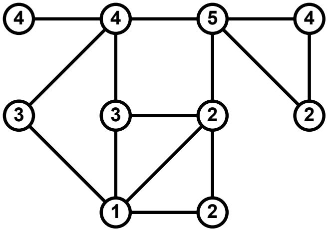
Color Coding

- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



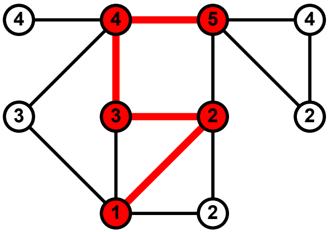
Color Coding

- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



Color Coding

- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



- Check if there is a path colored $1 - 2 - \cdots - k$; output "YES" or "NO".
  - If there is no k-path: no path colored $1 - 2 - \cdots - k$ exists $\Rightarrow$ "NO".
  - If there is a k-path: the probability that such a path is colored $1 - 2 - \cdots - k$ is k^{-k} thus the algorithm outputs "YES" with at least that probability.

Error probability

Useful fact

If the probability of success is at least p, then the probability that the algorithm **does not** say "YES" after $1/p$ repetitions is at most

$$(1-p)^{1/p} < \left(e^{-p}\right)^{1/p} = 1/e \approx 0.38$$

Error probability

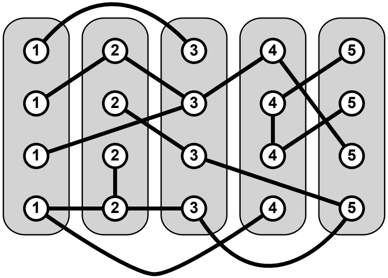
Useful fact

If the probability of success is at least p, then the probability that the algorithm **does not** say "YES" after $1/p$ repetitions is at most

$$(1-p)^{1/p} < \left(e^{-p}\right)^{1/p} = 1/e \approx 0.38$$

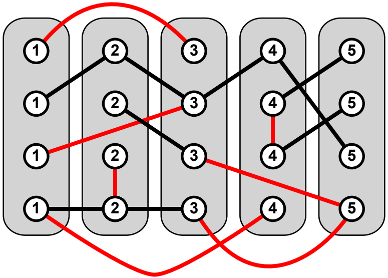
- Thus if $p > k^{-k}$, then error probability is at most $1/e$ after k^k repetitions.
- Repeating the whole algorithm a constant number of times can make the error probability an arbitrary small constant.
- For example, by trying $100 \cdot k^k$ random colorings, the probability of a wrong answer is at most $1/e^{100}$.

Finding a path colored $1 - 2 - \cdots - k$



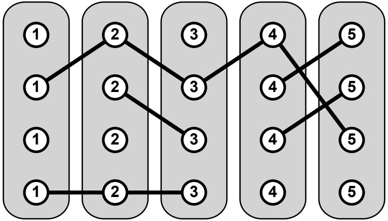
- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed towards the larger class.
- All we need to check if there is a directed path from class 1 to class k.

Finding a path colored $1 - 2 - \cdots - k$



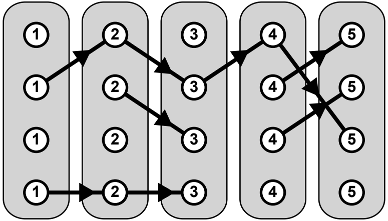
- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed towards the larger class.
- All we need to check if there is a directed path from class 1 to class k.

Finding a path colored $1 - 2 - \cdots - k$



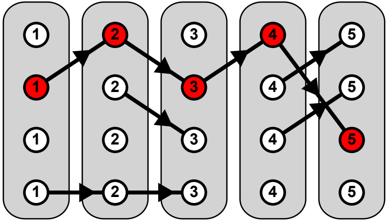
- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed towards the larger class.
- All we need to check if there is a directed path from class 1 to class k.

Finding a path colored $1 - 2 - \cdots - k$



- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed towards the larger class.
- All we need to check if there is a directed path from class 1 to class k.

Finding a path colored $1 - 2 - \cdots - k$

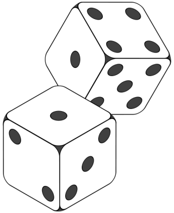


- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed towards the larger class.
- All we need to check if there is a directed path from class 1 to class k.

Color Coding

k-**PATH**

Color Coding
success probability: k^{-k}

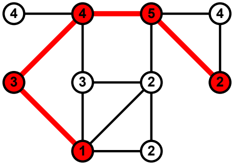


Finding a
$1 - 2 - \cdots - k$ **colored**
path

polynomial-time solvable

Improved Color Coding

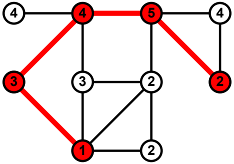
- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



- Check if there is a **colorful** path where each color appears exactly once on the vertices; output "YES" or "NO".

Improved Color Coding

- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



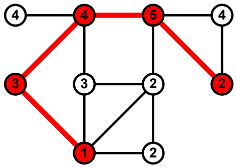
- Check if there is a **colorful** path where each color appears exactly once on the vertices; output "YES" or "NO".
  - If there is no k-path: no **colorful** path exists $\Rightarrow$ "NO".
  - If there is a k-path: the probability that it is **colorful** is

$$\frac{k!}{k^k} > \frac{\left(\frac{k}{e}\right)^k}{k^k} = e^{-k},$$

thus the algorithm outputs "YES" with at least that probability.

Improved Color Coding

- Assign colors from $[k]$ to vertices $V(G)$ uniformly and independently at random.



- Repeating the algorithm $100e^k$ times decreases the error probability to e^{-100}.

How to find a colorful path?

- Try all permutations ($k! \cdot n^{O(1)}$ time)
- Dynamic programming ($2^k \cdot n^{O(1)}$ time)

Finding a colorful path

Subproblems:

We introduce $2^k \cdot |V(G)|$ Boolean variables:

$$x(v, C) = \text{TRUE for some } v \in V(G) \text{ and } C \subseteq [k]$$
$$\Updownarrow$$
There is a path P ending at v such that each color in C appears on P exactly once and no other color appears.

Answer:

There is a colorful path $\iff$ $x(v, [k]) = \text{TRUE}$ for some vertex v.

Finding a colorful path

Subproblems:

We introduce $2^k \cdot |V(G)|$ Boolean variables:

$x(v, C) = \text{TRUE}$ for some $v \in V(G)$ and $C \subseteq [k]$

$\Updownarrow$

There is a path P ending at v such that each color in C appears on P exactly once and no other color appears.

Initialization:

For every v with color r, $x(v, \{r\}) = \text{TRUE}$.

Recurrence:

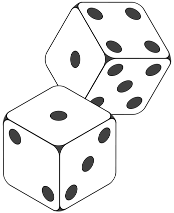
For every v with color r and set $C \subseteq [k]$

$$x(v, C) = \bigvee_{u \in N(v)} x(u, C \setminus \{r\}).$$

Improved Color Coding

k-**PATH**

Color Coding
success probability: e^{-k}



Finding a colorful path

Solvable in time $2^k \cdot n^{O(1)}$

Derandomization

Definition

A family $\mathcal{H}$ of functions $[n] \to [k]$ is a k-**perfect** family of hash functions if for every $S \subseteq [n]$ with $|S| = k$, there is an $h \in \mathcal{H}$ such that $h(x) \neq h(y)$ for any $x, y \in S$, $x \neq y$.

Theorem

There is a k-perfect family of functions $[n] \to [k]$ having size $2^{O(k)} \log n$ (and can be constructed in time polynomial in the size of the family).

Derandomization

Definition

A family $\mathcal{H}$ of functions $[n] \to [k]$ is a k-**perfect** family of hash functions if for every $S \subseteq [n]$ with $|S| = k$, there is an $h \in \mathcal{H}$ such that $h(x) \neq h(y)$ for any $x, y \in S$, $x \neq y$.

Theorem

There is a k-perfect family of functions $[n] \to [k]$ having size $2^{O(k)} \log n$ (and can be constructed in time polynomial in the size of the family).

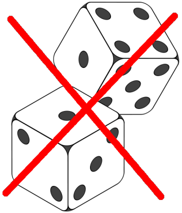
Instead of trying $O(e^k)$ **random colorings,** we go through a k-**perfect family** $\mathcal{H}$ of functions $V(G) \to [k]$.

If there is a solution S
$\Rightarrow$ The vertices of S are colorful for at least one $h \in \mathcal{H}$
$\Rightarrow$ Algorithm outputs "YES".
$\Rightarrow$ k-Path can be solved in **deterministic** time $2^{O(k)} \cdot n^{O(1)}$.

Derandomized Color Coding

k-PATH

k-perfect family
$2^{O(k)} \log n$ functions



Finding a colorful path

Solvable in time $2^k \cdot n^{O(1)}$

Summary

- **Branching**
  - $2^{O(k)} \cdot n^{O(1)}$ time algorithms for Vertex Cover and Triangle Free Deletion.
  - $2^{O(k \log k)} n^{O(1)}$ time algorithms for Feedback Vertex Set and Closest String
- **Kernelization**
  - $O(k^2)$ kernel for Vertex Cover.
- **Color Coding**
  - $2^{O(k)} \cdot n^{O(1)}$ (randomized) algorithm for k-Path.

The race for better FPT algorithms