Parameterization: basics classes and algorithms

Maria Serna

Spring 2024

1. Parameterization
2. Bounded search tree
3. Kernelization

Three NP complete problems

VERTEX COLORING

Given a graph G and an integer k,
$\exists\sigma : V(G) \to \{1,\dots,k\} \mid \forall\{u,v\} \in E(G)\sigma(u) \neq \sigma(v)$?

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall\{u,v\} \in E(G)|\{u,v\} \cap S| \leq 1$?

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall\{u,v\} \in E(G)|\{u,v\} \cap S| \geq 1$?

Three NP complete problems

VERTEX COLORING

Given a graph G and an integer k,
$\exists \sigma : V(G) \to \{1, \dots, k\} \mid \forall \{u, v\} \in E(G) \sigma(u) \neq \sigma(v)$?

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k \text{ and } \forall \{u, v\} \in E(G) |\{u, v\} \cap S| \leq 1$?

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k \text{ and } \forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

Is there any difference from a computational point of view?

Three NP complete problems

VERTEX COLORING

Given a graph G and an integer k,
$\exists\sigma : V(G) \to \{1, \dots, k\} \mid \forall\{u, v\} \in E(G)\sigma(u) \neq \sigma(v)$?

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall\{u, v\} \in E(G)|\{u, v\} \cap S| \leq 1$?

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall\{u, v\} \in E(G)|\{u, v\} \cap S| \geq 1$?

Is there any difference from a computational point of view?

Let's look to exact algorithms.

Vertex Coloring

VERTEX COLORING

Given a graph G and an integer k,
$\exists\sigma : V(G) \to \{1, \dots, k\} \mid \forall\{u, v\} \in E(G)\sigma(u) \neq \sigma(v)$?

Vertex Coloring

VERTEX COLORING

Given a graph G and an integer k,
$\exists\sigma : V(G) \to \{1, \ldots, k\} \mid \forall\{u, v\} \in E(G)\sigma(u) \neq \sigma(v)$?

- Brute force algorithm that checks all color assignments:

Vertex Coloring

VERTEX COLORING
Given a graph G and an integer k,
$\exists\sigma : V(G) \to \{1,\ldots,k\} \mid \forall\{u,v\} \in E(G)\sigma(u) \neq \sigma(v)$?

- Brute force algorithm that checks all color assignments:
- takes time $O(n^2 k^n)$ time.

Independent Set

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \leq 1$?

Independent Set

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \leq 1$?

- Brute force algorithm that checks all subsets with k vertices

Independent Set

INDEPENDENT SET

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \leq 1$?

- Brute force algorithm that checks all subsets with k vertices
- takes time $O(n^{k+1})$ time.

Vertex Cover

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

Vertex Cover

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

- Brute force algorithm that checks all subsets with k vertices

Vertex Cover

VERTEX COVER

Given a graph G and an integer k,
$\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

- Brute force algorithm that checks all subsets with k vertices
- takes time $O(n^{k+1})$ time.

A better algorithm?

Vertex Cover

function $\textsc{algVC}(G, k)$
    if $|E(G)| = 0$ **then**
        return true
    end if
    if $k = 0$ **then**
        return false
    end if
    Select and edge $e = \{u, v\} \in E(G)$
    return $\textsc{algVC}(G - u, k - 1)$ or $\textsc{algVC}(G - v, k - 1)$
end function

Vertex Cover

function $\textsc{algVC}(G, k)$
  if $|E(G)| = 0$ **then**
    return true
  end if
  if $k = 0$ **then**
    return false
  end if
  Select and edge $e = \{u, v\} \in E(G)$
  return $\textsc{algVC}(G - u, k - 1)$ or $\textsc{algVC}(G - v, k - 1)$
end function

Correctly solves the problem and takes time $O(m2^k)$

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

Algorithms cost

Given a graph G and an integer k:

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!

For constant k:

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!

For constant k:

- VERTEX COLORING: $O(n^2 k^n)$

Algorithms cost

Given a graph G and an integer k:

- Vertex coloring: $O(n^2 k^n)$
- Independent Set: $O(n^{k+1})$
- Vertex Cover: $O(m2^k)$

The dependence on $|G|$ and k are different!
For constant k:

- Vertex coloring: $O(n^2 k^n)$ exponential

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!

For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!
For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$ polynomial

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!
For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$ polynomial
- VERTEX COVER: $O(m2^k)$

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!
For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$ polynomial
- VERTEX COVER: $O(m2^k)$ polynomial

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m2^k)$

The dependence on $|G|$ and k are different!
For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$ polynomial
- VERTEX COVER: $O(m2^k)$ polynomial (even for $k = O(\log n)$)

Algorithms cost

Given a graph G and an integer k:

- VERTEX COLORING: $O(n^2 k^n)$
- INDEPENDENT SET: $O(n^{k+1})$
- VERTEX COVER: $O(m 2^k)$

The dependence on $|G|$ and k are different!

For constant k:

- VERTEX COLORING: $O(n^2 k^n)$ exponential
- INDEPENDENT SET: $O(n^{k+1})$ polynomial
- VERTEX COVER: $O(m 2^k)$ polynomial
  (even for $k = O(\log n)$)

Objective: Find slices of the problem having efficient algorithms

Slice: The instances with a particular value of a parameter

Natural small parameters

- VLSI design: the number of layers in a chip is below 10.
- Biology: DNA chains in many cases have path width below 11
- Robotics: The robot movements have small dimension
- Compilers: Type compatibility is usually EXP-complete, however typical type declaration have small depth

Natural small parameters

- VLSI design: the number of layers in a chip is below 10.
- Biology: DNA chains in many cases have path width below 11
- Robotics: The robot movements have small dimension
- Compilers: Type compatibility is usually EXP-complete, however typical type declaration have small depth
- Optimization problem: the measure of the optimal solution is small
- A problem might have more than one parameter of interest and the behavior with respect to different parameters might be different.

Parameterized problems

Parameterized problems

Given an alphabet Σ to represent the inputs to decision problems,

Parameterized problems

Given an alphabet Σ to represent the inputs to decision problems,

- A parameterization of Σ^* is a mapping $\kappa : \Sigma^* \to \mathbb{N}$ that can be computed in polynomial time.

Parameterized problems

Given an alphabet Σ to represent the inputs to decision problems,

- A parameterization of Σ^* is a mapping $\kappa : \Sigma^* \rightarrow \mathbb{N}$ that can be computed in polynomial time.
- A parameterized problem (with respect to Σ) is a pair (L, κ) where $L \subseteq \Sigma^*$ and κ is a parameterization of Σ^*.

Parameterized problems

Given an alphabet Σ to represent the inputs to decision problems,

- A parameterization of Σ^* is a mapping $\kappa : \Sigma^* \to \mathbb{N}$ that can be computed in polynomial time.
- A parameterized problem (with respect to Σ) is a pair (L, κ) where $L \subseteq \Sigma^*$ and κ is a parameterization of Σ^*.

- Parameterized problems are decision problems together with a parameterization.
- A problem can be analyzed under different parameterizations.

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \#\text{ of variables in } F & \text{if } w \text{ codifies } F \\ -3 & \text{otherwise} \end{cases}$$

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \#\text{ of variables in } F & \text{if } w \text{ codifies } F \\ -3 & \text{otherwise} \end{cases}$$

- κ is a parameterization

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \#\text{ of variables in } F & \text{if } w \text{ codifies } F \\ -3 & \text{otherwise} \end{cases}$$

- κ is a parameterization Why?

Parameterized problem: An example

SAT
Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \#\text{ of variables in } F & \text{if } w \text{ codifies } F \\ -3 & \text{otherwise} \end{cases}$$

- κ is a parameterization Why?

  P#VAR-SAT
  Input: A CNF formula F,
  Parameter: The number of variables in F
  Question: is there a satisfying assignment for F?

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \text{max \# of literals in a clause in } F & \text{if } w \text{ codifies } F \\ 0 & \text{otherwise} \end{cases}$$

Parameterized problem: An example

SAT

Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \text{max \# of literals in a clause in } F & \text{if } w \text{ codifies } F \\ 0 & \text{otherwise} \end{cases}$$

- κ is a parameterization.

Parameterized problem: An example

SAT
Given a CNF formula F,
is there a satisfying assignment for F?

- Consider $\kappa : \Sigma^* \to \mathbb{N}$

$$\kappa(w) = \begin{cases} \text{max } \# \text{ of literals in a clause in } F & \text{if } w \text{ codifies } F \\ 0 & \text{otherwise} \end{cases}$$

- κ is a parameterization.
  PMAX#LIT-SAT
  Input: A CNF formula F
  Parameter: The maximum number of literals in a clause in F
  Question: is there a satisfying assignment for F?

The NPO class: Natural parameterization

The NPO class: Natural parameterization

- Recall that an optimization problem is a structure $\mathcal{P} = (\mathsf{I}, \mathsf{sol}, \mathsf{m}, \mathsf{goal})$

The NPO class: Natural parameterization

- Recall that an optimization problem is a structure $\mathcal{P} = (\mathsf{I}, \mathsf{sol}, \mathsf{m}, \mathsf{goal})$
- The bounded version of an optimization problem is the decision problem

The NPO class: Natural parameterization

- Recall that an optimization problem is a structure $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$
- The bounded version of an optimization problem is the decision problem
  - Given $x \in \mathrm{I}$ and an integer k
    Is there $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \leqslant k$?
  - Given $x \in \mathrm{I}$ and an integer k
    Is there a solution $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \geqslant k$?

The NPO class: Natural parameterization

- Recall that an optimization problem is a structure $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$
- The bounded version of an optimization problem is the decision problem
  - Given $x \in \mathrm{I}$ and an integer k
    Is there $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \leqslant k$?
  - Given $x \in \mathrm{I}$ and an integer k
    Is there a solution $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \geqslant k$?
- The natural parameterization is the function $\kappa(x, k) = k$ (basically deals with x with small $\mathrm{opt}(x)$)

The NPO class: Natural parameterization

- Recall that an optimization problem is a structure $\mathcal{P} = (\mathrm{I}, \mathrm{sol}, \mathrm{m}, \mathrm{goal})$
- The bounded version of an optimization problem is the decision problem
  - Given $x \in \mathrm{I}$ and an integer k
    Is there $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \leqslant k$?
  - Given $x \in \mathrm{I}$ and an integer k
    Is there a solution $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \geqslant k$?
- The natural parameterization is the function $\kappa(x, k) = k$ (basically deals with x with small $\mathrm{opt}(x)$)

  P-Π

  Input: $x \in \mathrm{I}$ and an integer k,

  Parameter: k

  Question: Is there a solution $y \in \mathrm{sol}(x)$ such that $\mathrm{m}(x, y) \leqslant (\geqslant) k$?

Graph problems and parameters

- Let G be a graph and k a natural number.
- The function $\kappa(G, k) = k$ is used to define the parameterized problems
  - p-Independent Set
  - p-Vertex Coloring
  - p-Vertex Cover
  - p-Dominating Set
  - p-Clique
  - etc.

Graph problems and parameters

- Let G be a graph and k a natural number.
- The function $\kappa(G, k) = k$ is used to define the parameterized problems
  - p-Independent Set
  - p-Vertex Coloring
  - p-Vertex Cover
  - p-Dominating Set
  - p-Clique
  - etc.
- For problems on graphs we can use other graph properties to define graph parameters like max degree or diameter.
  Or any other graph parameter of interest.

FPT: Fixed Parameter Tractable Parameterized Problems

FPT: Fixed Parameter Tractable Parameterized Problems

- For an alphabet Σ and a parameterization κ.

FPT: Fixed Parameter Tractable Parameterized Problems

- For an alphabet Σ and a parameterization κ.
- $\mathcal{A}$ is an FPT algorithm with respect to κ if there are a computable function f and a polinomial function p such that for each $x \in \Sigma^*$, $\mathcal{A}$ on input x requires time $f(\kappa(x))p(|x|)$

FPT: Fixed Parameter Tractable Parameterized Problems

- For an alphabet Σ and a parameterization κ.
- $\mathcal{A}$ is an FPT algorithm with respect to κ if there are a computable function f and a polinomial function p such that for each $x \in \Sigma^*$, $\mathcal{A}$ on input x requires time $f(\kappa(x))p(|x|)$
- A parameterized problem (L, κ) belongs to FPT if there is an FPT-algorithm with respect to κ that decides L.

FPT: Fixed Parameter Tractable Parameterized Problems

- For an alphabet Σ and a parameterization κ.
- $\mathcal{A}$ is an FPT algorithm with respect to κ if there are a computable function f and a polinomial function p such that for each $x \in \Sigma^*$, $\mathcal{A}$ on input x requires time $f(\kappa(x))p(|x|)$
- A parameterized problem (L, κ) belongs to FPT if there is an FPT-algorithm with respect to κ that decides L.
- We have show that there is an algorithm for VERTEX COVER requiring $O(|E(G)|2^k)$ time

FPT: Fixed Parameter Tractable Parameterized Problems

- For an alphabet Σ and a parameterization κ.
- $\mathcal{A}$ is an FPT algorithm with respect to κ if there are a computable function f and a polinomial function p such that for each $x \in \Sigma^*$, $\mathcal{A}$ on input x requires time $f(\kappa(x))p(|x|)$
- A parameterized problem (L, κ) belongs to FPT if there is an FPT-algorithm with respect to κ that decides L.
- We have show that there is an algorithm for VERTEX COVER requiring $O(|E(G)|2^k)$ time
  P-VERTEX COVER belongs to FPT!

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.
  - paraNP is the counterpart of NP in classic complexity.
- XP
  - (L, κ) belongs to (uniform) XP if there is an algorithm $\mathcal{A}$ that decides $x \in L$ in time $O(|x|^{f(\kappa(x))})$, for a computable function f.

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.
  - paraNP is the counterpart of NP in classic complexity.
- XP
  - (L, κ) belongs to (uniform) XP if there is an algorithm $\mathcal{A}$ that decides $x \in L$ in time $O(|x|^{f(\kappa(x))})$, for a computable function f.
  - p-CLIQUE, p-VERTEX COVER, p-HITTING SET, p-DOMINATING SET belong to XP.

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.
  - paraNP is the counterpart of NP in classic complexity.
- XP
  - (L, κ) belongs to (uniform) XP if there is an algorithm $\mathcal{A}$ that decides $x \in L$ in time $O(|x|^{f(\kappa(x))})$, for a computable function f.
  - p-Clique, p-Vertex Cover, p-Hitting Set, p-Dominating Set belong to XP.
  - XP is the counterpart of EXP in classic complexity.

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.
  - paraNP is the counterpart of NP in classic complexity.
- XP
  - (L, κ) belongs to (uniform) XP if there is an algorithm $\mathcal{A}$ that decides $x \in L$ in time $O(|x|^{f(\kappa(x))})$, for a computable function f.
  - p-Clique, p-Vertex Cover, p-Hitting Set, p-Dominating Set belong to XP.
  - XP is the counterpart of EXP in classic complexity.
- In between FPT and those classes it is placed the W-hierarchy W[1], W[2] ...

Other classes (hard parameterized problems)

- paraNP
  - (L, κ) belongs to paraNP if there is a non-deterministic algorithm $\mathcal{A}$ that decides $x \in L$ in time $f(\kappa(x))p(|x|)$, for computable function f and polynomial function p.
  - If $L \in$ NP, for each parameterization κ, $(L, \kappa) \in$ paraNP p-Clique, p-Vertex Cover, ... belong to paraNP.
  - paraNP is the counterpart of NP in classic complexity.
- XP
  - (L, κ) belongs to (uniform) XP if there is an algorithm $\mathcal{A}$ that decides $x \in L$ in time $O(|x|^{f(\kappa(x))})$, for a computable function f.
  - p-Clique, p-Vertex Cover, p-Hitting Set, p-Dominating Set belong to XP.
  - XP is the counterpart of EXP in classic complexity.
- In between FPT and those classes it is placed the W-hierarchy W[1], W[2] ... defined through logic/circuit characterizations

p-Vertex Cover

P-VC

Input: a graph G and an integer k,

Parameter: k

Question: $\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

p-Vertex Cover

P-VC
Input: a graph G and an integer k,
Parameter: k
Question: $\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

function $\textsc{algVC}(G, k)$
  if $|E(G)| = 0$ **then**
    return true
  end if
  if $k = 0$ **then**
    return false
  end if
  Select and edge $e = \{u, v\} \in E(G)$
  return $\textsc{algVC}(G - u, k - 1)$ or $\textsc{algVC}(G - v, k - 1)$
end function

p-Vertex Cover

p-Vertex Cover

- ALGVC correctly solves the problem and takes time $O((n+m)2^k)$ thus p-VERTEX COVER belongs to FPT

p-Vertex Cover

- ALGVC correctly solves the problem and takes time $O((n+m)2^k)$ thus p-VERTEX COVER belongs to FPT
- ALGVC is a branching algorithm (two recursive calls) of bounded (by the parameter) depth

p-Vertex Cover

- ALGVC correctly solves the problem and takes time $O((n+m)2^k)$ thus p-Vertex Cover belongs to FPT
- ALGVC is a branching algorithm (two recursive calls) of bounded (by the parameter) depth
- As usual recursive calls are made to smaller instances (in some sense).

p-Vertex Cover

- ALGVC correctly solves the problem and takes time $O((n+m)2^k)$ thus p-VERTEX COVER belongs to FPT
- ALGVC is a branching algorithm (two recursive calls) of bounded (by the parameter) depth
- As usual recursive calls are made to smaller instances (in some sense).
- Such type of recursive algorithm is called a bounded search tree algorithm.

p-Vertex Cover

- ALGVC correctly solves the problem and takes time $O((n+m)2^k)$ thus p-VERTEX COVER belongs to FPT
- ALGVC is a branching algorithm (two recursive calls) of bounded (by the parameter) depth
- As usual recursive calls are made to smaller instances (in some sense).
- Such type of recursive algorithm is called a bounded search tree algorithm.
- If we have a constant bound on the number of recursive calls, depth bounded by the parameter, and polynomial cost per call, the resulting algorithm is an FPT algorithm.

Hitting Set

HITTING SET

Input: a collection of subsets $\mathcal{S} = (S_1, \ldots, S_m)$ of $U = \{1, \ldots, n\}$ and an integer k.

Question: $\exists A \subseteq U \mid |A| = k$ and $\forall X \in \mathcal{S}\ |X \cap A| \geq 1$?

Hitting Set

HITTING SET

Input: a collection of subsets $\mathcal{S} = (S_1, \ldots, S_m)$ of $U = \{1, \ldots, n\}$ and an integer k.

Question: $\exists A \subseteq U \mid |A| = k$ and $\forall X \in \mathcal{S}\ |X \cap A| \geq 1$?

- For a set family $\mathcal{S}$, let $d(\mathcal{S}) = \max\{|A| \mid A \in \mathcal{S}\}$

Hitting Set

HITTING SET

Input: a collection of subsets $\mathcal{S} = (S_1, \ldots, S_m)$ of $U = \{1, \ldots, n\}$ and an integer k.

Question: $\exists A \subseteq U \mid |A| = k$ and $\forall X \in \mathcal{S}\ |X \cap A| \geq 1$?

- For a set family $\mathcal{S}$, let $d(\mathcal{S}) = \max\{|A| \mid A \in \mathcal{S}\}$
- The function $\kappa(\mathcal{S}, k) = k + d(\mathcal{S})$ is a parameterization

Hitting Set

HITTING SET

Input: a collection of subsets $\mathcal{S} = (S_1, \ldots, S_m)$ of $U = \{1, \ldots, n\}$ and an integer k.

Question: $\exists A \subseteq U \mid |A| = k$ and $\forall X \in \mathcal{S}\ |X \cap A| \geq 1$?

- For a set family $\mathcal{S}$, let $d(\mathcal{S}) = \max\{|A| \mid A \in \mathcal{S}\}$
- The function $\kappa(\mathcal{S}, k) = k + d(\mathcal{S})$ is a parameterization

P-HITTING SET

Input: A collection of subsets $\mathcal{S} = (S_1, \ldots, S_m)$ of $U = \{1, \ldots, n\}$ and an integer k,

Parameter: $k + d(\mathcal{S})$

Question: $\exists A \subseteq U \mid |A| = k$ and $\forall X \in \mathcal{S}\ |X \cap A| \geq 1$?

p-Hitting Set

function $\text{ALGHS}(U, \mathcal{S}, k)$
  if $|\mathcal{S}| = 0$ **then**
    return true
  end if
  if $k = 0$ **then**
    return false
  end if
  Select a set $X \in \mathcal{S}$
  for all $v \in X$ **do**
    $V = U - \{v\}$; $\mathcal{S}_v = \{X \in \mathcal{S} \mid v \notin X\}$
    if $\text{ALGHS}(V, \mathcal{S}_v, k-1)$ **then**
      return (true)
    end if
  end for
  return false
end function

p-Hitting Set

```
function ALGHS(U, S, k)
    if |S| = 0 then
        return true
    end if
    if k = 0 then
        return false
    end if
    Select a set X ∈ S
    for all v ∈ X do
        V = U − {v}; S_v = {X ∈ S | v ∉ X}
        if ALGHS(V, S_v, k − 1) then
            return (true)
        end if
    end for
    return false
end function
```

p-Hitting Set

```
function ALGHS(U, S, k)
    if |S| = 0 then
        return true
    end if
    if k = 0 then
        return false
    end if
    Select a set X ∈ S
    for all v ∈ X do
        V = U − {v}; S_v = {X ∈ S | v ∉ X}
        if ALGHS(V, S_v, k − 1) then
            return (true)
        end if
    end for
    return false
end function
```

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$
- Let $T(s, k, d)$ be the number of steps of $\textsc{AlgHS}$ for inputs with $d(\mathcal{S}) \leq d$.

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$
- Let $T(s, k, d)$ be the number of steps of $\textsc{AlgHS}$ for inputs with $d(\mathcal{S}) \leq d$.
- $T(s, 0, d) = O(1)$
  $T(s, k, d) \leq dT(s, k-1, d) + O(s)$, for $k > 0$

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$
- Let $T(s, k, d)$ be the number of steps of $\textsc{AlgHS}$ for inputs with $d(\mathcal{S}) \leq d$.
- $T(s, 0, d) = O(1)$
  $T(s, k, d) \leq dT(s, k-1, d) + O(s)$, for $k > 0$
- When $d \geq 2$ and $k \geq 0$, there is a constant c (with respect to s and k) such that the above terms $O(1)$ and $O(s)$ are $\leq c\, s$.

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$
- Let $T(s, k, d)$ be the number of steps of $\textsc{AlgHS}$ for inputs with $d(\mathcal{S}) \le d$.
- $T(s, 0, d) = O(1)$
  $T(s, k, d) \le dT(s, k-1, d) + O(s)$, for $k > 0$
- When $d \ge 2$ and $k \ge 0$, there is a constant c (with respect to s and k) such that the above terms $O(1)$ and $O(s)$ are $\le c\, s$.

$$\begin{aligned} T(s,k,d) &\le dT(s,k-1,d) + c\,s \\ &\le d(dT(s,k-2,d) + c\,s) + c\,s \\ &\le d^2 T(s,k-2,d) + (d+1)c\,s \end{aligned}$$

p-Hitting Set

- Let $s = |U| + \sum_{j=1}^{m} |S_j|$
- Let $T(s, k, d)$ be the number of steps of $\textsc{AlgHS}$ for inputs with $d(\mathcal{S}) \leq d$.
- $T(s, 0, d) = O(1)$
  $T(s, k, d) \leq dT(s, k-1, d) + O(s)$, for $k > 0$
- When $d \geq 2$ and $k \geq 0$, there is a constant c (with respect to s and k) such that the above terms $O(1)$ and $O(s)$ are $\leq c\, s$.

$$\begin{aligned} T(s,k,d) &\leq dT(s,k-1,d) + c\,s \\ &\leq d(dT(s,k-2,d) + c\,s) + c\,s \\ &\leq d^2 T(s,k-2,d) + (d+1)c\,s \end{aligned}$$

- using the above inequalities it is easy to prove that $T(s, k, d) \leq (2d^k - 1)c\,s$.

p-Hitting Set

Lemma

p-Hitting Set *belongs to FPT*

Bounded search tree technique

Bounded search tree technique

- The FPT algorithms for p-Vertex Cover and p-card-Hitting Set are exact algorithms for Vertex Cover and Hitting Set respectively.

Bounded search tree technique

- The FPT algorithms for p-Vertex Cover and p-card-Hitting Set are exact algorithms for Vertex Cover and Hitting Set respectively.
- When the parameter is unbounded the algorithms take exponential time.

Bounded search tree technique

- The FPT algorithms for p-VERTEX COVER and p-CARD-HITTING SET are exact algorithms for VERTEX COVER and HITTING SET respectively.
- When the parameter is unbounded the algorithms take exponential time.
- We get FPT algorithm because the depth and/or branching of the recursion are function of the parameter.

Bounded search tree technique

- The FPT algorithms for p-Vertex Cover and p-card-Hitting Set are exact algorithms for Vertex Cover and Hitting Set respectively.
- When the parameter is unbounded the algorithms take exponential time.
- We get FPT algorithm because the depth and/or branching of the recursion are function of the parameter.
- This algorithmic technique is called bounded search trees.
- As a design tool we have to look for parameterizations allowing a recursive algorithm with those characteristics.

A faster algorithm for p-VC

Recall some notation

A faster algorithm for p-VC

Recall some notation

- For a graph G and $v \in V(G)$, $G - v$ denotes the graph obtained by deleting v (and all incident edges).
- For a set S, $S + v$ denotes $S \cup \{v\}$, and $S - v$ denotes $S \setminus \{v\}$.
- For a vertex $v \in V(G)$, $N(v)$ denotes the set of neighbors of v. $N[v] = N(v) + v$. $d(v) = |N(v)|$.
- For a graph $G = (V, E)$, $\delta(G) = \min_{v \in V} d(v)$, and $\Delta(G) = \max_{v \in V} d(v)$.

Vertex with degree 1

Vertex with degree 1

- If G contains a vertex u with $N(u) = \{v\}$, then there is a minimum vertex cover of G that contains v (but not u) .

Vertex with degree 1

- If G contains a vertex u with $N(u) = \{v\}$, then there is a minimum vertex cover of G that contains v (but not u) .
- In such a case,

Vertex with degree 1

- If G contains a vertex u with $N(u) = \{v\}$, then there is a minimum vertex cover of G that contains v (but not u) .
- In such a case,
  G has a k-VC iff $G - u - v$ has a $(k-1)$-VC

Vertex with degree 1

- If G contains a vertex u with $N(u) = \{v\}$, then there is a minimum vertex cover of G that contains v (but not u) .
- In such a case,
  G has a k-VC iff $G - u - v$ has a $(k-1)$-VC
- The recursion can skip a branching!

Vertex with degree 2

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then
  - there is a minimum vertex cover of G that contains all neighbors of v and w, or
  - there is a minimum vertex cover of G that contains v and w.

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then
  - there is a minimum vertex cover of G that contains all neighbors of v and w, or
  - there is a minimum vertex cover of G that contains v and w.

  Let S be a minimum vertex cover. If $v, w \notin S$, S must contains all neighbors of v and w. If S contains v but not w, S must contain u. But then, $S - u + w$ is also a minimum vertex cover, which contains v and w.

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then
  - there is a minimum vertex cover of G that contains all neighbors of v and w, or
  - there is a minimum vertex cover of G that contains v and w.

  Let S be a minimum vertex cover. If $v, w \notin S$, S must contains all neighbors of v and w. If S contains v but not w, S must contain u. But then, $S - u + w$ is also a minimum vertex cover, which contains v and w.
- In such a case,

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then
  - there is a minimum vertex cover of G that contains all neighbors of v and w, or
  - there is a minimum vertex cover of G that contains v and w.

  Let S be a minimum vertex cover. If $v, w \notin S$, S must contains all neighbors of v and w. If S contains v but not w, S must contain u. But then, $S - u + w$ is also a minimum vertex cover, which contains v and w.
- In such a case,
  G has a k-VC iff $G - u - v$ has a $(k-2)$-VC or
  $G - N[v] - N[w]$ has a $(k-x)$-VC, for $x = |N(v) \cup N(w)|$.

Vertex with degree 2

- If G contains a vertex u with $N(u) = \{v, w\}$, then
  - there is a minimum vertex cover of G that contains all neighbors of v and w, or
  - there is a minimum vertex cover of G that contains v and w.

  Let S be a minimum vertex cover. If $v, w \notin S$, S must contains all neighbors of v and w. If S contains v but not w, S must contain u. But then, $S - u + w$ is also a minimum vertex cover, which contains v and w.
- In such a case,
  G has a k-VC iff $G - u - v$ has a $(k-2)$-VC or
  $G - N[v] - N[w]$ has a $(k-x)$-VC, for $x = |N(v) \cup N(w)|$.
- If $\delta(G) \geq 2$, $x \geq 2$. The recursion can jump to a smaller problem in one step!

Vertex with degree ≥ 3

Vertex with degree ≥ 3

- If G contains a vertex u with $d(u) \geq 3$, then
  - there is a minimum vertex cover of G that contains u, or
  - there is a minimum vertex cover of G that contains $N(u)$.

Vertex with degree ≥ 3

- If G contains a vertex u with $d(u) \geq 3$, then
  - there is a minimum vertex cover of G that contains u, or
  - there is a minimum vertex cover of G that contains $N(u)$.
- In such a case,

Vertex with degree ≥ 3

- If G contains a vertex u with $d(u) \geq 3$, then
  - there is a minimum vertex cover of G that contains u, or
  - there is a minimum vertex cover of G that contains $N(u)$.
- In such a case,
  G has a k-VC iff $G - u$ has a $(k-1)$-VC or $G - N[u]$ has a $(k - d(u))$-VC.

Vertex with degree ≥ 3

- If G contains a vertex u with $d(u) \geq 3$, then
  - there is a minimum vertex cover of G that contains u, or
  - there is a minimum vertex cover of G that contains $N(u)$.
- In such a case,
  G has a k-VC iff $G - u$ has a $(k-1)$-VC or $G - N[u]$ has a $(k - d(u))$-VC.
- The recursion can jump to a smaller problem in one branch!

A faster algorithm for p-VC

- FASTVC:

A faster algorithm for p-VC

- FastVC:
  - If there is a vertex with degree one, use recursion of degree 1 vertices.
  - If there is a vertex with degree two, use recursion of degree 2 vertices.
  - Otherwise, use recursion of degree ≥ 3 vertices.

A faster algorithm for p-VC

- FASTVC:
  - If there is a vertex with degree one, use recursion of degree 1 vertices.
  - If there is a vertex with degree two, use recursion of degree 2 vertices.
  - Otherwise, use recursion of degree ≥ 3 vertices.
  - Stop recursion on base cases, graph has no edges (yes), $k = 0$ and edges (no).

A faster algorithm for p-VC

- FASTVC:
  - If there is a vertex with degree one, use recursion of degree 1 vertices.
  - If there is a vertex with degree two, use recursion of degree 2 vertices.
  - Otherwise, use recursion of degree ≥ 3 vertices.
  - Stop recursion on base cases, graph has no edges (yes), $k = 0$ and edges (no).
- How to get a bound in the cost?

A faster algorithm for p-VC

- FASTVC:
  - If there is a vertex with degree one, use recursion of degree 1 vertices.
  - If there is a vertex with degree two, use recursion of degree 2 vertices.
  - Otherwise, use recursion of degree ≥ 3 vertices.
  - Stop recursion on base cases, graph has no edges (yes), $k = 0$ and edges (no).
- How to get a bound in the cost? Guess and prove by induction!

A faster algorithm for p-VC

A faster algorithm for p-VC

Theorem

The search tree corresponding to FASTVC *has at most* 1.47^k *leaves.*

A faster algorithm for p-VC

Theorem

The search tree corresponding to FastVC *has at most* 1.47^k *leaves.*

Proof.

- By induction over k.

A faster algorithm for p-VC

Theorem

The search tree corresponding to FastVC *has at most* 1.47^k *leaves.*

Proof.

- By induction over k.
- If $k = 0$, we can decide in polynomial time if there is a 0-VC (there are no edges), so no recursive calls, only one node in the recursive search tree.

A faster algorithm for p-VC

Theorem

The search tree corresponding to FASTVC *has at most* 1.47^k *leaves.*

Proof.

- By induction over k.
- If $k = 0$, we can decide in polynomial time if there is a 0-VC (there are no edges), so no recursive calls, only one node in the recursive search tree.
- If $k \geq 1$, then there are 3 cases:

A faster algorithm for p-VC

Proof.

A faster algorithm for p-VC

Proof.

- G contains a degree 1 vertex, continue with the single instance $(G - v, k - 1)$, which by induction yields $1.47^{k-1} < 1.47^k$ leaves.

A faster algorithm for p-VC

Proof.

- G contains a degree 1 vertex, continue with the single instance $(G - v, k - 1)$, which by induction yields $1.47^{k-1} < 1.47^k$ leaves.
- G contains a degree 2 vertex, branch into two cases $(G', k - 2)$ and $(G'', k - x)$, but as $\delta(G) > 1$, $x \geq 2$. By induction, the total number of leaves is at most $2 \cdot 1.47^{k-2} \leq 1.47^k$.

A faster algorithm for p-VC

Proof.

- G contains a degree 1 vertex, continue with the single instance $(G - v, k - 1)$, which by induction yields $1.47^{k-1} < 1.47^k$ leaves.
- G contains a degree 2 vertex, branch into two cases $(G', k - 2)$ and $(G'', k - x)$, but as $\delta(G) > 1$, $x \geq 2$. By induction, the total number of leaves is at most $2 \cdot 1.47^{k-2} \leq 1.47^k$.
- G contains a degree $d \geq 3$ vertex, branch into two cases $(G', k - 1)$ and $(G'', k - d)$. By induction, the total number of leaves is at most $1.47^{k-1} + 1.47^{k-3} \leq 1.47^k$.

☺

A faster algorithm for p-VC

A faster algorithm for p-VC

Theorem

FastVC *has cost* $O(1.47^k p(n+m))$, *for some polynomial* p *besides the constant in* O *is also constant with respect to the parameter* k.

1. Parameterization

2. Bounded search tree

3. Kernelization

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.
    x and x' are equivalent instances

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.
    x and x' are equivalent instances
  - the size of x' is upperbounded by $f(\kappa(x))$, for some computable function f.

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.
    x and x' are equivalent instances
  - the size of x' is upperbounded by $f(\kappa(x))$, for some computable function f.
- An algorithm that computes x' and solves by brute force this instance has cost

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.
    x and x' are equivalent instances
  - the size of x' is upperbounded by $f(\kappa(x))$, for some computable function f.
- An algorithm that computes x' and solves by brute force this instance has cost
  $O(p(|x|) + g(f(\kappa(x)))$

Kernelization

- Kernelization is a technique to obtain FPT algorithms for a parameterized problem (L, κ).
- Based in auto-reductions
- We look for a polynomial time algorithm that transforms an instance x in another instance x' of the problem (the kernel). So that
  - x' is a yes instance iff x is a yes instance.
    x and x' are equivalent instances
  - the size of x' is upperbounded by $f(\kappa(x))$, for some computable function f.
- An algorithm that computes x' and solves by brute force this instance has cost
  $O(p(|x|) + g(f(\kappa(x)))$
  So, it is an FPT algorithm provided the problem is decidible.

k-Vertex Cover: reduction rules?

- Often a kernelization is defined through reduction rules that, either allow us to produce an smaller equivalent instance or to show that, the original instance is a NO instance.

k-Vertex Cover: reduction rules?

- Often a kernelization is defined through reduction rules that, either allow us to produce an smaller equivalent instance or to show that, the original instance is a NO instance.
- Technically, we could produce a NO instance of constant size, however we often see the construction as a preprocesing step that has the possibility of saying NO, and will do that as soon as possible.

k-Vertex Cover: reduction rules?

- Often a kernelization is defined through reduction rules that, either allow us to produce an smaller equivalent instance or to show that, the original instance is a NO instance.
- Technically, we could produce a NO instance of constant size, however we often see the construction as a preprocesing step that has the possibility of saying NO, and will do that as soon as possible.
- Let's look at a first kernelization for p-VC.

k-Vertex Cover: reduction rules?

- Often a kernelization is defined through reduction rules that, either allow us to produce an smaller equivalent instance or to show that, the original instance is a NO instance.
- Technically, we could produce a NO instance of constant size, however we often see the construction as a preprocesing step that has the possibility of saying NO, and will do that as soon as possible.
- Let's look at a first kernelization for p-VC.

p-VERTEX COVER
Input: a graph G and an integer k,
Parameter: k
Question: $\exists S \subseteq V(G) \mid |S| = k$ and $\forall \{u, v\} \in E(G) |\{u, v\} \cap S| \geq 1$?

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".
- An isolated vertex has degree zero. Therefore it does not cover any edge!

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".
- An isolated vertex has degree zero. Therefore it does not cover any edge!

Obs 1

If v is an isolated vertex, (G, k) and $(G - v, k)$ are equivalent.

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".
- An isolated vertex has degree zero. Therefore it does not cover any edge!

Obs 1

If v is an isolated vertex, (G, k) and $(G - v, k)$ are equivalent.

- A vertex with degree $\geq k + 1$

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".
- An isolated vertex has degree zero. Therefore it does not cover any edge!

Obs 1

If v is an isolated vertex, (G, k) and $(G - v, k)$ are equivalent.

- A vertex with degree $\geq k + 1$ must be part of a vertex cover of size $\leq k$.

k-Vertex Cover: reduction rules?

- Let (G, k) be a k-VC instance.
- recall: Two instances x_1 and x_2 of decision probem P are equivalent when "$x_1 \in \mathrm{P}$ iff $x_2 \in \mathrm{P}$".
- An isolated vertex has degree zero. Therefore it does not cover any edge!

Obs 1

If v is an isolated vertex, (G, k) and $(G - v, k)$ are equivalent.

- A vertex with degree $\geq k + 1$ must be part of a vertex cover of size $\leq k$.

Obs 2

If v has degree $\geq k + 1$, (G, k) and $(G - v, k - 1)$ are equivalent.

Reduction rules

- The previous observations suggest a preprocessing of the input:

Reduction rules

- The previous observations suggest a preprocessing of the input:
  Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.

Reduction rules

- The previous observations suggest a preprocessing of the input:
  Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- By Obs 1 and 2, the resulting instance (G', k') is equivalent to the original instance.

Reduction rules

- The previous observations suggest a preprocessing of the input:
  Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- By Obs 1 and 2, the resulting instance (G', k') is equivalent to the original instance.
- Furthermore, it can be computed in polynomial time.

Reduction rules

- The previous observations suggest a preprocessing of the input:
  Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- By Obs 1 and 2, the resulting instance (G', k') is equivalent to the original instance.
- Furthermore, it can be computed in polynomial time.
- How big is (G', k')?

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- In (G', k') all the vertices have degree $\leq k$.

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- In (G', k') all the vertices have degree $\leq k$.

Obs 3

If G has a vertex cover with $\leq k$ vertices and all the vertices have degree $\leq k$,

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- In (G', k') all the vertices have degree $\leq k$.

Obs 3

If G has a vertex cover with $\leq k$ vertices and all the vertices have degree $\leq k$, $|E(G')| \leq k^2$.

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- In (G', k') all the vertices have degree $\leq k$.

Obs 3

If G has a vertex cover with $\leq k$ vertices and all the vertices have degree $\leq k$, $|E(G')| \leq k^2$.

- So, we can filter as NO instances those leading to reduced instances with a high number of edges!

Reduced instance

- Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
- In (G', k') all the vertices have degree $\leq k$.

Obs 3

If G has a vertex cover with $\leq k$ vertices and all the vertices have degree $\leq k$, $|E(G')| \leq k^2$.

- So, we can filter as NO instances those leading to reduced instances with a high number of edges!
- By Obs 3, if $|E(G')| > k^2$, we replace (G', k') by a trivial small NO-instance, which is again equivalent.

Kernel

Theorem

Let (G, k) be an instance to p-VC. *In polynomial time we can obtain an equivalent* p-VC *instance (G', k') with $|V(G')|, |E(G')| \leq O(k^2)$.*

Kernel

Theorem

Let (G, k) be an instance to p-VC. *In polynomial time we can obtain an equivalent* p-VC *instance (G', k') with $|V(G')|, |E(G')| \leq O(k^2)$.*

- Such an instance is called a kernel.

Kernel

Theorem

Let (G, k) be an instance to $\mathrm{P\text{-}VC}$. *In polynomial time we can obtain an equivalent* $\mathrm{P\text{-}VC}$ *instance* (G', k') *with* $|V(G')|, |E(G')| \leq O(k^2)$.

- Such an instance is called a kernel.
- A kernel

Kernel

Theorem

Let (G,k) be an instance to p-VC*. In polynomial time we can obtain an equivalent* p-VC *instance (G',k') with $|V(G')|,|E(G')| \leq O(k^2)$.*

- Such an instance is called a kernel.
- A kernel
  - is an equivalent instance,

Kernel

Theorem

Let (G, k) be an instance to p-VC*. In polynomial time we can obtain an equivalent* p-VC *instance (G', k') with $|V(G')|, |E(G')| \leq O(k^2)$.*

- Such an instance is called a kernel.
- A kernel
  - is an equivalent instance,
  - can be computed in polynomial time, and

Kernel

Theorem

Let (G, k) be an instance to p-VC. *In polynomial time we can obtain an equivalent* p-VC *instance (G', k') with $|V(G')|, |E(G')| \leq O(k^2)$.*

- Such an instance is called a kernel.
- A kernel
    - is an equivalent instance,
    - can be computed in polynomial time, and
    - has size bounded by a function of the parameter

Kernelization algorithm

Kernelization algorithm

- Assume that KER-P is a polynomial time algorithm computing a kernel for a given instance of problem P and that ALG-P is an exact (exponential time) algorithm for P.

Kernelization algorithm

- Assume that $\textsc{Ker-P}$ is a polynomial time algorithm computing a kernel for a given instance of problem P and that $\textsc{Alg-P}$ is an exact (exponential time) algorithm for P.

function $\textsc{AlgKernel-P}(x)$
    $z = \textsc{Ker-P}(x)$
    return $(\textsc{Alg-P}(z))$
end function

Kernelization algorithm

- Assume that $\textsc{Ker-P}$ is a polynomial time algorithm computing a kernel for a given instance of problem P and that $\textsc{Alg-P}$ is an exact (exponential time) algorithm for P.

  function $\textsc{AlgKernel-P}(x)$
      $z = \textsc{Ker-P}(x)$
      return $(\textsc{Alg-P}(z))$
  end function

- $\textsc{AlgKer-p-VC}$ is an FPT algorithm for P.

A kernelization algorithm for p-VC

function $\textsc{AlgKernel-p-VC}(G, k)$

$(G', k') =$ Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.

if $|E(G')| > k^2$ **then return** $\textsc{no}$

end if

for each $S \subseteq V'$ with $|S| = k'$ **do**

  if S is a vertex cover **then return** $\textsc{si}$

  end if

end for

return $\textsc{no}$

end function

A kernelization algorithm for p-VC

function $\textsc{AlgKernel-p-VC}(G, k)$
    $(G', k') =$ Iteratively remove isolated vertices and vertices with degree at least $k+1$, decreasing the parameter by one in the second case.
    if $|E(G')| > k^2$ **then return** $\textsc{no}$
    end if
    for each $S \subseteq V'$ with $|S| = k'$ **do**
        if S is a vertex cover **then return** $\textsc{si}$
        end if
    end for
    return $\textsc{no}$
end function

$\textsc{AlgKernel-p-VC}$ runs in $O(n^c + k^{2k}k^2) = O(n^c) + O(k^{2k+2})$

p-MaxSat

P-MAXSAT

Input: a Boolean CNF formula F and an integer k.

Parameter: k.

Question: Is there a variable assignment satisfying at least k clauses?

p-MaxSat

P-MAXSAT
Input: a Boolean CNF formula F and an integer k.
Parameter: k.
Question: Is there a variable assignment satisfying at least k clauses?

Recall that the size of a CNF formula is the sum of clause lengths (# literals); we ignore as usual log-factors.

p-MaxSat: Reduction rules

p-MaxSat: Reduction rules

- A clause in F is trivial if it contains both a positive and negative literal in the same variable.

p-MaxSat: Reduction rules

- A clause in F is trivial if it contains both a positive and negative literal in the same variable.

Obs 1

Let F' be obtained from formula F by removing all t trivial clauses. $(F', k-t)$ and (F, k) are equivalent.

p-MaxSat: Reduction rules

p-MaxSat: Reduction rules

- A clause in (F, k) is long if it contains at least k literals, and short otherwise.

p-MaxSat: Reduction rules

- A clause in (F, k) is long if it contains at least k literals, and short otherwise.
- If F contains at least k long clauses, (F, k) is a YES instance of p-MaxSat.

p-MaxSat: Reduction rules

- A clause in (F, k) is long if it contains at least k literals, and short otherwise.
- If F contains at least k long clauses, (F, k) is a YES instance of p-MAXSAT.

Obs 2

Let F_s be obtained from formula F by removing all $\ell < k$ long clauses. $(F_s, k - \ell)$ and (F, k) are equivalent.

p-MaxSat: Reduction rules

p-MaxSat: Reduction rules

Obs 3

If F contains at least $2k$ clauses, (F, k) is a YES instance of p-MaxSat.

p-MaxSat: Reduction rules

Obs 3

If F contains at least $2k$ clauses, (F, k) is a YES instance of p-MAXSAT.

Proof.

Take an arbitrary truth assignment x and its complement $\overline{x}$ obtained by flipping all variables. Every clause of F is satisfied by x or by *overlinex* (or by both). The one that satisfies most clauses satisfies at least k clauses. ☺

A kernelization algorithm for p-MaxSat

```
function AlgKernel-p-MaxSat(F, k)
    Remove from F all t trivial clauses and set k = k − t
    if F has at least k long clauses then return YES
    end if
    Remove from F all ℓ long clauses and set k = k − ℓ
    if F has at least 2k clauses then return YES
    end if
    for each set of k clauses do
        for each selection of one literal per clause in the set do
            if selection has a compatible truth assignment then
                return YES
            end if
        end for
    end for
    return NO
```

A kernelization algorithm for p-MaxSat

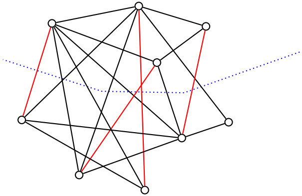
```
function AlgKernel-p-MaxSat(F, k)
    Remove from F all t trivial clauses and set k = k − t
    if F has at least k long clauses then return YES
    end if
    Remove from F all ℓ long clauses and set k = k − ℓ
    if F has at least 2k clauses then return YES
    end if
    for each set of k clauses do
        for each selection of one literal per clause in the set do
            if selection has a compatible truth assignment then
                return YES
            end if
        end for
    end for
    return NO
```

A kernelization algorithm for p-MaxSat

function ALGKERNEL-P-MAXSAT(F, k)
  Remove from F all t trivial clauses and set $k = k - t$
  if F has at least k long clauses **then return** YES
  end if
  Remove from F all ℓ long clauses and set $k = k - \ell$
  if F has at least $2k$ clauses **then return** YES
  end if
  for each set of k clauses **do**
    for each selection of one literal per clause in the set **do**
      if selection has a compatible truth assignment **then**
        return YES
      end if
    end for
  end for
  return NO

- *Crown decomposition* is a general kernelization technique based on some results on matchings.

- *Crown decomposition* is a general kernelization technique based on some results on matchings.
- For disjoint vertex subsets U, W of a graph G, M is a matching of U into W if every edge of M connects a vertex of U and a vertex of W and every vertex of U is an endpoint of some edge of M.
  We also say that M saturates U.



If M saturates U, $|U| \leq |W|$

Crown decomposition: Definition

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

Crown decomposition: Definition

- A crown decomposition of a graph $G = (V, E)$ is a partitioning of V into three parts C, H and R, such that
  - $C \neq \emptyset$ is an independent set.

Crown decomposition: Definition

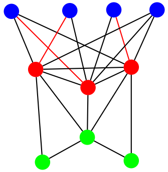
- A crown decomposition of a graph $G = (V, E)$ is a partitioning of V into three parts C, H and R, such that
  - $C \neq \emptyset$ is an independent set.
  - There are no edges between vertices of C and R. Removing H separates C from R.

Crown decomposition: Definition

- A crown decomposition of a graph $G = (V, E)$ is a partitioning of V into three parts C, H and R, such that
  - $C \neq \emptyset$ is an independent set.
  - There are no edges between vertices of C and R. Removing H separates C from R.
  - Let E' be the set of edges between vertices of C and H. Then E' contains a matching of H into C.

Crown decomposition: Definition

- A crown decomposition of a graph $G = (V, E)$ is a partitioning of V into three parts C, H and R, such that
  - $C \neq \emptyset$ is an independent set.
  - There are no edges between vertices of C and R. Removing H separates C from R.
  - Let E' be the set of edges between vertices of C and H. Then E' contains a matching of H into C.



Computing a crown decomposition

Theorem (König's theorem)

In every undirected bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover.

Theorem (Hall's theorem)

Let $G = (V_1, V_2, E)$ be an undirected bipartite graph. G has a matching saturating V_1 iff for all $X \subseteq V_1$, we have $|N(X)| \geq |X|$.

Can you obtain a minimum vertex cover in a bipartite graph in polynomial time?

Computing a crown decomposition

Theorem (König's theorem)

In every undirected bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover.

Theorem (Hall's theorem)

Let $G = (V_1, V_2, E)$ be an undirected bipartite graph. G has a matching saturating V_1 iff for all $X \subseteq V_1$, we have $|N(X)| \geq |X|$.

Can you obtain a minimum vertex cover in a bipartite graph in polynomial time? YES!

Computing a crown decomposition

Theorem ((Hopcroft-Karp, SIAM J. Computing 2, 225–231 (1973))

Let $G = (V_1, V_2, E)$ be an undirected bipartite graph on n vertices and m edges. Then we can find a maximum matching as well as a minimum vertex cover of G in time $O(m\sqrt{n})$. Furthermore, in time $O(m\sqrt{n})$ either we can find a matching saturating V_1 or an inclusion-wise minimal set $X \subseteq V_1$ such that $|N(X)| < |X|$.

Crown lemma

Lemma

Let $G = (V, E)$ be a graph without isolated vertices and with at least $3k + 1$ vertices. There is a polynomial-time algorithm that either

- *finds a matching of size $k + 1$ in G; or*
- *finds a crown decomposition of G.*

Proof

We compute a maximal matching M in G.
If $|M| \geq k + 1$, we are done.

Now, $1 \leq |M| \leq k+1$.

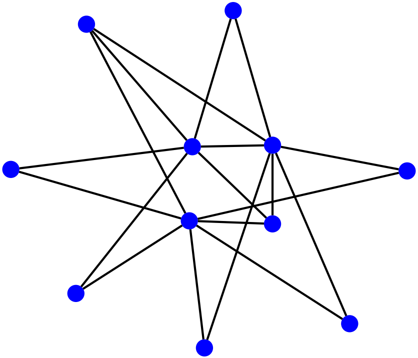
Let V_M be the end points of M and $I = V - V_M$.

- M is a maximal matching, so I is an independent set.
- Let G_{I,V_M} be the bipartite subgraph induced in G by I and V_M.
- In polynomial time, we compute a minimum size vertex cover X and a maximum matching M' in G_{I,V_M}.
- If $|M'| \geq k$, we are done. From now on, $|M'| \leq k$ and also $|X| \leq k$.
- If $X \cap V_M = \emptyset$, $X = I$. Then, $|I| = |X| \leq k$ and $|V| = |I| + |X| \leq k + 2k \leq 3k$!
- Then, $X \cap V_M \neq \emptyset$

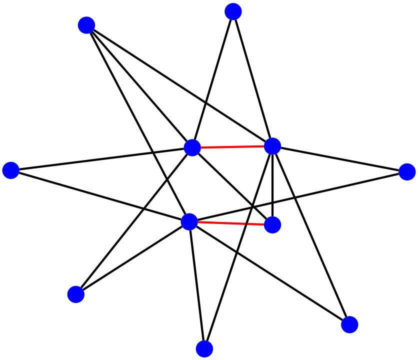
- We obtain a crown decomposition (C, H, R) as follows.
- Since $|X| = |M'|$, every edge of the matching M' has exactly one endpoint in X.
- Let M^* be the subset of M' such that every edge from M^* has exactly one endpoint in $X \cap V_M$ and let V_{M^*} denote the set of endpoints of edges in M^*.
- Set head $H = X \cap V_M = X \cap V_{M^*}$, crown $C = V_{M^*} \cap I$, and the remaining part is R.
- C is an independent set and, by construction, M^* is a matching of H into C.
- Since X is a vertex cover of G_{I,V_M}, every vertex of C can be adjacent only to vertices of H.

End proof

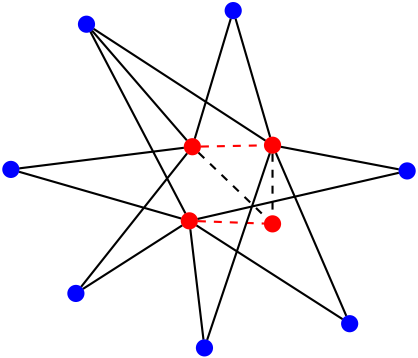
An example with $k = 3$



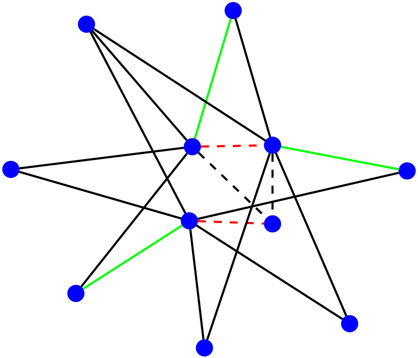
An example with $k = 3$



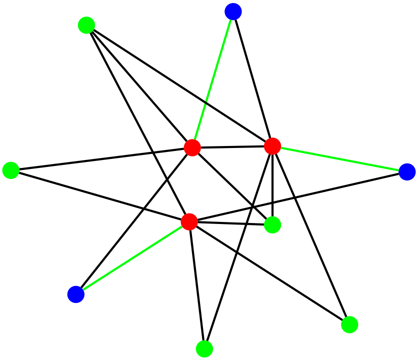
An example with $k = 3$



An example with $k = 3$



An example with $k = 3$



Crown decomposition: Vertex cover

Consider a Vertex Cover instance (G, k).

- By an exhaustive application of the isolated vertex reduction rule, we may assume that G has no isolated vertices.
- If $|V(G)| > 3k$, we use the crown lemma to get either
- a matching of size $k + 1$, (so (G, k) is a no-instance) or a crown decomposition C, H, R.

Crown decomposition: Vertex cover

From the crown decomposition C, H, R of G, let M be a matching of H into C.

- The matching M witnesses that, for every vertex cover X of G, X contains at least $|M| = |H|$ vertices of $H \cap C$ to cover the edges of M.
- H covers all edges of G that are incident to $H \cup C$.
- So, there exists a minimum vertex cover of G that contains H, and we may reduce (G, k) to $(G - H, k - |H|)$.
- Further, in $(G - H, k - |H|)$, $c \in C$ is isolated and can be eliminated.

Crown decomposition: Vertex cover

As the crown lemma promises that $H \neq \emptyset$, we can always reduce the graph as long as $|V(G)| > 3k$.

Lemma

Vertex Cover admits a kernel with at most $3k$ vertices.

Crown decomposition: Max SAT

Lemma

Max SAT admits a kernel with at most k variables and $2k$ clauses.

Kernelization: summary

Kernelization: summary

- For parameterized problems, kernelization algorithms are a method to obtain FPT algorithms.

Kernelization: summary

- For parameterized problems, kernelization algorithms are a method to obtain FPT algorithms.
- These are preprocessing algorithms that can add to any algorithmic method (e.g. approximation/exact algorithms).

Kernelization: summary

- For parameterized problems, kernelization algorithms are a method to obtain FPT algorithms.
- These are preprocessing algorithms that can add to any algorithmic method (e.g. approximation/exact algorithms).
- Kernelization algorithms usually consist of reduction rules, which reduce simple local structures (degree 1 vertices / high degree vertices / long clauses, etc), and a bound $f(k)$ for irreducible instances (X, k) that allows us to

Kernelization: summary

- For parameterized problems, kernelization algorithms are a method to obtain FPT algorithms.
- These are preprocessing algorithms that can add to any algorithmic method (e.g. approximation/exact algorithms).
- Kernelization algorithms usually consist of reduction rules, which reduce simple local structures (degree 1 vertices / high degree vertices / long clauses, etc), and a bound $f(k)$ for irreducible instances (X, k) that allows us to
  - return NO if $|X| > f(k)$, for minimization problems, or
  - return YES if $|X| > f(k)$, for maximization problems.

Designing kernelization algorithms

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?
- Can a bound be proved for irreducible instances? If not, which structures are problematic? Etc...

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?
- Can a bound be proved for irreducible instances? If not, which structures are problematic? Etc...
- Any problem in FPT admits a kernelization.

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?
- Can a bound be proved for irreducible instances? If not, which structures are problematic? Etc...
- Any problem in FPT admits a kernelization.

- Hardness notion?

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?
- Can a bound be proved for irreducible instances? If not, which structures are problematic? Etc...
- Any problem in FPT admits a kernelization.

- Hardness notion?
- We would like to get a kernel as small as possible.

Designing kernelization algorithms

- What are the trivial substructures, where an optimal solution of a certain form can be guaranteed?
- Is there a reduction rule reflecting this?
- Can a bound be proved for irreducible instances? If not, which structures are problematic? Etc...
- Any problem in FPT admits a kernelization.

- Hardness notion?
- We would like to get a kernel as small as possible.
- Statements like: (L, κ) does not admit a linear (quadratic) kernel unless some complexity assumption fails are the kind of results showing kernelization hardness.