Parametrized Complexity and Graph Minor Theory

Pierre Aboulker - pierreaboulker@gmail.com

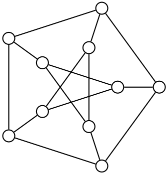
6 Hours' Programm

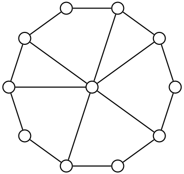
- Definitions of parametrized complexity (FPT, XP, W[1])
- Branching method
  - VERTEX COVER in time $O(1.46^k n^{0(1)})$
  - Branching vector
  - GRAPH MODIFICATION PROBLEM
  - FEEDBACK VERTEX SET in time $(3k)^k \cdot n^{O(1)}$
- Kernelization
  - k-VERTEX COVER has a $k^2 + k$ kernel
  - VERTEX COVER has a $3k$ kernel (crown decomposition)
  - VERTEX COVER has a $2k$ kernel (Linear Programming)
  - d-HITTING SET PROBLEM has a $d!k^d d^2$ kernel (Sunflower Lemma)
- Color Coding
  - LONGEST PATH in time $2^k n^{0(1)}$
- Iterative Compression
  - FEEDBACK VERTEX SET in time $5^k n^{0(1)}$

Graphs

A graph $G = (V, E)$:

- V is the set of vertices
- $E \subseteq V \times V$ is the set of edges.





All along the course, particularly for complexity analysis,

- n is the number of vertices,
- m is the number of edges.

An algorithm going in time $O(n + m)$ is said to be linear.

Basic Definitions and Terminology

In this course, all graphs are simple (no parallel edges) and without loop, unless expressly stated.

If G is a graph, we denote $V(G)$ its set of vertices and $E(G)$ its set of edges.

A vertex v is adjacent with a vertex u if $uv \in E(G)$. The neighbourhood of u, denoted $N(u)$ is the set of neighbours of u.
Its degree, denoted tcdarkred$d(u)$ is the cardinality of its neighbourhood. The maximum degree of a graph is denoted $\Delta(G)$. Given a set of vertices X, $N(X)$ is the set of vertices not in X that have at least one neighbour in X.

A graph with no edge is a stable set, or independent set, and a graph with all possible edges ($\binom{n}{2}$) is a clique, or complete graph. The complete graph on n vertices is denoted K_n. The complete bipartite graph with parts of size a and b is denoted $K_{a,b}$.

The path P_k is a graph with $V(P_k) = \{x_1, x_2, \ldots, x_k\}$ and $E(P_k) = \{x_i x_{i+1},\ 1 \leq i \leq k-1\}$. The vertices x_1 and x_k are called the endpoints of the path. If we add the edge $x_k x_1$ to P_k, then the resulting graph is the cycle on k vertices, denoted C_k.

Some graph parameters

- $\delta(G)$: minimum degree.
- $\Delta(G)$: maximum degree.
- $\omega(G)$: clique number.
- $\alpha(G)$: size of a maximum independent set.
- $\chi(G)$: chromatic number.
- $\tau(G)$: vertex cover.
- $\kappa(G)$: vertex connectivity.
- $tw(G)$: treewidth, measure how much a graph looks like a tree.

Parametrized Complexity and FPT Algorithms

Slides are inspired by a course of Daniel Marx, and another course of Marcin Pilipczuk.

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 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, maximum flow, 2-SAT etc

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, maximum flow, 2-SAT etc
- It is unlikely that polynomial-time algorithms exist for **NP-hard** problems.

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, maximum flow, 2-SAT 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, 3-SAT, etc.

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, maximum flow, 2-SAT 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, 3-SAT, etc.
- We expect that these problems can be solved only in exponential time (i.e., $O(c^n)$).

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, maximum flow, 2-SAT 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, 3-SAT, etc.
- We expect that these problems can be solved only in exponential time (i.e., $O(c^n)$).

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, maximum flow, 2-SAT 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, 3-SAT, 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?

What can you do in front of a hard problem

If a problem is NP-hard, then there is no algorithm that solves

- all instances
- optimally
- in poly-time

What can you do in front of a hard problem

If a problem is NP-hard, then there is no algorithm that solves

- all instances
- optimally
- in poly-time

But why is a problem hard to solve?

It is certainly easy to solve on some easy instances.

But how to capture the notion of easy instances?

Maybe some parameter of the input play an important role, and if this parameter is small we can solve the problem efficiently.

How to cheat in front of a hard problem?

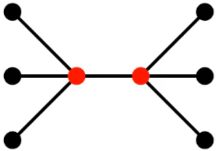
The size of the input is never the only thing that affects the running time of an algorithm.

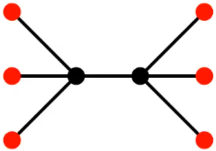
Main idea: measure the complexity in term of the input size and something else.

Formally: Instead of expressing the running time by a function $T(n)$ of the input size n, express it by a function $T(n, k)$ of the input size n and of a parameter k of the input.

Parametrized 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?



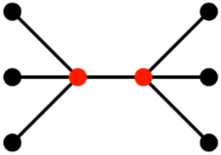


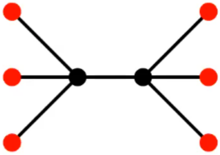
Complexity:	NP-complete	NP-complete
Brute force:	$O(n^k)$ possibilities	$O(n^k)$ possibilities

figure by Daniel Marx

Parametrized 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**

figure by Daniel Marx

Parametrized complexity, definitions

- A **parametrized algorithmic** problem is a problem where a certain parameter k is given in addition to the input (of size n).
- The complexity is studied as a function of n and k.
- k can be the size of the solution, or an implicit parameter of the input graph (diameter, maximum degree, treewidth...).

Parametrized complexity, definitions

- A **parametrized algorithmic** problem is a problem where a certain parameter k is given in addition to the input (of size n).
- The complexity is studied as a function of n and k.
- k can be the size of the solution, or an implicit parameter of the input graph (diameter, maximum degree, treewidth...).

There are roughly three possibilities for the complexity of a parametrized algorithmic problem.

Parametrized complexity, definitions

- A **parametrized algorithmic** problem is a problem where a certain parameter k is given in addition to the input (of size n).
- The complexity is studied as a function of n and k.
- k can be the size of the solution, or an implicit parameter of the input graph (diameter, maximum degree, treewidth...).

There are roughly three possibilities for the complexity of a parametrized algorithmic problem.

- Either the problem is already hard for fixed k.
  Example: decide if $\chi(G) \leq k$ is NP-hard for $k = 3$. (Brute force gives k^n)

Parametrized complexity, definitions

- A **parametrized algorithmic** problem is a problem where a certain parameter k is given in addition to the input (of size n).
- The complexity is studied as a function of n and k.
- k can be the size of the solution, or an implicit parameter of the input graph (diameter, maximum degree, treewidth...).

There are roughly three possibilities for the complexity of a parametrized algorithmic problem.

- Either the problem is already hard for fixed k.
  Example: decide if $\chi(G) \leq k$ is NP-hard for $k = 3$. (Brute force gives k^n)
- Or the problem is NP-hard for k in the input but polynomial for k fixed.
  Example: Decide if $\alpha(G) \leq k$ with parameter k by exhaustive search needs : $O(n^k)$ (we say it is **XP**).

Parametrized complexity, definitions

- A **parametrized algorithmic** problem is a problem where a certain parameter k is given in addition to the input (of size n).
- The complexity is studied as a function of n and k.
- k can be the size of the solution, or an implicit parameter of the input graph (diameter, maximum degree, treewidth...).

There are roughly three possibilities for the complexity of a parametrized algorithmic problem.

- Either the problem is already hard for fixed k.
  Example: decide if $\chi(G) \leq k$ is NP-hard for $k = 3$. (Brute force gives k^n)
- Or the problem is NP-hard for k in the input but polynomial for k fixed.
  Example: Decide if $\alpha(G) \leq k$ with parameter k by exhaustive search needs : $O\left(n^k\right)$ (we say it is **XP**).
- Or it is **Fixed Parameter Tractable** (**FPT**) for k: Algorithm in time $O\left(f(k) \cdot n^{O(1)}\right)$

Formal definition

- We consider only decision problem.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.
- A parametrized problem P is XP if there is an algorithm that, given an input (x, k)

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.
- A parametrized problem P is XP if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.
- A parametrized problem P is XP if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $n^{f(k)}$ for some computable function f and a constant c.

Formal definition

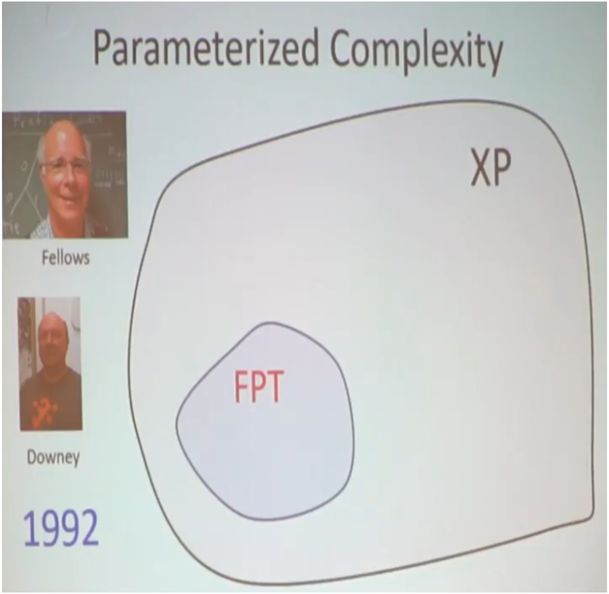
- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.
- A parametrized problem P is XP if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $n^{f(k)}$ for some computable function f and a constant c.

Formal definition

- We consider only decision problem.
- Let Σ be a finite alphabet used to encode the input.
  - $\Sigma = \{0, 1\}$ for binary encoding.
- A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$.
  - $=\{(x_1, k_1), (x_2, k_2), \dots\}$.
  - The set P contains the couples (x, k) for which the answer to the question encoded by (x, k) is YES; k is the parameter.
- A parametrized problem $\mathcal{P}$ is Fixed-Parameter Tractable if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $f(k)n^c$ for some computable function f and a constant c.
- A parametrized problem P is XP if there is an algorithm that, given an input (x, k)
  - Decide if (x, k) belongs to $\mathcal{P}$ or not, and
  - run in time $n^{f(k)}$ for some computable function f and a constant c.

For example, the set of tuples $\{(G, k) \in \mathcal{G} \times \mathbb{N} : vc(G) \leq k\}$ is the problem VERTEX-COVER parametrized by the size of the solution.

Parametrized Complexity



Parametrized Complexity

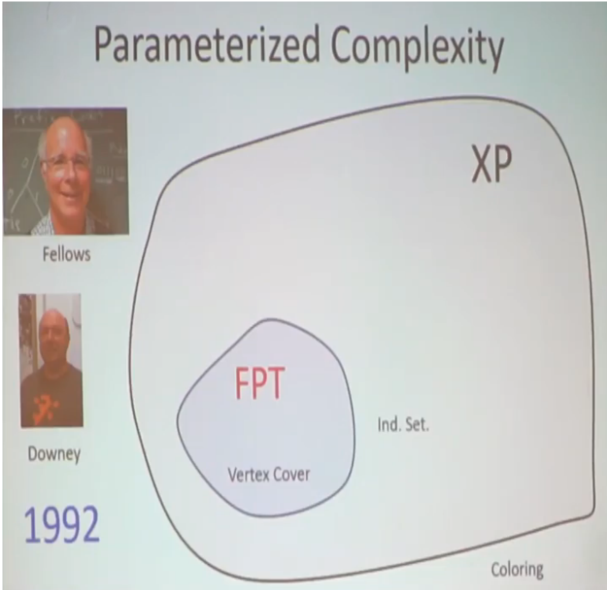
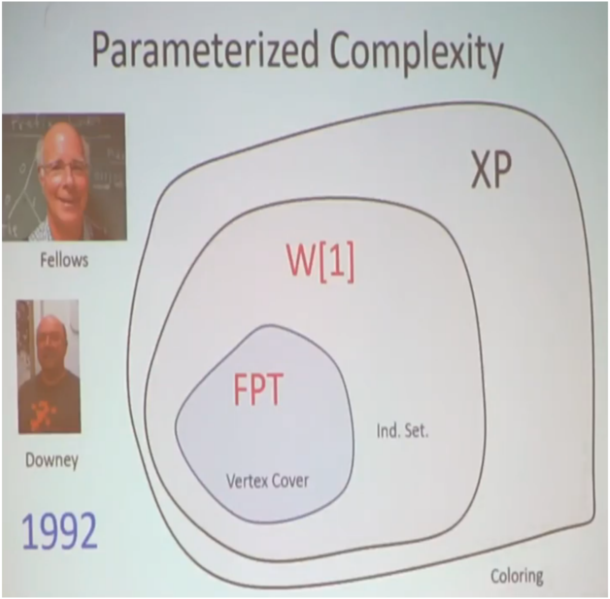
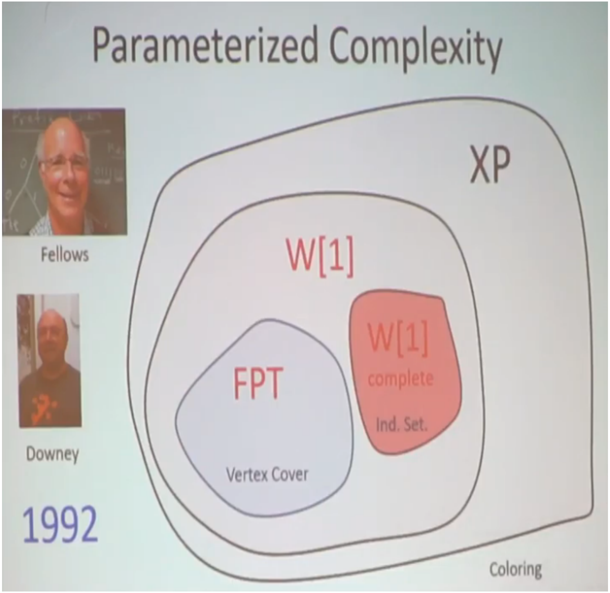


figure by Daniel Lokshtanov

Parametrized Complexity



Parametrized Complexity



$W[1]$-hardness

Negative evidence similar to NP-completeness: if a (parametrized) problem is $W[1]$-hard, then the problem is not FPT unless $FPT = W[1]$.

Some $W[1]$-hard problem:

- Find a clique/stable set of size k.
- Find a dominating set of size k
- Set cover
- ...

$W[1]$-hardness

Negative evidence similar to NP-completeness: if a (parametrized) problem is $W[1]$-hard, then the problem is not FPT unless $FPT = W[1]$.

Some $W[1]$-hard problem:

- Find a clique/stable set of size k.
- Find a dominating set of size k
- Set cover
- ...

General Principal to prove 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

$W[1]$-hardness

Negative evidence similar to NP-completeness: if a (parametrized) problem is $W[1]$-hard, then the problem is not FPT unless $FPT = W[1]$.

Some $W[1]$-hard problem:

- Find a clique/stable set of size k.
- Find a dominating set of size k
- Set cover
- ...

General Principal to prove 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

Exponential Time Hypothesis (ETH):
n-variable 3-SAT cannot be solved in time $2^{o(n)}$.

Clique parametrized by maximum degree

Problem (CLIQUE parametrized by Δ)

Input : A graph G with maximum degree Δ and an integer k
Question : Does G has a clique of size at least k?

Clique parametrized by maximum degree

Problem (CLIQUE parametrized by Δ)

Input : A graph G with maximum degree Δ and an integer k
Question : Does G has a clique of size at least k?

Algorithm: For each vertex v, check for a maximum clique in $N(v)$

Clique parametrized by maximum degree

Problem (CLIQUE parametrized by Δ)

Input : A graph G with maximum degree Δ and an integer k
Question : Does G has a clique of size at least k?

Algorithm: For each vertex v, check for a maximum clique in $N(v)$

Running time: $O(2^{\Delta} n)$, FPT!!

So CLIQUE parametrized by $\Delta(G)$ is FPT.

But CLIQUE parametrized by solution size k is $W[1]$-hard. That is, probably no algorithm in time $f(k) \cdot n^{O(1)}$.

Parametrized Complexity

Rod G. Downey
Michael R. Fellows

Parameterized Complexity

Springer 1999

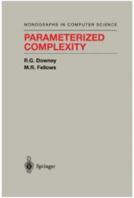
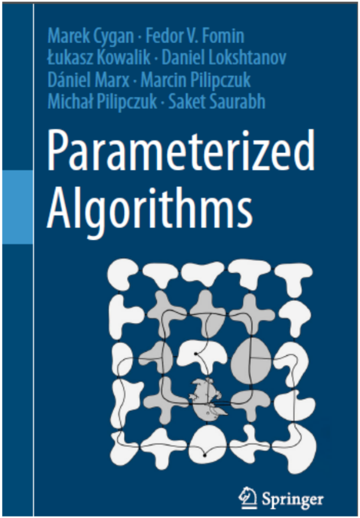


figure by Daniel Marx

- 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.

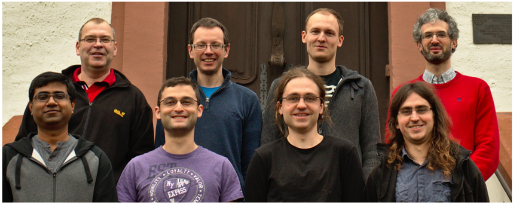
Source for this class



Parameterized Algorithms

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

Springer 2015



Algorihtmic techniques to design FPT algorithm

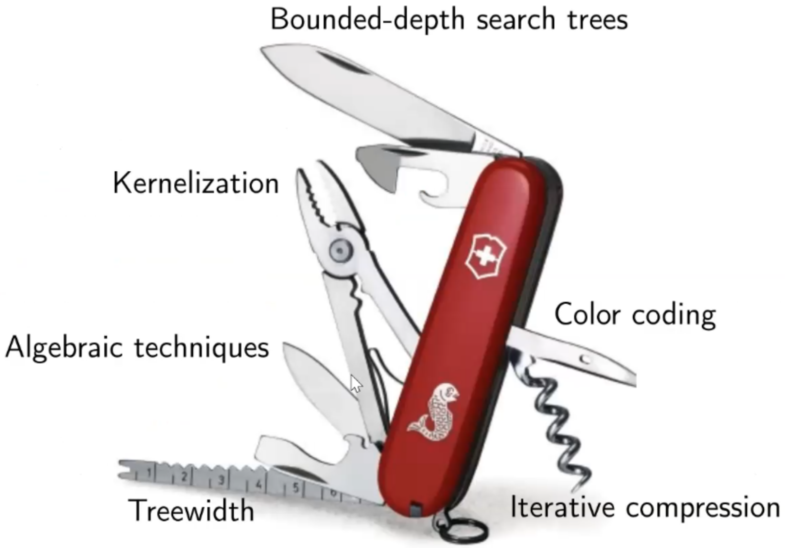


Figure by Daniel Marx

1 - Branching Method

First problem:

Vertex Cover

A **vertex cover** of a graph G is a set S of vertices such that $G \setminus S$ is edgeless.
In other words S hits all edges.

Vertex Cover

A **vertex cover** is a set S of vertices such that $G \setminus S$ is edgeless. In other words S hits all edges.

Problem (VERTEX COVER parametrized by the size of the solution)

Question: Given (G, k), does G have a vertex cover of size at most k?

Brute force: For every set S of k vertices, check if $G \setminus S$ is edgeless.
Running time: $O(n^k \cdot n^2) = O(n^{k+2})$.

So VERTEX COVER parametrized by the size of the solution is in *XP*.

But is it in *FPT*?

Thinking about the problem

Thinking about the problem

- For each edge uv, either u or v is in the solution.

Thinking about the problem

- For each edge uv, either u or v is in the solution.
- So G has a VC of size at most k if and only if $G \setminus \{u\}$ or $G \setminus \{v\}$ has a VC of size at most $k-1$.

Thinking about the problem

- For each edge uv, either u or v is in the solution.
- So G has a VC of size at most k if and only if $G \setminus \{u\}$ or $G \setminus \{v\}$ has a VC of size at most $k-1$.
- In other words, for every edge uv:

  (G, k) is a YES instance if and only if $(G \setminus \{u\}, k-1)$ or $(G \setminus \{v\}, k-1)$ is

Thinking about the problem

- For each edge uv, either u or v is in the solution.
- So G has a VC of size at most k if and only if $G \setminus \{u\}$ or $G \setminus \{v\}$ has a VC of size at most $k-1$.
- In other words, for every edge uv:

  (G, k) is a YES instance if and only if $(G \setminus \{u\}, k-1)$ or $(G \setminus \{v\}, k-1)$ is

- The tree search has depth at most k, so has at most 2^k vertices.

Thinking about the problem

- For each edge uv, either u or v is in the solution.
- So G has a VC of size at most k if and only if $G \setminus \{u\}$ or $G \setminus \{v\}$ has a VC of size at most $k-1$.
- In other words, for every edge uv:

  (G, k) is a YES instance if and only if $(G \setminus \{u\}, k-1)$ or $(G \setminus \{v\}, k-1)$ is

- The tree search has depth at most k, so has at most 2^k vertices.
- (G, k) is a YES-instance if and only if the graph on the leaves are edgeless.

Thinking about the problem

- For each edge uv, either u or v is in the solution.
- So G has a VC of size at most k if and only if $G \setminus \{u\}$ or $G \setminus \{v\}$ has a VC of size at most $k-1$.
- In other words, for every edge uv:

  (G, k) is a YES instance if and only if $(G \setminus \{u\}, k-1)$ or $(G \setminus \{v\}, k-1)$ is

- The tree search has depth at most k, so has at most 2^k vertices.
- (G, k) is a YES-instance if and only if the graph on the leaves are edgeless.
- So the running time: $O(2^k \cdot n^{O(1)})$.

Branching method, size of the search tree and complexity

To solve instance (G, k) of Vertex Cover:

- **Main idea**: reduce the problem to solving a bounded number of problems with paramater $k' < k$.
- We need to be able to solve instance (G, k) in poly-time knowing the solution of the new instances.
- Since the parameter decrease in every recursive call, the depth of the search tree is at most k.
- Size of the seach tree:
  - If we branch into c directions: c^k
  - If we branch into k directions: $k^k = 2^{k \log(k)}$
  - If we branch into $\log(n)$ directions: $n + 2^{k \log(k)}$

Branching method, size of the search tree and complexity

To solve instance (G, k) of VERTEX COVER:

- **Main idea**: reduce the problem to solving a bounded number of problems with paramater $k' < k$.
- We need to be able to solve instance (G, k) in poly-time knowing the solution of the new instances.
- Since the parameter decrease in every recursive call, the depth of the search tree is at most k.
- Size of the seach tree:
  - If we branch into c directions: c^k
  - If we branch into k directions: $k^k = 2^{k \log(k)}$
  - If we branch into $\log(n)$ directions: $n + 2^{k \log(k)}$

We are now going to solve VERTEX COVER in time $1.46^k \cdot n^{O(1)}$!

Notation: $1.46^k \cdot n^{O(1)} = O^*(1.46^k)$

More thinking about the problem

Idea: instead of branching on edges, we are going to branch on vertices of degree at least 3. It is going to work faster because in some of the branches, the parameter is going to decrease faster.

More thinking about the problem

Idea: instead of branching on edges, we are going to branch on vertices of degree at least 3. It is going to work faster because in some of the branches, the parameter is going to decrease faster.

- For each vertex u of degree at least 3:
  - either u is in the solution $\Rightarrow$ parameter decreases by 1
  - or all the neighbors of u are in the solution $\Rightarrow$ parameter decrease by at least 3

More thinking about the problem

Idea: instead of branching on edges, we are going to branch on vertices of degree at least 3. It is going to work faster because in some of the branches, the parameter is going to decrease faster.

- For each vertex u of degree at least 3:
  - either u is in the solution $\Rightarrow$ parameter decreases by 1
  - or all the neighbors of u are in the solution $\Rightarrow$ parameter decrease by at least 3
- If every vertex has degree at most 2, we can solve VERTEX COVER in poly-time because the graph is the disjoint union of paths and cycles. Such graphs will correspond to the leaf node of our searchtree.

More thinking about the problem

Idea: instead of branching on edges, we are going to branch on vertices of degree at least 3. It is going to work faster because in some of the branches, the parameter is going to decrease faster.

- For each vertex u of degree at least 3:
  - either u is in the solution $\Rightarrow$ parameter decreases by 1
  - or all the neighbors of u are in the solution $\Rightarrow$ parameter decrease by at least 3
- If every vertex has degree at most 2, we can solve VERTEX COVER in poly-time because the graph is the disjoint union of paths and cycles. Such graphs will correspond to the leaf node of our searchtree.

(G, k) is a YES instance if and only if $(G \setminus \{u\}, k - 1)$ or $(G \setminus N[u], k - d(u))$ is

Algebraic resolution

Let $T(k)$ be the number of leaves in the search tree, and $T(k) = 0$ if $k \leq 1$. Then:

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

Algebraic resolution

Let $T(k)$ be the number of leaves in the search tree, and $T(k) = 0$ if $k \leq 1$.
Then:

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

Let us prove by induction that $T(k) \leq c^k$ for some constant $c \geq 1$ as small as possible.
What is a good value for c? We are happy if it satisfies:

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

Algebraic resolution

Let $T(k)$ be the number of leaves in the search tree, and $T(k) = 0$ if $k \leq 1$.
Then:

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

Let us prove by induction that $T(k) \leq c^k$ for some constant $c \geq 1$ as small as possible.
What is a good value for c? We are happy if it satisfies:

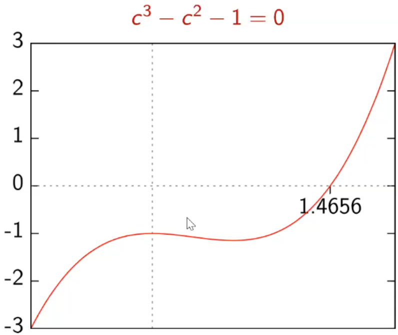
$$c^k \geq c^{k-1} + c^{k-3}$$

and in particular:

$$c^3 - c^2 - 1 \geq 0$$

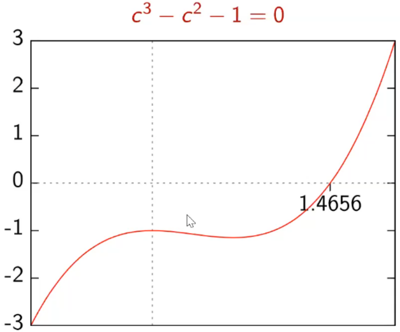
So we want to find the smallest positive root of this equation.
Actually, such equations have a unique postive root.

Solving the equation



$c = 1.4656$ is a good value, so we get $T(k) \leq 1.4656^k$.
And thus we get a $O^*(1.4656^k)$ algorithm for VERTEX COVER

Solving the equation



$c = 1.4656$ is a good value, so we get $T(k) \leq 1.4656^k$.
And thus we get a $O^*(1.4656^k)$ algorithm for VERTEX COVER

Best known FPT algorithm: $O^*(1.2738^k)$, by J. Chen, I. A. Kanj and G. Xia, *Simplicity is beauty: improved upper bounds for Vertex Cover*.

Branching method

The branching vector of our $O^*(1.4656k)$ 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:

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

And thus c satisfies:

$$c^7 - c^5 - c^2 - 2c - 2 \geq 0$$

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

In general, it is hard to compare branching vectors intuitively.

Next problem:

GRAPH MODIFICATION PROBLEM

Definition: Given a graph property $\mathcal{P}$, find a set of vertices S such that $G \setminus S$ satisfies $\mathcal{P}$.

If $\mathcal{P}$ is the property of being edgeless, we recover vertex cover.

Triangle-free deletion problem

Problem (Triangle-free deletion)

Given: a graph G and an integer k,
Question: is there a set of at most k vertices such that $G \setminus S$ is triangle-free?

Triangle-free deletion problem

Problem (Triangle-free deletion)

Given: a graph G and an integer k,
Question: is there a set of at most k vertices such that $G \setminus S$ is triangle-free?

Key idea showing that the branching method is going to work:

If $v_1 v_2 v_3$ is a triangle of G, then:

$$(G, k) \text{ is a YES instance}$$
$$\Leftrightarrow$$
$$(G \setminus \{v_i\}, k-1) \text{ is a YES instance for some } i \in \{1, 2, 3\}$$

Triangle-free deletion problem

Problem (Triangle-free deletion)

Given: a graph G and an integer k,
Question: is there a set of at most k vertices such that $G \setminus S$ is triangle-free?

Key idea showing that the branching method is going to work:

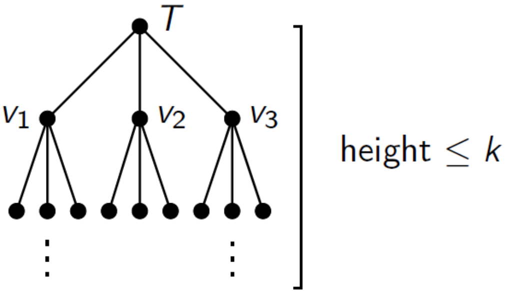
If $v_1 v_2 v_3$ is a triangle of G, then:

$$(G, k) \text{ is a YES instance}$$
$$\Leftrightarrow$$
$$(G \setminus \{v_i\}, k-1) \text{ is a YES instance for some } i \in \{1, 2, 3\}$$

Algo:

- Find a triangle $v_1 v_2 v_3$ (time: $O(n^3)$)
- Solve the instance $(G \setminus v_i, k-1)$ for $i = 1, 2, 3$.

Complexity analysis



The search tree has depth at most k and thus has at most 3^{k+1} vertices.

Find a triangle or check if a graph is triangle-free: n^3,

Running time: $O(3^k \cdot n^3)$.

Graph modification problem

Problem (Graph modification problem)

Given: (G, k)
Question: do at most k allowed operation on G can make G to have property $\mathcal{P}$?

- Allowed operations: vertex deletion, edge deletion, edge contraction, edge addition...
- Property $\mathcal{P}$: edgeless, no triangle, no cycles, disconnected...

Graph modification problem

Problem (Graph modification problem)

Given: (G, k)
Question: do at most k allowed operation on G can make G to have property $\mathcal{P}$?

- Allowed operations: vertex deletion, edge deletion, edge contraction, edge addition...
- Property $\mathcal{P}$: edgeless, no triangle, no cycles, 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 a forest.
- CHORDAL COMPLETION: add k edges to make the graph chordal.

Subgraphs and induce subgraph

1. Remove a vertex v (and all its incident edges), denoted $G \setminus v$.
2. Remove an edge e (but not its end vertices), denoted $G \setminus e$.

Subgraphs and induce subgraph

1. Remove a vertex v (and all its incident edges), denoted $G \setminus v$.
2. Remove an edge e (but not its end vertices), denoted $G \setminus e$.

- H is an induced subgraph of G if H obtained from G by the repeated use of 1.
- H is a subgraph of G if H obtained from G by the repeated use of 1 and 2.

Hereditary property

Definition: a graph property $\mathcal{P}$ is hereditary or closed under taking induced subgraph if whenever $G \in \mathcal{P}$, every induced subgraph H of G are also in $\mathcal{P}$.

small-*Deleting vertices do not ruin the property*-

Examples: edgeless, triangle-free, bipartite, planar...

Hereditary property

Definition: a graph property $\mathcal{P}$ is hereditary or closed under taking induced subgraph if whenever $G \in \mathcal{P}$, every induced subgraph H of G are also in $\mathcal{P}$.

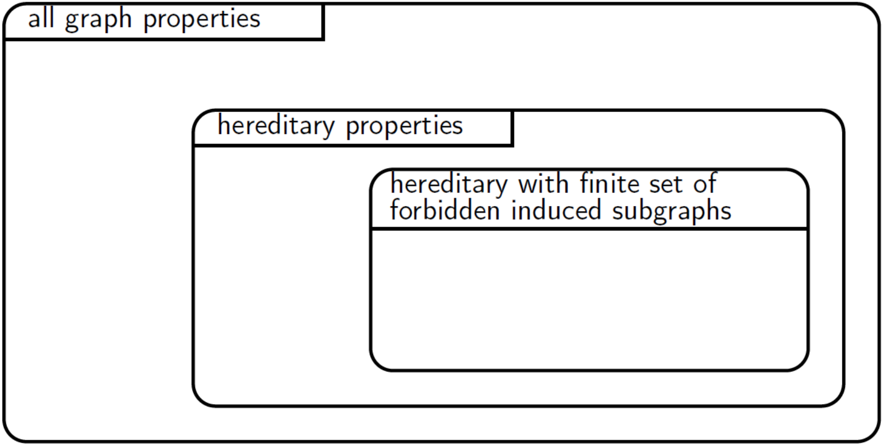
small-*Deleting vertices do not ruin the property*-

Examples: edgeless, triangle-free, bipartite, planar...

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

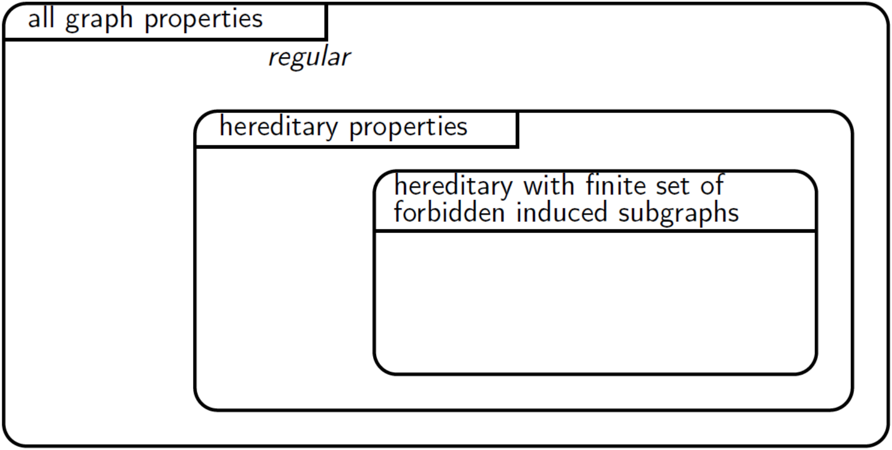
Example: a graph is bipartite if and only if it does not contain odd cycles as induced subgraph.

Graph properties



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

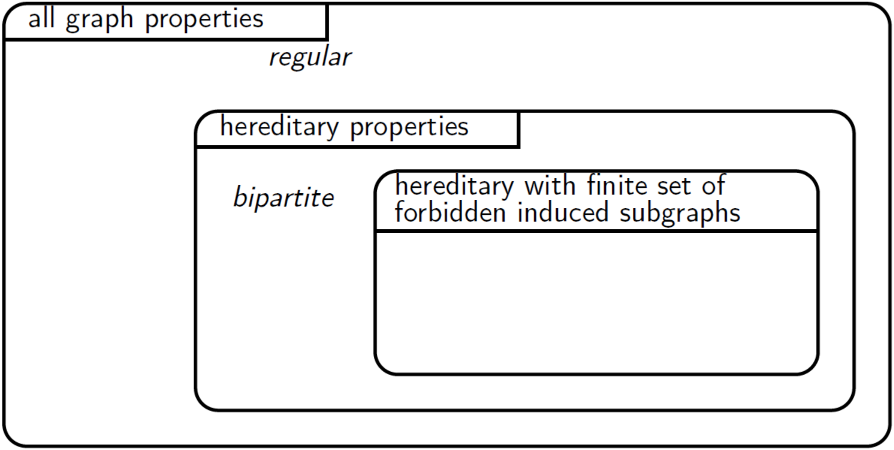
Graph properties



bipartite *triangle free* *connected*

planar *empty* *complete* *acyclic*

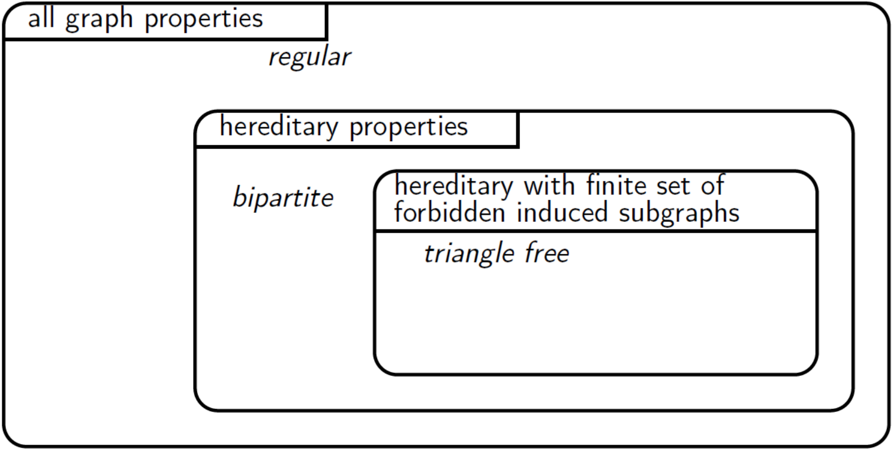
Graph properties



triangle free *connected*

planar *empty* *complete* *acyclic*

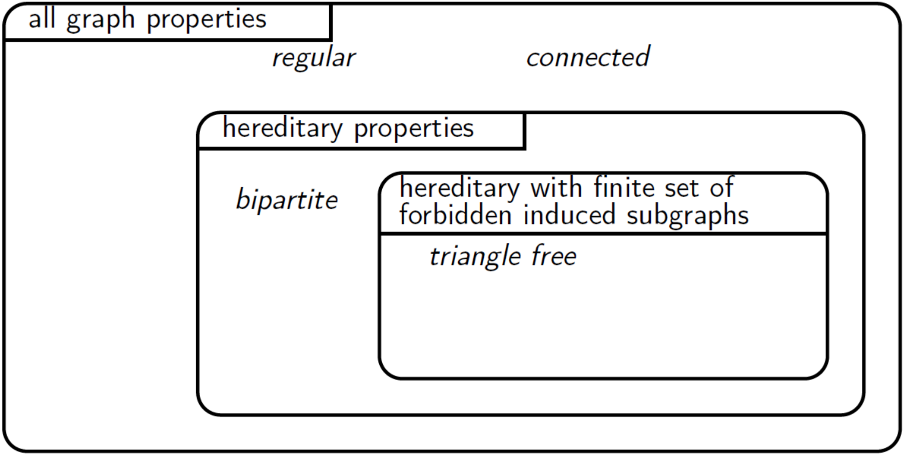
Graph properties



connected

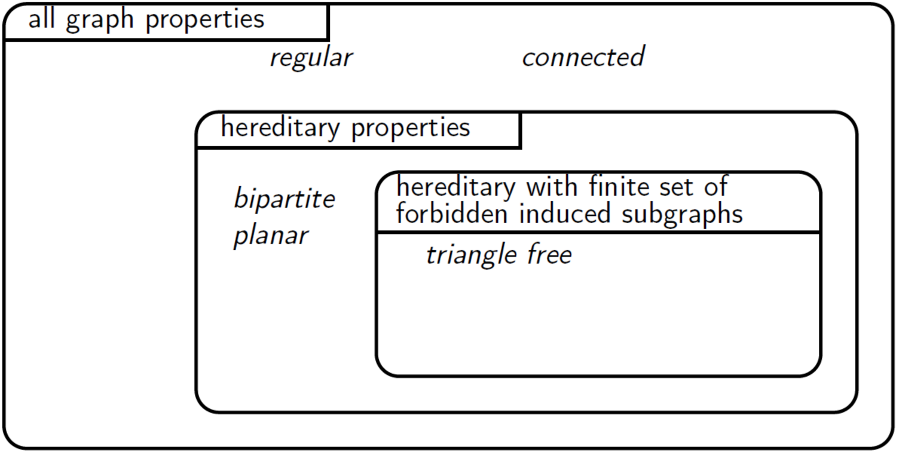
planar *empty* *complete* *acyclic*

Graph properties



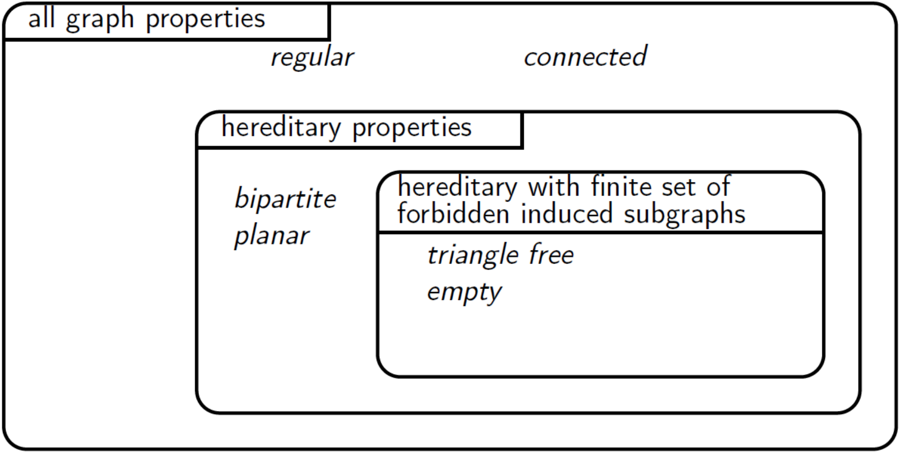
planar *empty* *complete* *acyclic*

Graph properties



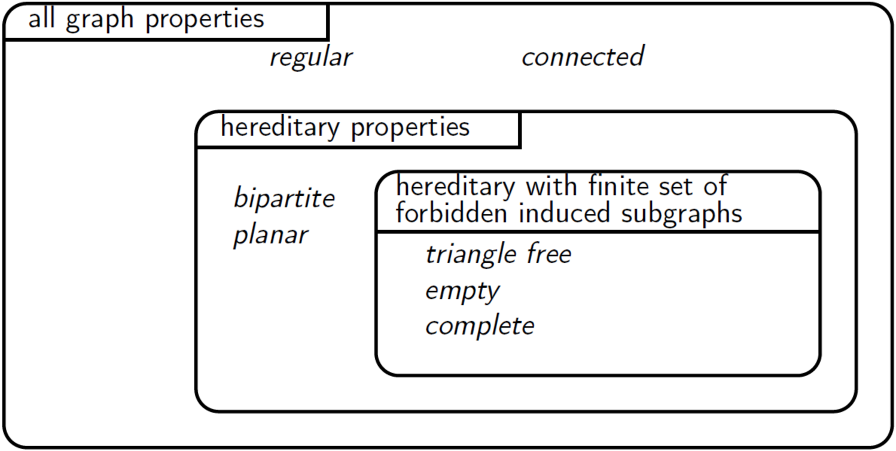
empty *complete* *acyclic*

Graph properties



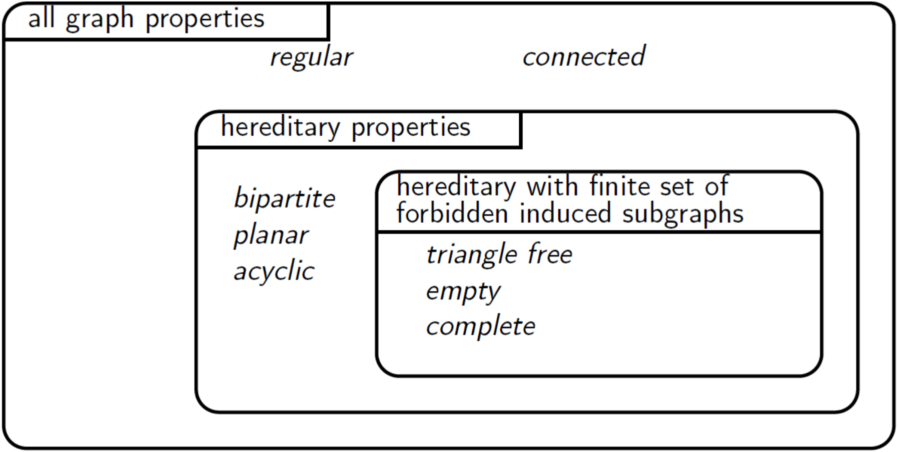
complete *acyclic*

Graph properties

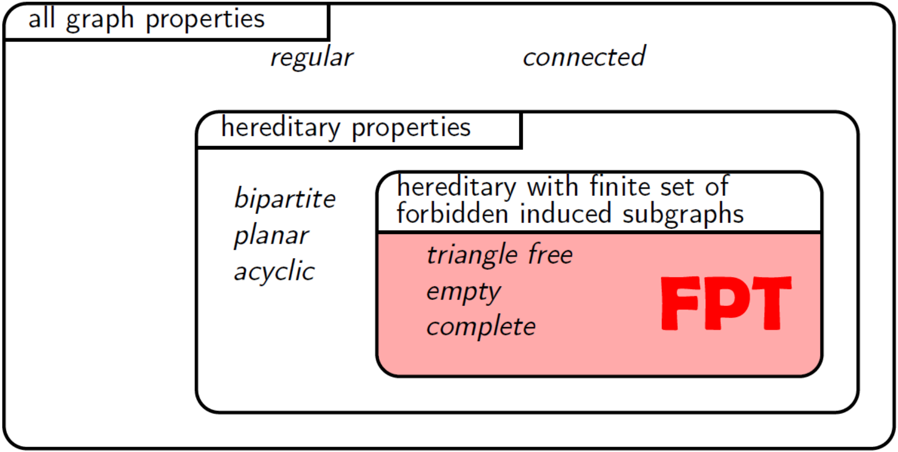


acyclic

Graph properties



Graph properties



Finite set of obstructions

Theorem

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

Finite set of obstructions

Theorem

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

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Observe that r is a constant.

Finite set of obstructions

Theorem

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

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Observe that r is a constant.
- Check if G contains a forbidden graphs. This can be done by brute force in time $|\mathcal{F}| \cdot n^r = O(n^r)$.

Finite set of obstructions

Theorem

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

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Observe that r is a constant.
- Check if G contains a forbidden graphs. This can be done by brute force in time $|\mathcal{F}| \cdot n^r = O(n^r)$.
- If a forbidden subgraph F exists, then we have to delete one of the at most r vertices of the copy of F.

Finite set of obstructions

Theorem

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

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Observe that r is a constant.
- Check if G contains a forbidden graphs. This can be done by brute force in time $|\mathcal{F}| \cdot n^r = O(n^r)$.
- If a forbidden subgraph F exists, then we have to delete one of the at most r vertices of the copy of F.
- The tree has at most r^{k+1} vertices, and the work to be done at each vertex is $O(n^r)$.

Finite set of obstructions

Theorem

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

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Observe that r is a constant.
- Check if G contains a forbidden graphs. This can be done by brute force in time $|\mathcal{F}| \cdot n^r = O(n^r)$.
- If a forbidden subgraph F exists, then we have to delete one of the at most r vertices of the copy of F.
- The tree has at most r^{k+1} vertices, and the work to be done at each vertex is $O(n^r)$.
- Total running time: $O(r^{k+1} \cdot n^r)$.

An active area of research

Graph modification problem is 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).

Next problem:

FEEDBACK VERTEX SET

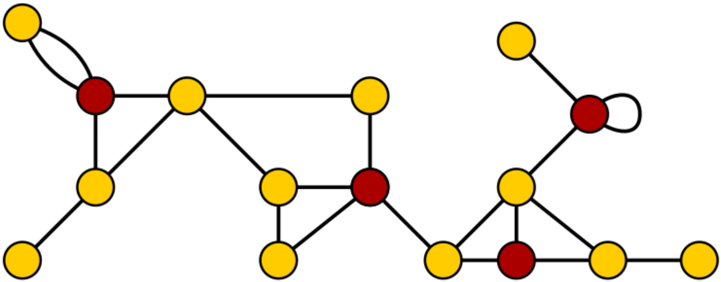
A **Feedback Vertex Set** (**FVS**) of a graph G is a set S of vertices such that $G \setminus S$ is a forest.
In other words S hits all cycles.

Feeback Vertex set

Problem (Feedback Vertex set (FVS))

Question: Given (G, k), find a set S of at most k vertices such that $G \setminus S$ has no cycle (i.e. $G \setminus S$ is a forest).

- We allow **loop**, and **multiple edges** (G is a **multigraph**).
- A Feedback Vertex Set is a set of vertices that hits every cycle of the graph.

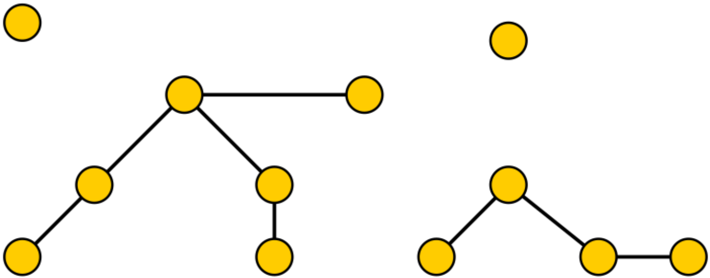


Feeback Vertex set

Problem (Feedback Vertex set (FVS))

Question: Given (G, k), find a set S of at most k vertices such that $G \setminus S$ has no cycle (i.e. $G \setminus S$ is a forest).

- We allow **loop**, and **multiple edges** (G is a **multigraph**).
- A Feedback Vertex Set is a set of vertices that hits every cycle of the graph.



Feeback Vertex set

Problem (Feedback Vertex set (FVS))

Question: Given (G, k), find a set S of at most k vertices such that $G \setminus S$ has no cycle (i.e. $G \setminus S$ is a forest).

- We allow **loop**, and **multiple edges** (G is a **multigraph**).
- A Feedback Vertex Set is a set of vertices that hits every cycle of the graph.

Link with vertex cover: a vertex cover is a set of vertices that hits every edge of the graph.

Thinking about the problem

- In Vertex Cover, at least one extremity of each edge must be in the solution.
- In Feedback Vertex set, at least one vertex of each cycle must be in the solution. But the size of a cycle can be arbitrarily large.

Thinking about the problem

- In Vertex Cover, at least one extremity of each edge must be in the solution.
- In Feedback Vertex set, at least one vertex of each cycle must be in the solution. But the size of a cycle can be arbitrarily large.

- We are going to: identify a set of $O(k)$ vertices such that any size-k feedback vertex set has to contain one of these vertices, and branch on it.

Thinking about the problem

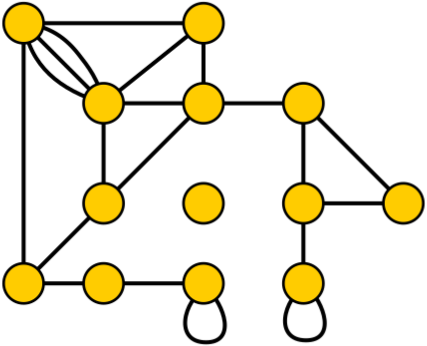
- In Vertex Cover, at least one extremity of each edge must be in the solution.
- In Feedback Vertex set, at least one vertex of each cycle must be in the solution. But the size of a cycle can be arbitrarily large.

- We are going to: identify a set of $O(k)$ vertices such that any size-k feedback vertex set has to contain one of these vertices, and branch on it.
- But first, as often, some reduction rules.

  The reduction rules are here to simplify the input in such a way that the new input is a YES-instance if and only if the orginal one is.

Reduction rules for FVS

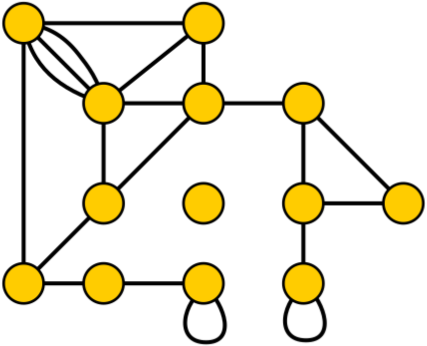
(R1) If there is a loop at v, then delete v and decrease k by one.



Reduction rules for FVS

(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.

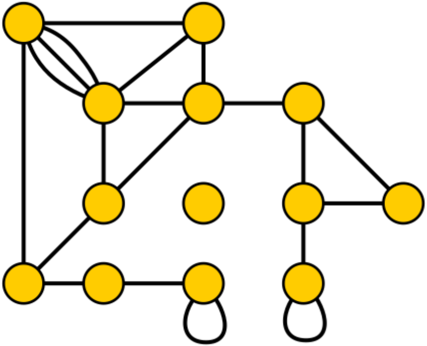


Reduction rules for FVS

(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 0 or 1, then delete v.



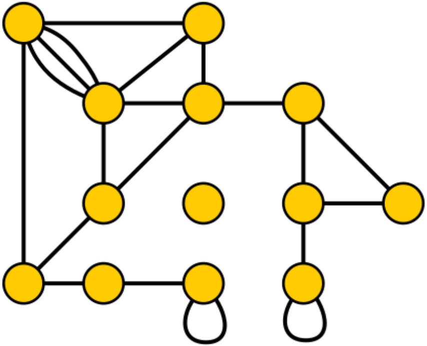
Reduction rules for FVS

(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 0 or 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.



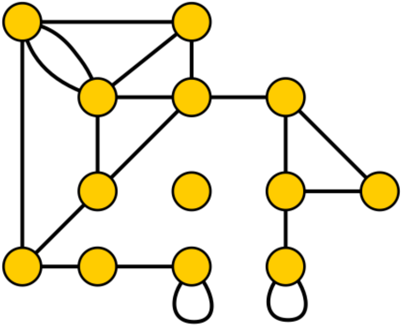
Reduction rules for FVS

(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 0 or 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.



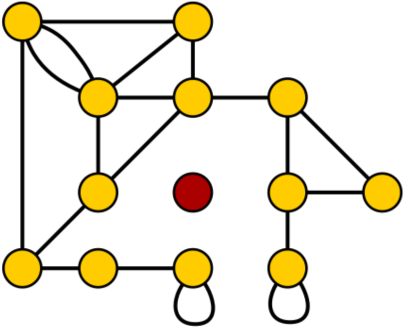
Reduction rules for FVS

(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 0 or 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.



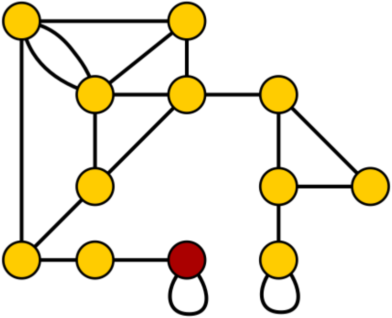
Reduction rules for FVS

(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 0 or 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.



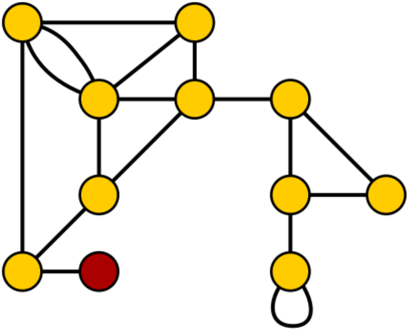
Reduction rules for FVS

(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 0 or 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.



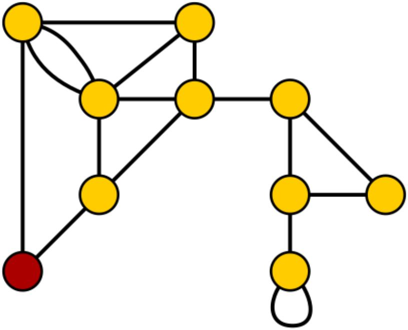
Reduction rules for FVS

(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 0 or 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.



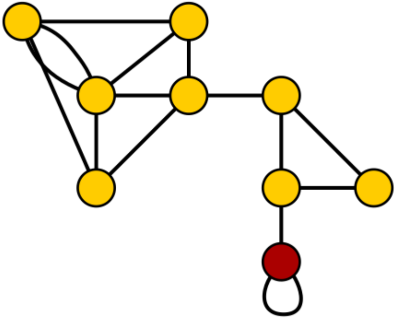
Reduction rules for FVS

(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 0 or 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.



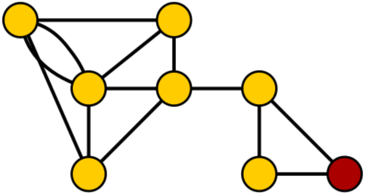
Reduction rules for FVS

(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 0 or 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.



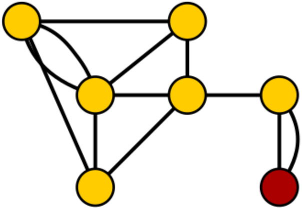
Reduction rules for FVS

(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 0 or 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.



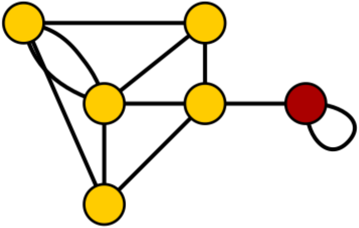
Reduction rules for FVS

(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 0 or 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.



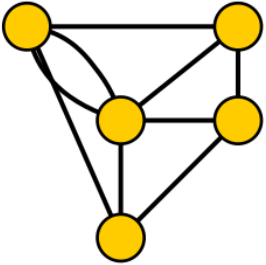
Reduction rules for FVS

(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 0 or 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 for FVS

(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 0 or 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.

After exhaustively applying these reduction rules, the resulting graph G satisfies:

- no loop,
- edge multiplicity is 1 or 2,
- minimum degree 3

Key property of reduction rules

Key Property of the reduction rules:
If (G, k) is an instance of FVS graph and (G', k') is the instance obtained after applying the reduction rules as much as we can, then

- G has a FVS of size at most k if and only if G' has a FVS of size at most k' and
- If S is a FVS of G', then it is a FVS of G together with the vertices deletes by R1. (not necessary if we don't care about the set and just want a YES/NO answer).

In other words, we can safely apply the reduction rules and work on the resulting graph.

Branching

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Branching

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Assuming the Lemma we can easily design our FPT algorithm:

- Apply reduction rules to obtain G' and compute V_{3k}.
- Branch on each vertex $x \in V_{3k}$, that is solve the problems for the k instances: $(G' \setminus \{x\}, k-1)$.
- Branching into $3k$ directions $\Rightarrow$ $O^*((3k)^k)$
- Applying reduction rules and finding the $3k$ largest degree vertices can easily be done in poly-time.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.
- So sum of the degree in $G[X \cup V_{3k}]$ is at most $2|X| + 6k - 2$.

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.
- So sum of the degree in $G[X \cup V_{3k}]$ is at most $2|X| + 6k - 2$.
- And now, the number of edges between S and $X \cup V_{3k}$ is:

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.
- So sum of the degree in $G[X \cup V_{3k}]$ is at most $2|X| + 6k - 2$.
- And now, the number of edges between S and $X \cup V_{3k}$ is:
  - $\geq 3kd + 3|X| - (2|X| + 6k - 2) > 3kd - 6k$

Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.
- So sum of the degree in $G[X \cup V_{3k}]$ is at most $2|X| + 6k - 2$.
- And now, the number of edges between S and $X \cup V_{3k}$ is:
  - $\geq 3kd + 3|X| - (2|X| + 6k - 2) > 3kd - 6k$
  - $\leq dk$ because S has k vertices, each of degree at most d.

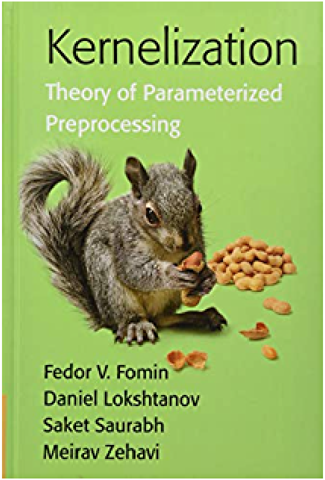
Proof of the Lemma

Lemma: Let G be a graph with minimum degree 3, and let V_{3k} be the $3k$ largest degree vertices. Then every Feedback Vertex set of size at most k contains at least one vertex of V_{3k}.

Proof:

- Let S be a solution disjoint from V_{3k}.
- Let d be the min degree of vertices in V_{3k}.
- Let $X = V(G) - (S \cup V_{3k})$
- $$\sum_{v \in X \cup V_{3k}} d(v) \geq 3|X| + 3kd$$
- $G[X \cup V_{3k}]$ is a forest, so the number of edges in $G[X \cup V_{3k}] \leq |X| + 3k - 1$.
- So sum of the degree in $G[X \cup V_{3k}]$ is at most $2|X| + 6k - 2$.
- And now, the number of edges between S and $X \cup V_{3k}$ is:
  - $\geq 3kd + 3|X| - (2|X| + 6k - 2) > 3kd - 6k$
  - $\leq dk$ because S has k vertices, each of degree at most d.
- So $3kd - 6k < kd \Leftrightarrow 2kd - 6k < 0$ which is false because $d \geq 3$.

2 - Kernelization


Kernelization
Theory of Parameterized Preprocessing
Fedor V. Fomin
Daniel Lokshtanov
Saket Saurabh
Meirav Zehavi

Data reduction

- 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.

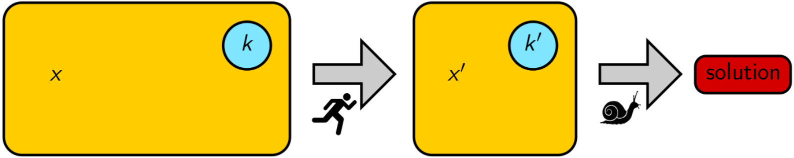


Figure by Daniel Marx

Kernelization: formal definition

- Let $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$ be a parametrized problem and let $f : \mathbb{N} \to \mathbb{N}$ be a computable function.

Kernelization: formal definition

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

Kernelization: formal definition

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

Kernelization: formal definition

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

Kernelization: formal definition

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

Question: which problem has a kernel??

A crazy equivalence

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

A crazy equivalence

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

Proof:

• If the problem has a kernel:
reduce the size of the instance in poly-time and use brute force on it $\Rightarrow$ FPT.

A crazy equivalence

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

Proof:

• If the problem has a kernel:
reduce the size of the instance in poly-time and use brute force on it $\Rightarrow$ FPT.

• If the problem can be solved in time $f(k) \cdot |x|^c$:

- If $|x| \leq f(k)$, then we already have our kernel.
- If $|x| \geq f(k)$, then we can solve the problem in time $f(k) \cdot |x|^c \leq |x|^{c+1}$ (which is polynomial in $|x|$) and then output a trivial YES or NO answer.

A crazy equivalence

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

Proof:

• If the problem has a kernel:
reduce the size of the instance in poly-time and use brute force on it $\Rightarrow$ FPT.

• If the problem can be solved in time $f(k) \cdot |x|^c$:

- If $|x| \leq f(k)$, then we already have our kernel.
- If $|x| \geq f(k)$, then we can solve the problem in time $f(k) \cdot |x|^c \leq |x|^{c+1}$ (which is polynomial in $|x|$) and then output a trivial YES or NO answer.

- So asking if there is a kernel is the same question as asking for an FPT algorithm.
- The important question: is there a polynomial kernel?

Back to vertex cover

Let us prove that Vertex Cover has a polynomial kernel.

A **vertex cover** of a graph G is a set S of vertices such that $G \setminus S$ is edgeless. In other words S hits all edges.

Thinking about the problem

Observe that if a vertex v has degree 0, then:
G has a vertex cover of size k if and only if $G - \{v\}$ has a vertex cover of size k.

Thinking about the problem

Observe that if a vertex v has degree 0, then:
G has a vertex cover of size k if and only if $G - \{v\}$ has a vertex cover of size k.

Observe that if a vertex v has degree $k+1$, then v must be in all vertex cover of size at most k. So:
G has a vertex cover of size k if and only if $G - \{v\}$ has a vertex cover of size $k-1$.

Thinking about the problem

Observe that if a vertex v has degree 0, then:
G has a vertex cover of size k if and only if $G - \{v\}$ has a vertex cover of size k.

Observe that if a vertex v has degree $k+1$, then v must be in all vertex cover of size at most k. So:
G has a vertex cover of size k if and only if $G - \{v\}$ has a vertex cover of size $k-1$.

This leads us to define the two following reduction rules:

(R1) If v has degree 0, then reduce to $(G - v, k)$

(R2) If v has degree at least $k+1$, then reduce to $(G - v, k-1)$.

Now, if (G, k) is an instance of VERTEX COVER and (G', k') is the instance obtained after an exhaustive application of $R1$ and $R2$, then:

(G, k) is a YES-instance if and only if (G', k') is a YES-instance.

Kernel for vertex cover

Reduction rules:

(R1) If v has degree 0, then reduce to $(G - v, k)$

(R2) If v has degree at least $k + 1$, then reduce to $(G - v, k - 1)$.

Kernel for vertex cover

Reduction rules:

(R1) If v has degree 0, then reduce to $(G - v, k)$

(R2) If v has degree at least $k + 1$, then reduce to $(G - v, k - 1)$.

Lemma: If (G, k) is a *YES*-instance for k-vertex cover on which reduction rules 1 and 2 cannot be applied, then G has at most k^2 edges and at most $k^2 + k$ vertices.

Kernel for vertex cover

Reduction rules:

(R1) If v has degree 0, then reduce to $(G - v, k)$

(R2) If v has degree at least $k + 1$, then reduce to $(G - v, k - 1)$.

Lemma: If (G, k) is a *YES*-instance for k-vertex cover on which reduction rules 1 and 2 cannot be applied, then G has at most k^2 edges and at most $k^2 + k$ vertices.

Proof:

- Let S be a vertex cover of G of size at most k.
- Each vertex hits at most k edges because (R2) does not apply. So there is at most k^2 edges.
- Each vertex is either in S, or is one of the k neighbors of a vertex in S. So $|V(G)| \leq k^2 + k$.

Kernel for vertex cover

Reduction rules:

(R1) If v has degree 0, then reduce to $(G - v, k)$

(R2) If v has degree at least $k + 1$, then reduce to $(G - v, k - 1)$.

Lemma: If (G, k) is a *YES*-instance for k-vertex cover on which reduction rules 1 and 2 cannot be applied, then G has at most k^2 edges and at most $k^2 + k$ vertices.

Kernelization for VERTEX COVER:

- Apply rules $(R1)$ and $(R2)$ exhaustively. We get a new instance (G', k') with $k' \leq k$ and such that (G, k) is a YES-instance if and only if (G', k') is.
- If $|E(G')| > k'^2$ or $|V(G)| > k'^2 + k'$, output NO.
- Otherwise we have a kernel of size $O(2k^2 + k)$.

Crown decomposition

Theorem: VERTEX COVER has a kernel with at most $3k$ vertices.

Crown decomposition

A crown decomposition of a graph G is a partitioning of $V(G)$ into three parts C, H and R such that:

1. C is a nonempty independent set;
2. There are no edge between C and R;
3. There is a matching between C and H of size $|H|$.

C is the crown, H the head, and R the rest.

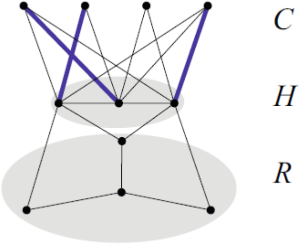


Figure from Parametrized Algorithm by CFKLMPPS

Matching in bipartite graphs

Let G be a bipartite graph with partition (V_1, V_2).

Kőnig's Theorem: The size of a maximum matching of G equal the size of a minimum vertex cover.

Hall's Theorem: G has a matching saturating V_1 if and only if for all $X \subseteq V_1$, $|N(X)| \geq |X|$.

Hopcroft-Karp algorithm: There is a $O(m\sqrt{n})$-time algorithm that finds a maximum matching as well as a minimum vertex cover in G. It furthermore finds a matching saturating V_1, or a inclusion-wise minimal set $X \subseteq V_1$ such that $|N(X)| < |X|$.

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Corollary: Vertex Cover has a kernel with at most $3k$ vertices.

Proof: Consider a Vertex Cover instance (G, k). By an exhaustive application of (R1), we may assume G has no isolated vertex. If $|V(G)| \geq 3k+1$, by the crown lemma applied to (G, k), either G has a $(k+1)$ matching, or a crown decomposition (C, H, R). In a former case, output NO. In the latter case, let M be a matching between H and C of size $|H|$. Observe that the matching M witnesses that, for every vertex cover X of G, X contains at least $|M| = |H|$ vertices of $H \cup C$ to cover the edges of M. On the other hand, H covers all edges of G that are incident to $H \cup C$. Consequently, there exists a minimum vertex cover of G that contains H. Moreover, vertices in C are isolated in $G - H$. Hence, (G, k) is a YES-instance if and only if $(G - (C \cup H), k - |H|)$ is. As $H \neq \emptyset$, we can run the crown algorithm until it outputs a matching of size $k+1$ or until $|V(G)| \leq 3k$. $\square$

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.
- Hence $|X \cap V_M| \neq \emptyset$.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.
- Hence $|X \cap V_M| \neq \emptyset$.
- Let M^* the edges of M' with an extremity in $X \cap V_M$, and $V_{M^*} = V(M^*)$.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.
- Hence $|X \cap V_M| \neq \emptyset$.
- Let M^* the edges of M' with an extremity in $X \cap V_M$, and $V_{M^*} = V(M^*)$.
- Set $H = X \cap V_M = X \cap V_{M^*}, \quad C = V_{M^*} \cap I, \quad R = V(G) \setminus (C \cup H)$.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.
- Hence $|X \cap V_M| \neq \emptyset$.
- Let M^* the edges of M' with an extremity in $X \cap V_M$, and $V_{M^*} = V(M^*)$.
- Set $H = X \cap V_M = X \cap V_{M^*}$, $C = V_{M^*} \cap I$, $R = V(G) \setminus (C \cup H)$.
- Prove that it is a crown decomposition and check that it gives a poly-time algorithm.

Proof of the crown lemma

Crown Algorithm: Let G be a graph with no isolated vertex and with at least $3k+1$ vertices. There is a poly-time algorithm that either:

- find a matching of size $k+1$, or
- find a crown decomposition of G.

Proof:

- Greedily find a maximal matching M.
- If $|M| \geq k+1$, we are done, so $|M| \leq k$.
- Set $V_M = V(M)$ and $I = V(G) \setminus V(M)$. Note that I is an independent set.
- Let G_{I,V_M} the bipartite graph formed by edges between I and V_M.
- Compute a maximum matching M' and a minimum vertex cover X of G_{I,V_M}.
- $|M'| \leq k$ (for otherwise we are done) and by Kőnig's Theorem $|X| = |M'|$.
- If $X \subseteq I$, then $X = I$ (because no isolated vertex) and thus $|V(G)| \leq 3k$ and we are done.
- Hence $|X \cap V_M| \neq \emptyset$.
- Let M^* the edges of M' with an extremity in $X \cap V_M$, and $V_{M^*} = V(M^*)$.
- Set $H = X \cap V_M = X \cap V_{M^*}$, $C = V_{M^*} \cap I$, $R = V(G) \setminus (C \cup H)$.
- Prove that it is a crown decomposition and check that it gives a poly-time algorithm.

Kernels based on linear programming

Theorem: VERTEX COVER has a kernel with at most $2k$ vertices.

Integer Linear Programming

Many combinatorial problems can be expressed in the language of Integer Linear Programming (ILP).

In an ILP instance, we are given a set of integer-valued variables, a set of linear inequalities (called constraints) and a linear cost function.
The goal is to minimize or maximize the value of the cost function respecting the constraints.

$$\begin{array}{llll} \text{Minimise :} & \sum_{j=1}^{n} c_j \cdot x_j & & \\ \text{Subject to:} & \sum_{j=1}^{n} a_{ij} \cdot x_j & \leq b_i & \text{for } 1 \leq i \leq m \\ & x_j & \in \mathbb{Z} & \text{for } 1 \leq j \leq m \end{array}$$

The a_{ij}, b_i and c_i are constants, the x_i are the variables.

Encode VERTEX COVER as an ILP

Introduce a variable $x_v \in \{0,1\}$ for each $v \in V(G)$.
Setting $x_v = 0$ means that x_v is not in the solution, while $x_v = 1$ means it is.

$$
\begin{array}{lrll}
\text{Minimise :} & \sum_{v \in V(G)} x_v & & \\
\text{Subject to:} & x_u + x_v & \geq 1 & \text{for all } uv \in E(G) \\
 & x_v & \in \{0,1\} & \text{for all } v \in V(G)
\end{array}
$$

Encode VERTEX COVER as an ILP

Introduce a variable $x_v \in \{0,1\}$ for each $v \in V(G)$.
Setting $x_v = 0$ means that x_v is not in the solution, while $x_v = 1$ means it is.

$$\begin{array}{lrll} \text{Minimise :} & \sum_{v \in V(G)} x_v & & \\ \text{Subject to:} & x_u + x_v & \geq 1 & \text{for all } uv \in E(G) \\ & x_v & \in \{0,1\} & \text{for all } v \in V(G) \end{array}$$

Beautifull, but how is it helpful? ILP is extremely hard to solve.

Fractional relaxation

Linear Programming is famously known for being solvable in (weakly) poly-time, so let us relax our problem. Call it $LPVC(G)$.

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

$x_v = \frac{1}{3}$ is understood as we take one third of the vertex.
A solution to $LPVC(G)$ is a called a fractional vertex cover of G. Its size if dentoed by $VC_f(G)$.

Fractional relaxation

Linear Programming is famously known for being solvable in (weakly) poly-time, so let us relax our problem. Call it $LPVC(G)$.

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

$x_v = \frac{1}{3}$ is understood as we take one third of the vertex.
A solution to $LPVC(G)$ is a called a fractional vertex cover of G. Its size if dentoed by $VC_f(G)$.

We of course have

$$VC_f(G) \leq VC(G)$$

Fractional relaxation

Linear Programming is famously known for being solvable in (weakly) poly-time, so let us relax our problem. Call it $LPVC(G)$.

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

$x_v = \frac{1}{3}$ is understood as we take one third of the vertex.
A solution to $LPVC(G)$ is a called a fractional vertex cover of G. Its size if dentoed by $VC_f(G)$.

We of course have

$$VC_f(G) \leq VC(G)$$

and for example, if G is a triangle, $VC_f(G) = \frac{3}{2} < 2 = VC(G)$.

Let $(x_v)_{v \in V(G)}$ be a minimum fractional vertex cover, i.e. an optimal solution to:

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

Partition the vertices with respect to their value as follows:

- $V_0 = \{v : x_v < \frac{1}{2}\}$
- $V_{\frac{1}{2}} = \{v : x_v = \frac{1}{2}\}$
- $V_1 = \{v : x_v > \frac{1}{2}\}$

Let $(x_v)_{v \in V(G)}$ be a minimum fractional vertex cover, i.e. an optimal solution to:

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

Partition the vertices with respect to their value as follows:

- $V_0 = \{v : x_v < \frac{1}{2}\}$
- $V_{\frac{1}{2}} = \{v : x_v = \frac{1}{2}\}$
- $V_1 = \{v : x_v > \frac{1}{2}\}$

Key Observations:

- V_0 is an independent set, and
- there is no edge between V_0 and $V_{\frac{1}{2}}$.

Let $(x_v)_{v \in V(G)}$ be a minimum fractional vertex cover, i.e. an optimal solution to:

$$\begin{array}{lll} \text{Minimise :} & \sum_{v \in V(G)} x_v & \\ \text{Subject to:} & x_u + x_v \geq 1 & \text{for all } uv \in E(G) \\ & 0 \leq x_v \leq 1 & \text{for all } v \in V(G) \end{array}$$

Partition the vertices with respect to their value as follows:

- $V_0 = \{v : x_v < \frac{1}{2}\}$
- $V_{\frac{1}{2}} = \{v : x_v = \frac{1}{2}\}$
- $V_1 = \{v : x_v > \frac{1}{2}\}$

Key Observations:

- V_0 is an independent set, and
- there is no edge between V_0 and $V_{\frac{1}{2}}$.

Theorem (Nemhauser-Trotter, 1975)
There is a minimum vertex cover S of G such that: $V_1 \subseteq S \subseteq V_{\frac{1}{2}} \cup V_1$

Theorem (Nemhauser-Trotter, 1975)
There is a minimum vertex cover S of G such that: $V_1 \subseteq S \subseteq V_{\frac{1}{2}} \cup V_1$

Proof:

- Let S^* be a minimum vertex cover of G.
- Set $S = V_1 \cup (V_{\frac{1}{2}} \cap S^*)$, and observe that $V_1 \subseteq S \subseteq V_{\frac{1}{2}} \cup V_1$.
- Since there is no edge between V_0 and $V_{\frac{1}{2}}$, S is a VC of G.
- It remains to prove that S is a minimal VC. Assume $|S| > |S^*|$.
- So

$$|V_0 \cap S^*| < |V_1 \setminus S^*| \tag{1}$$

- Set $\varepsilon = \min(|x_v - \frac{1}{2}| : v \in V_0 \cup V_1)$ and define:

$$y_v = \begin{cases} x_v - \varepsilon & \text{if } v \in V_1 \setminus S^* \\ x_v + \varepsilon & \text{if } v \in V_0 \cap S^* \\ x_v & \text{otherwise} \end{cases}$$

- It is easy to check that $(y_v)_{v \in V(G)}$ is a fractional vertex cover.
- But by (1), $\sum_{v \in V(G)} y_v < \sum_{v \in V(G)} x_v$, a contradiction.

Nemhauser-Trotter's theorem allows the following reduction rule:

(R3) Given an minimum fractional vertex cover $(x_v)_{v \in V(G)}$ and the partition $(V_0, V_{\frac{1}{2}}, V_1)$:

- if $\sum_{v \in V(G)} x_v > k$, output NO.
- Otherwise, solve $(G[V_{\frac{1}{2}}], k - |V_1|)$.

Nemhauser-Trotter's theorem allows the following reduction rule:

(R3) Given an minimum fractional vertex cover $(x_v)_{v \in V(G)}$ and the partition $(V_0, V_{\frac{1}{2}}, V_1)$:

- if $\sum_{v \in V(G)} x_v > k$, output NO.
- Otherwise, solve $(G[V_{\frac{1}{2}}], k - |V_1|)$.

This is a safe rule in the sense that:

- if $\sum_{v \in V(G)} x_v > k$, then (G, k) is indeed a NO-instance.
- $(G[V_{\frac{1}{2}}], k - |V_1|)$ is a YES-instance if and only (G, k) is.

Moreover, if (G, k) is a YES-instance, then
$|V_{\frac{1}{2}}| = \sum_{v \in V_{\frac{1}{2}}} 2x_v \leq 2 \sum_{v \in V(G)} x_v \leq 2k$.

Theorem: Vertex Cover has a kernel with at most $2k$ vertices.

Lemma: An minimum fractional vertex cover with each weight in $\{0, \frac{1}{2}, 1\}$ can be found in time $O(m\sqrt(n)$

Proof: We reduce fractional vertex cover to $\textsc{Vertex Cover}$ in the following bipartite graph H: take two copies V_1 and V_2 of $V(G)$ (if $u \in V(G)$, there is a copy u_1 of u in V_1 and a copy u_2 of u in V_2.) and if $uv \in E(G)$, then $u_1v_2, v_1u_2 \in E(G)$.

Lemma: An minimum fractional vertex cover with each weight in $\{0, \frac{1}{2}, 1\}$ can be found in time $O(m\sqrt(n)$

Proof: We reduce fractional vertex cover to $\textsc{Vertex Cover}$ in the following bipartite graph H: take two copies V_1 and V_2 of $V(G)$ (if $u \in V(G)$, there is a copy u_1 of u in V_1 and a copy u_2 of u in V_2.) and if $uv \in E(G)$, then $u_1v_2, v_1u_2 \in E(G)$.
Find a minimum vertex cover S in H with the Hopcroft-Karp algorithm: $O(m\sqrt(n)$.

Lemma: An minimum fractional vertex cover with each weight in $\{0, \frac{1}{2}, 1\}$ can be found in time $O(m\sqrt(n)$

Proof: We reduce fractional vertex cover to $\textsc{Vertex Cover}$ in the following bipartite graph H: take two copies V_1 and V_2 of $V(G)$ (if $u \in V(G)$, there is a copy u_1 of u in V_1 and a copy u_2 of u in V_2.) and if $uv \in E(G)$, then $u_1v_2, v_1u_2 \in E(G)$.

Find a minimum vertex cover S in H with the Hopcroft-Karp algorithm: $O(m\sqrt(n)$.

Define a vector $(x_v)_{v \in V(G)}$ as follows:

- if both v_1 and v_2 are in S, set $x_v = 1$,
- if exactly one of v_1 and v_2 are in S, set $x_v = \frac{1}{2}$,
- $x_v = 0$ otherwise.

Lemma: An minimum fractional vertex cover with each weight in $\{0, \frac{1}{2}, 1\}$ can be found in time $O(m\sqrt(n)$

Proof: We reduce fractional vertex cover to $\textsc{Vertex Cover}$ in the following bipartite graph H: take two copies V_1 and V_2 of $V(G)$ (if $u \in V(G)$, there is a copy u_1 of u in V_1 and a copy u_2 of u in V_2.) and if $uv \in E(G)$, then $u_1v_2, v_1u_2 \in E(G)$.
Find a minimum vertex cover S in H with the Hopcroft-Karp algorithm: $O(m\sqrt(n))$.
Define a vector $(x_v)_{v \in V(G)}$ as follows:

- if both v_1 and v_2 are in S, set $x_v = 1$,
- if exactly one of v_1 and v_2 are in S, set $x_v = \frac{1}{2}$,
- $x_v = 0$ otherwise.

We have: $\sum_{v \in V(G)} x_v = \frac{|S|}{2}$.

Lemma: An minimum fractional vertex cover with each weight in $\{0, \frac{1}{2}, 1\}$ can be found in time $O(m\sqrt(n)$

Proof: We reduce fractional vertex cover to VERTEX COVER in the following bipartite graph H: take two copies V_1 and V_2 of $V(G)$ (if $u \in V(G)$, there is a copy u_1 of u in V_1 and a copy u_2 of u in V_2.) and if $uv \in E(G)$, then $u_1v_2, v_1u_2 \in E(G)$.
Find a minimum vertex cover S in H with the Hopcroft-Karp algorithm: $O(m\sqrt(n)$.
Define a vector $(x_v)_{v \in V(G)}$ as follows:

- if both v_1 and v_2 are in S, set $x_v = 1$,
- if exactly one of v_1 and v_2 are in S, set $x_v = \frac{1}{2}$,
- $x_v = 0$ otherwise.

We have: $\sum_{v \in V(G)} x_v = \frac{|S|}{2}$.
Since S is a vertex cover of H, at least two of the vertices u_1, v_1, u_2, v_2 are in S, and thus, for every edge uv, $x_u + x_v \geq 1$. So $(x_v)_{v \in V(G)}$ is a fractional vertex cover G. Let us prove it is minimum.

Let $(y_v)_{v \in V(G)}$ be a minimum fractional vertex cover G.
We define a weight on $V(H)$ as follows:
For every $v \in V(G)$, $w(v_1) = w(v_2) = y_v$.
This weight assignment is a fractionnal vertex cover of H, i.e., for every edge $u_1 v_2$ of H, we have $w(u_1) + w(v_2) \geq 1$. Hence, $\sum_{v \in V(H)} w(v)$ is at least the size of a maximum matching M of H.
Now, by Kőnig Theorem, $|M| = |S|$, so:

$$\sum_{v \in V(G)} y_v = \frac{1}{2} \sum_{v \in V(G)} (w(v_1) + w(v_2)) = \frac{1}{2} \sum_{v \in V(H)} w(v) \geq \frac{|S|}{2} = \sum_{v \in V(G)} x_v$$

The sunflower Lemma

Theorem: d-HITTING SET has a kernel with at most $d!k^d$ hyperedges and $d!k^d d^2$ vertices.

The d-HITTING SET PROBLEM

Let V be a finite set. A set system $\mathcal{F}$ on V is a collection of subsets of X. We call $\mathcal{F}$ a d-set system if each set has size at most d. A hitting set of $\mathcal{F}$ is a set of vertices that intersects (hits) every set of $\mathcal{F}$.

Problem (d-HITTING SET PROBLEM)

Given: a d-set system $\mathcal{F}$ and a an integer k.
Question: does $\mathcal{F}$ admits a hitting set of size at most k?.

Note that when $d = 2$, it is vertex cover!

Sunflower

A collection of sets $S_1, \ldots S_k$ is a k-sunflower if

$$S_i \cap S_j = S_1 \cap S_2 \cap \ldots S_k \quad \forall\, i \neq j$$

The set $K = S_1 \cap S_2 \cap \ldots S_k$ is the core of the sun flower and the sets $S_i \setminus K$ are its petals.

Note that a set of k pairwise disjoint sets is a sunflower with k petals and an empty core.

The Sunflower Lemma, or Erdős-Rado Lemma

Lemma [The Sunflower Lemma, or Erdős-Rado Lemma, 1960]
Let $\mathcal{F}$ be a d-set system on a set V. If $|\mathcal{F}| > d!(k-1)^d$, then $\mathcal{F}$ has a sunflower with k petals.
Moreover, it can be found in time polynomial in $|V| + |\mathcal{F}| + k$.

The Sunflower Lemma, or Erdős-Rado Lemma

Lemma [The Sunflower Lemma, or Erdős-Rado Lemma, 1960]
Let $\mathcal{F}$ be a d-set system on a set V. If $|\mathcal{F}| > d!(k-1)^d$, then $\mathcal{F}$ has a sunflower with k petals.
Moreover, it can be found in time polynomial in $|V| + |\mathcal{F}| + k$.

Proof: We proceed by induction on d. For $d = 1$ it is trivial. Assume $d \geq 2$. Let $\mathcal{M} = \{S_1, \dots, S_\ell\}$ be a maximal collection of pairwise disjoint sets of $\mathcal{F}$. If $\ell \geq k$ we are done, we may assume $k < \ell$. Set $S = S_1 \cup \dots \cup S_\ell$ and observe $|S| \leq d(k-1)$. Moreover, every set of $\mathcal{F}$ intersects S. Hence, there is $u \in S$ that belongs to at least

$$\frac{d!(k-1)^d}{d(k-1)} = (d-1)!(k-1)^{d-1}$$

sets of $\mathcal{F}$. Construct a $(d-1)$-set system by taking all these sets and removing u from each of them. By induction it has a k-sunflower and thus, puting u back in, we get a k-sunflower in $\mathcal{F}$.

The Sunflower Conjecture

Sunflower Conjecture (Erdős-Rado, 1960)
Let $k \geq 3$. There exists $c = c(k)$ such that every d-set system $\mathcal{F}$ with $|\mathcal{F}| \geq c^d$ contains a d-sunflower.

Theorem (Alweiss, Lovett, Wu and Zhang, 2021):
Let $k \geq 3$. There exists c such that every d-set system $\mathcal{F}$ with $|\mathcal{F}| \geq (ck^3 \log d \log\log d)^d$ contains a k-sunflower.

Trendy topic:
Blog of Terry Tao
Polymath10

Kernel for d-HITTING SET

Problem (d-HITTING SET PROBLEM)

Given: a d-set system $\mathcal{F}$ and an integer k.
Question: does $\mathcal{F}$ admits a hitting set of size at most k?.

Kernel for d-HITTING SET

Problem (d-HITTING SET PROBLEM)

Given: a d-set system $\mathcal{F}$ and an integer k.
Question: does $\mathcal{F}$ admits a hitting set of size at most k?.

Theorem: d-HITTING SET has a kernel with at most $d!k^d$ sets and $d!k^d d^2$ vertices.

Kernel for d-HITTING SET

Problem (d-HITTING SET PROBLEM)

Given: a d-set system $\mathcal{F}$ and an integer k.
Question: does $\mathcal{F}$ admits a hitting set of size at most k?.

Theorem: d-HITTING SET has a kernel with at most $d!k^d$ sets and $d!k^d d^2$ vertices.

Crucial Observation: If $\mathcal{F}$ has a $(k+1)$-sunflower with core K, then every hitting set of $\mathcal{F}$ intersects K.

Reduction rule: Given an instance $(V, \mathcal{F}, k)$, if $\mathcal{F}$ has a $(k+1)$-sunflower $S = \{S_1, \dots, S_{k+1}\}$ with core K, return $(V', \mathcal{F}', k)$ where:

- $\mathcal{F}' = (\mathcal{F} \setminus S) \cup K$ and
- $V' = \cup_{F \in \mathcal{F}'} F$

3 - Color coding

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k! 2^k \cdot n^{0(1)}$, using treewidth.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k!2^k \cdot n^{0(1)}$, using treewidth.
- Alon, Yuster, Zwik, 1994: $((2e)^k)m$ using color coding.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k!2^k \cdot n^{0(1)}$, using treewidth.
- Alon, Yuster, Zwik, 1994: $((2e)^k)m$ using color coding.
- Kneiss, Molle, Richter, Rossmanith, 2006: $4^k \cdot n^{O(1)}$ using color coding.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k!2^k \cdot n^{0(1)}$, using treewidth.
- Alon, Yuster, Zwik, 1994: $((2e)^k)m$ using color coding.
- Kneiss, Molle, Richter, Rossmanith, 2006: $4^k \cdot n^{O(1)}$ using color coding.
- Koutis 2008: $2^{3k/2} \cdot n^{O(1)}$, algebraic method.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k!2^k \cdot n^{0(1)}$, using treewidth.
- Alon, Yuster, Zwik, 1994: $((2e)^k)m$ using color coding.
- Kneiss, Molle, Richter, Rossmanith, 2006: $4^k \cdot n^{O(1)}$ using color coding.
- Koutis 2008: $2^{3k/2} \cdot n^{O(1)}$, algebraic method.
- Williams 2009: $2^k \cdot n^{0(1)}$, algebraic method.

k-PATH PROBLEM

Problem (k-PATH)

Given (G, k), decide if G contains a (simple) path on k vertices as a subgraph.

A long history:

- This problem is NP-complete (it is hamiltonian path for $k = n$).
- No trivial FPT algorithm exists.
- Monien 1985: $k! \cdot n^{O(1)}$ using representative set.
- Bodlaender 1989: $k!2^k \cdot n^{0(1)}$, using treewidth.
- Alon, Yuster, Zwik, 1994: $((2e)^k)m$ using color coding.
- Kneiss, Molle, Richter, Rossmanith, 2006: $4^k \cdot n^{O(1)}$ using color coding.
- Koutis 2008: $2^{3k/2} \cdot n^{O(1)}$, algebraic method.
- Williams 2009: $2^k \cdot n^{0(1)}$, algebraic method.
- Bjorklund, Husfeldt, Kaski, Koivisto 2010: $1.66^k n^{O(1)}$

Randomized algorithm

- A randomized algorithm is an algorithm that employes randomness.

Randomized algorithm

- A randomized algorithm is an algorithm that employes randomness.

- IRL, a guaranteed error probability of 10^{-100} is as good as a deterministic algorithm (probability of hardware failure is larger!)

Randomized algorithm

- A randomized algorithm is an algorithm that employes randomness.

- IRL, a guaranteed error probability of 10^{-100} is as good as a deterministic algorithm (probability of hardware failure is larger!)

- Randomized algorithm can be more efficient and/or conceptually simpler.

Randomized algorithm

- A randomized algorithm is an algorithm that employes randomness.
- IRL, a guaranteed error probability of 10^{-100} is as good as a deterministic algorithm (probability of hardware failure is larger!)
- Randomized algorithm can be more efficient and/or conceptually simpler.
- It can be the first step towards a deterministic algorithm

Randomized algorithm

- A randomized algorithm is an algorithm that employes randomness.

- IRL, a guaranteed error probability of 10^{-100} is as good as a deterministic algorithm (probability of hardware failure is larger!)

- Randomized algorithm can be more efficient and/or conceptually simpler.

- It can be the first step towards a deterministic algorithm
  - Standard derandomization techniques exist.

Monte-carlo algorithm

A typical situation in randomized algorithm is the so-called Monte-Carlo algorithm with one-sided error:

- NO instance: always output NO.
- YES instance: output YES with probability p (and NO with probability $1 - p$).
- The time complexity is deterministic, and depends on p.

Monte-carlo algorithm

A typical situation in randomized algorithm is the so-called Monte-Carlo algorithm with one-sided error:

- NO instance: always output NO.
- YES instance: output YES with probability p (and NO with probability $1 - p$).
- The time complexity is deterministic, and depends on p.

Question: Are we happy with a probability $p = \frac{1}{10}$?

Monte-carlo algorithm

A typical situation in randomized algorithm is the so-called Monte-Carlo algorithm with one-sided error:

- NO instance: always output NO.
- YES instance: output YES with probability p (and NO with probability $1 - p$).
- The time complexity is deterministic, and depends on p.

Question: Are we happy with a probability $p = \frac{1}{10}$?

Answer: Yes! because of Probability Amplification:

Repeat the algorithm 100 times and output YES if there was at least one YES. Then:

$$Pr[error] \leq \frac{9}{10^{100}}$$

Monte-carlo algorithm

A typical situation in randomized algorithm is the so-called Monte-Carlo algorithm with one-sided error:

- NO instance: always output NO.
- YES instance: output YES with probability p (and NO with probability $1 - p$).
- The time complexity is deterministic, and depends on p.

Question: Are we happy with a probability $p = \frac{1}{10}$?

Answer: Yes! because of Probability Amplification:

Repeat the algorithm 100 times and output YES if there was at least one YES.
Then:

$$Pr[error] \leq \frac{9}{10^{100}}$$

Morality: any constant probability is ok.

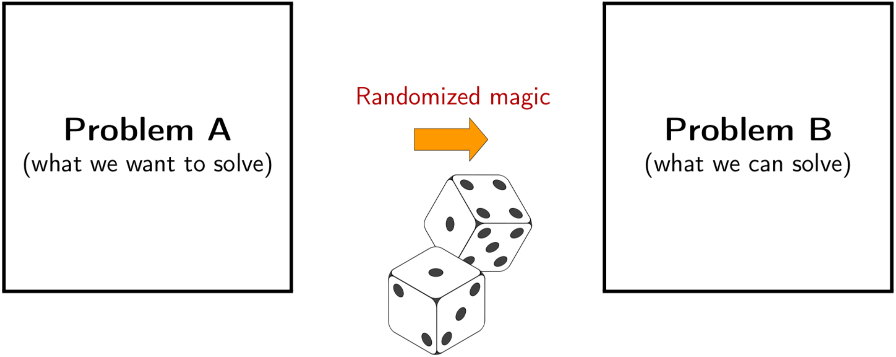
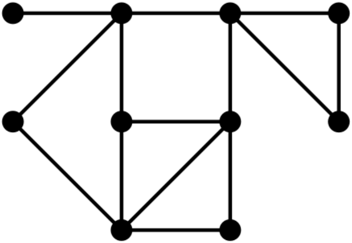


Figure by Daniel Marx

Color coding

Surprising idea: transform the problem into the following:

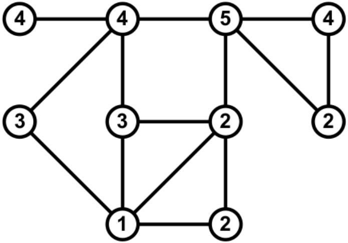
- Assume the vertices are colored randomly with $\{1, \ldots, k\}$
- **Problem**: find a path colored $1 - 2 - \cdots - k$.



Color coding

Surprising idea: transform the problem into the following:

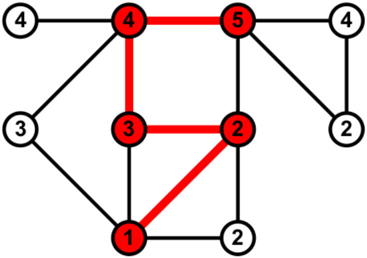
- Assume the vertices are colored randomly with $\{1, \ldots, k\}$
- **Problem**: find a path colored $1 - 2 - \cdots - k$.



Color coding

Surprising idea: transform the problem into the following:

- Assume the vertices are colored randomly with $\{1, \ldots, k\}$
- **Problem**: find a path colored $1 - 2 - \cdots - k$.



Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.

Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.
- Output YES if there is a path colored $1 - 2 - \cdots - k$, and NO otherwise.

Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.
- Output YES if there is a path colored $1 - 2 - \cdots - k$, and NO otherwise.
- If G has a k-path, the probability that this k-path is colored 1-2-…-k is $1/k^k$.

Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.
- Output YES if there is a path colored $1 - 2 - \cdots - k$, and NO otherwise.
- If G has a k-path, the probability that this k-path is colored 1-2-$\ldots$-k is $1/k^k$.
- So if G is a YES instance, the algo output YES with probability at least $1/k^k$

Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.
- Output YES if there is a path colored $1 - 2 - \cdots - k$, and NO otherwise.
- If G has a k-path, the probability that this k-path is colored 1-2-$\ldots$-k is $1/k^k$.
- So if G is a YES instance, the algo output YES with probability at least $1/k^k$
- And if it is a NO instance, the algorithm output NO.

Color coding

- Assign color from $[k]$ to the vertices of G uniformly and independently at random.
- Output YES if there is a path colored $1 - 2 - \cdots - k$, and NO otherwise.
- If G has a k-path, the probability that this k-path is colored 1-2-...-k is $1/k^k$.
- So if G is a YES instance, the algo output YES with probability at least $1/k^k$
- And if it is a NO instance, the algorithm output NO.
- This looks very bad, but since k is considered as a constant maybe it is not that bad!

Brillant idea: do it a lot of times

Useful fact

If the probability of success of a (Monte-Carlo) algorithm is at least p, then the probability that, given a YES-instance, the algorithm return NO $1/p$ times in a row is at most:

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

Brillant idea: do it a lot of times

Useful fact

If the probability of success of a (Monte-Carlo) algorithm is at least p, then the probability that, given a YES-instance, the algorithm return NO $1/p$ times in a row is at most:

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

Thus if $p \geq \frac{1}{k^k}$, then after k^k repetitions error probability is at most $1/e$:

$$\left(1 - \frac{1}{k^k}\right)^k < \frac{1}{e}$$

Hence, by trying $100 \cdot k^k$ random colorings, the probability of a wrong answer is at most $1/e^{100}$.

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

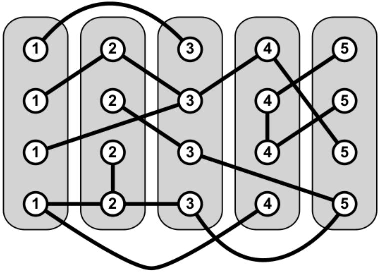


Figure by Daniel Marx

- Let V_i be the set of vertices colored i (color class)
- Delete edge linking non-consecutive color classes.
- Orient the edges toward the larger class
- Check if there is a path from color class 1 to color class k: this can be done in linear time with BFS.

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

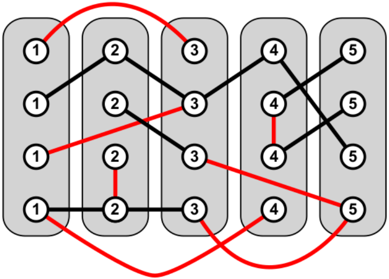


Figure by Daniel Marx

- Let V_i be the set of vertices colored i (color class)
- Delete edge linking non-consecutive color classes.
- Orient the edges toward the larger class
- Check if there is a path from color class 1 to color class k: this can be done in linear time with BFS.

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

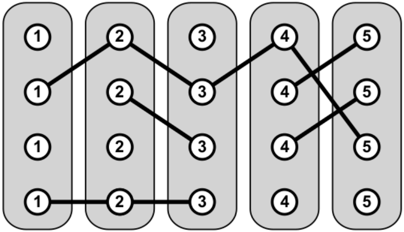


Figure by Daniel Marx

- Let V_i be the set of vertices colored i (color class)
- Delete edge linking non-consecutive color classes.
- Orient the edges toward the larger class
- Check if there is a path from color class 1 to color class k: this can be done in linear time with BFS.

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

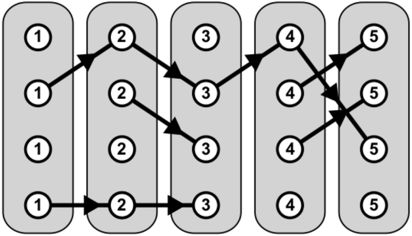


Figure by Daniel Marx

- Let V_i be the set of vertices colored i (color class)
- Delete edge linking non-consecutive color classes.
- Orient the edges toward the larger class
- Check if there is a path from color class 1 to color class k: this can be done in linear time with BFS.

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

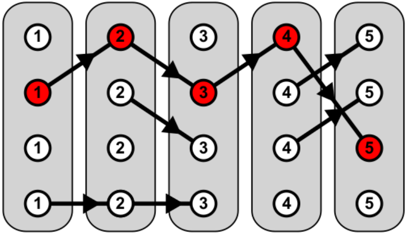


Figure by Daniel Marx

- Let V_i be the set of vertices colored i (color class)
- Delete edge linking non-consecutive color classes.
- Orient the edges toward the larger class
- Check if there is a path from color class 1 to color class k: this can be done in linear time with BFS.

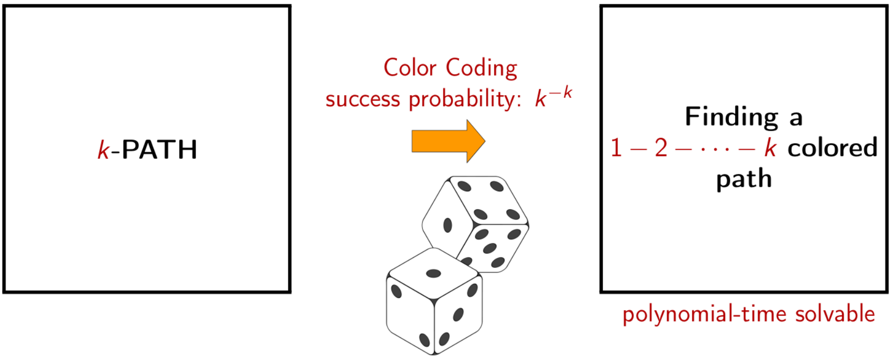
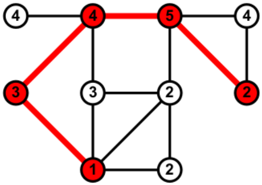


Figure by Daniel Marx

Complexity: $O(c \cdot k^k \cdot (n + m))$.
Probability of sucess: $1/e^c$

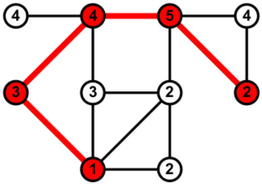
Improved color coding

- Assign colors from $[k]$ to the vertices uniformly and independantly at random.



Improved color coding

- Assign colors from $[k]$ to the vertices uniformly and independantly at random.



- Output YES if there is a colorfull k-path.
  - If there is no k-path, no colorfull path exist, and the algo output NO.
  - If there is a k-path, probability that it is colorfull is

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

- Repeat the algorithm $100e^k$ times decrease the error probability to e^{-100}.

Improved color coding

So replacing the problem "Find a k-path colored $1 - 2 - \cdots - k$?" by "Is there a k-path coloured with k colours?" allowed us to go from probability of sucess of $1/k^k$ to $1/e^k$.

Recall that this means that we need to solve the problem e^k times instead of k^k.

But how hard is it to solve colorfull path problem?

Find a colorfullpath with dynamic programming

Subproblem: For each vertex v and each set of color $C \subseteq [k]$, define:

$D(v, C)$ to be YES if there is a path ending at v and using each color of C.

Find a colorfullpath with dynamic programming

Subproblem: For each vertex v and each set of color $C \subseteq [k]$, define:

$D(v, C)$ to be YES if there is a path ending at v and using each color of C.

Denote by $\chi : V \to [k]$ the random coloring.

$D(v, C)$ is YES if and only if $\chi(v) \in C$ and there is an edge uv for which $D(u, C \setminus \chi(v))$ is YES.

Find a colorfullpath with dynamic programming

Subproblem: For each vertex v and each set of color $C \subseteq [k]$, define:

$D(v, C)$ to be YES if there is a path ending at v and using each color of C.

Denote by $\chi : V \to [k]$ the random coloring.

$D(v, C)$ is YES if and only if $\chi(v) \in C$ and there is an edge uv for which $D(u, C \setminus \chi(v))$ is YES.

Now, we can solve this DP in time $2^k \cdot |E|$

Recap

The algorithm: Repeat e^k times:

1. Sample a coloring $c : V \leftarrow \{1, \ldots, k\}$
2. Check if G contains a colorfull k-path in time $O(2^k) \cdot |E|$ and return YES if it does.

If no colorfull k-path was found, return NO.

Recap

The algorithm: Repeat e^k times:

1. Sample a coloring $c : V \leftarrow \{1, \dots, k\}$
2. Check if G contains a colorfull k-path in time $O(2^k) \cdot |E|$ and return YES if it does.

If no colorfull k-path was found, return NO.

Analysis:

- If no solution, the answer is correct,

Recap

The algorithm: Repeat e^k times:

1. Sample a coloring $c : V \leftarrow \{1, \ldots, k\}$
2. Check if G contains a colorfull k-path in time $O(2^k) \cdot |E|$ and return YES if it does.

If no colorfull k-path was found, return NO.

Analysis:

- If no solution, the answer is correct,
- If there is a solution $(u_1 u_2 \ldots u_k)$,

$$Pr(\text{single try sucess}) \geq \frac{k!}{k^k} \simeq \frac{(\frac{k}{e})^k}{k^k} = \frac{1}{e^k}$$

Recap

The algorithm: Repeat e^k times:

1. Sample a coloring $c : V \leftarrow \{1, \dots, k\}$
2. Check if G contains a colorfull k-path in time $O(2^k) \cdot |E|$ and return YES if it does.

If no colorfull k-path was found, return NO.

Analysis:

- If no solution, the answer is correct,
- If there is a solution $(u_1 u_2 \dots u_k)$,

$$Pr(\text{single try sucess}) \geq \frac{k!}{k^k} \simeq \frac{(\frac{k}{e})^k}{k^k} = \frac{1}{e^k}$$

$$Pr[error] = Pr[e^k \text{ single failures}] \leq \left(1 - \frac{1}{e^k}\right)^{e^k} \leq \frac{1}{e} < \frac{1}{2}$$

Recap

The algorithm: Repeat e^k times:

1. Sample a coloring $c : V \leftarrow \{1, \dots, k\}$
2. Check if G contains a colorfull k-path in time $O(2^k) \cdot |E|$ and return YES if it does.

If no colorfull k-path was found, return NO.

Analysis:

- If no solution, the answer is correct,
- If there is a solution $(u_1 u_2 \dots u_k)$,

$$Pr(\text{single try sucess}) \geq \frac{k!}{k^k} \simeq \frac{(\frac{k}{e})^k}{k^k} = \frac{1}{e^k}$$

$$Pr[error] = Pr[e^k \text{ single failures}] \leq \left(1 - \frac{1}{e^k}\right)^{e^k} \leq \frac{1}{e} < \frac{1}{2}$$

- Total running time: $O((2e)^k \cdot |E|)$.

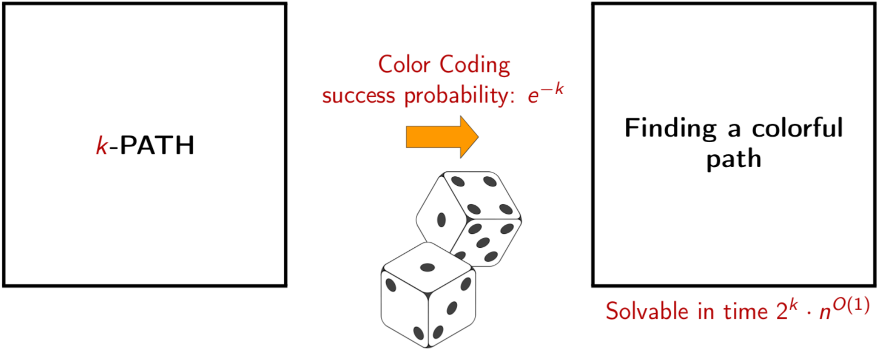


Figure by Daniel Marx

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

k-PATH

k-perfect family

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

Finding a colorful path

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

Figure by Daniel Marx

4 - Iterative Compression

Iterative compression

General technique used for graph modification problems: Find a set S of k vertices/edges such that $G \setminus S$ has a particular property.

Iterative compression

General technique used for graph modification problems: Find a set S of k vertices/edges such that $G \setminus S$ has a particular property.

We'll do it for Feedback Vertex Set:

- Goal: find a set S of at most k vertices such that $G \setminus S$ is a forest.
- Running time: $5^k \cdot n^{O(1)}$.

Recall that we have seen an algorithm runing in $(3k)^k n^{O(1)}$ using the branching method.

Best known algorithm: $2.7^k \cdot n^{O(1)}$, Li and Nederlof, 2020.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.
- We are going to compute a solution S_i for each G_i.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.
- We are going to compute a solution S_i for each G_i.
- Assume we have a solution S_{i-1} for G_{i-1} (i.e. $G_{i-1} \setminus S_{i-1}$ is a forest).

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \dots, v_n$.
- Set $G_i = G[\{v_1, v_2, \dots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.
- We are going to compute a solution S_i for each G_i.
- Assume we have a solution S_{i-1} for G_{i-1} (i.e. $G_{i-1} \setminus S_{i-1}$ is a forest).
- Key: $S_{i-1} \cup \{v_i\}$ is a solution for G_i but potentially of size $k+1$.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.
- We are going to compute a solution S_i for each G_i.
- Assume we have a solution S_{i-1} for G_{i-1} (i.e. $G_{i-1} \setminus S_{i-1}$ is a forest).
- Key: $S_{i-1} \cup \{v_i\}$ is a solution for G_i but potentially of size $k + 1$.

General idea of iterative compression

Main idea: introduce vertices one by one and maintain a solution

- Order the vertices: $v_1, v_2, \ldots, v_n$.
- Set $G_i = G[\{v_1, v_2, \ldots, v_i\}]$.
- Oberve that if G is a YES-instance, then each G_i is a YES-instance.
- We are going to compute a solution S_i for each G_i.
- Assume we have a solution S_{i-1} for G_{i-1} (i.e. $G_{i-1} \setminus S_{i-1}$ is a forest).
- Key: $S_{i-1} \cup \{v_i\}$ is a solution for G_i but potentially of size $k+1$.

So we can focus on the following problem:

Problem (FVS COMPRESSION)

Input: (G, k) and a vertex set S with $|S| \leq k+1$ and $G \setminus S$ is a forest.
Output: A FVS of size at most k (if it exists).

Problem (FVS COMPRESSION)

Input: (G, k) and a vertex set S with $|S| \leq k + 1$ and $G \setminus S$ is a forest.
Output: A FVS of size at most k.

Observation: if we can solve FVS COMPRESSION in time $f(k) \cdot n^c$, then we can solve FVS in time $f(k) \cdot n^{c+1}$.

So we can assume that we have a FVS of size $k + 1$ essentially for free

This FVS of size $k + 1$ gives us a lot of structure that will help us to find a smaller FVS.

Solve FVS COMPRESSION with Branching

Branching: 'guess' a set $X_S \subseteq S$ (2^{k+1} choices) that goes into the solution X.

- Delete X_S from G.
- Set $W = S - X_S$ and $\ell = |W| = k + 1 - |X_S|$
- It remains to solve the following:

Problem (Disjoint FVS)

Input: G, $W \subseteq V(G)$ such that $G \setminus W$ is a forest.
Output: a FVS X such that $|X| \leq |W| - 1$ and $X \cap W = \emptyset$.
Parameter: $|W| = \ell$.

Solve FVS compression with Branching

Branching: 'guess' a set $X_S \subseteq S$ (2^{k+1} choices) that goes into the solution X.

- Delete X_S from G.
- Set $W = S - X_S$ and $\ell = |W| = k + 1 - |X_S|$
- It remains to solve the following:

Problem (Disjoint FVS)

Input: G, $W \subseteq V(G)$ such that $G \setminus W$ is a forest.
Output: a FVS X such that $|X| \leq |W| - 1$ and $X \cap W = \emptyset$.
Parameter: $|W| = \ell$.

$f(\ell) \cdot n^c$ for Disjoint FVS

Solve FVS compression with Branching

Branching: 'guess' a set $X_S \subseteq S$ (2^{k+1} choices) that goes into the solution X.

- Delete X_S from G.
- Set $W = S - X_S$ and $\ell = |W| = k + 1 - |X_S|$
- It remains to solve the following:

Problem (Disjoint FVS)

Input: G, $W \subseteq V(G)$ such that $G \setminus W$ is a forest.
Output: a FVS X such that $|X| \leq |W| - 1$ and $X \cap W = \emptyset$.
Parameter: $|W| = \ell$.

$f(\ell) \cdot n^c$ for Disjoint FVS

$$\Downarrow$$

$\sum_{\ell=0}^{k} \binom{k+1}{\ell} f(\ell) \cdot n^c = \hat{f}(k) \cdot n^c$ for FVS compression

Solve FVS COMPRESSION with Branching

Branching: 'guess' a set $X_S \subseteq S$ (2^{k+1} choices) that goes into the solution X.

- Delete X_S from G.
- Set $W = S - X_S$ and $\ell = |W| = k + 1 - |X_S|$
- It remains to solve the following:

Problem (Disjoint FVS)

Input: G, $W \subseteq V(G)$ such that $G \setminus W$ is a forest.
Output: a FVS X such that $|X| \leq |W| - 1$ and $X \cap W = \emptyset$.
Parameter: $|W| = \ell$.

$f(\ell) \cdot n^c$ for Disjoint FVS

$$\Downarrow$$

$\sum_{\ell=0}^{k} \binom{k+1}{\ell} f(\ell) \cdot n^c = \hat{f}(k) \cdot n^c$ for FVS COMPRESSION

$$\Downarrow$$

$\hat{f}(k) \cdot n^{c+1}$ for FVS

Disjoint FVS

$f(\ell) \cdot n^c$ for Disjoint FVS

Disjoint FVS

$$f(\ell) \cdot n^c \text{ for } \textsc{Disjoint FVS}$$

$$\Downarrow$$

$$\sum_{\ell=0}^{k} \binom{k+1}{\ell} f(\ell) \cdot n^c = \hat{f}(k) \cdot n^c \text{ for } \textsc{FVS compression}$$

Disjoint FVS

$$f(\ell) \cdot n^c \text{ for Disjoint FVS}$$
$$\Downarrow$$
$$\sum_{\ell=0}^{k} \binom{k+1}{\ell} f(\ell) \cdot n^c = \hat{f}(k) \cdot n^c \text{ for FVS compression}$$
$$\Downarrow$$
$$\hat{f}(k) \cdot n^{c+1} \text{ for FVS}$$

Computation: If $f(\ell) = c^\ell$, then $\sum_{\ell=0}^{k} \binom{k+1}{\ell} f(\ell) = (c+1)^{k+1}$

Goal: Disjoint FVS in $4^\ell \cdot n^{0(1)}$ ($\Rightarrow 5^k \cdot n^{0(1)}$ for FVS compression
$\Rightarrow 5^{k+1} \cdot n^{0(1)}$ for FVS).

Solving DISJOINT FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Let u be a leaf of F

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Let u be a leaf of F
Case 1: u has no neighbor in W: delete u (it does not participate in any cycle).

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Let u be a leaf of F
Case 1: u has no neighbor in W: delete u (it does not participate in any cycle).
Case 2: u has a unique neighbor in W: delete u and add en edge between the two neighbors of u.

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Let u be a leaf of F
Case 1: u has no neighbor in W: delete u (it does not participate in any cycle).
Case 2: u has a unique neighbor in W: delete u and add en edge between the two neighbors of u.
Case 3: u has at least 2 neighbors in W:

- if there exists $w_1, w_2 \in N(u) \cap W$ such that w_1 and w_2 are in the same connected component of W, then u must be in the solution. So we may delete u, and solve Disjoint FVS on $(G \setminus u, \ell - 1)$

Solving Disjoint FVS

Input: G, $W \subseteq V(G)$, $|W| = \ell$, $G - W = F$ is a forest.
Goal: Find a FVS disjoint from W of size at most $\ell - 1$.

Idea: look at leaves of F and how they interact with W.

Let u be a leaf of F
Case 1: u has no neighbor in W: delete u (it does not participate in any cycle).
Case 2: u has a unique neighbor in W: delete u and add en edge between the two neighbors of u.
Case 3: u has at least 2 neighbors in W:

- if there exists $w_1, w_2 \in N(u) \cap W$ such that w_1 and w_2 are in the same connected component of W, then u must be in the solution. So we may delete u, and solve Disjoint FVS on $(G \setminus u, \ell - 1)$
- otherwise, branch on u:
  - u is in the solution, solve $(G - u, \ell - 1)$, or
  - u is not in the solution, add u into W.
    Then the number of connected components of W decreseases, which make us happy.
    Also observe that at the beginning, W has at most ℓ connected components.

Solving Disjoint FVS

Branch on u:

- u is in the solution, solve $(G - u, \ell - 1)$, or
- u is not in the solution, add u into W.
  Then the number of connected components of W decreseases by 1.

Formally: for an instance $I = (G, W, \ell)$, define a potential function

$$\mu(I) = \ell + \text{number of connected components of } G[W]$$

At the beginning: $\mu(I) \leq 2\ell$.
In each branch, μ decreases strictly in both branches,
So the tree has depth at most 2ℓ, and thus has at most $2^{2\ell} = 4^\ell$ vertices.
So the running time is $4^\ell \cdot n^{O(1)}$.

Recap

Generic: Problem ⇒ Problem Compression ⇒ Disjoint Problem.

Recap

Generic: Problem ⇒ Problem Compression ⇒ Disjoint Problem.

Let $\mathcal{C}$ be a class of graphs. The $\mathcal{C}$-vertex deletion problem is:
Input: A graph G and an integer k.
Question: is there $S \subseteq V(G)$ such that $G \setminus S \in \mathcal{C}$.

If $\mathcal{C} = \{\text{edgeless graphs}\} \Rightarrow$ Vertex Cover
If $\mathcal{C} = \{\text{forest graphs}\} \Rightarrow$ Feedback Vertex Set

Recap

Generic: Problem $\Rightarrow$ Problem Compression $\Rightarrow$ Disjoint Problem.

Let $\mathcal{C}$ be a class of graphs. The $\mathcal{C}$-vertex deletion problem is:
Input: A graph G and an integer k.
Question: is there $S \subseteq V(G)$ such that $G \setminus S \in \mathcal{C}$.

If $\mathcal{C} = \{\text{edgeless graphs}\} \Rightarrow$ VERTEX COVER
If $\mathcal{C} = \{\text{forest graphs}\} \Rightarrow$ FEEDBACK VERTEX SET

Disjoint $\mathcal{C}$-vertex deletion in FPT time $\Rightarrow$

$\mathcal{C}$-vertex deletion Compression in FPT time $\Rightarrow$

$\mathcal{C}$-vertex deletion in FPT time

If we are only interested to know if the problem is FPT or not, this is for free!