Kernelization

Astrid Pieterse

October 21, 2016

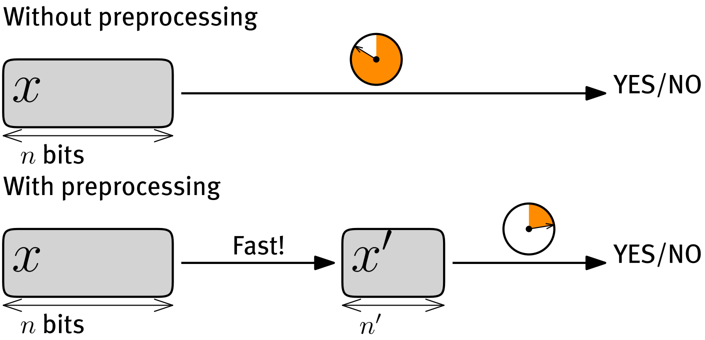
Algorithms and running time

- Algorithms
  - Today we consider decision problems: YES/NO output
- Running time
  - Number of steps compared to input size
  - Polynomial vs Exponential
  - Denoted using big-O notation, $O(n^5)$ or $O(3^n)$
- NP-hardness
  - No polynomial-time algorithm

Why preprocessing

Suppose the problem is NP-hard, what can we do?

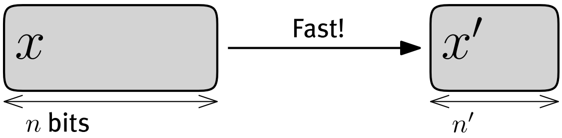
- Provide a way to preprocess an input
- Simplify the input efficiently, while keeping the answer
- Apply the “slow” algorithm to the simplified input



Great preprocessing

A great preprocessing algorithm would:

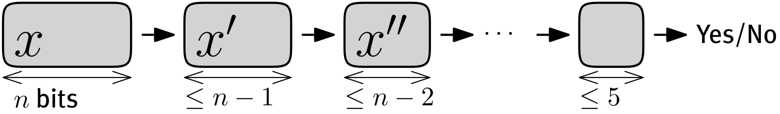
- Run in polynomial time
- Not change the answer
- Reduce the size of the instance to size $n' < n$



Great preprocessing: does not exist

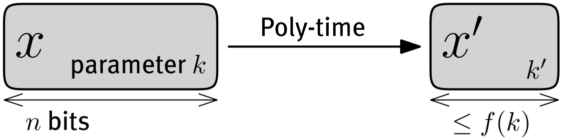
For NP-hard problems, such algorithms are not believed to exist:

- We can use them to solve the problem in polynomial time
  - As soon as the instance x has constant size (say, $\leqslant 5$), solve it in constant time
  - Otherwise, run Preprocess(x) to find a smaller instance x'
  - Recurse on x'
- Does $\leqslant n$ steps in polynomial time $\to$ polynomial time



Parameterized preprocessing

- Consider an additional parameter k, and measure the size of the preprocessed instance as a function of k
- k denotes the complexity of the instance in some way
  - Measure of how tree-like a graph is
  - Number of variables in a problem
  - ... More examples later
- An instance that is large compared to k must somehow contain many irrelevant parts



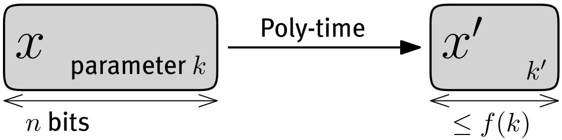
Parameterized preprocessing: kernels

Kernelization

Find an algorithm that

- Runs in polynomial time
- Outputs an equivalent instance of the same problem
- The new instance is *small*
  - Bounded by some function $f(k)$
  - Independent of n

Such an algorithm is called a *kernel*

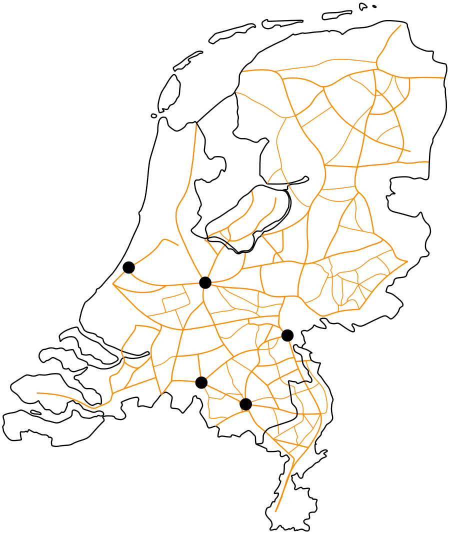


Kernelization: Example

- Visit all my k friends in a tour
- $k!$ possible orders
  - Hard problem
- Input size
  - Huge!
- Simplify input
  - Replace the road network by a graph
  - Edge $\{A, B\}$ keeps the distance from A to B
  - Equivalently, a distance table
- Kernel size
  - Number of cities squared
  - Much smaller

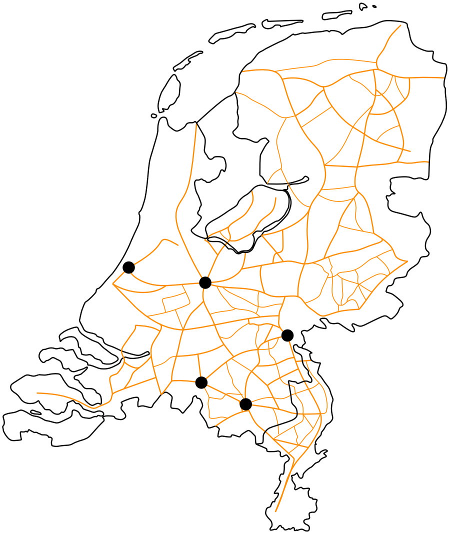
Kernelization: Example

- Visit all my k friends in a tour
- $k!$ possible orders
  - Hard problem
- Input size
  - Huge!
- Simplify input
  - Replace the road network by a graph
  - Edge $\{A, B\}$ keeps the distance from A to B
  - Equivalently, a distance table
- Kernel size
  - Number of cities squared
  - Much smaller



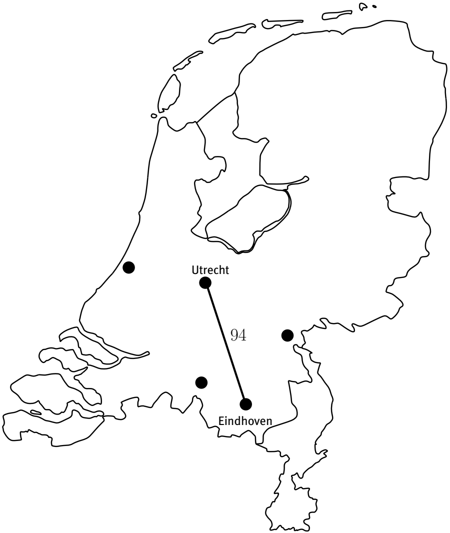
Kernelization: Example

- Visit all my k friends in a tour
- $k!$ possible orders
  - Hard problem
- Input size
  - Huge!
- Simplify input
  - Replace the road network by a graph
  - Edge $\{A, B\}$ keeps the distance from A to B
  - Equivalently, a distance table
- Kernel size
  - Number of cities squared
  - Much smaller



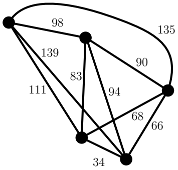
Kernelization: Example

- Visit all my k friends in a tour
- $k!$ possible orders
    - Hard problem
- Input size
    - Huge!
- Simplify input
    - Replace the road network by a graph
    - Edge $\{A, B\}$ keeps the distance from A to B
    - Equivalently, a distance table
- Kernel size
    - Number of cities squared
    - Much smaller



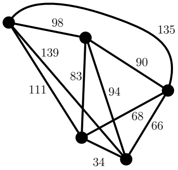
Kernelization: Example

- Visit all my k friends in a tour
- $k!$ possible orders
  - Hard problem
- Input size
  - Huge!
- Simplify input
  - Replace the road network by a graph
  - Edge $\{A, B\}$ keeps the distance from A to B
  - Equivalently, a distance table
- Kernel size
  - Number of cities squared
  - Much smaller



Kernelization: Example

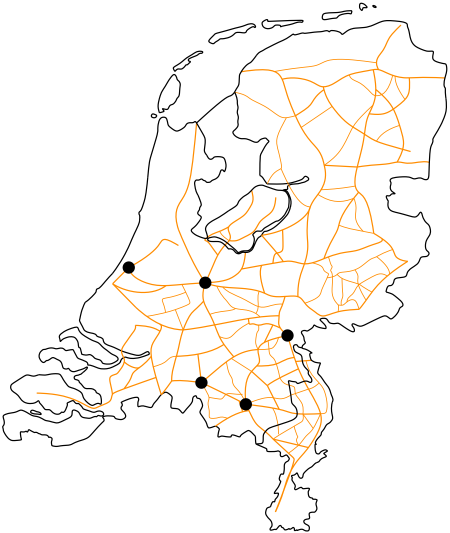
- Visit all my k friends in a tour
- $k!$ possible orders
  - Hard problem
- Input size
  - Huge!
- Simplify input
  - Replace the road network by a graph
  - Edge $\{A, B\}$ keeps the distance from A to B
  - Equivalently, a distance table
- Kernel size
  - Number of cities squared
  - Much smaller



Kernelization: Example

Correctness

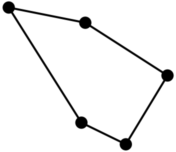
- The new instance is small
- Can solve shortest path in polynomial time
- Correctness?



Kernelization: Example

Suppose the kernel has some tour

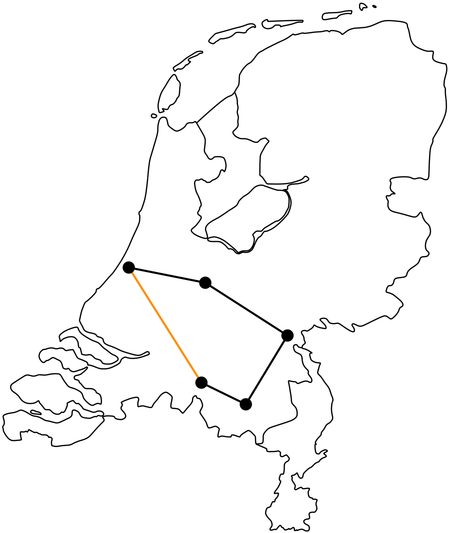
- If this tour uses edge $\{A, B\}$, replace by shortest path



Kernelization: Example

Suppose the kernel has some tour

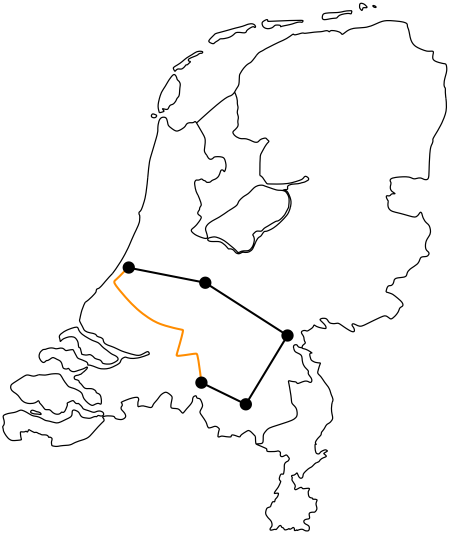
- If this tour uses edge $\{A, B\}$, replace by shortest path



Kernelization: Example

Suppose the kernel has some tour

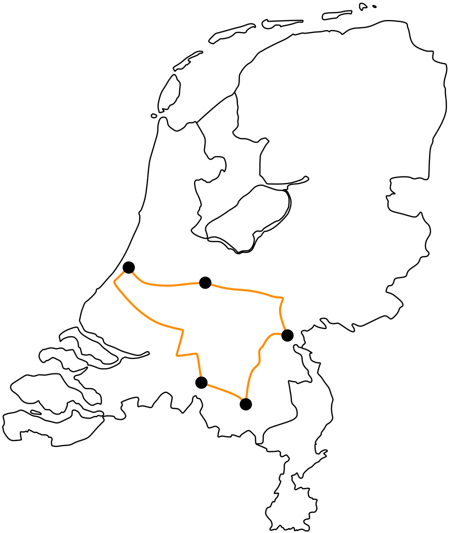
- If this tour uses edge $\{A, B\}$, replace by shortest path



Kernelization: Example

Suppose the kernel has some tour

- If this tour uses edge $\{A, B\}$, replace by shortest path



Exercises

1. We just saw a simple example of a kernel. But does it actually follow all the rules?
   You may assume that the decision problem asks whether there is a tour of length at most ℓ

Kernel:

- Runs in polynomial time
- x is a yes-instance $\Leftrightarrow$ x' is a yes-instance
- $|x'| \leqslant f(k)$ and $k' \leqslant f(k)$

Solutions

1. Technically, the size is not completely correct. How large are the edge-weights (length of shortest paths) that we store??
   They are not bounded by k.
   But, there are ways to solve the problem, there is a way to “rescale” our weights such that they can be stored efficiently

Satisfiability variants

- Input: some formula with boolean variables

$$(x \vee y) \wedge (z \vee \neg y \vee w) \wedge \ldots$$

- Used to verify specifications
- Rule out incorrect behaviour in chip design
- Type of formula depends on practical problem
  - influence running times of algorithms
  - influences kernel size?

Exact Satisfiability

Input CNF-formula F consisting of *clauses* each consisting of a number of *literals*, a literal is a variable or its negation

$$F = \underbrace{\{\neg x, \neg y\}}_{\text{clause}} \wedge \{\neg y, z\} \wedge \{x, z\}$$

Parameter The number of variables

Question Does there exist an assignment to the variables, such that each clause contains *exactly* one true literal?

With few variables, exponentially many clauses can be made

To obtain a kernel, we need to reduce the formula such that we can we bound the number of remaining clauses

Exact Satisfiability: Example

Input:

$$F = \underbrace{\{\neg x, \neg y\}}_{\text{clause}} \wedge \{\neg y, z\} \wedge \{x, z\}$$

has satisfying assignment

$$x \leftarrow \textit{true}, y \leftarrow \textit{false}, z \leftarrow \textit{false}$$

Substituted in F this gives:

$$\{\textit{false}, \textit{true}\} \wedge \{\textit{true}, \textit{false}\} \wedge \{\textit{true}, \textit{false}\}$$

Exactly 1 *true* literal in each clause, as required

Kernel for Exact Satisfiability

Example

Let $x, y, z \in \{0, 1\}$ (where 0 is false, 1 is true), then

$$
\begin{array}{lclcrcl}
\{\neg x, \neg y\} & \wedge & & (1-x)+(1-y)=1 & & x+y=1 \\
\{\neg y, z\} & \wedge & \Leftrightarrow & (1-y)+z=1 & \Leftrightarrow & z-y=0 \\
\{x, z\} & & & x+z=1 & & x+z=1
\end{array}
$$

is exact-sat

- Clause $\{x, z\}$ is satisfied if the other two clauses are satisfied

Kernel for Exact Satisfiability

Example

Let $x, y, z \in \{0, 1\}$ (where 0 is false, 1 is true), then

$$\begin{array}{llcll}
\{\neg x, \neg y\} & \wedge & & (1 - x) + (1 - y) = 1 & & x + y = 1 \\
\{\neg y, z\} & \wedge & \Leftrightarrow & (1 - y) + z = 1 & \Leftrightarrow & z - y = 0 \\
\{x, z\} & & & x + z = 1 & & x + z = 1
\end{array}$$

is exact-sat

- Clause $\{x, z\}$ is satisfied if the other two clauses are satisfied

Example

Let $x, y, z \in \{0, 1\}$ (where 0 is false, 1 is true), then

$$
\begin{array}{llcrcr}
\{\neg x, \neg y\} & \wedge & & (1 - x) + (1 - y) = 1 & & x + y = 1 \\
\{\neg y, z\} & \wedge & \Leftrightarrow & (1 - y) + z = 1 & \Leftrightarrow & z - y = 0 \\
\{x, z\} & & & x + z = 1 & & x + z = 1
\end{array}
$$

is exact-sat

- Clause $\{x, z\}$ is satisfied if the other two clauses are satisfied

Kernel for Exact Satisfiability

Example

Let $x, y, z \in \{0, 1\}$ (where 0 is false, 1 is true), then

$$\begin{array}{lclclcl}
\{\neg x, \neg y\} & \wedge & & (1 - x) + (1 - y) & = 1 & & x + y = 1 \\
\{\neg y, z\} & \wedge & \Leftrightarrow & (1 - y) + z & = 1 & \Leftrightarrow & z - y = 0 \\
\{x, z\} & & & x + z & = 1 & & \overline{x + z = 1}\;+
\end{array}$$

is exact-sat

- Clause $\{x, z\}$ is satisfied if the other two clauses are satisfied

Kernel for Exact Satisfiability

Example

Let $x, y, z \in \{0, 1\}$ (where 0 is false, 1 is true), then

$$
\begin{array}{lcclcl}
\{\neg x, \neg y\} \quad \wedge & & (1 - x) + (1 - y) = 1 & & x + y = 1 \\
\{\neg y, z\} \quad \wedge & \Leftrightarrow & (1 - y) + z = 1 & \Leftrightarrow & z - y = 0 \\
\{x, z\} & & x + z = 1 & & \overline{x + z = 1}^{\,+}
\end{array}
$$

is exact-sat

- Clause $\{x, z\}$ is satisfied if the other two clauses are satisfied

Kernel for Exact Satisfiability

Algorithm

- Given formula F
- Rewrite by giving an linear equation for each clause
- Find a basis of the row-space
  - Use Gaussian elimination
- Remove constraints not in the basis

$$\begin{pmatrix} 1 & 0 & \dots & 1 \\ 1 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ 0 & -1 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ \vdots \\ 0 \end{pmatrix}$$

Kernel for Exact Satisfiability

Algorithm

- Given formula F
- Rewrite by giving an linear equation for each clause
- Find a basis of the row-space
  - Use Gaussian elimination
- Remove constraints not in the basis

$$\begin{pmatrix} 1 & 0 & \dots & 1 \\ 1 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ 0 & -1 & \dots & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ \vdots \\ 0 \end{pmatrix}$$

Algorithm

- Given formula F
- Rewrite by giving an linear equation for each clause
- Find a basis of the row-space
  - Use Gaussian elimination
- Remove constraints not in the basis

$$\left(\begin{array}{cccc|c} 1 & 0 & \ldots & 1 & 1 \\ 1 & 1 & \ldots & 0 & 1 \\ \ldots & \ldots & \ldots & \ldots & \\ \ldots & \ldots & \ldots & \ldots & \\ 0 & -1 & \ldots & 1 & 0 \end{array}\right)$$

Kernel for Exact Satisfiability

Algorithm

- Given formula F
- Rewrite by giving an linear equation for each clause
- Find a basis of the row-space
  - Use Gaussian elimination
- Remove constraints not in the basis

$$\left(\begin{array}{cccc|c} 1 & 0 & \dots & 1 & 1 \\ 1 & 1 & \dots & 0 & 1 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 0 & -1 & \dots & 1 & 0 \end{array}\right)$$

Kernel for Exact Satisfiability

Algorithm

- Given formula F
- Rewrite by giving an linear equation for each clause
- Find a basis of the row-space
  - Use Gaussian elimination
- Remove constraints not in the basis

$$\left(\begin{array}{cccc|c} 1 & 0 & \ldots & 1 & 1 \\ 1 & 1 & \ldots & 0 & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots \end{array}\right)$$

Kernel for Exact Satisfiability

Correctness

- Polynomial time
- yes-instance after removing constraints $\Rightarrow F$ is a yes-instance

Size

- Matrix size $(\text{\#clauses}) \times (n+1)$
- At most $n+1$ clauses remaining

Kernel for Exact Satisfiability: Recap

- Problem can be represented by linear equations
- Gives a linear kernel

Q: What about problems represented by equations of higher-degree polynomials?

Kernel for Exact Satisfiability: Recap

- Problem can be represented by linear equations
- Gives a linear kernel

Q: What about problems represented by equations of higher-degree polynomials?

Exercise

Find a kernel for the 3-NAE-SAT problem that has $O(n^2)$ constraints.

Input A formula in CNF-form where every clause contains exactly 3 variables, such as

$$F = \underbrace{\{x, y, z\}}_{\text{clause}} \wedge \{y, z, w\} \wedge \ldots$$

You may assume that the input contains no negations.

Parameter The number of variables.

Question Does there exist an assignment to the variables, such that each clause contains at least one *true* and at least one *false* variable?

Hint: Find a degree-2 polynomial f to represent a clause, for example

$$f(x, y, z) = 0 \Leftrightarrow \{x, y, z\} \text{ contains 1 or 2 } \textit{true} \text{ variables}$$

Solution: 3-NAE-SAT

Find a degree-2 polynomial to represent a clause. Start simple:

1. Single variable: $\{x\}$. Never satisfied.
2. Small clause: $\{x, y\}$. Satisfied iff $x + y = 1$
3. Consider $\{x, y, z\}$. Satisfied iff $x + y + z \in \{1, 2\}$,

$$\underbrace{x + y + z}_{\text{\#true vars.}} - \underbrace{(xy + xz + yz)}_{\text{\#true pairs}} = 1$$

d. Generally for clause $\{x_1, x_2, \ldots, x_d\}$ let
$g(t) := (t-1)(t-2)\cdots(t-d+1)$, define

$$f(x_1, \ldots, x_d) := g(x_1 + x_2 + \ldots + x_d)$$

General solution: d-NAE-SAT

Find a degree-$(d-1)$ polynomial g to represent a clause

Kernel:

- Construct such a polynomial for each clause
- Construct matrix A with one row for each clause
  - Row i contains all coefficients of the polynomial of clause i
- Do Gaussian elimination, find a basis
- Remove all clauses belonging to rows NOT in this basis

#clauses = #coefficients in the polynomials = $O(n^{d-1})$

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

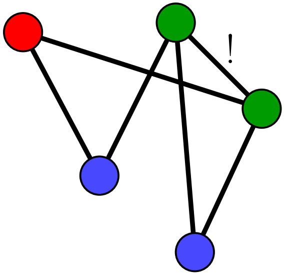
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

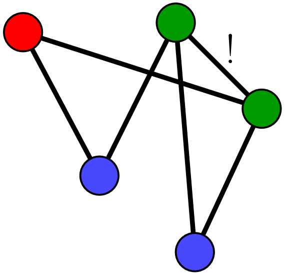
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

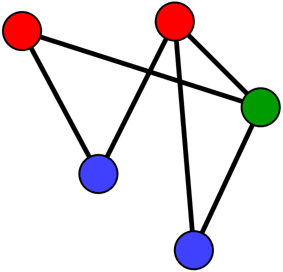
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

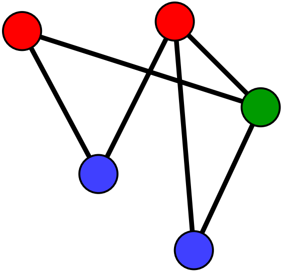
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

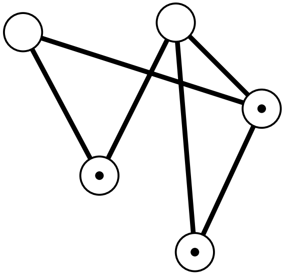
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

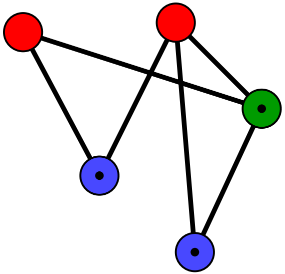
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



Problem:

There are many graphs with a small vertex cover

Example 3: 3-Coloring

Problem

Given: Graph G

Question: Is it possible to color each vertex of G with red, green or blue such that any two endpoints of an edge have a different color

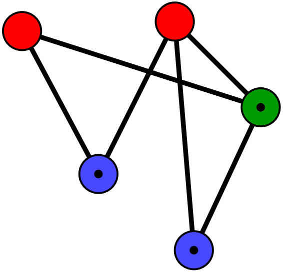
Parameter

The size of a vertex cover in G: a set S of vertices, such that for every edge at least one endpoint is in S

Goal

Find a small kernel (wrt $|S|$)

We assume S is given



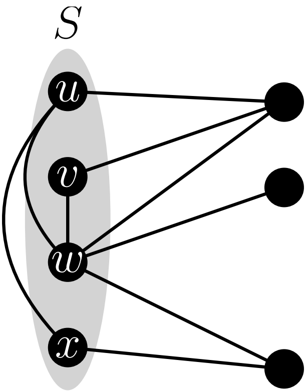
Problem:

There are many graphs with a small vertex cover

Kernel for 3-coloring

Put vertices in the vertex cover on the left, all others on the right

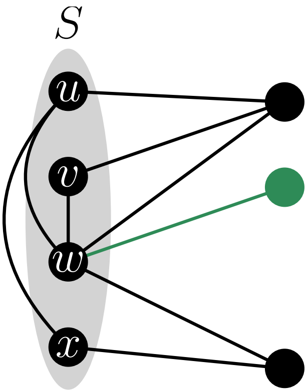
- Vertices not in S cannot be connected to each other
- When can we color a vertex not in S?
- Vertices with degree at least 3 in $V(G) \setminus S$ can cause a problem
    - Low-degree vertices in $V(G) \setminus S$ could safely be removed



Kernel for 3-coloring

Put vertices in the vertex cover on the left, all others on the right

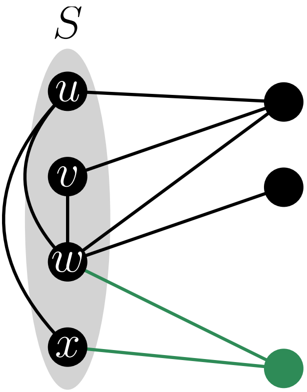
- Vertices not in S cannot be connected to each other
- When can we color a vertex not in S?
- Vertices with degree at least 3 in $V(G) \setminus S$ can cause a problem
  - Low-degree vertices in $V(G) \setminus S$ could safely be removed



Kernel for 3-coloring

Put vertices in the vertex cover on the left, all others on the right

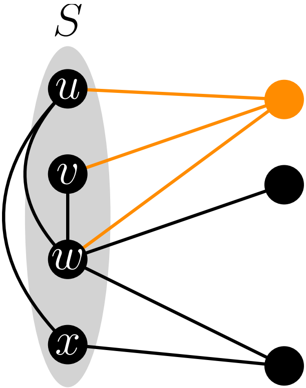
- Vertices not in S cannot be connected to each other
- When can we color a vertex not in S?
- Vertices with degree at least 3 in $V(G) \setminus S$ can cause a problem
  - Low-degree vertices in $V(G) \setminus S$ could safely be removed



Kernel for 3-coloring

Put vertices in the vertex cover on the left, all others on the right

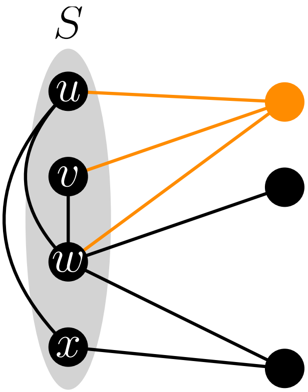
- Vertices not in S cannot be connected to each other
- When can we color a vertex not in S?
- Vertices with degree at least 3 in $V(G) \setminus S$ can cause a problem
  - Low-degree vertices in $V(G) \setminus S$ could safely be removed



Kernel for 3-coloring

Put vertices in the vertex cover on the left, all others on the right

- Vertices not in S cannot be connected to each other
- When can we color a vertex not in S?
- Vertices with degree at least 3 in $V(G) \setminus S$ can cause a problem
    - Low-degree vertices in $V(G) \setminus S$ could safely be removed



Kernel for 3-coloring

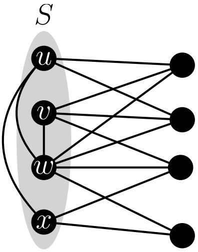
Algorithm

Given: graph G, vertex cover S

First, we mark vertices we want to keep:

- For any three distinct vertices u, v, w in S
- Mark one common neighbor in $V(G) \setminus S$

Remove all unmarked vertices in $V(G) \setminus S$



For u, v, w

- Mark one vertex

For v, w, x

- Mark one vertex.

Other triples have no common neighbor

Kernel for 3-coloring

Algorithm

Given: graph G, vertex cover S

First, we mark vertices we want to keep:

- For any three distinct vertices u, v, w in S
- Mark one common neighbor in $V(G) \setminus S$

Remove all unmarked vertices in $V(G) \setminus S$

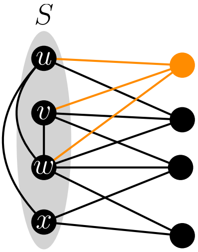
For u, v, w

- Mark one vertex

For v, w, x

- Mark one vertex.

Other triples have no common neighbor



Kernel for 3-coloring

Algorithm

Given: graph G, vertex cover S

First, we mark vertices we want to keep:

- For any three distinct vertices u, v, w in S
- Mark one common neighbor in $V(G) \setminus S$

Remove all unmarked vertices in $V(G) \setminus S$

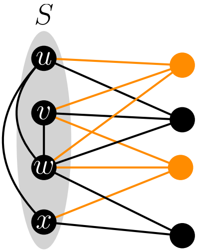
For u, v, w

- Mark one vertex

For v, w, x

- Mark one vertex.

Other triples have no common neighbor



Kernel for 3-coloring

Algorithm

Given: graph G, vertex cover S

First, we mark vertices we want to keep:

- For any three distinct vertices u, v, w in S
- Mark one common neighbor in $V(G) \setminus S$

Remove all unmarked vertices in $V(G) \setminus S$

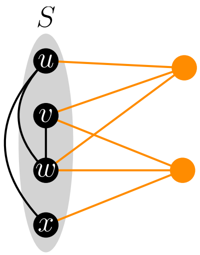
For u, v, w

- Mark one vertex

For v, w, x

- Mark one vertex.

Other triples have no common neighbor



Correctness

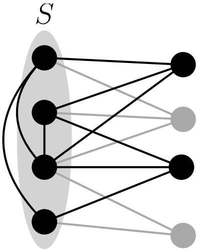
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $V(G) \setminus V(G')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

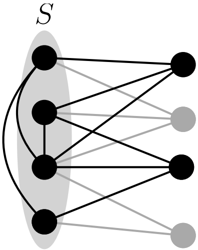
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $V(G) \setminus V(G')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

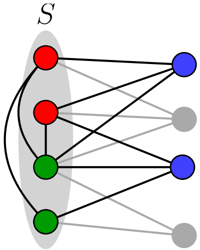
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $V(G) \setminus V(G')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

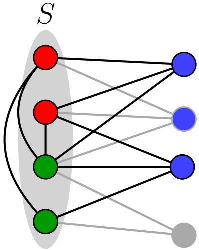
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $\mathrm{V}(\mathrm{G}) \setminus \mathrm{V}(\mathrm{G}')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

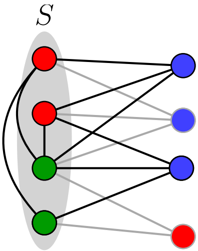
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $V(G) \setminus V(G')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

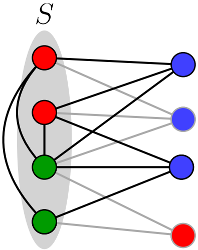
Let the original input be G and the new graph G'

Clearly if G is 3-colorable, so is G'. Suppose G' is 3-colorable, show how to color G

- Copy the coloring of G' to G
  - This colors all vertices in S
- For every vertex in $V(G) \setminus V(G')$ assign it a different color than any of its neighbors
  - All neighbors are in S, those colorings are known

Clearly this yields a valid coloring for G

- But we need to show that the last step is always possible!



Kernel for 3-coloring

Correctness

Suppose vertex y in $V(G) \setminus V(G')$ cannot be colored

- Thus, it has three neighbors u, v, w which are red, green and blue respectively
- But then there is a node y' in G' with neighbors u, v, w
  - since some vertex was *marked* by u, v, w
- Thus y' cannot be colored properly; contradiction

G' has a 3-coloring if and only if G has a 3-coloring

Kernel for 3-coloring

Size

Remember: parameter $|S|$. Count the vertices in G':

- All vertices in S, clearly there are $|S|$
- One vertex for all triples in S, at most $|S|^3$
- In total $O(|S|^3)$ vertices and
  - Possibly $O(|S|^4)$ edges with this kernel

Can we do better? Maybe we can reuse some previous insights:

- Find which vertices outside S are redundant

Kernel for 3-coloring

Size

Remember: parameter $|S|$. Count the vertices in G':

- All vertices in S, clearly there are $|S|$
- One vertex for all triples in S, at most $|S|^3$
- In total $O(|S|^3)$ vertices and
  - Possibly $O(|S|^4)$ edges with this kernel

Can we do better? Maybe we can reuse some previous insights:

- Find which vertices outside S are redundant

Kernel for 3-coloring

To find redundant vertices, use a method that finds redundant clauses

- Replace each vertex in $V(G) \setminus S$ by a constraint
- For every vertex $v \in S$ we have three $0/1$ variables
  - r_v, g_v and b_v indicating its color
- For every $v \notin S$ we have a constraint
  - Let v have neighbors 1, 2 and 3

$$\sum_{1 \leqslant i < j \leqslant 3} (r_i r_j + b_i b_j + g_i g_j) \equiv 1 \pmod 2$$

  - If all distinct colors: result is 0
  - Else, sum is 1 or 3
- Degree-2 polynomial, this allows us to find $O(n^2)$ relevant vertices and remove all others

Size: $O(|S|^2)$ vertices (and edges) remain

Lower bounds

Is a given kernel for a parameterized problem optimal?

- or is there a smaller kernel wrt the same parameter?

Therefore, we want lower bounds on kernel sizes.

- For which we need some assumptions
- For example; $\mathrm{P} \neq \mathrm{NP}$

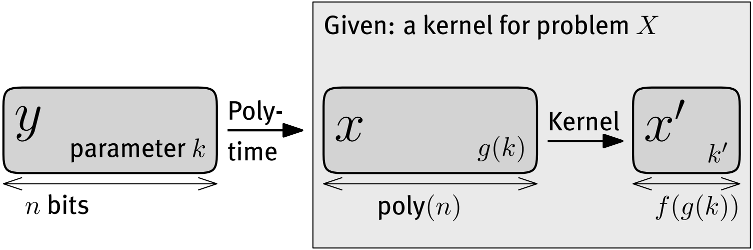
First idea:

Suppose: you already know some lower bound for some problem
Can we use it to get lower bounds for other problems?

Re-using existing kernels

A kernel for problem X can sometimes be used to get a (general type of) kernel for problem Y

3-Coloring $\longrightarrow$ Quadratic equations over $0/1$-variables

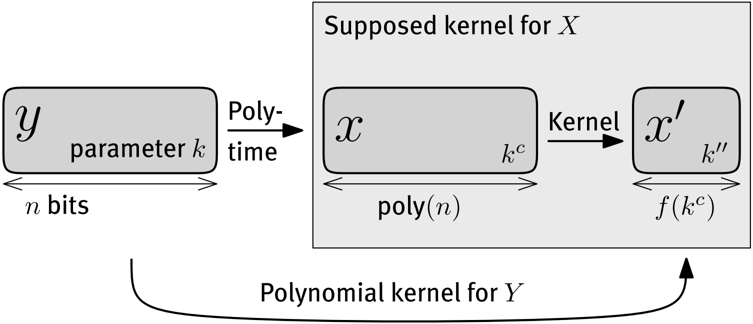


We can use this to transfer lower bounds
Very similar to giving NP-hardness reductions!

Re-using Lower bounds

GIVEN: problem Y has no (generalized) kernel of polynomial size
Suppose there is an algorithm that has as input (y, k) for problem Y and

- Outputs an instance (x, k') for problem X in polynomial time
- (x, k') is a yes-instance $\Leftrightarrow$ (y, k) is a yes-instance
- k' is bounded by k^c for a constant c

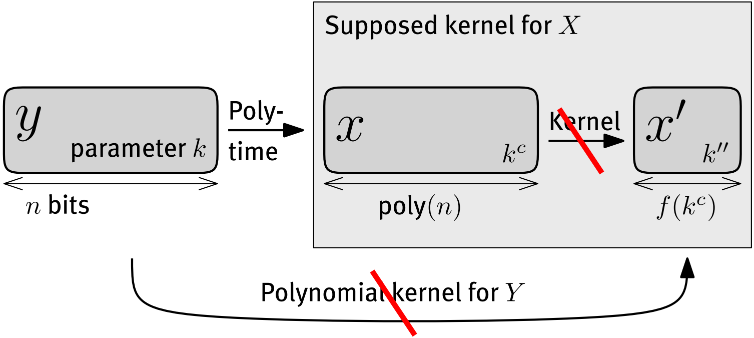


Re-using Lower bounds

GIVEN: problem Y has no (generalized) kernel of polynomial size

Suppose there is an algorithm that has as input (y, k) for problem Y and

- Outputs an instance (x, k') for problem X in polynomial time
- (x, k') is a yes-instance $\Leftrightarrow$ (y, k) is a yes-instance
- k' is bounded by k^c for a constant c



Lower bound: NAE-SAT

It is known that CNF-SAT with unbounded clause length does not have a kernel of size polynomial in the number of variables

- Transform an instance of CNF-SAT to NAE-SAT, to get a lower bound
  Add new variable t to each clause

  $$F = (x \vee y) \wedge (x \vee \neg y \vee z) \wedge (\neg w \vee x)$$

  becomes

  $$F' = \{t, x, y\} \wedge \{t, x, \neg y, z\} \wedge \{t, \neg w, x\}$$

- Can be done in polynomial time
- The number of variables only increased by 1
- It remains to show F is satisfiable iff F' is

Lower bound: NAE-SAT

$$F = (\quad x \vee y) \wedge (\quad x \vee \neg y \vee z) \wedge (\quad \neg w \vee x)$$
$$F' = \{t, x,\quad y\} \wedge \{t, x,\quad \neg y,\quad z\} \wedge (t, \neg w,\quad x)$$

Suppose F is satisfiable

- Keep the same assignment for F', extended with $t :=$ *false*
- Every clause in F' has a *false* variable (namely, t)
- and one *true* variable (from satisfying F)

Suppose F' is satisfiable

- If $t =$ *false*, keep the same assignment for F
- If $t =$ *true*, take the opposite assignment for F

Exercises

1. Does Exact-SAT have a polynomial kernel when parameterized by the number of variables?

2. If all clauses have size d, CNF-SAT with n variables does not have a kernel of size $O(n^{d-\varepsilon})$, for any ε.

   a. Can we use this bound to obtain a bound for NAE-SAT with d variables per clause? How?

   b. Can we hope to improve the kernel for 3-NAE-SAT with $O(n^2)$ constraints that we saw in a previous exercise?

Solutions

1. Does Exact-SAT have a polynomial kernel when parameterized by the number of variables?
   Yes, we have seen that the number of constraints can be reduced to $O(n)$ *giving a kernel of* $O(n^2 \log n)$. *This implies that there is no reduction from CNF-SAT to Exact-SAT*

2. If all clauses have size d, CNF-SAT with n variables does not have a kernel of size $O(n^{d-\varepsilon})$, for any ε.

   a. Can we use this bound to obtain a bound for NAE-SAT with d variables per clause? How?
      Yes (proof on whiteboard)

   b. Can we hope to improve the kernel for 3-NAE-SAT with $O(n^2)$ constraints that we saw in a previous exercise?
      Not really, there is no kernel of size $O(n^{2-\varepsilon})$. *But improvements by logarithmic factors could still be possible.*

Questions?